Refactoring the UrQMD Model for Many-Core Architectures

Dissertation

zur Erlangung des Doktorgrades

der Naturwissenschaften

vorgelegt beim Fachbereich Informatik und Mathematik

der Johann Wolfgang Goethe-Universität

in Frankfurt am Main

von

Jochen Gerhard

aus Idar-Oberstein

Frankfurt 2013

(D30)

vom Fachbereich Informatik und Mathematik der

Johann Wolfgang Goethe-Universität angenommen.

Dekan: Prof. Dr. Thorsten Theobald

Gutachter: Prof. Dr. Volker Lindenstruth, Prof. Dr. Marcus Bleicher

Datum der Disputation: 25.07.2013

Abstract

Ultrarelativistic Quantum Molecular Dynamics is a physics model to describe the transport, collision, scattering, and decay of nuclear particles. The UrQMD framework has been in use for nearly 20 years since its first development. In this period computing aspects, the design of code, and the efficiency of computation have been minor points of interest. Nowadays an additional issue arises due to the fact that the run time of the framework does not diminish any more with new hardware generations.

The current development in computing hardware is mainly focused on parallelism. Especially in scientific applications a high order of parallelisation can be achieved due to the superposition principle. In this thesis it is shown how modern design criteria and algorithm redesign are applied to physics frameworks. The redesign with a special emphasise on many-core architectures allows for significant improvements of the execution speed.

The most time consuming part of UrQMD is a newly introduced relativistic hydrodynamic phase. The algorithm used to simulate the hydrodynamic evolution is the SHASTA. As the sequential form of SHASTA is successfully applied in various simulation frameworks for heavy ion collisions its possible parallelisation is analysed. Two different implementations of SHASTA are presented.

The first one is an improved sequential implementation. By applying a more concise design and evading unnecessary memory copies, the execution time could be reduced to the half of the FORTRAN version's execution time. The usage of memory could be reduced by 80% compared to the memory needed in the original version.

The second implementation concentrates fully on the usage of many-core architectures and deviates significantly from the classical implementation. Contrary to the sequential implementation, it follows the *recalculate instead of memory look-up* paradigm. By this means the execution speed could be accelerated up to a factor of 460 on GPUs.

Additionally a stability analysis of the UrQMD model is presented. Applying metaprogramming UrQMD is compiled and executed in a massively parallel setup. The resulting simulation data of all parallel UrQMD instances were hereafter gathered and analysed. Hence UrQMD could be proven of high stability to the uncertainty of experimental data.

As a further application of modern programming paradigms a prototypical implementation of the worldline formalism is presented. This formalism allows for a direct calculation of Feynman integrals and constitutes therefore an interesting enhancement for the UrQMD model. Its massively parallel implementation on GPUs is examined.

Acknowledgements

Following a famous metaphor, we all are but dwarfs standing on the shoulders of giants. One does not climb this metaphorical shoulders alone, but one meets many helpful hands on the way. Here I wish to thank those hands for helping and pointing into the right direction.

Firstly, I thank my supervisor Prof. Dr. Volker Lindenstruth. He introduced me to the field of parallel computing with his unparalleled *down to the bare silicon* approach. An approach where he jokingly insisted that a Mathematician has to get his *hands dirty*, where pure abstraction is nice and fine, but *real world problems* need careful precise work on the details. Being a physicist by education and a computer scientist by profession himself, he states an ideal example for a scientist who builds a bridge between the faculties.

Secondly, I thank my supervisor Prof. Dr. Marcus Bleicher for introducing me to the field of heavy ion physics. In countless talks he explained the winding alleys of the simulation models, showed me gems in theoretical physics, and proposed different interesting possibilities for future research projects often pointing beyond the classic faculties.

I thank Hannah Petersen, author of the hybrid model, for providing me with the initial code base and various supporting and encouraging talks.

I wish to express my gratitude to my co-authors of various papers: Dr. Klaus Aehlig, Dr. Björn Bäuchle, Helge Dietert, and Dr. Thomas Fischbacher. I have cited different of our papers in this thesis, but even beyond our collective work distilled in this papers, fruitful discussions and ideas were helping hiking marks. Additionally I thank Sebastian Kalcher for proofreading this thesis.

I thank my parents, Karin and Karl Gerhard and my parents in law, Gisela and Erwin Seitz, for their constant moral support in this enterprise. The same support I got from Andrea and Harald Kropp, who I wish to express my thanks at this place. A special thank for perpetual encouragement goes to my wife Tamara.

Last but not least, I wish to thank the Frankfurt Institute for Advanced Studies. At this institute I found an ideal place for this interdisciplinary work between computer science and physics.

Contents

List of Figures

List of listings

1. Introduction

1.1. Molecular Dynamics and Relativistic Quantum Molecular Dynamics

Molecular Dynamics (MD) is a computational approach to study the collective movement and interaction of particles, such as molecules. In each time step the potentials and equations of motion for each particle are calculated. The resulting future position of each particle is calculated in discrete time steps. Molecular Dynamics (MD) is used in theoretical chemistry, biology and materials science. This model aims mainly, from a particle physicist's point of view, at macroscopic objects and their interacting and forming forces. First ideas of MD are mentioned in [6]. Nowadays MD is applied for medical research, e.g. protein folding, and the investigation of new surface structures in materials science [38, 36]. Due to the pressing need of compute power in research and development the usage of modern many-core architectures like GPUs is widely present in MD applications [64, 22, 32, 65].

Relativistic quantum molecular dynamics, e.g. UrQMD, is an effective model for simulations of nuclear collisions and scatterings. It combines approaches of different branches of physics. Starting from relativistic motions and the Boltzmann ansatz, it includes pQCD phenomenology, hadron physics, and a multitude of smaller models. The principal goal of relativistic quantum molecular dynamics (RQMD) is the study of matter under extreme conditions. The single most referred state is the Quark-Gluon-Plasma (QGP), a special state of matter that only existed at the very beginning of the Universe. The reproduction of the QGP is a goal of heavy ion collision experiments.

Simulation frameworks like the Frankfurt UrQMD framework have to incorporate these different approaches and areas of physics in order to provide a realistic description of the complex processes taking place in particle collisions. These frameworks obey a certain evolutionary process, as new findings from experimental data as well as new ideas from theoretical physics have to be integrated and often even have to substitute older approaches and algorithms. As always in physics, a model is only as good as its predictions, hence the results of software frameworks are compared to experimental data and thereby validated (and not verified) on their predictive outcome.

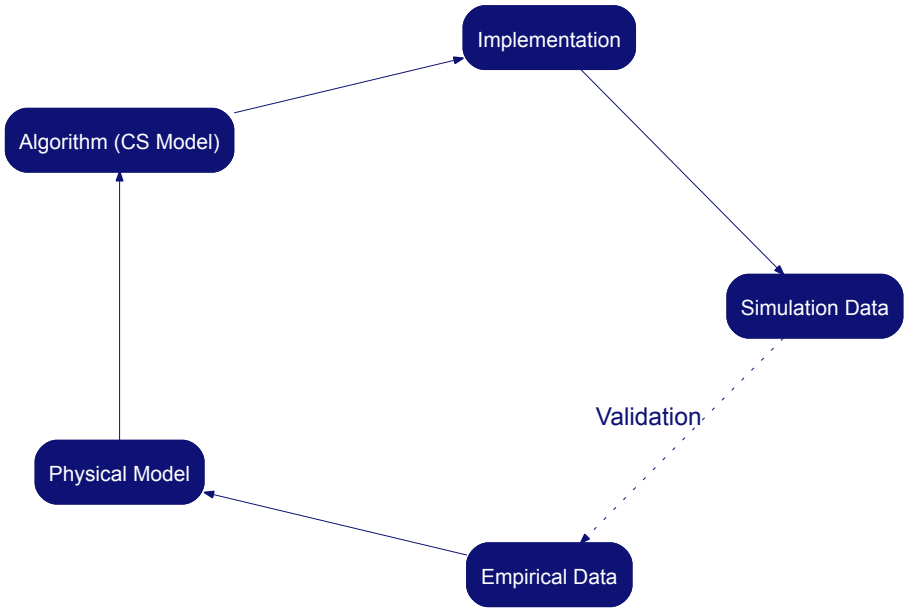


Figure 1.1.: The process of modelling and implementation. For a successful redesign of legacy systems (like UrQMD) on many-core architectures the implementation side is not the most crucial part. In order to gain significant benefits of modern architectures new models on the computational side, i.e. algorithms suited to hardware and physics model, must be examined.

In the next section I will outline the physics simulated in the Frankfurt UrQMD framework. In order to port legacy frameworks, like the UrQMD framework, to modern architectures an in-depth understanding of the underlying physics principles is crucial.

1.1.1. The underlying physics of MD and RQMD

In MD the modelled particles are often molecules or atoms. Usually MD simulations include an ensemble of empirical potentials. These potentials, such as the *Lennard-Jones* potential, are often calculated for each pair of participating particle in the simulation. The resulting potentials are used to compute the acting forces on all particles and finally provide the acceleration for the underlying Newtonian motion of these particles. The origin of this empirical potentials can be seen in *quantum electrodynamics* (QED), i.e. the theory of the *electrical force*.

On the other side, the modelled particles in RQMD are the constituents of atoms, i.e. protons and neutrons, and more general, all hadrons. Although QED has to be integrated into RQMD simulations, the dominating potentials between hadrons result from *quantum chromodynamics* (QCD), i.e. the theory of the *strong force*.

Although in both models first principal calculations, i.e. calculations directly with QED or QCD are possible for individual systems, this approach is hardly used in bigger systems. The simulation of bigger systems, e.g. protein folding in MD or the collision of lead-ions in RQMD, is carried out by so called effective models. Effective models apply empirical potentials or theoretical approximations to the full quantum field theoretical description of the potentials. Nevertheless the differences between QED and QCD necessitate the usage of completely different computational approaches.

1.1.2. A brief overview of QED and QCD

Quantum electrodynamics is the theory for the interaction of electrical charged particles, e.g. electron and its heavier partner the muon. The carrier of all forces, the gauge boson, is the (virtual) photon. Photons do not interact with each other directly. Self interaction by a splitting into virtual proton and electron pairs e^+e^- is possible in higher order, though. The Feynman diagram in figure 1.2 shows this reaction. However, it is suppressed by the electroweak coupling α_{ew}. The left side shows an incoming photon which splits up into an electron and a positron in the first vertex. They meet again in the second vertex and produce the outgoing photon. It is because of the non-interaction of photons that optics is linear, i.e. can be handled by geometrical optics.

Quantum chromodynamics is the theory of particles with colour[1] charge. These are e.g. the six quarks: up, down, strange, charm, top, and bottom. Each one of the quarks carries a fractional electric charge and one of three colour charges: red, green, and blue. The two lightest quarks, up and down, build the protons and the neutrons. However, in particle collisions additional hadrons are produced, consisting of various combinations of quarks.

The gauge bosons responsible for the forces between the quarks are the gluons. Contrary to the QED case, where the photon does not carry electrical charge, in the QCD case the gauge bosons carry a colour charge. This implies an additional complexity of this theory. In the QED case electrical charged particles do act on each other due to photon exchange. The photons however are not charged and rest unaffected of any electrical charge.

[1] Of course the "colour" in QCD is a different matter than the visible colour, as the size of the quarks makes them invisible to observable light.

Due to the colour charge of the gluons and their possibility to change their colour, four kinds of interactions have to be modelled:

- Quark with quark.
- Quark with gluon.
- Gluon with gluon.
- Gluon with itself.

Not only the increased number of gauge bosons renders QCD more complicated than QED, but a whole class of new interactions has to be modelled additionally. The basic case of this self interaction is shown in figure 1.2. On the right side such a process for an incoming gluon is shown. Here the gluon splits up into two gluons in the first node. They join again in the second node producing an outgoing gluon. The coupling constant measures, the interaction strength at each vertex. The vertices in QCD (figure 1.2, right side) couple by a factor of 137 stronger than the vertices in QED (figure 1.2, left side). Therefore the rare case of photon self interaction (via intermediate electron - positron pairs) can be neglected more easily than the gluon self interaction.

Another difference is given by the long range potentials in both theories. While single electric charges (e.g. single electrons) can be studied, single colour charges (e.g. quarks) never occur. They occur in *white* triplets and form the nucleons like protons and neutrons, or coupled with the respective anti-quark forming a meson. To examine the interaction purely between quarks, and not between protons and neutrons, big accelerator experiments like the LHC at CERN and FAIR at GSI are build. Only at high momentum transfers (i.e. temperatures) or at extreme densities the quarks within nuclei behave freely and one can study their interaction. This phenomenon is called asymptotic freedom. Due to the high collision energy, a simulation automatically has to cope with a high particle multiplicity and relativistic effects. Another entanglement in RQMD contrary to MD is given by the spacial extension of the systems. While most quantum effects average out on macroscopic objects, like molecules, they have to be carefully integrated to simulations concerning objects of the size of a proton or smaller. In addition to the high multiplicities created by the nuclei of atoms, any excess energy can create further particles during a collision. The complexity of accelerated and colliding nuclei is, for today's computers, far too high to be simulated by first-principle calculations, like real time Lattice-QCD.

Effective models on the other hand try to simulate the collective behaviour of many particles at once. For instance the collision of two lead ions taking place in the LHC. In order to cope with the combinatorical explosion due to high multiplicities[2] the effective models apply different physical principles. The non interacting movements of particles

[2]Even the detection and analysis is a computationally demanding problem, as can be seen in [31].

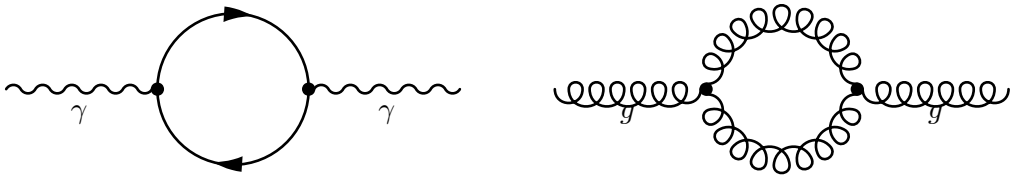


Figure 1.2.: Photon self interaction is only possible via intermediate electron-positron decomposition, which is suppressed by $\frac{1}{137^2}$. Gluons can interact with themselves due to colour charge.

can be modelled with relativistic mechanics, the particles' position in the phase space can be modelled by multidimensional Gaussians. The empirically found branching ratios, i.e. the probability of a particle to decay into other particles, can be implemented together with a bookkeeping and a pseudo random number generator in order to simulate the cascading scatterings and decays in a nucleus-nucleus collision. A part of the collision can be modelled by relativistic hydrodynamics, going back to a famous paper of Landau [9].

All these different approaches make use of a wide selection of algorithms and numerical methods. When simulating different energy regimes and phases of a particle collision different processes are dominating the reactions. Therefore these different processes have to be orchestrated very carefully to assure stability of the model and efficiency of its computation. An additionally important point is the extensibility and flexibility of the full integrated model. New results and theoretical developments should be integrated quickly into the model, and falsified approaches should be substituted by more accurate approaches.

Unfortunately many implementations lack the needed modularity, as they are created on the fly, when new approaches have to be tested out quickly. Often though, the prototypes reside in the model and become deeply entangled in the core routines over time. This leads to inconsistencies in interfaces. They are often not designed in order to encapsulate functionality, but in order to get the desired data out as fast as possible. Particularly the usage of `common blocks` in FORTRAN in this way leads to an humongous memory requirement of simulation software. Often these `common blocks` are enhanced by various fields which are of interest for a current project and later on relied on as recyclable memory region. This leads to the situation that the total size of all `common blocks` is limited only by the system's maximal stack size.

1.2. The Frankfurt UrQMD Framework

The Ultra-relativistic Quantum Molecular Dynamics (UrQMD) model [8, 10] is a simulation framework for heavy ion collisions. It originates in the *Quantum Molecular Dynamics*

model of J. Aichelin and H. Stöcker [5, 62]. To enhance the applicable energy regime it includes the PYTHIA framework [61]. A recent addition was a hydrodynamic module [52]. Hence, in more than twenty years of development a multitude of physical approaches, and accordingly algorithmic solutions, has been implemented. UrQMD is a valuable asset to the Frankfurt physics community and under constant development. However, recent developments in hardware and software design have not been implemented in the UrQMD core framework. Due to the usage of FORTRAN 77 with a rather "goal-oriented programming approach", a well planed redesign down to the algorithmic approach is necessary.

In my thesis I concentrate on four aspects of this redesign:

- The analysis and implementation of an approach for quantum field theoretical calculations based on the direct calculation of Feynman integrals. Using a separate hybrid Python and OpenCL code-base the *worldline formalism* [47] is analysed specially for its parallelisation capabilities on many-core architectures.
- A stability analysis of the UrQMD model. By application of an metaprogramming approach the stability of UrQMD is tested against variations of the experimental input data.
- An object oriented approach using C++. Carried out in the hydrodynamic subsystem of UrQMD, it illustrates how object orientation can help to maintain a high readability even in complex physics frameworks.
- A hardware oriented redesign of the hydrodynamic calculation in C++ and OpenCL which applies metaprogramming, object oriented design, and many-core specific optimisations.

1.3. Legacy Code

As many other packages in theoretical physics UrQMD is entirely coded in FORTRAN 77. The transition from the *problem oriented* coding used in UrQMD to *structured programming* [18] or even *object oriented programming* is aggravated by the numerous usage of `common blocks` and entangled `gotos` which leads to the infamous "spaghetti code".

In order to benefit from new developments in hardware architecture, a redesign to a more structured algorithm is necessary. An aim of this thesis is to demonstrate how legacy code, like UrQMD, benefits from such a structured redesign. The benefits are not only to be found in the optimised maintainability and extensibility of the code basis. In fact, only the redesign enables the usage of modern many-core architectures, like GPUs.

Different approaches can be taken to redesign a present code basis. The high performance (HPC) community relies heavily on FORTRAN-code. Therefore significant endeavours have been made in order to provide the compute power of GPUs even for legacy FORTRAN systems, e.g. a FORTRAN compiler of the Portland Group [53] suited to the usage of NVIDIA's *compute uniform device architecture* (CUDA). Additionally automated approaches, like `f2c` from FORTRAN to C++, are investigated, e.g. to port computational fluid dynamic (CFD) codes to CUDA with python-parsers [15].

However, these automated procedures can not provide the full spectrum of optimisations and often parallelise the simplest loop structures only. This is due the lack of knowledge of the underlying problem to the optimiser. The compiler can only work on the given source code, describing one possible implementation. To the programmer, on the other hand, the complete description of the physics model is known. He has therefore the choice of redesigning even on the complete algorithmic level and not only on the implementation level. Additionally the source codes generated by automated methods lack readability, as their generation is laboured to be correct for the compiler and often does not foresee any interaction with a programmer. Therefore the maintainability of partially converted simulation frameworks decreases further. Hence the first step towards the usage of GPUs is the redesign of legacy codes, which hinder parallelisation due to their nested structures [63].

In its continuous development different modules of UrQMD have been devised for parts of the simulation. These range from bookkeeping tasks over the interface routines for PYTHIA to various subroutines for string fragmentation and other quantum mechanical mechanisms. An overview of the necessary modules can be seen in figure 1.3. The introduction of the hydrodynamic module has posed an optimal starting point for the redesign of UrQMD. As can be seen in figure 1.3, the hydro module is self contained (`1fluid.f`) and does not interfere with other sub-modules of UrQMD.

Additionally the profiling information of UrQMD reveals that the hydrodynamic calculation dominates the execution time, as shown in figure 1.4 and figure 1.5. Therefore, as a

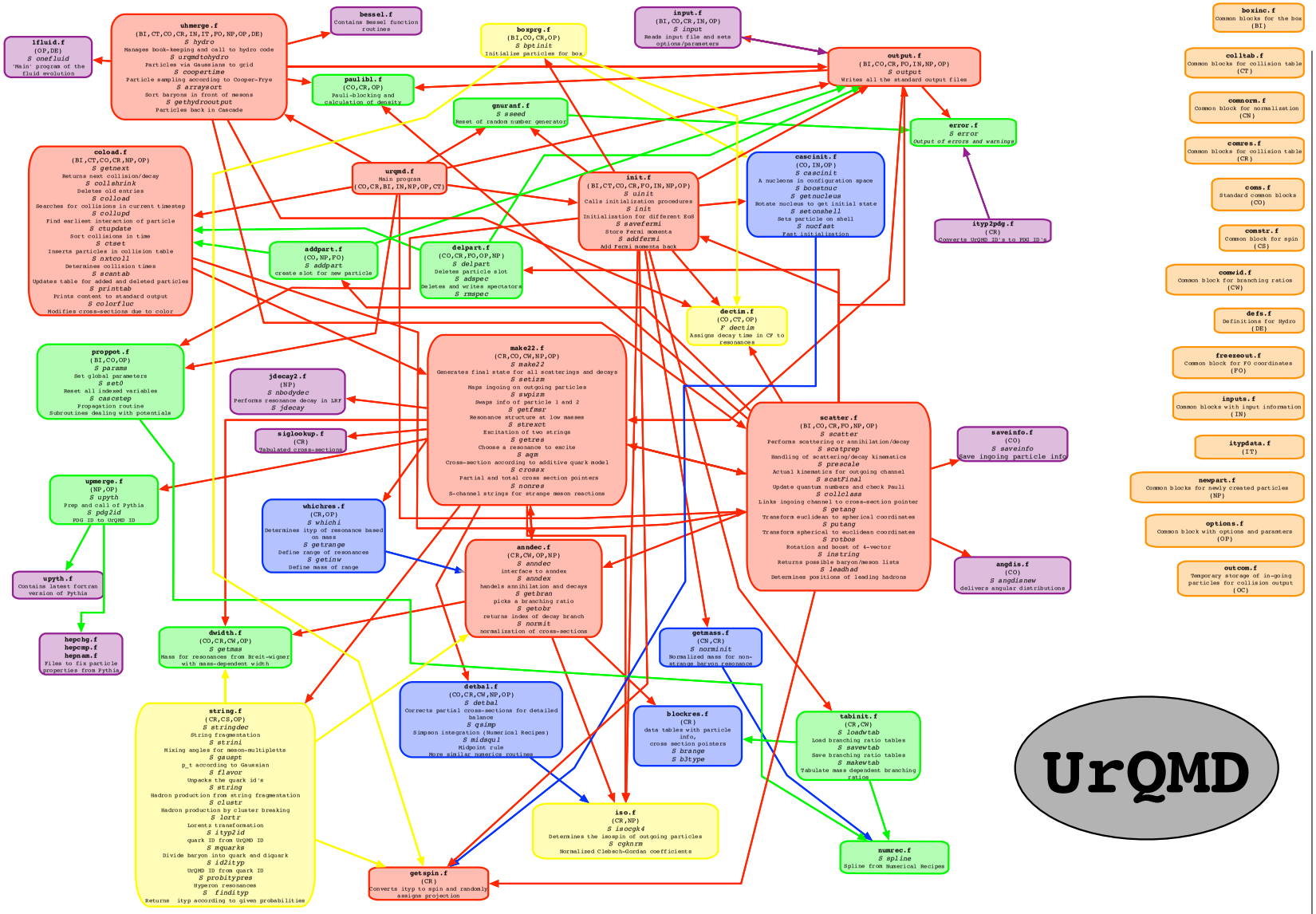


Figure 1.3.: The organisation chart of UrQMD on file level [51].

consequence of Amdahl's law, the parallelisation of `onefluid()` in figure 1.5 is the most promising candidate to gain an overall speedup for UrQMD.

The redesign of legacy frameworks, like UrQMD, including the developments of algorithms suited to the physics models and modern hardware architectures, is clearly an endeavour of several years of scientific work. However, this thesis will outline the analysis, redesign process, and integration of new features for different core aspects within the UrQMD framework under the consideration of many-core architectures.

1.4. Heterogeneous Computing

The usage of modern computer architecture poses a major challenge to legacy systems, like UrQMD. Notwithstanding that frameworks for parallel design of algorithms have been used widely in HPC applications of theoretical physics, the classic approaches of using MPI or even OpenMP are not sufficient to benefit from the vast choice of modern computer architectures.

Over the last years the increase in performance of computing hardware has not been realised by higher frequencies of the chips but by an increase of parallelisation (see figure 1.6). In its original form Moore's law is still correct and the increase of the transistor count still results in a gain of computational power. However, nowadays this computational power is accessible only by the usage of parallel algorithms.

Additionally the vast selection of different hardware types, not only different vendors, but completely different computing models poses a heavy burden to any possible refactoring attempt. Depending on the necessities different hardware types can be used:

- Different CPUs, with a number of virtual or real cores, accompanied by sophisticated hardware routines for branch prediction, pipelining, out-of-order execution etc. .
- GPUs, with a multitude of cores, optimised for floating point operations, and a distinct memory hierarchy. (A development driven mainly by the demands of modern computer games).
- FPGAs, that can be programmed to fit perfectly to the desired execution stream of the needed programs.
- Cell broadband engines, featuring a low latency interconnect.
- Accelerators like *Intel Many Integrated Core* (Intel MIC), designed only to fit the needs of the HPC community, offering large vector sizes.

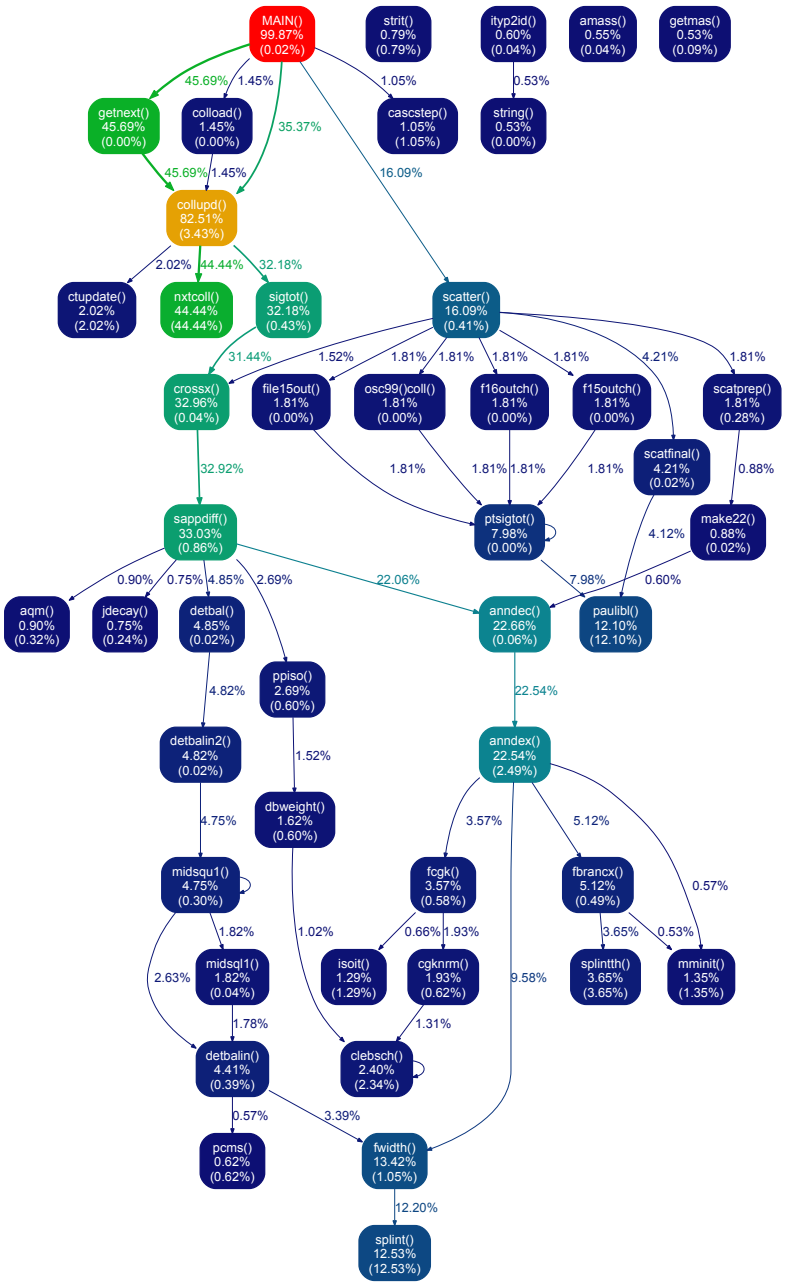


Figure 1.4.: The call graph of UrQMD without hydrodynamics for a 3 GeV Au+Au collision. Methods that consume less than 0.5% of time are suppressed. Due to the time consumption of the hydrodynamic routines in figure 1.5 a finer-grained resolution is gained in this graph.

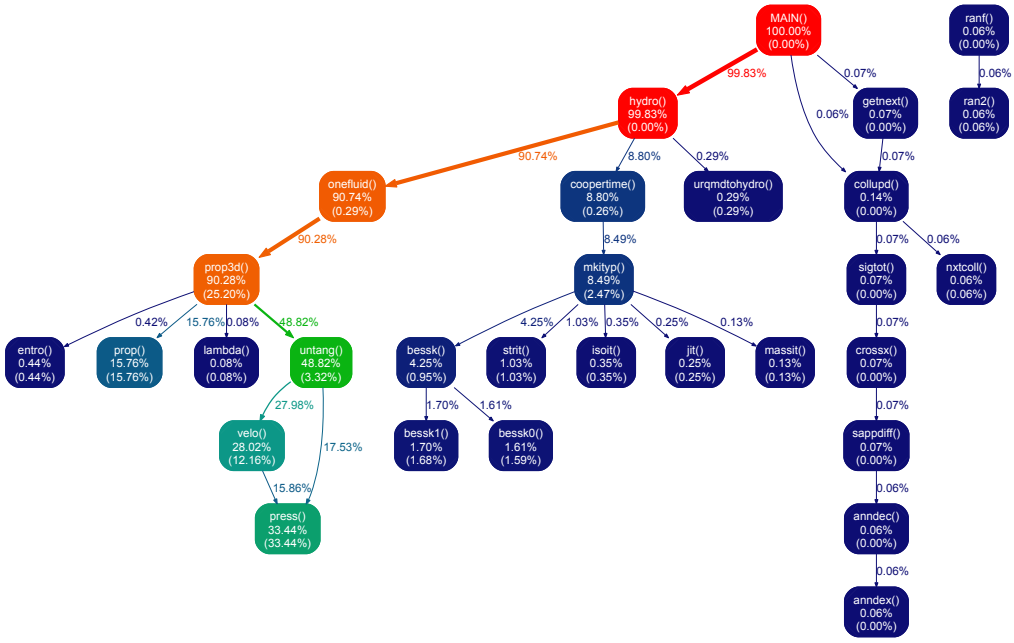


Figure 1.5.: The call graph of UrQMD with hydrodynamics for a 3 GeV Au+Au collision. Methods that consume less than 0.05% of time are suppressed. Each node contains the name of the method, the part of the execution time spend within this method (or below), and in parentheses the time exclusively spend in the method. The colour of the node is coded in a blue to red scheme according to the execution time. The edges of the graph are labelled with the percentage of calls to the child nodes.

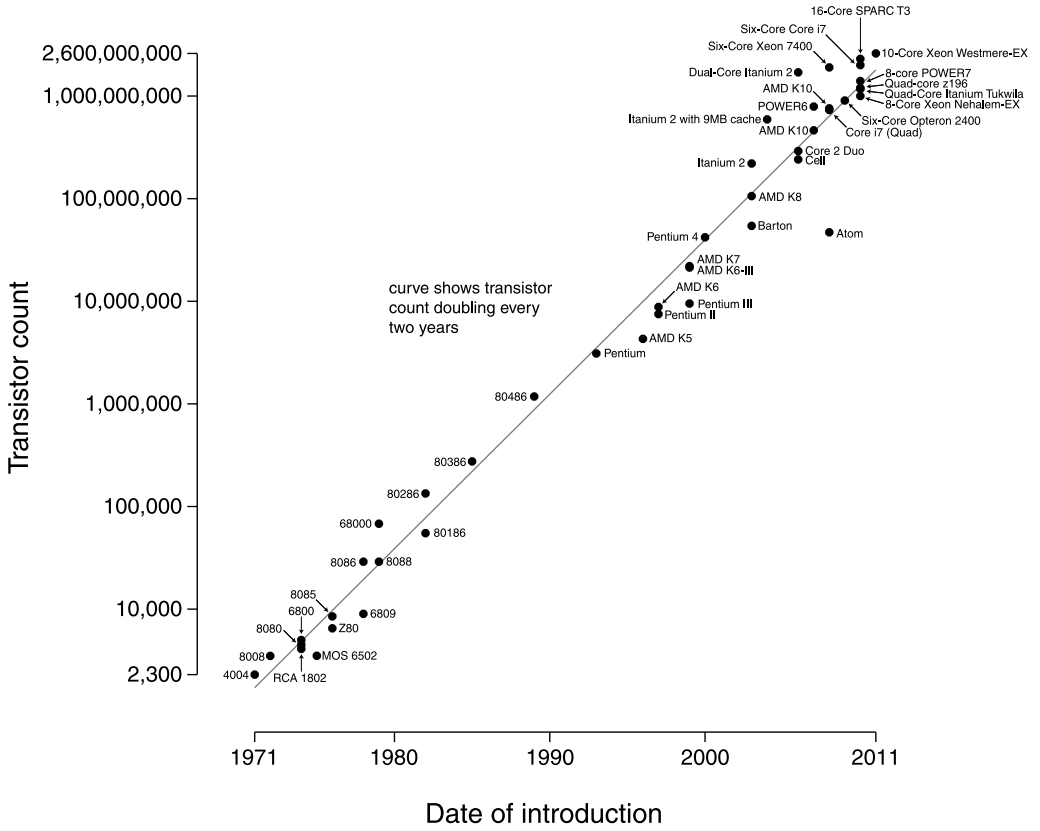


Figure 1.6.: Moore's law in its original form states the doubling of the transistor number of an integrated circuit every two years. (Figure taken from [66].)

	Single Data	Multiple Data
Single Instruction	single-core CPU	GPU
Multiple Instruction	—	multi-core CPU

Table 1.1.: Flynn's taxonomy.

Often a mixture of different devices offers the best capabilities for the needs of research groups[3]. In this heterogeneous compute architecture the choice of language and parallelisation concept is crucial. For this reason the *Open compute language* (OpenCL) is designed to put the necessary abstraction layers onto the physical existent hardware. Designed by a non profit industry consortium, OpenCL is devised to be a royalty-free and open standard. Thus a possible *vendor lock-in* can be avoided.

However, even before the development of state of the art programming frameworks, the promising performance in *number crunching* of GPUs was sought-after. Noteworthy in the high-energy physics community are the Lattice-QCD calculations by [19]. But only the usage of programming frameworks like OpenCL allows to rely on a clear algorithmic design of physics models, without the unnecessary obfuscation by complicated graphics operations.

The rather smooth transition from multi-core to many-core programming can best be characterised not by the mere number of compute cores, but by the capabilities of these cores. On the one hand, classic processors are equipped with different arithmetical-logical-units (ALUs) responsible for floating point and integer calculations. They use sophisticated cache structures, complex branching hardware (including branch predictions), efficient pipelines, out-of-order execution, and (comparatively) small vector registers. On the other hand, many-core hardware, like GPUs, makes use of highly efficient floating point ALUs, with small cache structures supporting an hierarchy of shared memories. Here the execution stream is grained, i.e. fixed groups of ALUs work always together and execute the same operations per cycle. Within Flynn's taxonomy the present GPU hardware can be seen as best suited for the *single instruction multiple data* (SIMD) case (compare table 1.1). As modern GPUs are more flexible than the classical SIMD definition, the term *single program multiple data* (SPMD) has been coined. This is due to the fact that the execution stream is not totally parallel on modern GPUs. Within a (model specific) granularity the execution streams of GPUs may deviate from each other.

Although for UrQMD I concentrate mainly on the usage of GPUs, the usage of OpenCL has allowed to execute the redesigned programs on other hardware types too. The performance details are discussed in chapter 5.

[3]The Frankfurt LOEWE-CSC is a realisation of this heterogeneous computing paradigm using AMD CPUs and GPUs.

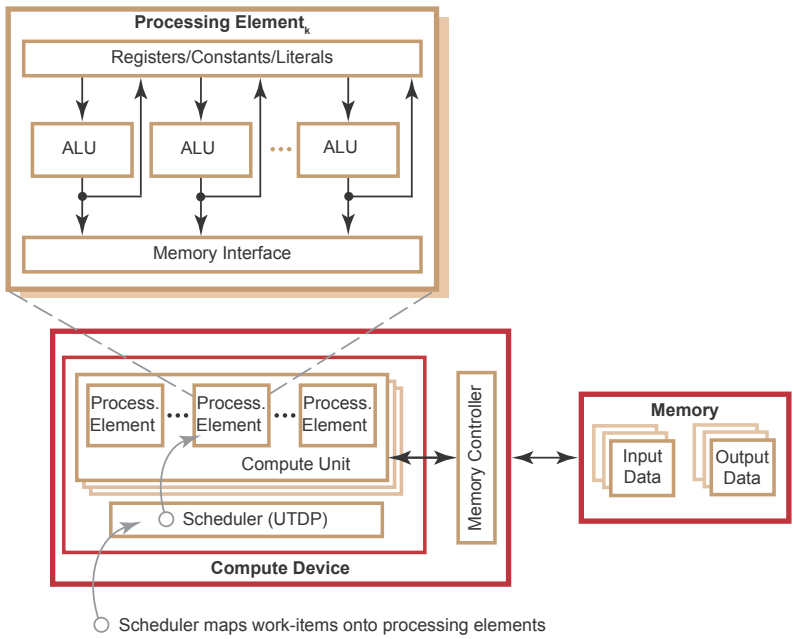


Figure 1.7.: A schematic diagram of an AMD GPU. (Figure taken from [1] and slightly modified.)

1.4.1. The OpenCL Abstraction of GPUs

The structure of GPUs still changes rapidly from model to model and from vendor to vendor, hence the nomenclature is often to be called blurry. I will follow mainly the nomenclature of current AMD GPUs, which deviates slightly from the NVIDIA nomenclature. OpenCL provides the compute capabilities of GPUs through an abstraction to the actual hardware. A common structure among all devices, cores, ALUs, and memory types is assumed in the OpenCL standard. Additional features of the actual hardware maybe accessed with hardware specific extensions to the standard. However the usage of this extensions comes at the price of reduced portability. To ensure the maximal portability I have avoided any hardware specific extension in the following.

The Platform Model

In figure 1.7 a schematic view of an AMD GPU is depicted. The shown structure is typical for all modern GPUs. A GPU consists of a number of *compute units* which are controlled independently by a scheduler, called *ultra-threaded dispatch processor* (UTDP) in AMD's nomenclature. The scheduler is responsible for the mapping of the tasks of the parallel program to the different compute units and *processing elements* on the GPU.

The compute units are subdivided into processing elements. A processing element can be seen as a (virtual) scalar core. However, it may be composed of different ALUs and offer e.g. instruction level parallelism.

The OpenCL standard defines an independence between different compute units, hence compute units must be able to work with different instruction streams in parallel. Within a compute unit, though, the standard allows both: a SIMD or a SPMD structure.

All processing elements within one compute unit can be executed in a lockstep manner, i.e. all elements in one unit execute the exact same instructions at the same time. In this case, a compute unit realises a SIMD architecture. In the other extreme, every processing element maintains an independent program counter, this realises a SPMD architecture even within one compute unit. However, often the ALUs within one compute unit are grouped to small functional units and are equipped with an independent program counter. The granularity given by this grouping is called a *wavefront*. A wavefront exhibits the finest granularity, in which the execution stream of a program may deviate from a pure lockstep execution of instructions, within the wavefront a pure SIMD behaviour is realised.

The present hardware in a compute node is managed through the OpenCL *platform model*, as shown in figure 1.8. Typically a classical C or C++ program uses the OpenCL framework to manage one or more *compute devices*. The compute devices can be different GPUs in one system, or even the present CPU. The management program is executed in the typical manner on the used node, called *host*, and makes the different compute devices available. The AMD implementation of the OpenCL standard allows the usage of the present (AMD) GPUs together with the CPUs of the node.

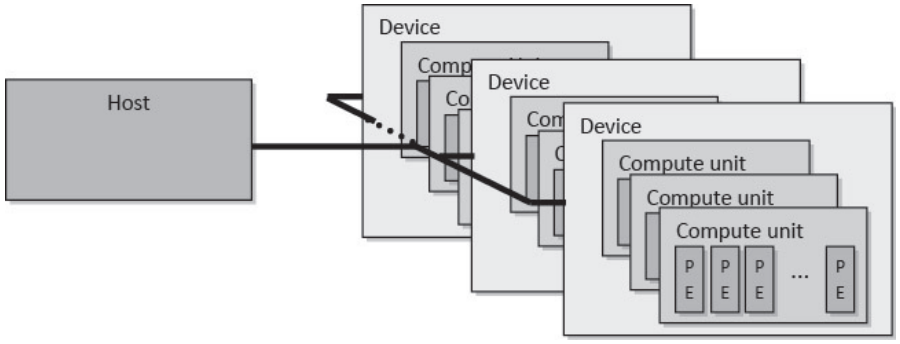


Figure 1.8.: The OpenCL Platform Model. (Figure from [23].)

The Execution Model

The host program, responsible for the management of the compute devices, provides the necessary *context* for the OpenCL computations. The context consists of the compute devices, memory objects, program objects, and command queues. It determines the environment of the executed program. All data transfers and calculations shall reside within the same context. The code responsible for the computations on the devices is organised in so called *kernels*. These kernels are programmed in the actual OpenCL-language, a derivation from the C99-language. Usually kernels are directly compiled before execution, this guarantees the high portability of OpenCL-programs to different hardware types. To launch the computation on the GPU, the kernels are enqueued in the command queue with a geometrical description of their computing range. This description spans an index space and the instances of the kernel, called *work-items* are executed for each point of the index space. Each work-item is attributed with an identifier, the global ID, which gives the position in the index space.

The geometrical description of the index space is given by the `NDRange`-argument (N-dimensional range). The simplest case is giving solely the number n of kernel instances without any additional structure. In this case, n work-items, instances of the kernel, are executed in parallel on the GPU. The index space may be one, two, or three dimensional. Therefore the dimension best suited to the domain of the physical model can be chosen and the global ID can be used to carry suitable geometrical information.

Additionally to the geometric structure of the index space, the work-items can be subdivided into *work-groups*. Each work-group is bound onto one compute unit. Therefore the execution stream of work-groups is quite uniform. A further subdivision into work-groups is often very beneficial for the performance of the executed program. Constraints are depending on the actual hardware used and its wavefront length. A schematic overview of the `NDRange` structure is shown in figure 1.9.

The Memory Model

The OpenCL memory model defines a hierarchical structure of memory on the compute devices, see figure 1.10. Four different memory regions exhibit different access limitations:

- *Global Memory.* Every work-item has a read/write access to the global memory.
- *Constant Memory.* Every work-item has a read only access to the constant memory.
- *Local Memory.* Only work-items within the same work-group have read/write access to the local memory.
- *Private Memory* Each work-item has exclusive read/write access to its private memory.

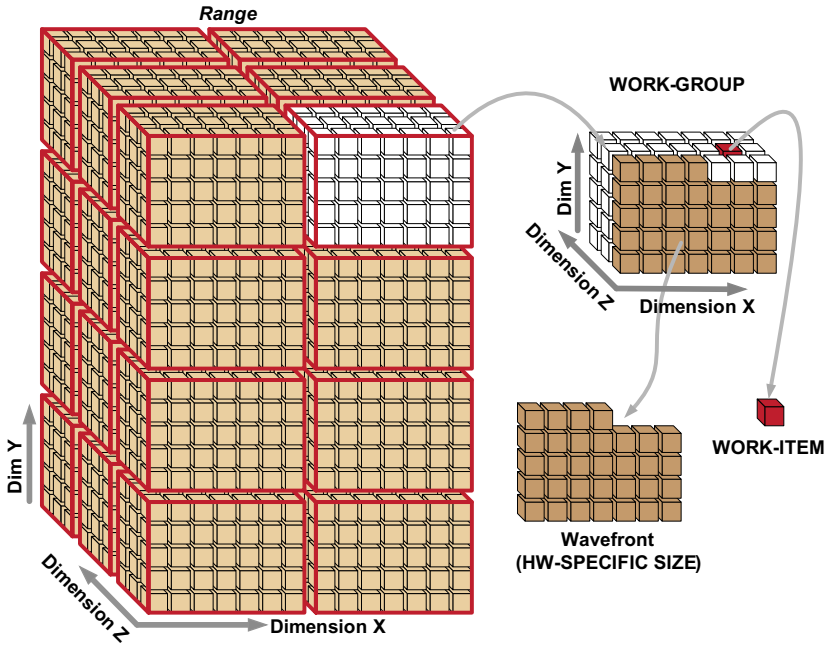


Figure 1.9.: Work-items can be enqueued with a three dimensional organisation by the `NDRange` argument. (Figure taken from [1].)

On graphics cards these memory regions provide significantly different access latencies. The fastest access is possible to the registers of the processing elements. Depending on the needed resources, private memory may be mapped onto these registers. However, if a kernel makes extensive use of private memory, it physically resides within the global memory with its reduced access speed. A small amount of memory is usually present on each compute unit. This amount of memory, called *local data store* (LDS) in AMD's nomenclature, is shared between all work-groups that are executed in parallel on a compute unit. The global memory has the slowest access speed. It can be accessed from all work-items independent of the work-group or compute unit. The constant memory usually resides within the global memory. Special cache structures can increase the access latencies. Often the constant memory has a privileged caching structure.

The memory management must be carried out with the appropriate API calls in the host program. The allocation of global memory, as well as all dynamic allocations of all memory types, must be handled on the host side. Static allocations of local and private memory are managed on the device side. Allocated sections of global memory used for computation are called buffers. Data transfer from host to device and vice versa must be handled by the host program.

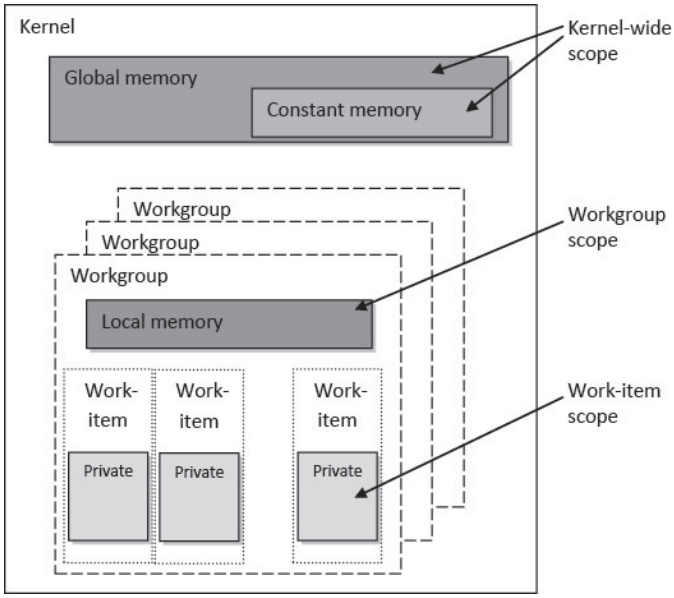


Figure 1.10.: The OpenCL Memory Model. (Figure taken from [23].)

Consistency and Synchronisation

In OpenCL a relaxed consistency scheme is realised, i.e. consistency can be enforced with special `barrier` and synchronisation points. Private memory is always consistent. Local memory read- and write-consistency can be enforced within a work-group with special `barrier` functions. The same holds also for all work-items within the same work-group and their access to the global memory. However, no consistency can be enforced for work-items of different work-groups.

All memory transfers and executions are controlled in command queues, which are assigned to one compute device only. However, different command queues can be assigned to the same compute device. Therefore the execution of independent different kernels or memory copies can be maintained independent. Kernels can be executed in order, i.e. each kernel waits for the completion of the preceding kernel in the queue. Additionally kernels can be executed out-of-order. In this case the execution of the kernel can be controlled by a dependency-list. The execution of each kernel generates an event object, which can be used to control the command queue.

2. Parallel computation of the Casimir Effect

2.1. Experimental Prototyping

The present chapter, and the according algorithms in the appendix, are mostly based on our paper [4], which was elaborated during a research stay at the university of Southampton. Contrary to the article, I will discuss the design criteria and their relevance for other fields of physics more thoroughly here.

During the research stay with an interdisciplinary team of physicists, computer scientists, and mathematicians the goal was to investigate on a novel approach in computational quantum theory, the worldline approach [28, 47, 26]. Not only the predictive power of the model has been of interest, but also its viability for massively parallel calculation of quantum phenomena. Rapid prototyping allowed to implement this approach to quantum field theory for Casimir forces in the three months the team stayed together. Instead of a redesign or incorporation into the UrQMD model, different code bases were created:

1. A Python prototype including all features, but too slow in execution for a fine grained grid.

2. A Scheme implementation strongly oriented to the Python implementation with the full feature spectrum and fast enough to produce numerical results on a workstation.

3. A Python + OpenCL hybrid version limited to selected geometries which is able to fully harvest the parallel computing capabilities of the LOEWE-CSC supercomputer.

Within this project we were able to explore new algorithms and important optimisations for the quantum field theoretical calculations. Experimental prototyping often allows only for a very small subset of desired features. Nevertheless various geometries could be studied with the Scheme implementation and the existence of frameworks like PyOpenCL [39] has allowed us to do massive parallel simulations on GPUs for selected geometries. Thus, numerical issues and artefacts could be estimated with concrete data of simulations and the efficiency of GPUs for this kind of algorithm could be examined.

2.2. Introduction to Path Integrals

Path integrals can be seen as an alternative approach to quantum mechanics and quantum field theory. They are used to calculate the probability amplitude and have a rather geometric viewpoint compared to the more standard Schrödinger approach. I will follow [20] for the heuristic introduction in this section.

In classical mechanics there is typically a unique path a particle describes. This path depends on boundary conditions, e.g. an electrical potential acting on an electron. The path can be found by applying the *principle of least action*. On the unique path, a quantity called action has its minimum. For the free particle this path, the *classical trajectory*, is a straight line. The principle of least action is described by the action integral

$$\mathrm{S} = \int_{\mathrm{t_a}}^{\mathrm{t_b}} \mathcal{L}(\dot{\mathrm{x}}, \mathrm{x}, \mathrm{t})\, \mathrm{dt} \quad . \tag{2.1}$$

Which leads to the classical Lagrangian equation of motion:

$$\frac{\mathrm{d}}{\mathrm{d}t}\left(\frac{\partial \mathcal{L}}{\partial \dot{x}}\right) - \frac{\partial \mathcal{L}}{\partial x} = 0 \quad . \tag{2.2}$$

This equation is of course a merely informal description and has significance only, if x and $\dot{x}$ are given by a curve $\Gamma(t)$ which is parameterised by t.

The path integral approach can be motivated by the double slit experiment for electrons. The typical setup, shown in figure 2.1, consists of an electron emitter, a double slit, and a wall with an electron detector.

The detector is measuring the well known interference pattern at plane B. This pattern can be described by interference calculation considering the *de Broglie* wavelength of electrons. Instead of calculating the interference patterns by the *Huygens-Fresnel* principle, the path integral approach assigns a probability amplitude depending on the action of each path an electron describes. However, as the interference pattern vanishes as soon as it is known whether an electron passes slit 1 or slit 2 the idea is to calculate as if an electron passes both slits.

Thus to calculate the correct probability amplitude, the path integral approach abandons the idea of exact one path per particle. The next step is a mere limit process by putting more and more masks between emitter and detector, and punching more and more wholes into this masks, as shown in figure 2.2. Reconsidering again the double slit experiment, all possible paths leading through the two given slits and the initial mask give a contribution to the final probability amplitude. A selection of possible paths can be modelled by Brownian motion, as shown in figure 2.3.

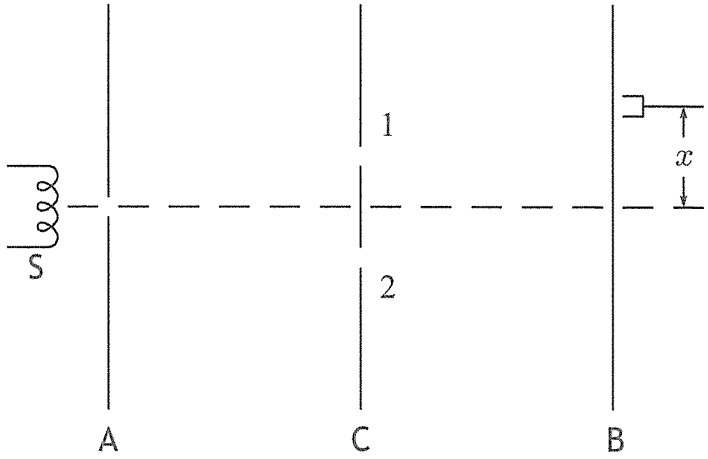


Figure 2.1.: Typical setup of the double slit experiment, consisting of an electron emitting coil S with a mask A to generate an electron ray with the wall C in the middle having two holes. At plane B the detector measures the arriving electrons and their distance to the central axis. (Figure taken from [20].)

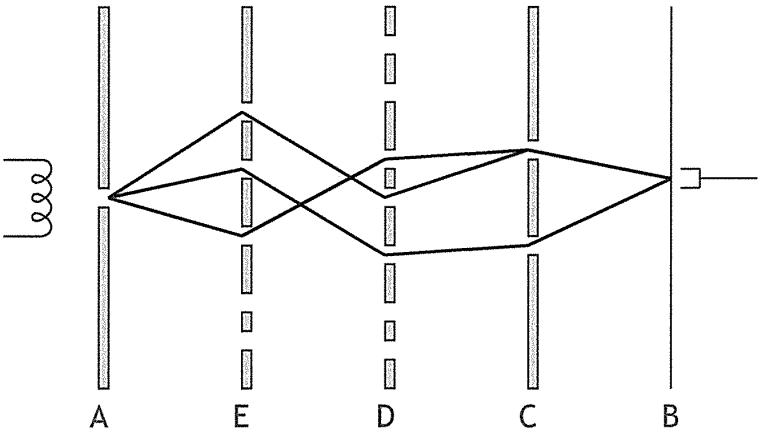


Figure 2.2.: Considering more masks in between emitter and detector and adding more slits into the masks. Again the interference pattern changes if the information about the actual way of the electrons can be gained. Thus all possible combinations have to be considered at once. (Figure taken from [20].)

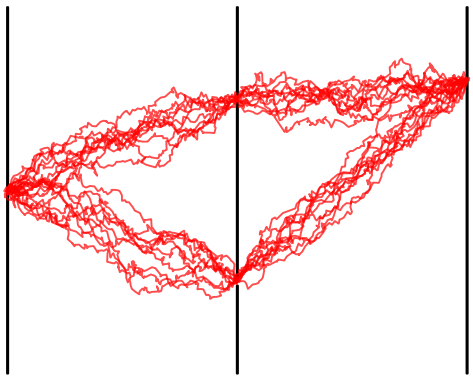


Figure 2.3.: The classical double slit setup with an central double slit mask. The red paths are a selection of possible paths of **one** electron originating in the centre of the left mask and hitting a detector on the right wall, the paths are modelled with a Brownian motion.

In the double slit experiment only paths leading through the initial mask, the double slit, up to their first impact onto the final detector wall are feasible. Following the path integral approach, all these paths contribute to the final calculation of the probability amplitude by the action equation 2.1 which is path dependent. The sum over all paths is referred by the new notation as:

$$\Phi = \int_a^b \mathrm{e}^{\frac{\mathrm{i}}{\hbar}\mathrm{S[a,b]}} \mathcal{D}\mathrm{x(t)} \quad . \tag{2.3}$$

Of course this integral has to converge and be well-defined, which is assured by special kernels suitable to the kind of Lagrangian applied. Additionally in a Monte Carlo simulation the ensemble of exemplary paths has to be *representative* for all possible paths, and again the question for the right measure is important. However for the simulation of the Casimir effect, the worldline approach uses the ensemble of closed loops within e.g. two parallel plates and uses as border conditions pure geometrical intersections with these two plates. Our simulation results mirror e.g. the effects measured between parallel plates.

The resulting attracting force between the plates in figure 2.4 is the integral over the contribution of the action of each loop. The loops are modelled by Brownian motion. As the length of each loop itself is infinite, a suitable measure is the variance of the

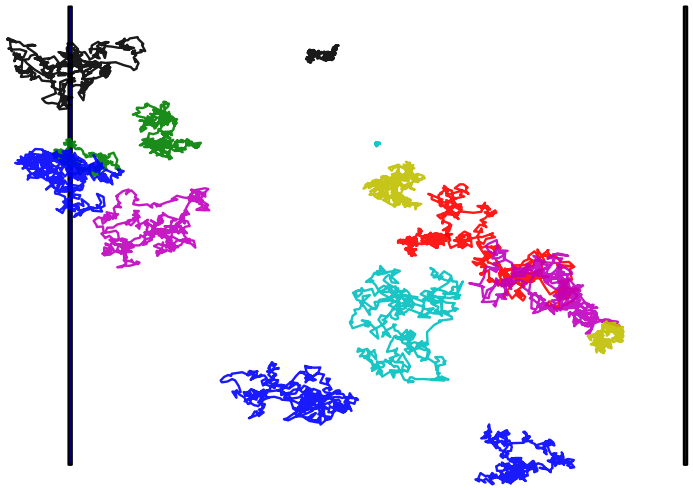


Figure 2.4.: The experimental setup for the parallel plates geometry. Between two parallel perfect conductors (black) loops of different length (coloured) are modelled by Brownian motion.

underlying distribution. The distribution of the possible *lengths* is discussed in 2.3.3 as it is crucial to the accuracy of the integration. The following overview of the physics model and the applied algorithms is taken from our article [4].

2.3. The Numerical Method of Worldline Numerics

Casimir forces are forces arising due to quantum effects in geometries with boundary conditions. They have first been predicted via theoretical considerations by Casimir in 1948 [13], and have since been verified experimentally to 1% accuracy [45]. At present, one major obstacle to research is that Casimir force calculations are often computationally very demanding. Nevertheless, the development of theoretical tools and methods must go hand in hand with progress in nano scale manufacturing, for it is clear that a sound understanding of the role of Casimir forces in nano machines will become increasingly important as we learn to manufacture on shorter length scales.

One approach to the calculation of Casimir forces is based on the *worldline approach* developed by Gies, Klingmüller, Langfeld and Moyaerts [28, 47, 26]. This approach has mostly been used to study simplified field theoretical models with massless scalars instead of vector gauge bosons (photons). Nowadays alternative methods are available to directly calculate electrodynamic effects even with frequency-dependent optical properties of materials [59, 44]. However, the worldline approach is interesting for a number of reasons:

- Due to the probabilistic nature of the method, it is sometimes computationally comparatively cheap (depending on the geometry) to obtain a rough estimate of Casimir forces.

- The calculation can be modified in such a way that it simultaneously gives all the forces on a number of bodies, making it potentially attractive for problems requiring a geometric shape optimisation approach.

- Finally, it is formulated in a way that is suggestive of a remarkably intuitive interpretation.

2.3.1. Monte Carlo Integration over Loops

The Casimir energy for a static geometry that can be modelled by a position-dependent potential $V(x)$ is given as the quantum effective action per unit time:

$$E_{\text{Casimir}} = \frac{\Gamma[V]}{\int_{\tau=\tau_-}^{\tau_+} d\tau} \quad . \tag{2.4}$$

The *worldline numerics* formalism by Gies, Langfeld, and Moyaerts is developed in [29] uses the Lagrangian:

$$\mathcal{L} = \frac{1}{2}\partial_\mu\phi\partial_\mu\phi + \frac{1}{2}m^2\phi^2 + \frac{1}{2}V(x)\phi^2 \quad , \tag{2.5}$$

for a real scalar field ϕ coupled to a potential $V(x)$. The quantum effective action for the potential V is given by:

$$\Gamma[V] = \frac{1}{2}\mathrm{Tr}\ln\frac{-\partial^2 + \mathrm{m}^2 + \mathrm{V(x)}}{-\partial^2 + \mathrm{m}^2} \quad . \tag{2.6}$$

The trace operator is re-expressed as a Feynman path integral. This leads, for a real scalar field of mass m interacting only with the external potential V, to an expression for the effective action that is numerically tractable via Monte Carlo methods. The key expression from [29] is:

$$\Gamma_\Lambda[V] = -\frac{1}{2}\frac{1}{(4\pi)^2}\int_{1/\Lambda^2}^{\infty}\frac{dT}{T^3}e^{-m^2T}\int d^4x\left[\langle W_V[y;x,T]\rangle_y - 1\right] \quad , \tag{2.7}$$

where an UV cut-off regulator Λ has been introduced. Here, the expectation value $\langle\cdot\rangle_y$ is the ensemble average over all closed loop (c.l.) Gaussian random walks $y : [0;1] \mapsto \mathbb{R}, y(0) = y(1)$ of *Wilson loops* rescaled to proper time T. Let the statistical weight of the loop y be

$$p[y] = \exp\left(-\int_{t=0}^{t=1} dt\,\dot{y}(t)^2/4\right) \quad , \tag{2.8}$$

then:

$$\langle W_V[y;x,T]\rangle_y = \frac{\int_{y\ \mathrm{c.l.}}\mathcal{D}y\,W_V[y;x,T]\,p[y]}{\int_{y\ \mathrm{c.l.}}\mathcal{D}y\,p[y]} \quad , \tag{2.9}$$

where W_V depends on the *path* y and on position shift x and proper time T:

$$W_V[y;x,T] = \exp\left(-T\int_{t=0}^{t=1} dt\,V(x+\sqrt{T}y(t))\right) \quad . \tag{2.10}$$

From this expression for the effective action of a free scalar interacting with a potential, Casimir forces can be obtained by using the position dependency of the potential $V(x)$ to model the geometry, and calculating energy changes associated with changes to the geometry.

While the applicability of this model for the calculation of real Casimir forces is questionable (even for perfect conductors) as the physics of photons is quite different from that of a scalar field, the remarkable conceptual simplicity of the above expressions certainly warrants a deeper investigation of its properties and potential utility, for it

might actually allow a (yet undiscovered) generalisation to the photon case. For the electromagnetic case, one would naturally want to start with investigations of perfect conductor surfaces, and the (obvious) scalar pendant of this is a potential $V(x)$ that suppresses all quantum fluctuations *inside* the given bodies. It is not difficult to see that one may alternatively restrict the potential to have non-zero values only close to the surfaces of bodies, taking

$$V(x) = \lambda \int_{\Sigma} d^2\sigma \delta^3(x - x_\sigma) \quad , \tag{2.11}$$

and considering the limit $\lambda \to \infty$. Then, $W_V[y; x, T]$ reduces to:

$$\exp\left[-T \int_0^1 dt V(x + \sqrt{T}y(t))\right] = \begin{cases} 1 & \text{Loop pierces a surface} \\ 0 & \text{Loop does not pierce a surface} \end{cases} \quad . \tag{2.12}$$

Substituting $s = \sqrt{T}$ to eliminate the square root, and using translation invariance of the integral to ensure all loops have centre of gravity at the origin, the expression for the geometry-dependent regularised Casimir energy is:

$$\Gamma_\Lambda = -\frac{1}{(4\pi)^2} \int_{1/\Lambda}^{\infty} \frac{ds}{s^5} \langle \Theta_V(sy, x) \rangle_y \quad , \tag{2.13}$$

where the mean value is over all unit loops, sy is the loop $y(t)$ scaled pointwise around its centre of gravity $\bar{y}$ by the factor s, and

$$\Theta_V(sy, x) = \begin{cases} 0 & \text{Loop does not pierce a surface} \\ 1 & \text{Loop pierces a surface} \end{cases} \quad . \tag{2.14}$$

The problem with this approach is that the Casimir energy attributed to the surface of any single body goes to infinity as we send the energy regulator Λ to infinity, due to the contribution from very short loops close to the surface. (This is, of course, a non-physical artefact related to the *geometry much larger than atomic scales* approximation.) In order to predict Casimir forces between different objects we are only interested in the dependency of the energy on the relative position of these objects. Therefore, it makes sense to modify this scheme in a way that (i) the contribution of each loop is taken into account relative to a configuration in which all objects are at infinite separation from one another, and (ii) the Casimir energy contribution attributed to loops piercing only one object surface (hence, *belonging* to that object) is taken as zero.

Consider n objects with potentials $V_1, \ldots, V_n$. Then the total potential is $V = V_1 + V_2 + \ldots + V_n$ and we use the freedom to shift the absolute energy level to define the

interaction Casimir energy E as in [29] as the energy difference relative to a configuration in which every body is at an effectively infinite distance from every other body:

$$E = \left(\Gamma[V] - \Gamma[V_1] - \Gamma[V_2] - \ldots - \Gamma[V_n]\right)/\Delta\tau \quad . \tag{2.15}$$

Using this, we get

$$E = -\frac{1}{(4\pi)^2}\int\limits_{1/\Lambda}^{\infty}\frac{ds}{s^5}\langle\Theta(sy,x)\rangle_y \quad , \tag{2.16}$$

with Θ given by

$$\Theta(sy,x) = \begin{cases} 0 & \text{Re-scaled loop does not pierce any surface} \\ 1-n & \text{Re-scaled loop pierces the surfaces of } n \geq 1 \text{ objects} \end{cases} \quad . \tag{2.17}$$

If n objects come close to one another, every loop that pierces all of them can be regarded as the image of n loops, each to be considered as being attached to (and moving with) that body. Hence, when objects are in proximity, we count a loop *once* that would have been counted n times instead for separated objects. (Note that the counting weight of both a loop that pierces no surface, and a loop that pierces only one surface, is zero.) If the objects are now spatially separated the integral $\int_{1/\Lambda}^{\infty}\frac{ds}{s^5}\langle\Theta(sy,x)\rangle_y$ is finite and well behaved for $\Lambda \to 0$, so we can safely set $\Lambda = 0$. One is easily convinced that this is indeed the correct expression by considering a simple geometry (such as two parallel flat slabs) and requesting that the Casimir force does not change if one object is instead thought of as being made of two adjacent bodies. The counting weights are dictated by the convention for the *zero energy* configuration.

For each loop y, the weight Θ, as a function of the rescaling factor s, is piece-wise constant. The s-integral hence can easily be performed analytically. Rather than being only a convenient simplification that saves computing time, this property plays a crucial role for the efficient simultaneous computation of multi-body forces.

2.3.2. Loop Generation

When trying to evaluate Equation 2.16, one naturally would try to discretise the loop as consisting of a finite number of straight sections. Taking the procedure literally, the presence of complicated curved geometries would mandate computationally fairly expensive ray-surface intersection checks. In many cases, a better investment of the computational effort may be to instead make the number of discretisation points on the loop sufficiently large to ensure that simple inside/outside checks applied to each point give a reasonably close approximation. Still, generic ray/surface intersection checks can become useful, especially if the complicated multiple integral in Equation 2.16 (over loop shapes, loop sizes, and loop centres of gravity) can partially be evaluated by analytic, or

rather semi-analytic means that involve numerical approximation of integral boundaries. This is relevant for the discussion in section 2.3.4, and may make major computational improvements of the method possible.

In order to generate a properly distributed random sample of loops, we first generalise the problem to finding a process that produces piecewise straight paths with Gaussian length distribution (of given standard deviation) for a given starting and end point (not necessarily coincident). This problem can be rephrased as finding pairs of such paths, each with its own starting point but at first without any constraint on their endpoints, and imposing the condition that they meet at their endpoints. Concatenating the first path to the reverse of the second solves the problem of finding a path with the correct distribution between two given points. One easily sees that the distribution of the midpoint is still Gaussian (being the product of two Gaussian distributions). Hence, we can sample a loop by recursively sampling an intermediate point in the interval between a given start and end point.

This method, known as the *d ('doubling') loop algorithm* [30], manages to generate closed loops with the desired distribution with very little effort.

Rather than choosing starting points randomly in space and then determining the location of the halfway-round-the-loop point, it makes sense to perform stratified spatial sampling on a lattice. To do so, we choose the first point to be the lattice point and take the halfway point to be Gaussian distributed with mean the first point and standard deviation a characteristic length. We then continue sampling a loop of that length and later scale it around the midpoint between the first point and the halfway point appropriately to obtain a unit loop. In that way, the midpoint is Gaussian distributed around the lattice point with a given length scale.

If we want unbiased integration by taking loops for each lattice point, the lattice points have to be representative for loops sampled in their vicinity, with a characteristic length being that of the grid. This is certainly true if the grid is fine compared to any characteristic length of the geometry. However, the same can be achieved for arbitrary grids, if we take the characteristic length in the just described stratified sampling to be that of the grid.

A different kind of lattice effects has to be taken into account for methods computing a force as a difference in energy for two given geometries. Such methods would typically put a fixed set of loops on each lattice point and add up their energy contributions. Then they would do the same for the same geometry with one object moved in a particular direction. The difference in energy is then proportional to the force component on the moved object in the given direction.

To focus on the net effect, as we do with symmetries (see 2.3.5), one typically would use the same set of loops for both geometries. Also, to have for each loop a corresponding shifted loop, the amount the object is moved has to be a multiple of the grid length

(in this direction). While doing otherwise would not necessarily yield a bias, doing so significantly improves the convergence speed of this method.

For our purposes, we typically only ask simple questions about each loop, such as *which objects does it hit?*, or (at most) *for what scaling intervals does this loop, centred at x_{cm} but rescaled in size, hit object O_n?* In order to answer these, only very little information needs to be stored when visiting the loop point by point. So it is possible to implement the relevant algorithms in such a way that the loop is generated on the fly, and we never have to store the entire loop in memory—the number of points we have to remember is about the binary logarithm on the loop length. This yields an algorithm with a very small memory footprint and attractive characteristics for computing architectures that emphasise a high degree of parallelism between very simple cores.

2.3.3. Numerical Integration over the Scaling Factor

One approach to obtain energies—and so ultimately, by comparing energies for different geometric configurations, forces—is to directly evaluate the integral in Equation 2.16 numerically. Naively, one would have to, for various values of s, estimate $\langle\Theta(sy, x)\rangle_y$ and summing up. Since the order of summing up does not matter, we can as well compute the expectation value of the following process:

- Choose s uniformly at random from the interval $[a, b]$.
- Randomly generate a loop sy of size s.
- Count $(1 - n)/s^5$ if the loop hits $n \geq 2$ objects, and 0 otherwise.

Here $[a, b]$ is an interval big enough so that integrating over that interval does not differ noticeably from integrating over all positive reals.

Looking at that random process more closely, one notes that the information about the random loop we use is the number of objects it hits. We have to pay particular attention to short loops that are just long enough to barely touch multiple objects, as they give the largest contribution to the sum. One should note that it is not possible to attribute a useful physical meaning to absolute differences in loop scaling factors s: for a loop that hits (at least) two objects, the effect of changing s to $s + 0.1$ very much depends on what the magnitude of s is. As relative changes of the scaling factor hence are more important than absolute changes, we much prefer a distribution, when sampling loops, that handles all orders of magnitude equally. In other words, we prefer a distribution where the logarithm of s is uniformly distributed on $[\ln(a), \ln(b)]$.

When changing the distribution of s, we also have to transform the weight attributed to each sample accordingly. Taking the logarithm of s to be uniformly distributed, rather than s itself, each value s will be $1/s$ times as likely as before. To still get the same

expectation, we have to multiply each value by s. Hence, we are finally left with estimating the expectation of the following process:

- Chose $\sigma = \ln s$ uniformly at random in the interval $[\ln(a), \ln(b)]$.
- Randomly generate a loop of size e^{σ}.
- Count $(1-n)e^{-4\sigma} = (1-n)s^{-4}$ if loop hits $n \geq 2$ elements, and 0 otherwise.

2.3.4. Symbolic Integration over the Scaling Factor

It makes sense to try to perform at least part of the integration needed to evaluate eq. 2.7 symbolically, for two independent reasons. While this may on the one hand help to simplify the problem, it also gives us a much more useful handle on problems that involve changing geometries. As we are much more interested in Casimir forces (and moments) than just energies, this is obviously desirable.

In particular, we can, as in [27], typically perform the integration over the loop scaling factor $\int_{s=0}^{\infty} \frac{ds}{s^5} \Theta(sy, x)$ symbolically.

If we have sampled a loop y, we can compute for each sampling point the values of s for which this sampling point is inside a given object. Often, this is just an interval, or at worst the union of a few intervals. By merging these intervals for each sampling point, we can compute the set of s values for which the loop hits the given object. Now, as Θ counts the number of objects hit by the loop, it is piecewise constant on the partitioning so obtained; if $\Theta = n$ for $T \in [a, b]$, we have $\int_a^b \frac{ds}{s^5} \Theta(sy, x) = n(a^{-4} - b^{-4})/4$.

Note that this means that we also do not need to specify the region of s which we want to sample, i.e. our method does not need to know a geometric length scale.

2.3.5. Numerical improvements

In a typical geometry, essentially the whole energy or force is contributed by few, comparably small regions. These are typically the regions where two objects come closely together. While we still have to sample loops in such a way that we integrate over all of the relevant region of space, it is worthwhile to focus effort mainly on these highly contributing areas, as the absolute uncertainty of our Monte Carlo estimation is much higher there. We achieve this in the following way: We first specify an absolute accuracy to which we want the density estimated to at every point. When later sampling the density at a given point, we first take a specified minimum of samples. From that we estimate the (unbiased) variance of our sampling at this point. We continue sampling until a pre-defined (95%) confidence interval for the sampling mean is smaller than the prespecified accuracy. This *adaptive sampling* scheme reduces significantly the execution time, while guaranteeing the necessary accuracy.

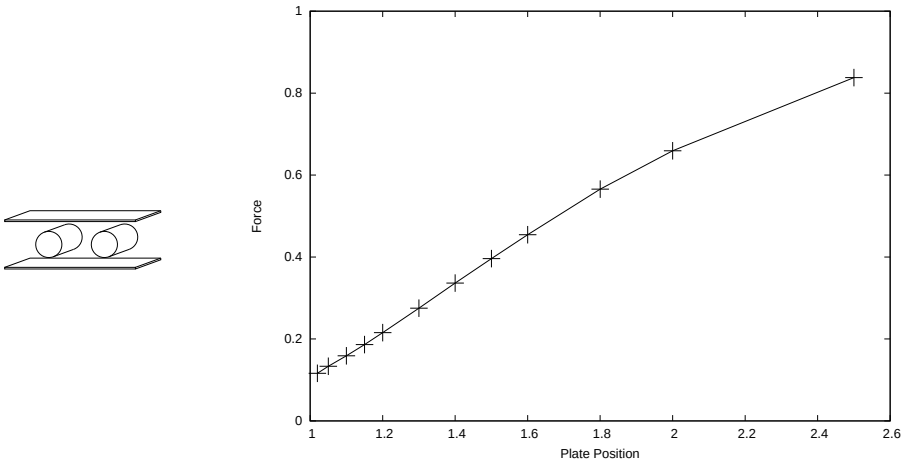


Figure 2.5.: The "cylinders with sidewalls" geometry and the dependency of the attractive scalar Casimir force between the cylinders on the Position of the plates. At position 1.0, the plates would touch the cylinders.

Some geometries, like the "cylinders with sidewalls" geometry studied in [54], show a high degree of symmetry. While perfect symmetry helps to reduce the computational effort as the calculation can be restricted to a fundamental domain, slightly non-symmetric configurations often are a problem if we want to compute the force on an object that gets pulled in different (perhaps opposing) directions: most of the contributions cancel, giving rise to a small residual force. A naive approach would compute the contribution at both sides of the object separately and then add up. This, however, would yield a huge variance for a comparably small resulting value. Fortunately, the force contribution of a loop and its mirrored image are highly correlated in these situations. Often, one is the negative value of the other. So we have a better way of estimating the contribution by estimating the expectation of the following process:

- Randomly pick a loop and also consider its mirror image under the symmetry.
- Add up the force contributions of both these loops.

By this *mirroring method*, we do not change the expectation value of the sum, but, due to the correlation, the variance is much smaller. Hence, it is possible to compute force contributions where a naive approach would require excessive effort due the huge variation, as in the system discussed in the next section.

2.3.6. Parallel Cylinders between Plates, revisited

The "cylinders with sidewalls" geometry studied in [54] (figure 2.5, on the left) has been shown to nicely demonstrate that Casimir forces are essentially multi-body forces. It is given by two parallel, infinitely long cylinders between two parallel infinite plates. Focusing on the attractive force between cylinders, one finds that this depends in a fairly subtle way on the distance between the plates. In figure 2.6 the energy densities of two different configurations are shown. On the left side, the parallel plates are near enough to suppress almost completely any energy contribution. When the distance of the plates is increased, the loops in the centre contribute significantly to the energy.

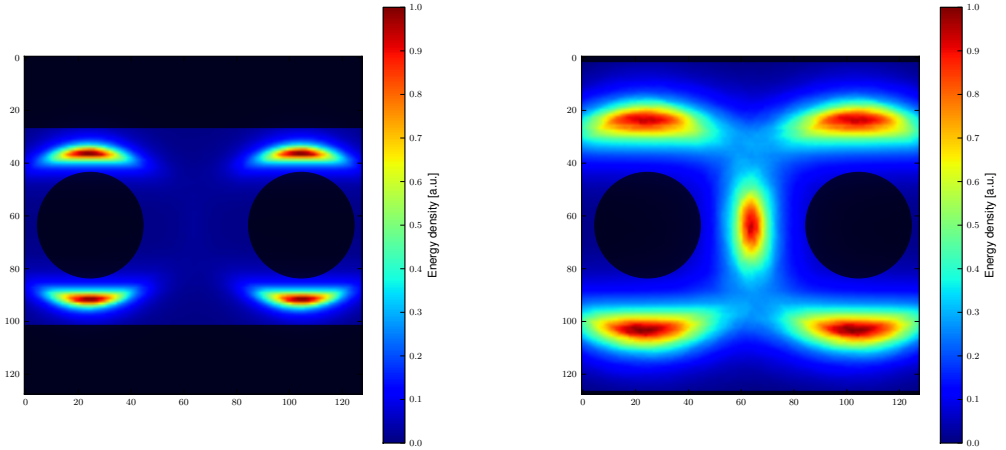
For the calculation whose results are shown in figure 2.5, the cylinders used have radius 1.0 and centres at $(x,\, y)$-coordinates $(-2.0, 0.0)$ and $(2.0, 0.0)$, respectively. The plates are given by the equation $z = p$, where p is varied in the range $1.02 \ldots 2.50$. The calculation of the force on the left cylinder used the scaling method, sampling around a grid with spacing 0.05 and exploiting the mirror method (section 2.3.5) by taking a reflection-symmetric loop ensemble w.r.t. the plane $x = -2.0$ to reduce the variance due to cancellation. This sampling was done in an adaptive manner (section 2.3.5). In total, between $1.9 \cdot 10^7$ and $3.9 \cdot 10^7$ loops of 2^{13} points were sampled for each geometry.

The results are shown at the right-hand side of Figure 2.5. As opposed to methods such as the proximity force approximation, we do see a dependency of the forces on the cylinders on the plate distance. We however could not find the non-monotonic behaviour reported in the literature [54] for the photon case. So, once again, we have an example where scalars behave in a qualitatively different way than photons [33].

2.3.7. Significance of Worldline Numerics for Engineering

From a microsystems engineering perspective, the worldline numerics have a number of attractive properties, such as the ability to quickly give crude estimates, considerable potential to solve geometric optimisation related problems, and of course its conceptual simplicity and intuitiveness that make it a useful educational tool with the potential to give a simple yet quantitatively correct mental model of the origin of Casimir forces. Quite remarkably, the operational procedure can be explained using very simple concepts only—in fact, even without having to use much linear algebra.

At present, the biggest obstacle to its utilisation for engineering applications is the method's inability to handle photons in the presence of conducting boundaries instead of scalar particles in the presence of Dirichlet boundary conditions. As we have demonstrated in section 2.3.6 through an example calculation, the problem is that any attempt to use scalars in order to approximate photon Casimir forces is questionable as this can easily give predictions that are wrong already at the qualitative level.



(a) A compact plates and cylinders configuration. (b) A loose plates and cylinders configuration.

Figure 2.6.: The Casimir energy density in the plates and cylinders geometry in arbitrary units. The region in the centre shows no energy contribution in the dense configuration on the left side (a). The loose configuration (b) on the right side shows a significant energy contribution of the loops which are situated in the centre.

2.4. Implementation

2.4.1. Design

The design decided upon in the Python prototype was later used in the Scheme version. As the Scheme version and the Python prototype aimed on a multitude of geometries, the basic building blocks (solid bodies) were designed as shown in figure 2.7. The OpenCL version, on the other side, applies a more specialised design concentrating on fewer geometries and massive parallel computation. Therefore the more general approach using the `oset`-calculations has been substituted by the direct Monte Carlo integration shown in section 2.3.3. The construction of the loops is encapsulated in the generator class `HostGen`, which provides the necessary format of loops, as well as the possibility to save and restore the state of the *Pseudo Random Number Generator* (PRNG). For each specialised geometry an extra module and kernel is devised. This is due to the need of different buffers holding the spacial information of the examined geometry.

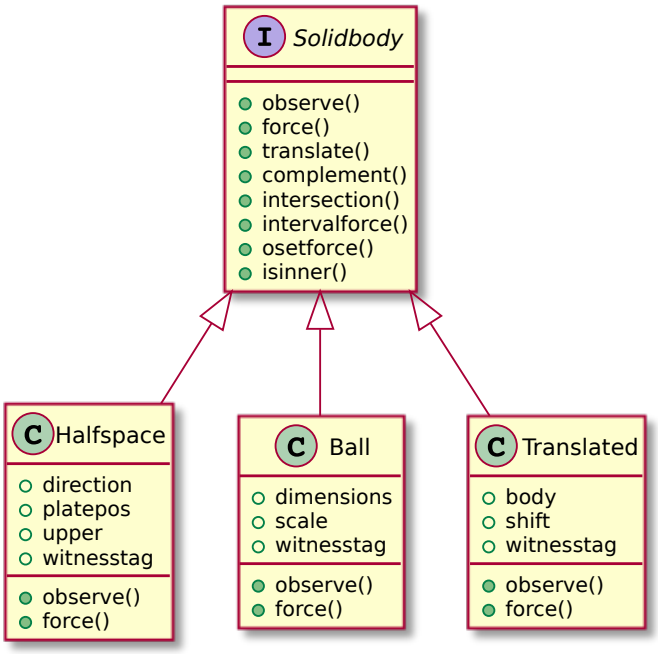


Figure 2.7.: The geometry in the Python prototype was realised by n-dimensional half-spaces and n-dimensional spheres (balls). With the `Translated`-class different approaches calculating force and energy could be studied. The classes `Translated`, `Ball`, and `Halfspace` implement the interface provided by `Solidbody`.

2.4.2. Many-Core implementation

In the worldline formalism the calculation of contributions to the total energy (or force) can be performed independently for each grid point. Also, for each grid point, the contribution of each loop to the energy (or force) does not depend on the contribution of the other loops. Thus the problem can be calculated in a massively parallel approach. Furthermore the basic component—processing a loop – does not require overly complex calculations (in the sense of memory requirements and deep branching). As the worldline method is a probabilistic approach, the accuracy of the calculation can be increased increasing the number of grid points, number of loops, and the number of points per loop in an appropriate way.

As the computation for each point and loop follows the same algorithm, this approach fits the *single program, multiple data* (SPMD) processing approach very well and under the prospect of using the LOEWE-CSC an implementation in OpenCL was a natural choice. This was specially interesting, as the PyOpenCL framework allowed us to combine the rapid-prototyping facilities of Python with the needed OpenCL in order to address the

GPUs.

On account of local memory size limitations, we have developed a version of the *d-loop algorithm* that generates, for each loop, the loop points on the fly—without ever storing the entire loop in memory. As OpenCL in its current specification [37] does not support recursion directly, it is advantageous to instead manage a stack of arcs yet to be split in half, as sketched in listing 1. This manages to reduce the memory footprint of loop

```
while (stacksize > 0) {
  pop(StartPos, EndPos, level);
  if (level > 0){
    MidPos = (StartPos + EndPos) / 2
    + gauss_normal(0, sigma(level));
    push(MidPos, EndPos, level-1);
    push(StartPos, MidPos, level-1);
  }
  else calc_contribution(StartPos);
}
```

Listing 1: Schematic structure of the code managing a stack of yet-to-be-split arcs.

generation and processing from $\mathcal{O}(N)$ to $\mathcal{O}(\log(N))$, N being the number of loop points. Therefore it is possible in principle to shift loop generation to GPU cores even for loops too large to fit into the GPU memory. Regrettably this comes at the cost of intensive branching and it depends highly on the used hardware, whether this version of the *d-loop algorithm* is efficient or not.

The generation of loops on the GPU depends additionally on a reliable fast PRNG. Here the usage of CLRANLUX, as proposed for Lattice-QCD [7], might be a possibility. In our case, however, we found it more appropriate generate loop shapes on the CPU with the Python provided Mersenne twister and subsequently upload them GPU global memory, thereby relying on a thoroughly tested PRNG.

The domain decomposition can follow directly the investigated geometry, as the energy (and force) contribution of each grid point is independent of the surrounding grid points. As the resulting energy (and force) is assigned only to the grid points, it is possible to calculate tiles of the geometry separately and compose the resulting forces in a post-processing step.

An additional level of parallelism can be achieved by running the Monte Carlo integration with different seeds on different GPUs. This comes only with a small risk of biased statistics, because of the period size of the built-in Mersenne twister ($2^{19937} - 1$) and the comparatively small number of generated loop-points ($\leq 2^{40}$).

After the initial setup of the GPU, including the buffers needed for loops and energies, the host-code subsequently produces new sets of loops and uploads them onto the GPU global memory, see figure 2.8. This is done during the calculation of the (partial) energy contribution of the current loop-set to the total energy, hence an additional level of parallelism can be used.

The kernel, shown in listing 15 found in the appendix, loops over all loops of a current loop-set. Each work-item changes then the underlying geometry (i.e. the plates are translated instead of all the loops) and performs the intersection checks and calculates the corresponding partial energy difference under an additional shift in the direction of the examined force. Thereby the contribution of each loop to the forces acting on the current pixel is computed. Using the same set of loops at all grid points also helps in terms of statistics as there then can be direct cancellation between opposing forces arising from similar geometric structures. (See also the discussion in section 2.3.5.)

We have used GPUs mainly with the direct numerical Monte Carlo integration method described in section 2.3.3. The symbolic integration with scaling intervals has been applied to the plate-plate geometry on the GPU and showed a better convergence than the direct numerical integration. However, the direct integration is more suited for parallelisation and can be applied to several geometries. The direct integration method requires a fixed number of sampling points, depending only on the desired accuracy. The symbolic integration, though, requires a different number of scaling intervals for different geometries.

The worldline formalism, used here to study a simplified field theoretical model, does not yet provide all the quantum mechanical effects (see section 2.3.7), desired for more complex simulations. However, as our many-core implementation proves, the worldline formalism is suitable to a massively parallel computation on modern hardware. Hence more complicated versions of the worldline formalism pose promising candidates for quantum mechanical calculations on large scales.

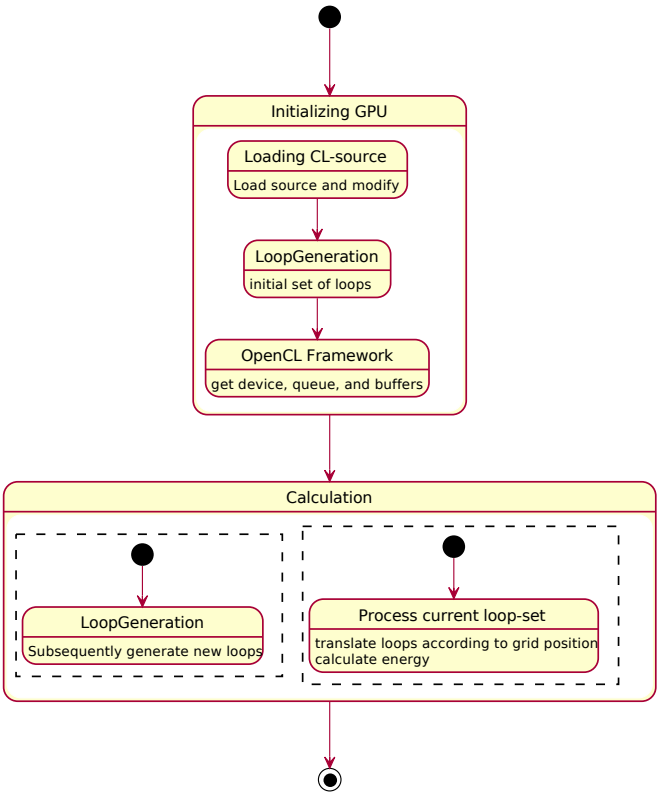


Figure 2.8.: The host program initialises the GPU, modifies by metaprogramming the given OpenCL source according to user specified parameters and generates an initial set of loops. Thereafter subsequently new sets of loops are generated and loaded onto the GPU during the computation of the partial energy contribution of the preceding loop-set.

3. Stability Analysis

Parts of this chapter have been previously published in [24]. In addition to the publication's results, regarding the implications for UrQMD, the computational approach is examined in greater detail here.

3.1. Model Analysis by Metaprogramming

Apart from the experimental difficulties, a major obstacle to pin down the properties of strongly interacting matter is the unambiguous interpretation of the experimental results. Unfortunately, first principle lattice QCD calculations are currently only feasible in thermal equilibrium and for very moderate μ_B/T values (T being the temperature and μ_B being the baryo-chemical potential). Therefore, transport approaches like UrQMD are employed to link the final state observables to the physics properties of the hot and dense stage of the reaction.

All transport models have in common that they rely as input on measured quantities like the hadron masses, the hadron decay widths, individual branching ratios and cross sections. Unfortunately, these quantities are very often not exactly known, as one can see from an inspection of the Particle Data Group (PDG) [48] tables.

In this chapter the dependence of UrQMD to the variation of some of these parameters is explored systematically. In the long run, these investigations will allow to obtain the systematic error of the simulations, which is needed to quantify the quality of the model results. Although applied on the UrQMD-model, the method is also transferable to other transport models based on similar physics assumptions.

As shown in chapter 1 the UrQMD-model contains various interacting subsystems. Hence performing parameter scans on single routines gives only small insights on the overall impact of varied parameters. Because of the intrinsic dependencies of the routines, responsible for distinct physical behaviours, the model has to be examined as a whole.

In a classically grown software package, like UrQMD, it is quite cumbersome to introduce the necessary variations with new routines or parameters. It does not suffice to change the main calling routine, or the configuration parameters; instead one would have to rewrite parts of the whole simulation framework. This lack of flexibility led to one of the starting ideas to *object oriented design* (OOD).

Although a complete rewrite and redesign of UrQMD is planned[1], it is possible to conduct the necessary studies with the current code base. The necessary degree of freedom can be achieved by considering the source code itself as a variable. Variations of the source code, controlled by a metaprogram, can then be compiled and tested. This approach by metaprogramming is usable to a certain extend with many legacy systems.

The usage of a metaprogram offers the possibility to examine *systematically* slight modifications of programs, e.g. in their constant settings. Thus, not only the stability of simulation frameworks can be tested, but also variations within physical theories can be examined. Therefore existing (legacy) simulation models can be reused to check their consistency with varied physics parameters, even if this possibility is not foreseen in the code base.

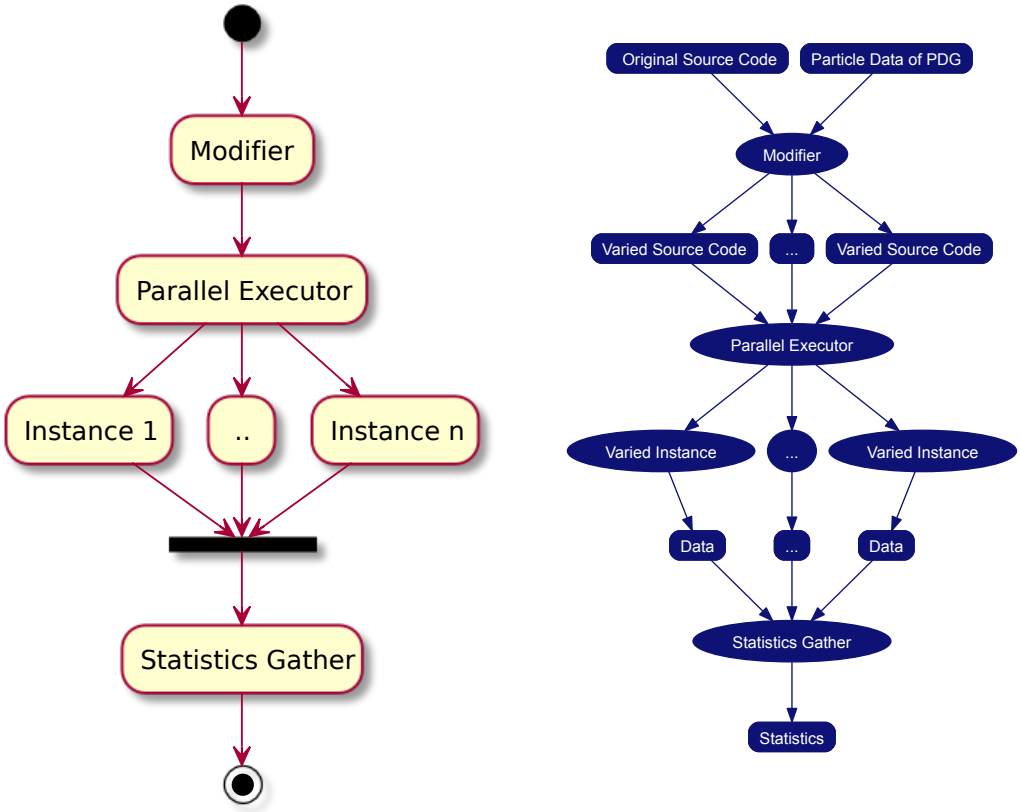
The combination of a metaprogram in Python with the target program UrQMD in FORTRAN benefits of the rich possibilities in text manipulation of a scripting language (Python), with the high execution speed in numerical calculations of a compiled language (FORTRAN). In addition, using this approach, the *varied constants* are variables only to the metaprogram and hence remain constant to the FORTRAN compiler, allowing it more aggressive optimisations.

3.2. Computational Setup

The main modifications within UrQMD is the variation of the hard-coded hadron masses and widths with automatically generated tables with varied parameters. Then a sufficient number of simulations is performed and evaluated. The activity diagram and data flow is shown in figure 3.1. To this aim, the analysis framework parses the up-to-date data from PDG web page [48] and generates intervals according to the PDG-data and user parameters. Thereafter the analysis framework automatically rewrites the UrQMD-source code according to these intervals. On the LOEWE-CSC the different variations of UrQMD are compiled and employed to carry out a systematic parameter scan and to check the stability of the model. After the computation the data files from different UrQMD runs are parsed and compressed to statistics files, which can easily be interpreted on local desktop systems.

In the following, the dependence of the UrQMD results on the particle data, within the estimated errors provided by the PDG is shown. Where possible the variation range is extended up to ±10% of the PDG-mass and width values. Separate scans for variations of mass and width of each baryon family are performed. Typically 10 000 events are simulated to stay clear of statistical errors.

[1] With the redesign of SHASTA as a first step.



(a) Activity Diagram (b) Data Flow

Figure 3.1.: The analysis framework compiles the varied sources of UrQMD and instantiates them in parallel on the LOEWE-CSC. After completion the simulations' results are gathered and put into compressed statistic files.

3.3. The Stability of the UrQMD-model

3.3.1. Cross Sections

Let us start by investigating the total pion-nucleon cross section as a function of energy. This cross section is of special importance for the dynamics of nuclear matter at intermediate energies. It also serves as direct benchmark to adjust the parameter sets since it is well measured experimentally. The total cross section is given by

$$\sigma_{N\pi}^{\mathrm{tot}} = \sum_{R=\Delta,N^*} \langle j_N, m_N, j_\pi, m_\pi \| J_R, M_R \rangle \times \frac{2S_R+1}{(2S_N+1)(2S_\pi+1)} \frac{\pi}{p_{\mathrm{CMS}}^2} \frac{\Gamma_{R\to N\pi}\Gamma_{\mathrm{tot}}}{(M_R-\sqrt{s})^2+\frac{\Gamma_{\mathrm{tot}}^2}{4}} \quad , \quad (3.1)$$

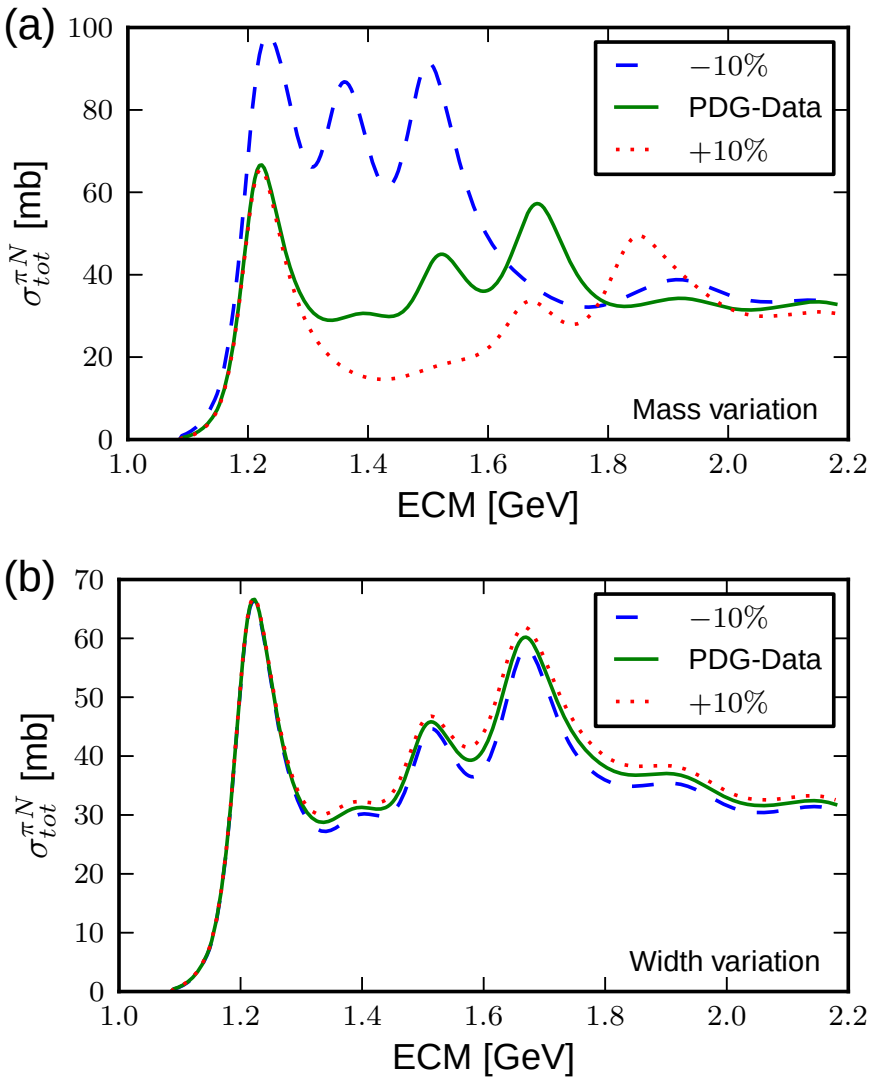


Figure 3.2.: (a): Total pion-nucleon cross section as a function of $\sqrt{s}$ for a systematic variation of the nucleon resonance masses. The full line depicts the PDG averages, the dotted line shows a variation of the PDG parameters by +10%, the dashed line a variation by −10%. (b): Total pion-nucleon cross section as a function of $\sqrt{s}$ for a systematic variation of the nucleon-resonance widths. The full line depicts the PDG averages, the dotted line shows a variation of the PDG parameters by +10%, the dashed line a variation by −10%.

with the total and partial decay widths Γ_{tot} and $\Gamma_{R\to N\pi}$, if the mass dependence of the widths is neglected[2]. Thus, the cross section depends on the widths and masses of all nucleon- and Delta-resonances N^* and $\Delta^{(*)}$. Figure 3.2 (a) depicts the total pion-nucleon cross section $\sigma_{\text{tot}}^{\pi\text{N}}$ as a function of the centre of mass energy $\sqrt{s}$ for a systematic variation of the nucleon-resonance masses. The full line depicts the PDG averages, the dotted line shows a variation of the PDG parameters by +10%, the dashed line a variation by −10%. One clearly observes that the πN cross section varies strongly if the resonance masses are changed. In turn, however, this allows to pin down the resonance masses rather precisely. In contrast a change of the resonance widths leaves the cross section unaltered. Figure 3.2 (b) shows the total pion-nucleon cross section as a function of $\sqrt{s}$ for a systematic variation of the nucleon-resonance widths. The full line depicts the PDG averages, the dotted line shows a variation of the PDG parameters by +10%, the dashed line a variation by −10%.

Figure 3.3 (a) depicts the total pion-nucleon cross section as a function of $\sqrt{s}$ for a systematic variation of the Δ masses. The full line depicts the PDG averages, the dotted line shows a variation of the PDG parameters by +10%, the dashed line a variation by −10%. Figure 3.3 (b) shows the total pion-nucleon cross section as a function of $\sqrt{s}$ for a systematic variation of the Δ widths. The full line depicts the PDG averages, the dotted line shows a variation of the PDG parameters by +10%, the dashed line a variation by −10%. One clearly observes that a variation of the nucleon-masses has a drastic effect on the pion-nucleon cross sections especially in the $\Delta(1232)$ region. In comparison to the available experimental data, strong constraints on the model parameters can be obtained. In fact, the employed parameters are based on the PDG data and re-adjusted within the limits of the PDG ranges.

3.3.2. Pion Production

3.3.3. Pion Yields

Let us next turn to the investigation of full Pb+Pb collisions and focus on the FAIR energy range of 2A GeV and 30A GeV. Here we investigate the total pion yield for a systematic variation of all nucleon-resonance masses m_{N^*} by up to 10%. We show the deviation of the pion yield compared to a UrQMD calculation with the mean values of the PDG data files. Figure 3.4 (a) shows the relative pion yield in Pb+Pb collisions at 2A GeV beam energy for a systematic variation of the masses and widths of the nucleon-resonances by ±10%. Figure 3.4 (b) shows the pion yield in Pb+Pb collisions at 30A GeV beam energy for a systematic variation of the masses and widths of the nucleon-resonances by ±10%.

[2]The full UrQMD simulation includes the mass dependence of the widths.

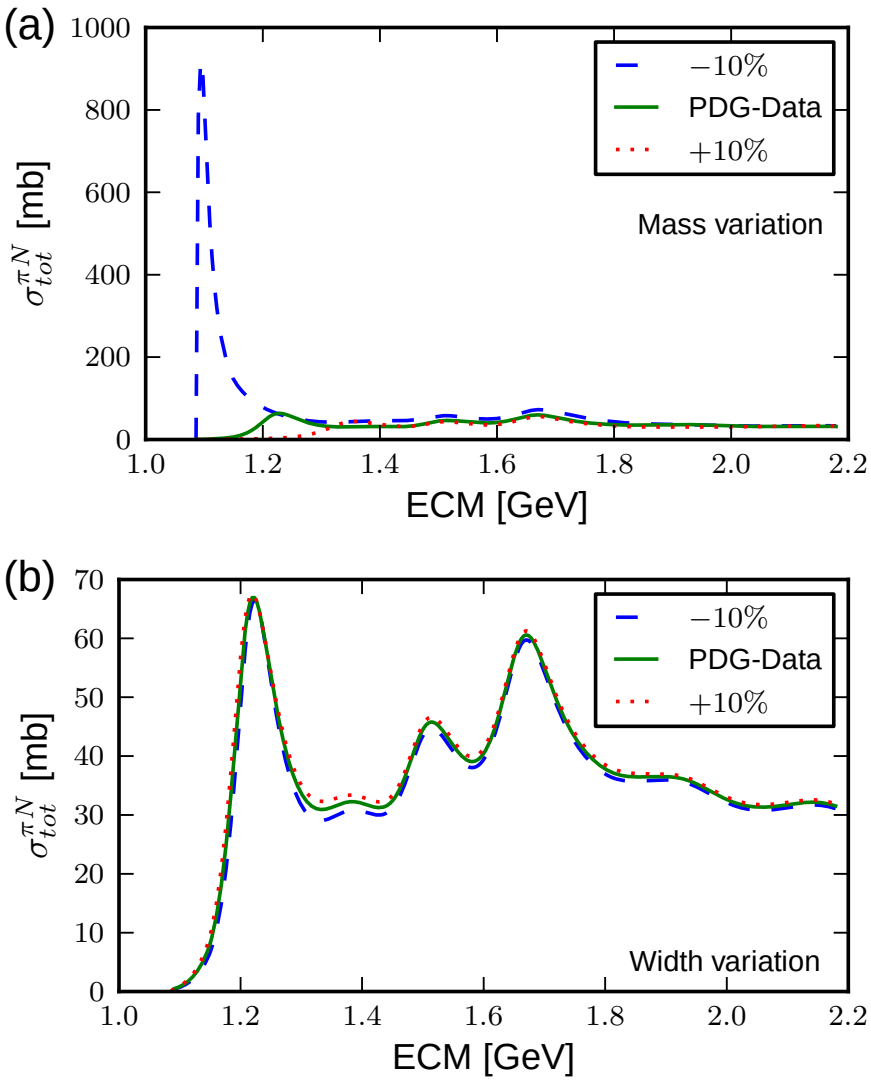


Figure 3.3.: (a): Total pion-nucleon cross section as a function of $\sqrt{s}$ for a systematic variation of the Δ masses. The full line depicts the PDG averages, the dotted line shows a variation of the PDG parameters by $+10\%$, the dashed line a variation by -10%. (b): Total pion-nucleon cross section as a function of $\sqrt{s}$ for a systematic variation of the Δ widths. The full line depicts the PDG averages, the dotted line shows a variation of the PDG parameters by $+10\%$, the dashed line a variation by -10%.

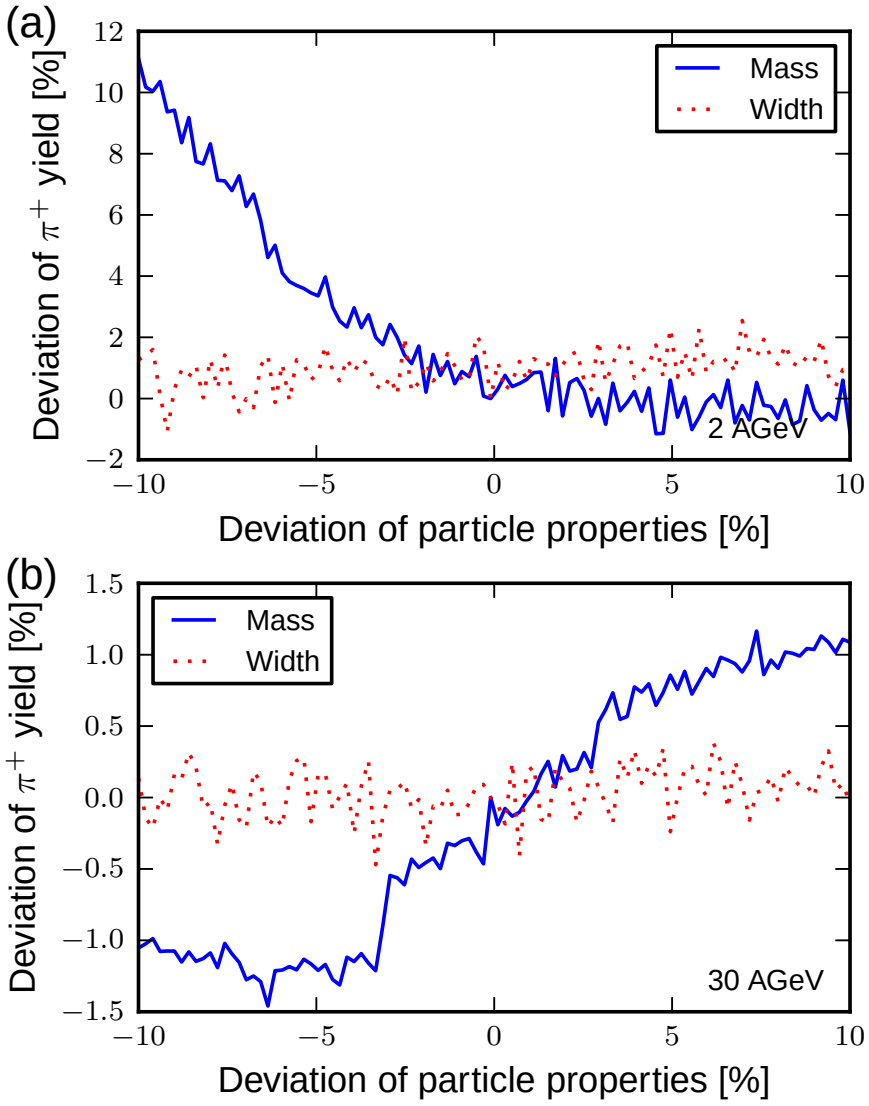


Figure 3.4.: (a): Relative pion yield in Pb+Pb collisions at 2A GeV beam energy. For a systematic variation of the masses and widths of the nucleon-resonances by ±10%. (b): Relative pion yield in Pb+Pb collisions at 30A GeV beam energy. For a systematic variation of the masses and widths of the nucleon-resonances by ±10%.

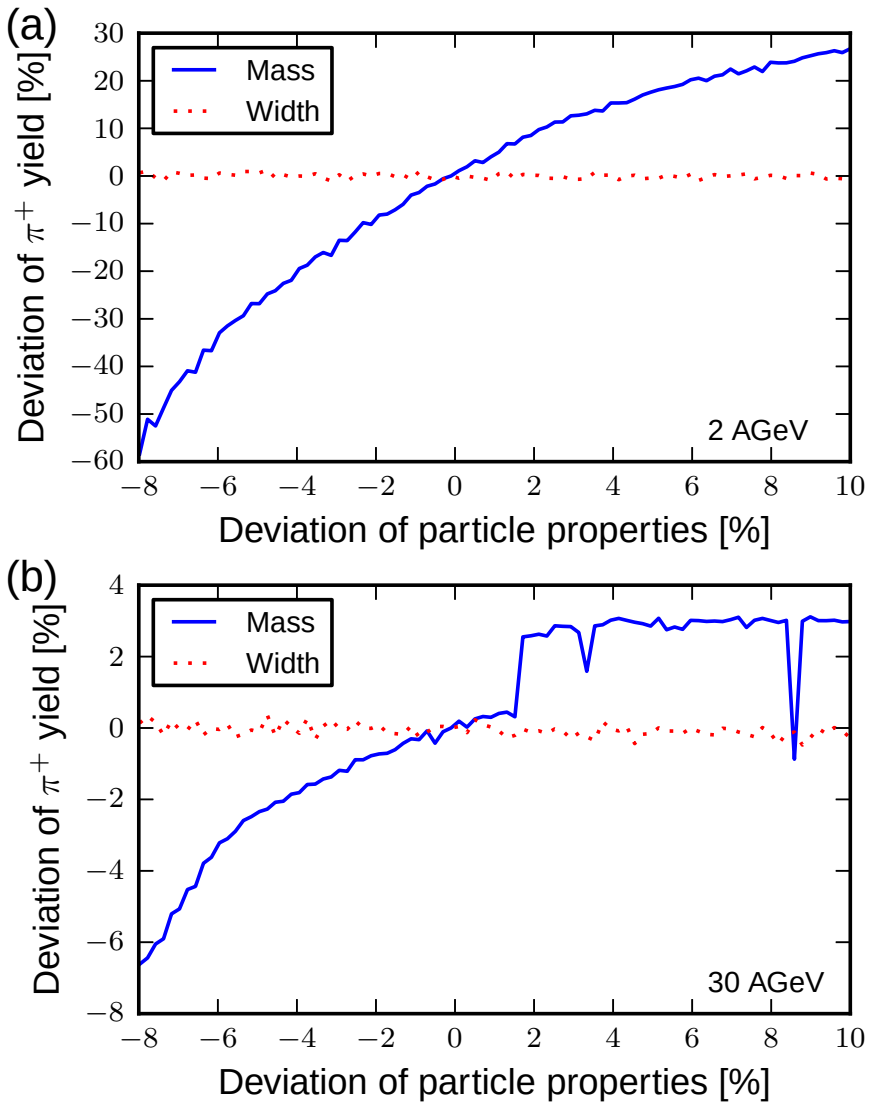


Figure 3.5.: (a): Relative pion yield in Pb+Pb collisions at 2A GeV beam energy. For a systematic variation of the masses and widths of the Δ resonances by $\pm 10\%$. (b): Relative pion yield in Pb+Pb collisions at 30A GeV beam energy. For a systematic variation of the masses and widths of the Δ resonances by $\pm 10\%$.

Even at the lowest energy, which is strongly dominated by resonance dynamics, the model results do at worst vary linearly with the variation of the model parameters. At 30A GeV, the model results are stable against a variation of the resonance parameters. The variation of the particle widths has no significant effect on the model results.

Figure 3.5 investigates the pion production as a function of varying masses of the Delta-resonances m_Δ and their widths Γ_Δ. The masses of all Delta-resonances have been scaled with the same factor. Here we limit the variation to -8% – $+10\%$, because the code becomes unstable for too low masses. Again a variation of the width leaves the results unchanged. The variation of the $\Delta^{(*)}$ masses, however results in a strong variation of the pion yield at 2A GeV. This effect is mainly attributed to the $\Delta(1232)$ resonance that is pushed toward the kinematic limit ($m_{\Delta(1232)} \to m_p + m_\pi$). At 30A GeV, the variance of the yield stays generally moderate. However, the pion yield shows a pronounced step if the masses are shifted by $\sim 1.8\%$ upward. While the magnitude of the effect is small it indicates that complex simulation models may exhibit discontinuous behaviours. The dip at $\sim 8.3\%$ however is a purely statistical effect, which can be seen in the production processes. Let us now have a closer look into the origin of the step.

For further analyses, we group different production processes into five classes, discriminating the decay of Delta-resonances (Δ), the decay of nucleon-resonances (N^*), the decay of strange baryons (accounting for all unstable baryons not included in the former two classes) (B_s), the decay of meson resonances (m) and scatterings ($XY \to \pi + R$). In Figure 3.6 (a) one observes that at $E_{\text{lab}} = 2\text{A GeV}$, the number of pions from scatterings stays constant as a function of Delta-mass, while the production of pions from Delta decays rises linearly. At very low Delta-masses (less than -6% of the standard value), the formation of Deltas absorbs on average more pions than are being produced by the decays thereof, while at higher Delta-masses, the opposite is true. The increase of pion production from Deltas is counteracted by a decrease of pion production from nucleon-resonances N^*, which start to be net-absorbing above $+6\%$ of the standard (Delta-)masses. This can be explained in a picture of detailed balance: When the Delta-resonances produce more pions, the equilibrium value of pion- and N^*-multiplicity is shifted toward the N^*. Thus, the N^*-phase space is populated more quickly than it is depleted. The same effect, though much weaker and not turning around completely, can be seen in the decrease of the number of pions from mesonic decays. In total, the rise of pions from Deltas counteracts the fall of pions from N^*, thus leading to a weak overall rise of the pion production.

At 30A GeV we focus now on the step like behaviours at a Δ-mass shift of $\sim +1.8\%$. Figure 3.6 (b) shows the contributions of different Δ-resonances to the final number of pions for varying Delta masses between 0 and $+10\%$ at high impact energy $E_{\text{lab}} = 30\text{A GeV}$. In an analysis similar to the one from Figure 3.6 (a), we trace the step to the Delta contribution. The step we discovered earlier consists of an increased pion production (less absorption) from the Δ_{1950}-resonance, which rises from -20 to 0, and a corresponding

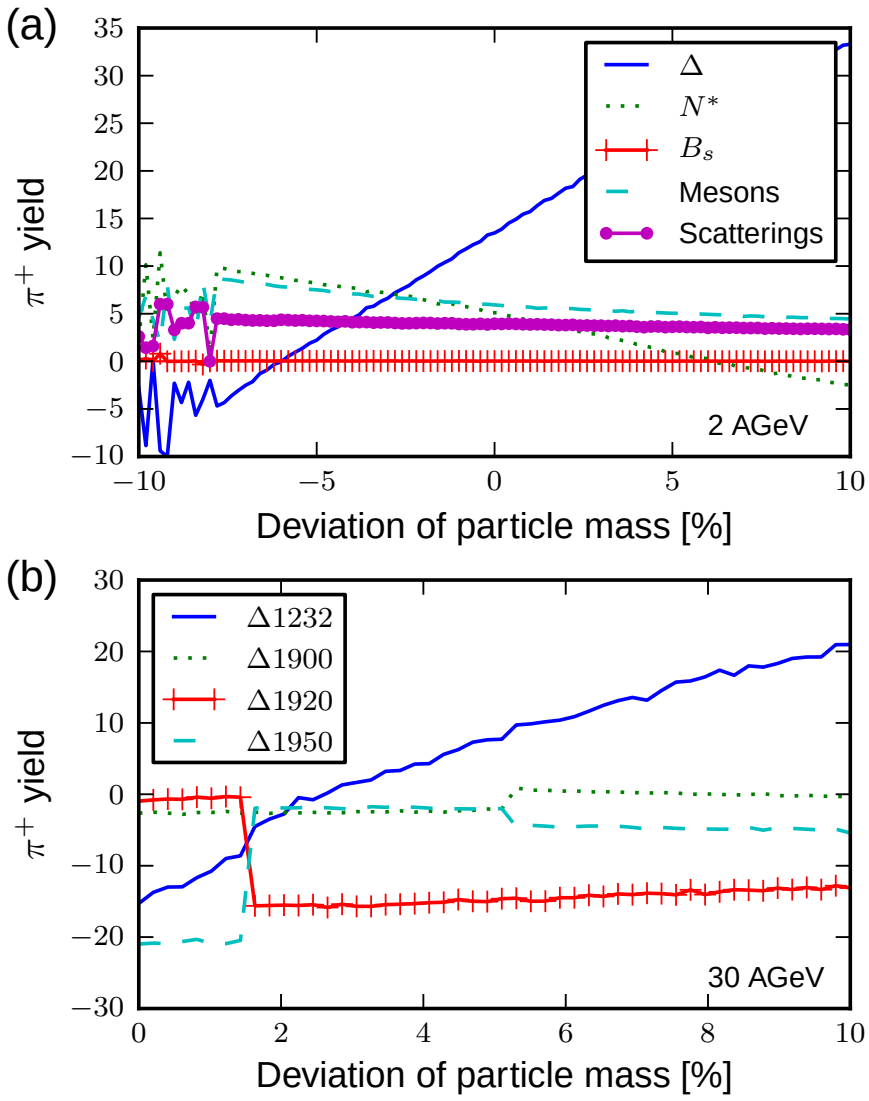


Figure 3.6.: (a): Pion yield in Pb+Pb collisions at 2A GeV beam energy itemising different production processes. (b): Pion production from various Δ resonances in Pb+Pb collisions at 30A GeV.

decreased production (increased net absorption) from the Δ_{1920}-resonance, which drops from 0 to -16. The $\sqrt{s}$-distribution of the underlying πN collision remains essentially unchanged as a function of m_Δ. Therefore, at the point of the discontinuities, the Δ_{1920} takes the role that Δ_{1950} had at lower masses. Since the branching ratios $\Delta_{1920} \to \pi\Delta_{1232}$ and $\Delta_{1950} \to \pi\Delta_{1232}$ differ by a factor of 2 (40% vs. 20%), the number of Δ_{1232} and thus the number of pions are changed over a small mass interval. We find a similar behaviours at $\sim +5\%$, where the pion production from Δ_{1950} decreases, while the production from Δ_{1900} increases. Superimposed is an approximately linear rise of pion production from the lowest Delta-resonance Δ_{1232}.

3.3.4. p_T-spectra of π^+

Finally, we discuss the transverse momentum distributions[3]. Here we investigate the pions transverse momentum distributions in Pb+Pb collisions at 2A and 30A GeV. Figure 3.7 (a) shows the deviation of the pion transverse momentum spectra (p_t) in Pb+Pb collisions at 2A GeV for variations of the nucleon-resonance masses.

At 2A GeV, a decrease of the nucleon-resonance masses shifts the pions to lower p_t, while an increase of the masses shifts it to higher transverse momenta. However, variations of the yields at higher p_t maybe up to $\pm 20\%$ for a variation of $\pm 5\%$.

Figure 3.7 (b) displays the deviation of the pion transverse momentum spectra (p_t) in Pb+Pb collisions at 30A GeV for variations of the nucleon-resonance masses. Figure 3.8 (a) shows the deviation of the pion transverse momentum spectra (p_t) in Pb+Pb collisions at 2A GeV for variations of the Δ resonance masses. Again, we do not observe and effect on the calculations at $E_{\mathrm{lab}} = 30$A GeV is within the statistical fluctuations. Figure 3.8 (b) displays the deviation of the pion transverse momentum spectra (p_t) in Pb+Pb collisions at 30A GeV for variations of the Δ resonance masses.

The effect of variations in Delta-resonance masses is strongly nonlinear. Both at high beam energies and at low beam energies, we can distinguish three transverse momentum regions. Pions from intermediate transverse momentum $0.2 < p_t < 0.6$ GeV are being shifted to low transverse momentum $p_t < 0.2$ GeV, if the masses are decreased and vice versa, if the masses are increased. This is expected, since the available kinetic energy in a Delta decay decreases with decreasing Delta-mass. At higher transverse momenta, lower masses lead to higher pion yields, while higher masses lead to lower pion yields, which is a reversal from the behaviours observed from varying the nucleon-resonance masses. Furthermore, we observe the effects to be a lot stronger in low-energy collisions. Also the variation of the Δ masses by $\pm 5\%$ results in modifications of the pion yield by $\pm 20\%$ in given p_t regions.

[3]We also analysed the rapidity spectrum distributions, but found no significant deviation.

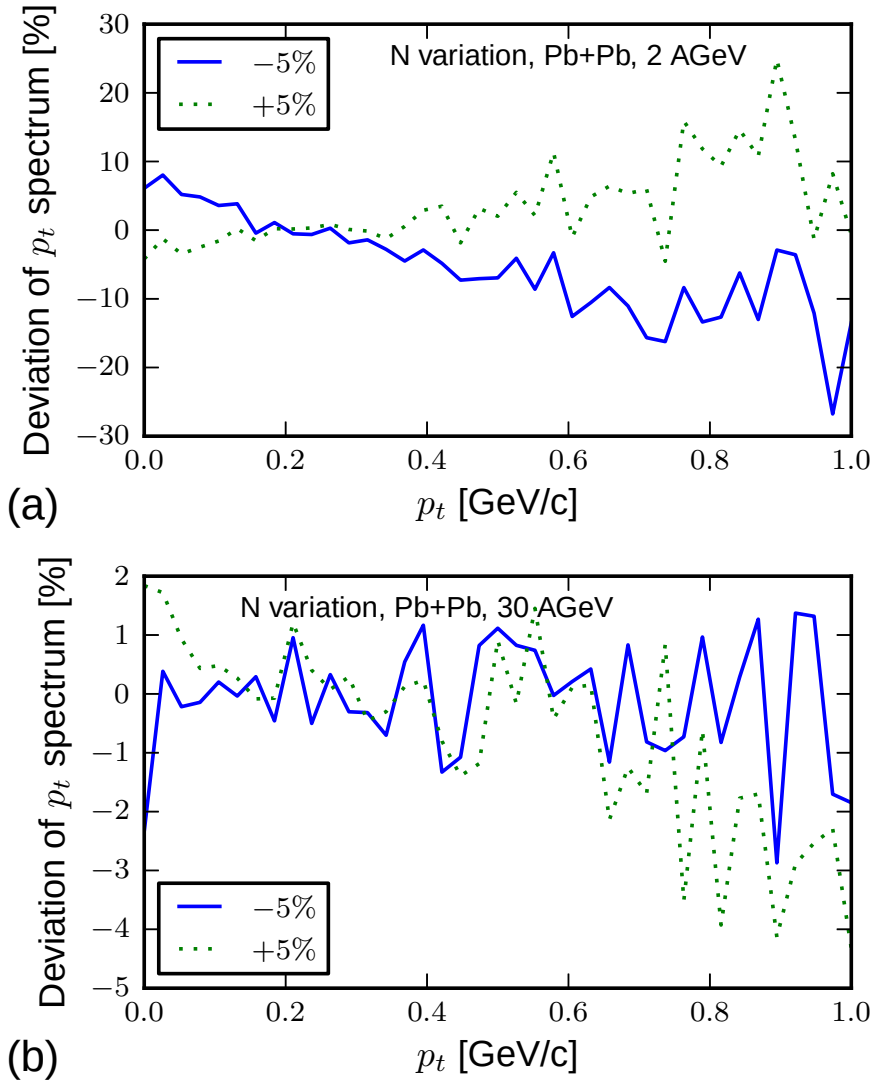


Figure 3.7.: (a): Deviation of pion transverse momentum spectra (p_t) in Pb+Pb collisions at 2A GeV for variations of the nucleon-resonance masses. (b): Deviation of pion transverse momentum spectra (p_t) in Pb+Pb collisions at 30A GeV for variations of the nucleon-resonance masses.

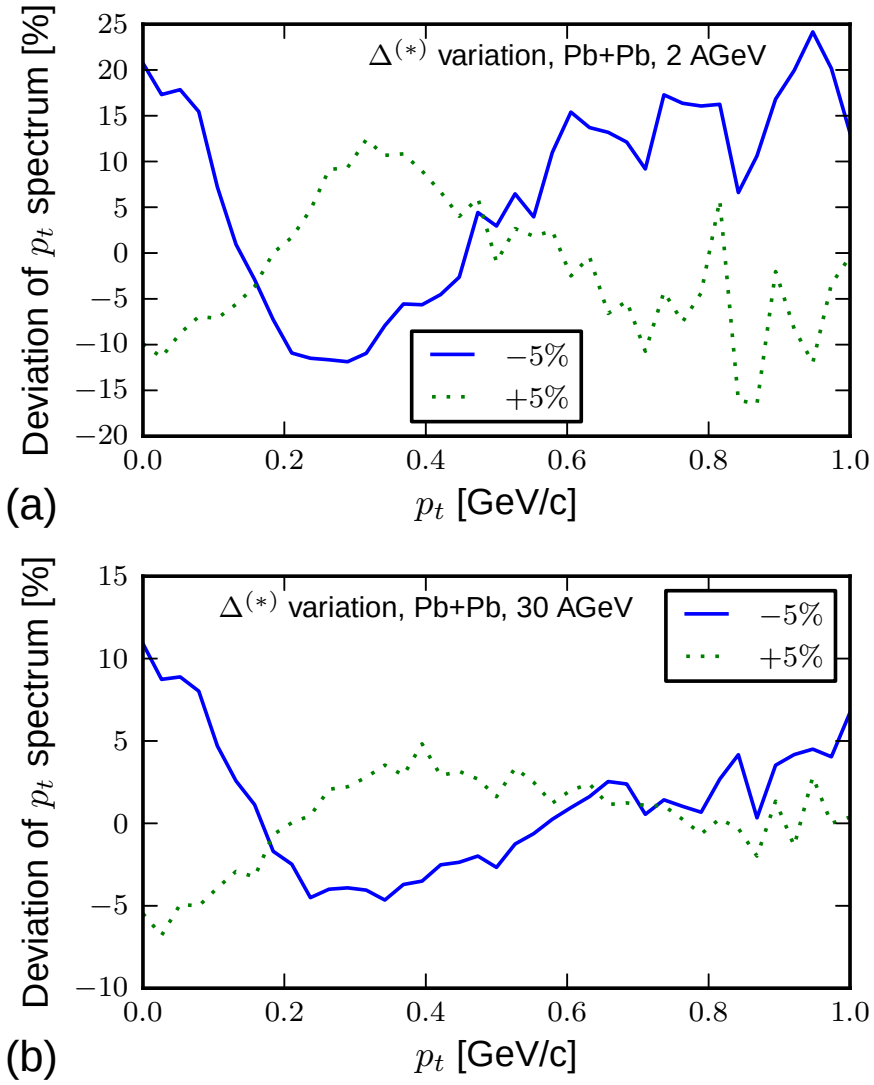


Figure 3.8.: (a): Deviation of pion transverse momentum spectra (p_t) in Pb+Pb collisions at 2A GeV for variations of the Δ resonance masses. (b): Deviation of pion transverse momentum spectra (p_t) in Pb+Pb collisions at 30A GeV for variations of the Δ resonance masses.

3.3.5. Consequences of the Stability Results

In light of the upcoming high precision experiments at FAIR, it is highly desirable to obtain better estimates on the systematic errors of transport simulations. As shown, this question could be addressed by using the UrQMD transport approach in nucleus-nucleus reactions in the FAIR energy regime from 2A to 30A GeV. We have analysed elementary cross sections in pion-nucleon reactions, as well as lead-lead collisions for various sets of input parameter variations of the hadron masses and widths. Although the analysed quantities show globally only a weak dependence, discontinuities like in Figure 3.5 (b) may occur and influence predictions made by the applied models. The dependence is strongest in low-energy collisions, where the collision dynamics is dominated by resonance production and decay. Here, one may encounter systematic errors on the order of $\pm 20\%$. At higher energies, the systematic errors are much smaller. One should note, that we have explored a *worst case scenario* where all parameters were shifted simultaneously in one direction. The error on the masses (and widths) are however uncorrelated and should therefore induce smaller systematic bias into the simulations as compared to this study. Therefore, we conclude that the predictive and analysis power of the present approach is better than a systematic error of 20%.

The method described in section 3.1 can be applied to other simulation frameworks in this field. Therefore stability analyses of these frameworks can be performed before start of the impending experiments at FAIR, without the need of a complete redesign of the frameworks.

4. Hydrodynamics

The usage of a relativistic hydrodynamic approach to describe heavy ion collisions goes back to an idea of Landau [9]. It is motivated by the fact that the mean free path of particles is small compared to the size of a system consistent of two colliding nuclei.

4.1. Theoretical Foundations: the Euler Equations

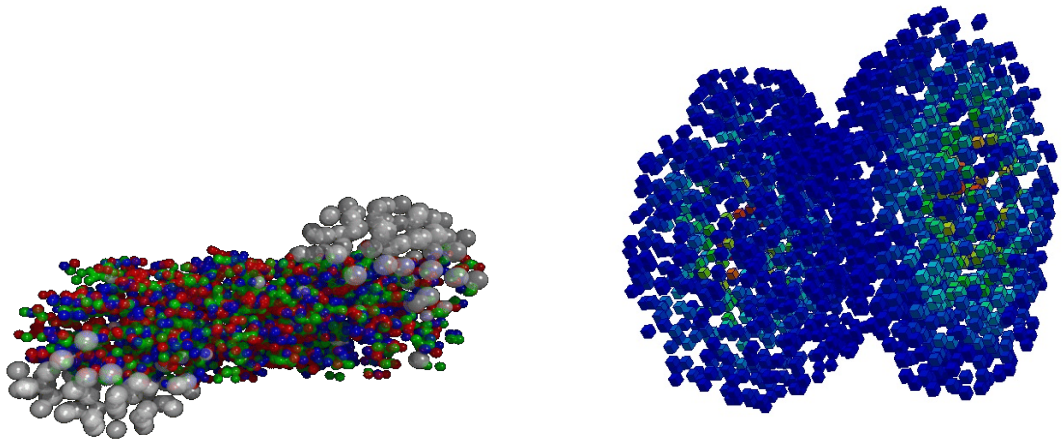
Experimental data, e.g. the elliptical flow observed in RHIC experiments, show that even at today's highest energies the dynamics of strongly interacting matter can be modelled with (ideal) hydrodynamics. The UrQMD hybrid model applies after an initial collision calculation hydrodynamics and uses a transport approach as afterburner [52]. The two borders of a hybrid model are the transition from the transport model (microscopic model) to the hydrodynamic model (macroscopic model) and after the hydrodynamic propagation the transition back again to the transport model. In UrQMD (version 3.3) the transition to the hydrodynamic model is carried out immediately after the initial baryon currents have decoupled, i.e. after $t = \frac{2R}{\gamma_{\mathrm{CM}}}$ with R being the radius of the nuclei and γ_{CM} the Lorentz-factor for the centre of mass system.

In the hydro phase different equations of state can be applied in different phases of the propagation, e.g. depending on the energy regime currently present in regions of the grid. The importance of different equations of state is shown in [58, 34]. According to certain criteria, e.g. average energy density, chemical potential, or merely propagation time, a transition back to the microscopic model is carried out. This process is called *freeze-out*. Let us explore the hydrodynamic phase in detail in the following.

After the transition from particles to fluid volumes certain quanta should be conserved:

- Energy,
- Momenta, and
- Quantum Numbers.

The classic concept of mass conservation is handled by the baryon number conservation, which is more suitable to simulations at this level. Neglecting thermal transport as well as viscosity leads to the *Euler equations* of fluid dynamics.



(a) Particles (picture from [43]).

(b) Fluid parcels.

Figure 4.1.: After the initial phase UrQMD makes a transition from particle degrees of freedom to fluid degrees of freedom.

As explained in section 4.2 neither the mass nor the number of particles are a conserved quantity, when considering relativistic effects. However the baryon number is conserved even in collisions, where relativistic effects take place. The following derivations follow closely [42] and [14]. The conservation form of the Euler equations is the most suitable for the later computational approach. The equations can be derived from a Lagrangian or an Eulerian viewpoint. The Lagrangian approach follows an arbitrary but fixed fluid parcel during the movement, while the Eulerian approach fixes a volume in space and evaluates the changes of quantities within this volume during the flow. These different viewpoints are mirrored in the differential operators $\frac{\partial}{\partial t}$ (the partial time derivative, which keeps the place fixed), and $\frac{\mathrm{D}}{\mathrm{D}t}$ (the total or material derivative). If the fluid is transported by a velocity field $\mathbf{u}$ relative to the fixed (Eulerian) volume, these operators have the following relationship:

$$\frac{\mathrm{D}}{\mathrm{D}t} = \frac{\partial}{\partial t} + \mathbf{u} \cdot \nabla \quad . \tag{4.1}$$

Where the left summand $\frac{\partial}{\partial t}$ describes the local temporal change, and the right summand $\mathbf{u} \cdot \nabla$ the change due to advection.

4.1.1. Baryon Conservation

The conserved baryon number resembles the net number of baryon charge in a certain fluid volume. Albeit the baryon density is subject to relativistic effects the net number is a conserved quantity. For a non-relativistic case it can best be compared to the mass of

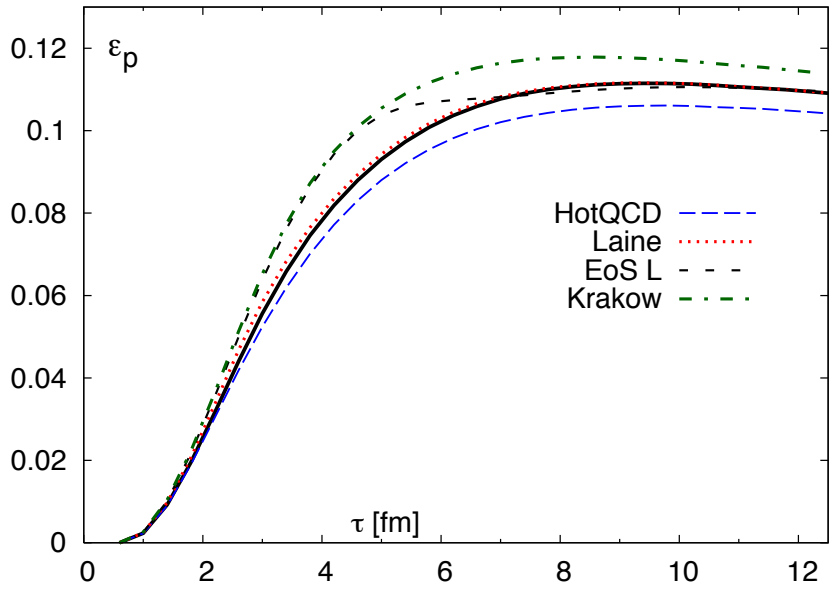


Figure 4.2.: Energy density as a function of time in a typical relativistic heavy ion collision. Different lattice equations of state are compared to the model developed in [34] (solid line). (Figure taken from [34])

the propagated fluids. Let $\rho(\mathbf{x}, t) \in \mathbb{R}$ be the baryon density, such that for an arbitrary Volume V the total baryon number at time t is:

$$R = \int_V \rho(\mathbf{x}, t)\, \mathrm{d}\mathbf{x} \quad . \tag{4.2}$$

As no baryons are created or destroyed the baryon number of volume V changes only by the flux of baryons across the border ∂V of the volume. If $\mathbf{n}$ is the outward normal vector on the surface of the volume, then the baryon flux is given by $\rho\mathbf{u} \cdot \mathbf{n}$. Therefore:

$$\frac{\partial}{\partial t} \int_V \rho(\mathbf{x}, t)\, \mathrm{d}\mathbf{x} + \int_{\partial V} \rho\mathbf{u} \cdot \mathbf{n}\, \mathrm{d}\mathbf{s} = 0 \quad . \tag{4.3}$$

The volume V is independent of time t and we can assume the border ∂V is sufficiently smooth and the velocity field is continuously differentiable. With the Gaussian integration theorem follows:

$$\int_V \left(\frac{\partial}{\partial t} \rho(\mathbf{x}, t) + \nabla \cdot (\rho(\mathbf{x}, t)\mathbf{u}) \right) \mathrm{d}\mathbf{x} = 0 \quad . \tag{4.4}$$

As this must hold for arbitrary volumes V the integrand has to vanish. Therefore we derive the *continuity equation*:

$$\frac{\partial}{\partial t} \rho + \nabla \cdot (\rho\mathbf{u}) = 0 \quad . \tag{4.5}$$

4.1.2. Conservation of Momenta

The equation for the conservation of momenta is often simply called Euler equation, as it was Euler's proposal to describe the movements of ideal fluids[1]. The model of ideal fluids neglects the viscosity of fluids. Similar to the case of the continuity equation not only the local changes of momentum due to pressure, but also the flux of momentum has to be considered. The local change on the momentum N within a volume V is due to the force acting on the surface of the volume and the flux. Therefore:

$$\partial_t N = \frac{\partial}{\partial t} \int_V \rho(\mathbf{x}, t)\mathbf{u}\,\mathrm{d}\mathbf{x} \tag{4.6}$$

$$= -\int_{\partial V} p\mathbf{n}\,\mathrm{d}\mathbf{s} - \int_{\partial V} (\rho(\mathbf{x}, t)\mathbf{u})\,\mathbf{u}\cdot\mathbf{n}\,\mathrm{d}\mathbf{s} \quad . \tag{4.7}$$

Again with the Gaussian integration theorem follows:

$$\int_V \frac{\partial}{\partial t}(\rho\mathbf{u}) + \nabla\cdot\left(p\bar{\bar{\mathbf{I}}} + \rho\mathbf{u}\otimes\mathbf{u}\right)\,\mathrm{d}\mathbf{x} = 0 \quad . \tag{4.8}$$

With $\bar{\bar{\mathbf{I}}}$ being the identity tensor of rank 2 and the divergence $(\nabla\cdot)$ extended to rank 2 tensor fields. Hence the differential form of the Euler equation is:

$$\frac{\partial}{\partial t}(\rho\mathbf{u}) + \nabla\cdot\left(p\bar{\bar{\mathbf{I}}} + \rho\mathbf{u}\otimes\mathbf{u}\right) = 0 \quad \Leftrightarrow \tag{4.9}$$

$$\frac{\partial}{\partial t}(\rho\mathbf{u}) + \nabla\cdot(\rho\mathbf{u}\otimes\mathbf{u}) + \nabla p = 0 \quad . \tag{4.10}$$

The conservation of momentum is a direct consequence of Newton's second law of mechanics $\mathbf{F} = \dot{\mathbf{N}}$. Similarly $\mathbf{F} = m\mathbf{a}$ can be used to derive the total change of the fluid velocity vector:

$$\rho\frac{\mathrm{D}\mathbf{u}}{\mathrm{D}t} + \nabla p = 0 \quad , \tag{4.11}$$

$$\rho\left[\frac{\partial}{\partial t} + \mathbf{u}\cdot\nabla\right]\mathbf{u} + \nabla p = 0 \quad , \tag{4.12}$$

$$\frac{\partial\mathbf{u}}{\partial t} + (\mathbf{u}\cdot\nabla)\,\mathbf{u} + \frac{1}{\rho}\nabla p = 0 \quad . \tag{4.13}$$

The derivation of the integral form, using the *material derivative* applies Reynold's transport theorem to bridge between the immutable *Eulerian* volumes and the time dependent *Lagrangian* fluid parcels.

[1] Feynman describes this as *dry water* [21].

4.1.3. Conservation of Energy

The equation for the conservation of energy for adiabatic processes was formulated by Laplace. The conservation of energy is derived applying the Gibbs relation:

$$\mathrm{d}\omega = T\,\mathrm{d}S + \frac{1}{\rho}\,\mathrm{d}p \quad . \tag{4.14}$$

For the enthalpy ω, temperature T, entropy S, pressure p, and density ρ. Supposing isentropic fluids one obtains

$$\mathrm{d}p = \rho\,\mathrm{d}\omega \quad . \tag{4.15}$$

For a fixed volume V the total energy within this volume is:

$$E = E_{\mathrm{kin}} + E_{inner} \quad \Leftrightarrow \tag{4.16}$$

$$E = \frac{1}{2}\left(\mathbf{u}\cdot\mathbf{u}\right) + \varepsilon \quad . \tag{4.17}$$

For the isentropic case the lack of $T\,\mathrm{d}S$ leads to the simpler derivatives for the kinetic and inner energy:

$$\frac{\partial}{\partial t}\left(\rho\frac{u^2}{2}\right) = -\frac{u^2}{2}\nabla\cdot(\rho\mathbf{u}) - \rho\mathbf{u}\cdot\nabla\left(\frac{u^2}{2}+\omega\right) \quad , \tag{4.18}$$

$$\frac{\partial}{\partial t}(\rho\varepsilon) = \omega\frac{\partial\rho}{\partial t} \quad . \tag{4.19}$$

For $u^2 = \mathbf{u}\cdot\mathbf{u}$. The inner energy relates here to the enthalpy by $\omega = \varepsilon + \frac{p}{\rho}$. The local change within a fixed volume V is thereby determined by two quantities, the flux of energy and the work done by the pressure:

$$\frac{\partial}{\partial t}\int\limits_V \rho\left(\frac{1}{2}\left(\mathbf{u}\cdot\mathbf{u}\right)+\varepsilon\right)\mathrm{d}\mathbf{x} = -\int\limits_{\partial V}\rho\mathbf{u}\left(\frac{1}{2}\mathbf{u}\cdot\mathbf{u}+\varepsilon\right)\cdot\mathbf{n}\,\mathrm{d}\mathbf{s} - \int\limits_{\partial V} p\mathbf{u}\cdot\mathbf{n}\,\mathrm{d}\mathbf{s} \quad . \tag{4.20}$$

Which holds as differential form:

$$\frac{\partial}{\partial t}E + \nabla\cdot\left[\mathbf{u}\left(E+p\right)\right] = 0 \quad . \tag{4.21}$$

4.1.4. Coordinate Form of Euler Equations

The Euler equations in differential form:

$$\frac{\partial}{\partial t}\rho + \nabla\cdot(\rho\mathbf{u}) = 0 \quad , \tag{4.22}$$

$$\frac{\partial}{\partial t}(\rho\mathbf{u}) + \nabla\cdot(\rho\mathbf{u}\otimes\mathbf{u}) + \nabla p = 0 \quad , \tag{4.23}$$

$$\frac{\partial}{\partial t}E + \nabla\cdot\left[\mathbf{u}\left(E+p\right)\right] = 0 \quad , \tag{4.24}$$

are, formulated in coordinate form, almost identically implemented in the explicit solver analysed in chapter 4.4:

$$\frac{\Delta}{\Delta t}\begin{pmatrix}\rho\\ \rho u\\ \rho v\\ \rho w\\ E\end{pmatrix}+\frac{\Delta}{\Delta x}\begin{pmatrix}\rho u\\ p+\rho u^2\\ \rho uv\\ \rho uw\\ u(E+p)\end{pmatrix}+\frac{\Delta}{\Delta y}\begin{pmatrix}\rho v\\ \rho uv\\ p+\rho v^2\\ \rho vw\\ v(E+p)\end{pmatrix}+\frac{\Delta}{\Delta z}\begin{pmatrix}\rho w\\ \rho uw\\ \rho vw\\ p+\rho w^2\\ w(E+p)\end{pmatrix}=\mathbf{0} \quad , \tag{4.25}$$

with $\mathbf{u}=(u,v,w)^{\mathrm{T}}$.

4.2. Relativistic Hydrodynamics

Collisions take usually place at an energy regime subject to significant relativistic effects, e.g. two protons colliding with a centre of mass energy of $\sqrt{s}=2\,\mathrm{GeV}$ have increased their mass by about 10%. A more exhaustive explanation of relativistic hydrodynamics can be found in [16] and [50]. I will follow these references in the next section's elaboration. In the formulation of special relativity used here, Greek indices count from 0 to 3 while Latin indices count from 1 to 3. I follow the Einstein summation convention, i.e. in terms with doubly used indices as super- and subscript, the summation of these terms is meant. If not otherwise stated in the following let $v=|\mathbf{v}|=\sqrt{\sum_{i=1}^{3}v_i^2}$ be the norm for the velocity three-vector $\mathbf{v}$.

4.2.1. Relativistic Conservation of Baryon Number

Fundamental changes due to relativistic effects are the contraction of volumes and the increase of mass. A conserved quantity is the baryon number $B=\frac{1}{3}\left(n_{\mathrm{q}}-n_{\bar{\mathrm{q}}}\right)$, with n_{q} being the number of quarks and $n_{\bar{\mathrm{q}}}$ the number of anti-quarks. Hence the baryon density is subject to relativistic corrections. Let $\mathbf{v}$ be the velocity, then

$$\gamma=\frac{1}{\sqrt{1-v^2}} \quad ,$$

is the so called Lorentz-factor. Instead of the non-relativistic density ρ the conserved quantity is then $\mathcal{N}=\gamma\rho$ therefore:

$$\frac{\partial}{\partial t}\mathcal{N}+\nabla\cdot(\mathcal{N}\mathbf{v})=0 \quad . \tag{4.26}$$

It is common to use tensor notation in special relativity. The relativistic equation of baryon conservation is thereby[2]:

$$\frac{\partial}{\partial x^{\mu}}N^{\mu}=0 \quad . \tag{4.27}$$

[2]Using the Einstein summation convention.

For the four-vector $N^\mu = \rho u^\mu$ defined by the four-velocity $u^\mu = (\gamma, \gamma\mathbf{v})$. Because of

$$0 = \frac{\partial}{\partial x^\mu} N^\mu = \sum_{\mu=0}^{3} \frac{\partial}{\partial x^\mu} N^\mu \tag{4.28}$$

$$= \frac{\partial}{\partial t} N^0 + \sum_{i=1}^{3} \frac{\partial}{\partial x^i} N^i \tag{4.29}$$

$$= \frac{\partial}{\partial t}(\gamma\rho) + \nabla \cdot (\gamma\rho\mathbf{v}) \quad , \tag{4.30}$$

the equations (4.26) and (4.27) are equal. They are the relativistic counterpart to equation (4.22).

4.2.2. Conservation of Energy-Momentum-Tensor

In the relativistic case, there is a flux between the momenta and the energy of a fluid volume. This is because the energy of a particle changes with its velocity. Hence momentum and energy are not independent, as in the non-relativistic case, but put together in the so called energy-momentum-tensor $T^{\mu\nu}$. The components of $T^{\mu\nu}$ are:

- T^{00} the energy density,
- T^{0j} the density of the momentum in x_j axis,
- T^{i0} the flux of energy in x_i axis, and
- T^{ij} the flux of the x_j-momentum to the x_i axis.

The energy-momentum-tensor is in its simplest form when calculated in the local rest frame, i.e. the frame where the fluid volume is at rest:

$$T^{\mu\nu}_{\mathrm{LRF}} = \begin{pmatrix} \varepsilon & 0 & 0 & 0 \\ 0 & p & 0 & 0 \\ 0 & 0 & p & 0 \\ 0 & 0 & 0 & p \end{pmatrix} \quad . \tag{4.31}$$

As the energy-momentum-tensor is symmetric in the local rest frame, it has to be symmetric in all frames[3]. Transforming into the computational frame one obtains:

$$T^{\mu\nu}_{\mathrm{CF}} = \begin{pmatrix} (\varepsilon+p)\,\gamma^2 - p & \gamma u_x\,(\varepsilon+p) & \gamma u_y\,(\varepsilon+p) & \gamma u_z\,(\varepsilon+p) \\ \gamma u_x\,(\varepsilon+p) & u_x^2\,(\varepsilon+p) + p & u_x u_y\,(\varepsilon+p) & u_x u_z\,(\varepsilon+p) \\ \gamma u_y\,(\varepsilon+p) & u_x u_y\,(\varepsilon+p) & u_y^2\,(\varepsilon+p) + p & u_y z_z\,(\varepsilon+p) \\ \gamma u_z\,(\varepsilon+p) & u_x u_y\,(\varepsilon+p) & u_y u_z\,(\varepsilon+p) & u_z^2\,(\varepsilon+p) + p \end{pmatrix} \tag{4.32}$$

$$= (\varepsilon+p)\,u^\mu u^\nu - p g^{\mu\nu} \quad . \tag{4.33}$$

[3]The symmetry is invariant under Lorentz-transformations.

While the non-relativistic momentum density is given by $\rho\mathbf{v}$ in the relativistic part it is determined by the energy density and additionally by the pressure $(\varepsilon + p)\mathbf{v}$. Thus, the conservation of energy and momentum together reads:

$$\frac{\partial}{\partial x^\mu} T^{\mu\nu} = 0 \quad . \tag{4.34}$$

4.2.3. Similarity to non-relativistic Euler Equations

For $w = \varepsilon + p$ the energy-momentum-tensor can be written as:

$$T^{ij} = w\gamma^2 v_i v_j + p\delta_{ij} \quad , \tag{4.35}$$
$$T^{0i} = w\gamma^2 v_i \quad , \tag{4.36}$$
$$T^{00} = (\varepsilon + pv^2)\gamma^2 \quad . \tag{4.37}$$

Setting then:

$$\mathcal{N} = \rho\gamma \quad , \tag{4.38}$$
$$\boldsymbol{\mathcal{M}} = w\gamma^2\mathbf{v} \quad , \text{and} \tag{4.39}$$
$$\mathcal{E} = (\varepsilon + pv^2)\gamma^2 \quad , \tag{4.40}$$

the relativistic Euler equations take the form similar to the non-relativistic differential equations (4.22), (4.23), and (4.24):

$$\frac{\partial}{\partial t}\mathcal{N} + \nabla \cdot (\mathcal{N}\mathbf{v}) = 0 \quad , \tag{4.41}$$
$$\frac{\partial}{\partial t}\boldsymbol{\mathcal{M}} + \nabla \cdot \left(\boldsymbol{\mathcal{M}}\mathbf{v} + p\bar{\bar{\mathbf{I}}}\right) = 0 \quad , \tag{4.42}$$
$$\frac{\partial}{\partial t}\mathcal{E} + \nabla \cdot (\mathbf{v}\,(\mathcal{E} + p)) = 0 \quad . \tag{4.43}$$

4.3. Algorithm Types for Hydrodynamics

Computational fluid dynamics is a widely studied field in applied mathematics. The vast choice of algorithms, ranging from explicit Eulerian solvers to implicit multigrid methods is the fruit of mathematical research ranging back to the works of d'Alembert, Euler, Lagrange, and Laplace. In the twentieth century different numerical methods solving partial differential equations have been developed. The selection of an appropriate solver for hyperbolic differential equations, depends on the boundary conditions of the underlying physics model. Although pure *finite volume methods* are well suited to transport problems, in the case of ideal relativistic hydrodynamics in [57] the SHASTA is shown to provide solutions best suited to physics conditions present. The SHASTA introduced in [12] is classified as a *flux-corrected transport* algorithm. Boris and Book derive their results

from Eulerian *finite difference* algorithms which they enhance by subsequent correction steps. The authors introduced with their paper [12] the family of *flux-corrected transport* methods. However the relationship of the SHASTA is certainly seen closer to the *finite volume* methods than to *finite difference* methods in modern nomenclature.

I concentrate in the following on the redesign and optimisation of the SHASTA. Nevertheless the optimisations seldom take advantage of special properties of the SHASTA. Hence, they can be applied in general to all explicit solvers and most of them even to implicit solvers. For our purpose, however, explicit methods are in the centre of interest. They allow a direct access to all physics quantities within each time step, instead of a more abstract iteration step. This direct access allows for a further development of in situ functions, like the relativistic corrector or the calculation of derived quantities for possible freeze-out routines.

4.4. The SHASTA

The invention [40] of flux-corrected transport algorithms dates back to 1971 and was widely presented in [12] with the *Sharp and Smooth Transport Algorithm* (SHASTA). Since then various algorithms have been derived using the original ideas of [12]. In UrQMD a relativistic implementation [57] of SHASTA is used. It is composed of five phases:

1. Geometric transport.
2. Anti-Diffusion.
3. Flux limiter.
4. Relativistic correction.
5. Relativistic calculation of the equation of state.

The SHASTA is specified for one dimensional transport. Although otherwise stated [12], its direct extension to three dimensions in its classic form is not possible. The major obstacle is the search for optima in three dimensions to compensate for nonphysical flows. The most straight forward approach to this problem is by dimension splitting, though numerical artefacts are produced with this approach[4].

4.4.1. Geometric Transport

In the geometric transport phase the algorithm makes a transition from the Eulerian grid to Lagrangian fluid parcels that are transported following a geometric scheme. In figure 4.3

[4]The problem of *quadratic* fireballs was a returning issue in presentations of the SHASTA authors [40].

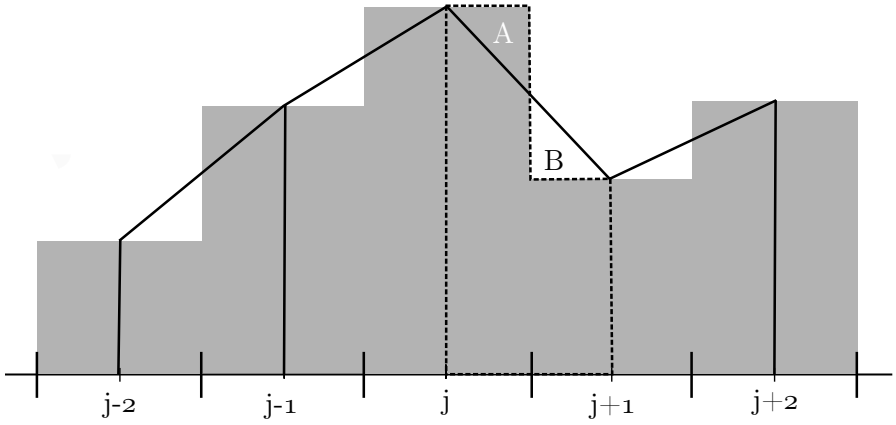


Figure 4.3.: The area of each rectangle, between two big lines on the scale, resemble the content of a cell in the Eulerian grid. The trapezoids, resembling Lagrangian fluid parcels, have the same area as their two underlying half rectangles.

each rectangle represents the content of a cell in the Eulerian grid, e.g. if the height is the mass density and its area is the mass attributed to a cell. As a consequence of the intercept theorem each trapezoid has the same area as the two underlying halves of the rectangles. Identical angles and same side lead to congruent triangles$A$ and B and therefore these triangles have the same area. Each trapezoid (and therefore Lagrangian fluid parcel) is now transported. This is done by moving the trapezoids' y-parallel sides with the velocities of the corresponding cells. In figure 4.3 the dotted trapezoid's side in cell j with v_j and the side in cell $j+1$ with v_{j+1}. As the velocities are not necessarily identical this leads to a distortion of the trapezoid's baseline. To guarantee the conservation of cell's content, the trapezoid has to maintain the same area as before the movement, hence each y-parallel side is scaled by the geometrical factor:

$$\mu = \frac{\Delta x}{\Delta x + \Delta t \left(v_{j+1} - v_j\right)} \quad . \tag{4.44}$$

Assuming $\left|v_i \frac{\Delta t}{\Delta x}\right| < \frac{1}{2}$ for all cells i, the scaling of the trapezoids is sufficient to guarantee positivity after the propagation step. Positivity is equivalent to:

$$0 < \frac{\Delta x}{\Delta x + \Delta t \left(v_{j+1} - v_j\right)} \quad \Leftrightarrow \tag{4.45}$$

$$0 < \Delta x + \Delta t \left(v_{j+1} - v_j\right) \quad \Leftrightarrow \tag{4.46}$$

$$1 > v_j \frac{\Delta t}{\Delta x} - v_{j+1} \frac{\Delta t}{\Delta x} \quad . \tag{4.47}$$

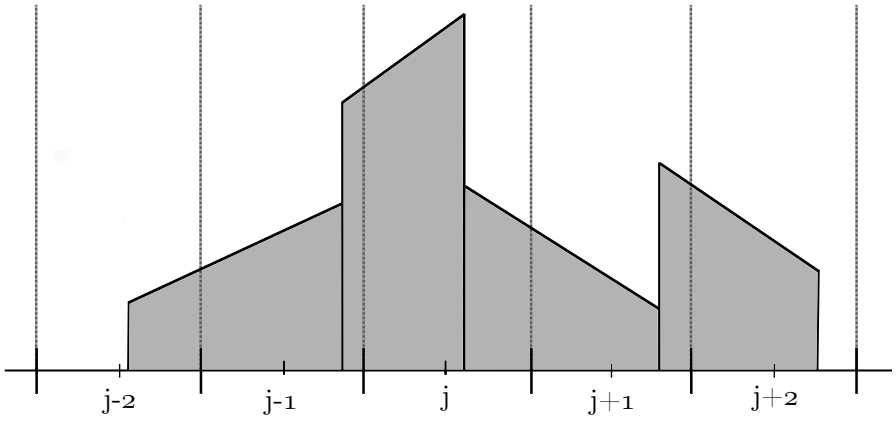


Figure 4.4.: The trapezoids are moved and scaled. The areas of the trapezoids residing in a cell are interpreted as the new content of a cell.

Using the assumption $\left|v_i \frac{\Delta t}{\Delta x}\right| < \frac{1}{2}$ twice:

$$v_j \frac{\Delta t}{\Delta x} - v_{j+1} \frac{\Delta t}{\Delta x} < \tag{4.48}$$

$$\frac{1}{2} - v_{j+1} \frac{\Delta t}{\Delta x} < \tag{4.49}$$

$$\frac{1}{2} - \left(-\frac{1}{2}\right) = 1 \quad . \tag{4.50}$$

One can see the strict inequality (4.47) holds because of $\Delta x > 0$, hence positivity is guaranteed. After the movement of the trapezoids (see figure 4.4) their area is reinterpolated to the underlying Eulerian cells. As all trapezoids and cells can be handled independently, this suggests first parallelisation possibilities. I will elaborate on that in chapter 5. The interpolation is done by calculating a convex combination of the trapezoids' area for each cell. The authors of SHASTA propose the usage of different approximations for the cell velocity [12]. In the relativistic case the centred time approach is used [57][5]. Combining the interpolation and geometric factors leads to:

$$Q_+ = \frac{\frac{1}{2} - v_j^{n+\frac{1}{2}} \cdot \lambda}{1 + (v_{j+1}^{n+\frac{1}{2}} \cdot \lambda - v_j^{n+\frac{1}{2}} \cdot \lambda)} \quad , \tag{4.51}$$

$$Q_- = \frac{\frac{1}{2} + v_j^{n+\frac{1}{2}} \cdot \lambda}{1 - (v_{j-1}^{n+\frac{1}{2}} \cdot \lambda - v_j^{n+\frac{1}{2}} \cdot \lambda)} \quad , \tag{4.52}$$

for $\lambda = \frac{\Delta t}{\Delta x}$, and the velocity v.

[5] Thus making use only of a staggered grid in time and not in space.

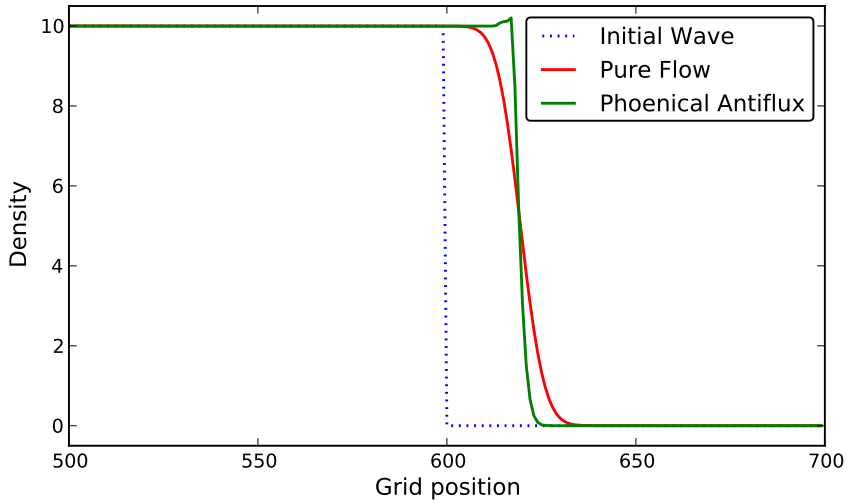


Figure 4.5.: One dimensional propagation of a square wave with $v = 0.5$, $\lambda = 0.4$, and $T = 100$ time steps. The diffusion occurring in the pure geometric propagation is almost completely suppressed when the *phoenical antiflux* is applied.

For an arbitrary density ρ_j^n at time-step n the propagated and interpolated cell content $\hat{\rho}_j^{n+1}$, without any source terms, takes the following form:

$$\hat{\rho}_j^{n+1} = \frac{1}{2}Q_+^2\left(\rho_{j+1}^n - \rho_j^n\right) - \frac{1}{2}Q_-^2\left(\rho_j^n - \rho_{j-1}^n\right) + (Q_+ + Q_-)\rho_j^n \quad . \tag{4.53}$$

4.4.2. Anti-Diffusion and Flux Limiter

Unfortunately the geometric transport in equation (4.53) generates inherently a numerical error[6](see figure 4.5). This can easily be calculated for the constant velocity $v = 0$ case, as the factors $Q_\pm$ both evaluate to $Q_\pm = \frac{1}{2}$:

$$\hat{\rho}_j^{n+1} = \rho_j^n + \frac{1}{8}\left(\rho_{j+1}^n - 2\rho_j^n + \rho_{j-1}^n\right) \quad , \tag{4.54}$$

whilst the analytical solution for the linear advection equation with velocity $v = 0$ is of cause $\rho_j^{n+1} = \rho_j$. Hence the quantity $\hat{\rho}_j^{n+1}$ can be seen as the *correct* quantity ρ_j^{n+1} plus an erroneous numerical diffusion flux composed of an incoming flux f_j and an outgoing flux f_{j-1}.

$$\hat{\rho}_j^{n+1} = \rho_j^{n+1} + f_j - f_{j-1} \quad . \tag{4.55}$$

[6]This error is often called numerical diffusion because of its form similar to the differential quotient used to express the second derivative. Although analogous (due a second derivative) to physical diffusion, this is a purely numerical artefact.

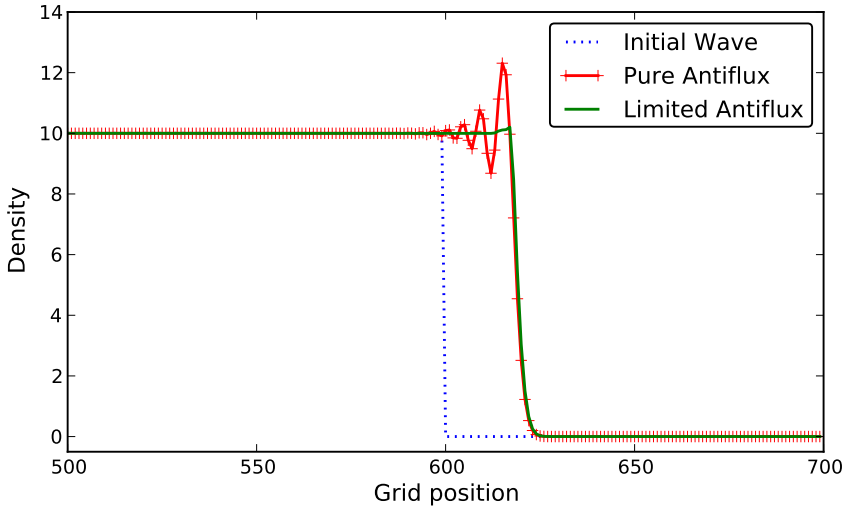


Figure 4.6.: One dimensional propagation of a square wave with $v = 0.5$, $\lambda = 0.4$, and $T = 100$ time steps.Without the flux limiter the *phoenical antiflux* creates significant over- and undershoots. These are mitigated by the *flux limiter*.

Therefore in [12] and [11] different anti-diffusion correctors are examined, to mitigate the erroneous flux between the cells. The relativistic implementation [57] makes use of the so called *phoenical* anti-diffusion:

$$A_j^{\mathrm{ph}} = \frac{1}{8}\left(\hat{\Delta}_j - \frac{1}{8}\left(\Delta_{j+1} - 2\Delta_j + \Delta_{j-1}\right)\right) \quad . \tag{4.56}$$

With $\Delta_j = \rho_{j+1} - \rho_j$ and $\hat{\Delta}_j = \hat{\rho}_{j+1} - \hat{\rho}_j$. Regrettably the anti-diffusion fluxes can not simply be added to compensate for the numerical diffusion, as they might lead to non positive results (compare figure 4.6). For physical concerns the application of anti-diffusion fluxes must not generate any new optima, nor emphasise existing optima in the density distribution. Thus, the anti-diffusion itself is corrected by a flux limiter depending on a neighbourhood of points: the final density after the application of anti-diffusion must not be higher than the density of neighbouring cells. Hence the flux limited anti-diffusion is defined by:

$$\sigma = \mathrm{sgn}\left(A_j^{\mathrm{ph}}\right) \quad , \tag{4.57}$$

$$A_j = \sigma \cdot \max\left\{0, \min\left\{\sigma\hat{\Delta}_{j+1}, \left|A_j^{\mathrm{ph}}\right|, \sigma\hat{\Delta}_{j-1}\right\}\right\} \quad , \tag{4.58}$$

and the transported and corrected quantity by:

$$\rho_j^{n+1} = \hat{\rho}_j^{n+1} - A_j + A_{j-1} \quad . \tag{4.59}$$

4.4.3. Relativistic Amendments

The numerical scheme can lead to spurious energy densities in cells, i.e. the physical constraint $|\boldsymbol{\mathcal{M}}| \leq |\mathcal{E}|$ can be violated, as energy and momenta are transported independently, but are closely entangled as seen in section 4.2. Following [57] this is corrected by symmetrically downscaling $\boldsymbol{\mathcal{M}}_j$ until $|\boldsymbol{\mathcal{M}}| = |\mathcal{E}|$ if it exceeds the energy content. As an additional stabiliser, all contents smaller than a fixed threshold are set to zero. Although this influences the conservation of momenta, the overall effect has proven to be negligible.

To gain the thermodynamic quantities, like pressure $p(\varepsilon, \rho)$, baryo-chemical potential $\mu(\varepsilon, \rho)$, and also the propagation velocity $\mathbf{v}(p, \mathbf{M}, E)$, the quantities $\mathcal{E}$, $\boldsymbol{\mathcal{M}}$, and $\mathcal{N}$ have to be boosted to their respective eigenframe. Nevertheless it is not necessary to invert the equations (4.38), (4.39), and (4.40) completely. As stated in [57], the following fixed point scheme can be applied.

Starting from Equation (4.38) the baryon density in the eigenframe is simply given by $\rho = \gamma^{-1}\mathcal{N}$. From equation (4.40) follows:

$$\begin{aligned}
\mathcal{E} &= \left(\varepsilon + v^2 p\right)\gamma^2 && (4.60)\\
&= \frac{\varepsilon + v^2 p}{1 - v^2} = \frac{\varepsilon + p - p + v^2 p}{1 - v^2} && (4.61)\\
&= (\varepsilon + p)\,\gamma^2 - \frac{1 - v^2}{1 - v^2} p && (4.62)\\
&= (\varepsilon + p)\,\gamma^2 - p && (4.63)
\end{aligned}$$

Deriving the norm of the momentum vector (equation (4.39)):

$$\begin{aligned}
\mathcal{M} = |\boldsymbol{\mathcal{M}}| &= \left|w\gamma^2 \mathbf{v}\right| && (4.64)\\
&= w\gamma^2 v = (\varepsilon + p)\,\gamma^2 v && (4.65)\\
&= v\left[(\varepsilon + p)\,\gamma^2 - p + p\right] = v\,(\mathcal{E} + p) && (4.66)
\end{aligned}$$

the energy density in the eigenframe can be obtained:

$$\begin{aligned}
\mathcal{E} - \mathcal{M}v &= \mathcal{E} - v^2\,(\mathcal{E} + p) && (4.67)\\
&= (\varepsilon + p)\,\gamma^2 - p - v^2\left[(\varepsilon + p)\,\gamma^2 - p + p\right] && (4.68)\\
&= \frac{(\varepsilon + p) - \left(1 - v^2\right) p - v^2\,(\varepsilon + p)}{1 - v^2} && (4.69)\\
&= \frac{\varepsilon - v^2\varepsilon}{1 - v^2} = \varepsilon && (4.70)
\end{aligned}$$

Thus the pressure can be calculated as ε and ρ are determined.

The velocity vector $\mathbf{v}$ points to the same direction as the momentum vector $\boldsymbol{\mathcal{M}}$, hence it suffices to calculate its norm. This can be done by the fixed point equation:

$$v = \frac{\mathcal{M}}{\mathcal{E} + p(\varepsilon, \rho)} \quad . \tag{4.71}$$

As $\mathcal{M} \leq \mathcal{E}$ and $p > 0$ the conditions of Banach's fixed-point theorem are met.

5. Implementation of the SHASTA

This chapter is mostly based on our paper [25]. The SHASTA has been used to enhance UrQMD to a hydrodynamics and transport hybrid model [52]. The implementation used is based on [57] and contains the necessary amendments for relativistic effects. As seen in the profiling shown in chapter 1, the hydrodynamic stage of the hybrid model is the most time demanding part of UrQMD and therefore the ideal candidate for optimisation and design studies. The implementation of the numerical scheme described in chapter 4 was completely revised in order to gain two important improvements:

1. The new implementation follows the criteria of good software design, in order to facilitate further improvements, optimisations, and to guarantee a higher usability.

2. The new implementation is more efficient and benefits from developments of new hardware architecture, namely the usage of accelerators like GPUs.

5.1. Software Design

The transition of the classic implementation of SHASTA (realised with FORTRAN) to the OpenCL-SHASTA (realised in OpenCL and C++) proceeded with two essential intermediate step-stones: At first different FORTRAN-C++ hybrid versions were built. The mixture between FORTRAN routines and C++ functions enabled a smooth transition with continuous testing. The second step was a full C++ version which supports all features of the FORTRAN version. The C++ version finally was used to design a many-core enabled OpenCL version.

A major issue was certainly the design transition from a very *problem-oriented* or *classic* implementation (see figure 5.1) in FORTRAN to an implementation following the object oriented design paradigm in C++. OpenCL in its current version [37] does not support class structures on the GPU side, hence the `EoSClass` and `GridClass` are not used in OpenCL-SHASTA. Nevertheless the organisation on the host-side of devices, buffers, queues, and the orchestration of kernels is done with a class structure, implemented in the `GPUClass`. The first step using a hybrid implementation was merely a prototype to test numerics directly, by mimicking the behaviours of the full classic SHASTA. I will come to this later in section 5.1.3.

Figure 5.1.: Dependency in the FORTRAN 77 version of SHASTA. Methods (blue) do not only depend on each other but are also entangled by the usage of `common blocks` (red), hence variables with a global scope. Obviously this hidden dependencies lead to different pitfalls in any parallelisation attempt.

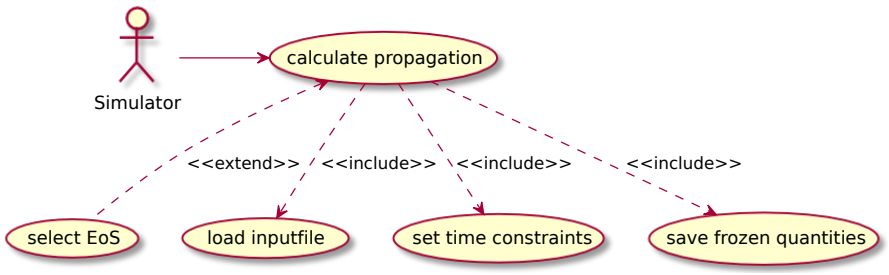


Figure 5.2.: The C++ implementation of SHASTA offers the selection of an equation of state from a fixed set.

5.1.1. From FORTRAN-SHASTA to C++-SHASTA

The main objective of the C++ version was to demonstrate *object oriented design* criteria in physics frameworks and to lay the foundations for the subsequent OpenCL version. The structure of the implemented methods mimics the numerics of the FORTRAN routines as closely as possible. Although the exact mimicking of the numerics imposed constraints to the program structure, the C++ implementation emphasises a clear encapsulation with classes. The functionality of the FORTRAN version is implemented (figure 5.2) almost completely. Two methods: the `changegrid()`-method, responsible for dynamic grid changes if necessary, and the `freezeout()` (omitted in figure 5.1 as implemented in `main()`), responsible for the transformation from fluid parcels to particles have been omitted. In the course of a modular redesign, the parts of the program responsible for the freeze-out should be excluded from the parts responsible for the hydrodynamic calculation. This is necessary, as different physics models and algorithms can be used for the freeze-out and are matter of further research in this area. Therefore the output of C++-SHASTA has the same format, as its input: the densities in a fixed grid. As shown in figure 5.2 the C++ version allows the user the choice of time parameters, input files and the desired equation of state. The propagated quantities are saved and can be loaded into UrQMD. The full integration is planned for a future release of UrQMD. The equations of state are chosen by a configuration file. They are implemented as objects that inherit all necessary methods from their abstract base class and are set before the computation starts (see listing 2). Therefore the rather complicated call-scheme in the FORTRAN version (listing 3 and figure 5.1) is substituted by the class structure shown in figure 5.3. The constructor handles the necessary input by parameters or analytic equation.

In the FORTRAN version the parameter determining the selected EoS is accessed various times, not only in the control logic in listing 3, but also later within the read-in routines, e.g. in listing 7. For the read-in case, the performance issues are negligible, as the reading

```
// initialise equation of state
// depending on choice in init-file
EqofState *EOS;
switch(Data->eos){
  case 1: EOS = new EqofState1(); break;
  case 2: EOS = new EqofState2(); break;
  case 3: EOS = new EqofState3(); break;
  case 4: EOS = new EqofState4(); break;
  case 5: EOS = new EqofState5(); break;
  default: EOS = new EqofState1(); break;
}
```

Listing 2: In the C++ version the `EOS` Object is declared to be from the abstract `EqofState` class and initialised to the class chosen by `Data->eos`.

from tables on the hard disk clearly dominates the execution time. Nevertheless, also in the calculation of thermodynamic quantities, like pressure, temperature, or chemical potential, branchings according to the EoS occur. This superfluous branchings, like in listing 4, are completely avoided in the C++ version due the class structure of the EoS (listing 8).

As SHASTA is implemented with a dimension split approach, the neighbour relation between cells differs with every time step. The index arithmetic is hidden in the `ray`-methods of `GridClass` and `Oriented`. In listing 5 the accessor method of the `fastCube`-objects is shown.

The method depends on the internal state of `stepsize` and `offset`. The member variables are set according to the current propagation direction in the propagation routines. Each ray is handled separately, as can be seen in listing 6. The step-size depends on the propagation direction and is set immediately at the beginning. Subsequently all cells have to be calculated, therefore the one dimensional subsets, or rays, are propagated with the `halfstep` procedure. The starting point, called `offset`, is calculated and passed to the object representing the grid. Therefore the accessor-method respects exactly the neighbourhood and the `halfstep` procedure is exactly the same for all directions and starting points.

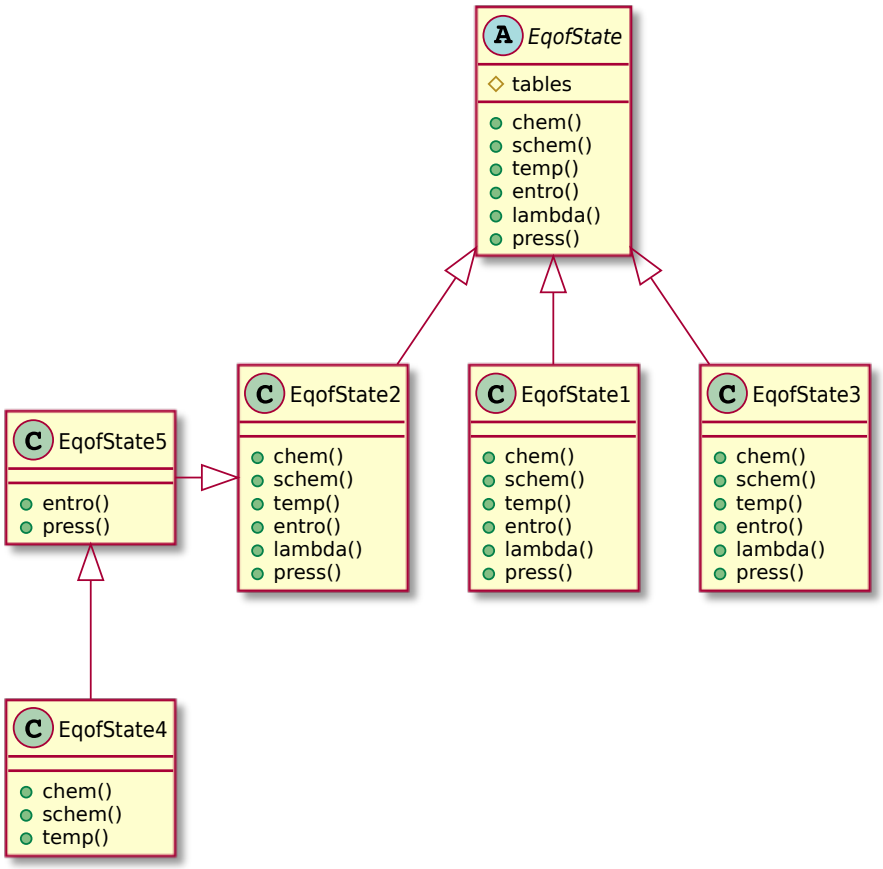


Figure 5.3.: Different equations of state vary often only in a small subset of methods. All methods are purely virtual in the abstract base class `EqofState`. Only the visibility of the tables in the abstract base class (A) is restricted and set to *protected* (denoted by the yellow diamond) all other methods in the different classes (C) are public (denoted by the green circles).

```
if (eos.eq.2) then
call readeos1()
else if (eos.eq.3) then
call readeos2()
else if (eos.eq.4) then
call readeos3()
else if (eos.eq.5) then
call readeos3()
end if
```

Listing 3: In the FORTRAN version the EoS is chosen according to the `eos` variable. Nevertheless there is an ambiguity for the EoS 4 and 5 which is further handled in the `radeos3()` method, shown in listing 7.

```
if (eos.eq.1) then
   chem = 0d0
else if (eos.eq.3) then
   ...
else if ((eos.eq.4).or.(eos.eq.5).or.(eos.eq.2)) then
   if (e.lt.400.0d0) then
      if((e.lt.0.1d0).and.(n.lt.0.02d0)) then
```

Listing 4: In the FORTRAN version the routine `chem()` is responsible for the chemical potential of the underlying EoS. A complicated branching pattern has to cope for the different EoS within the calculation, resulting in unnecessary operations.

5.1.2. OpenCL-SHASTA

While the aim of C++ version of SHASTA was to demonstrate that the usage of OOD the main goal of the OpenCL-SHASTA was to demonstrate the performance and reliability of GPUs for numerical calculations in heavy-ion physics. While a lot of classical fluid dynamical simulations have been carried to GPUs and even some relativistic calculations of astronomers exist, the heavy-ion community requires special properties which are not yet present on GPUs.

Unfortunately the present implementation [37] of OpenCL does not yet allow the usage of OOD on the device-side[1]. Nevertheless, there are C++-bindings on the host-side, allowing for a well organised core of the GPU program.

[1] The *C++ Static kernel language* introduced recently by AMD [60], which supports a subset of the OOD potential of C++, was not used due to the desired independence of any vendor.

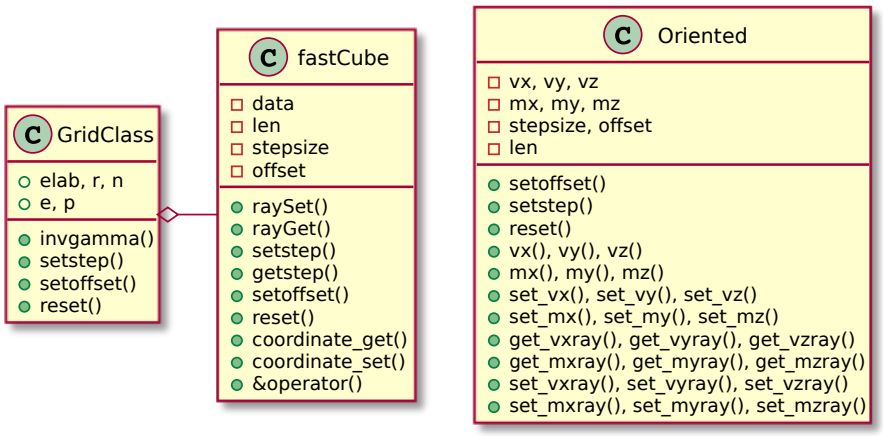


Figure 5.4.: The `GridClass` is aggregated of the `fastCube` class and contains scalar variables. While the `Oriented` class consists of vector variables It can not be aggregated by `fastCube`, as the reset method also permutes the velocity vector. Only the member variables of `fastCube` and `Oriented` are private (denoted by red boxes), other member variables and methods are public (denoted by green circles) to enable the usage of the getters and setters of the underlying data-type.

```
inline float ray_get(int i){
  return( data[i*stepsize+offset] );
};
```

Listing 5: The get-method for grid elements depends on the member variables `stepsize` and `offset`, which are set with accessor-methods.

The fundamental hardware configuration is contained in the `BasicGPU` class, while derived classes are used for profiling, testing and productive usage (see figure 5.5). Further classes can be derived and enhanced by the necessary methods to handle viscosity and other equations of state.

Although the kernels can not fully benefit of the OOD, the modularity and the structure of the C++ version has been maintained as far as applicable on the GPU. Different equations of state can be implemented with small additional kernels, without major modifications to the host code. Use cases for the OpenCL version have been separated further, as shown in figure 5.6.

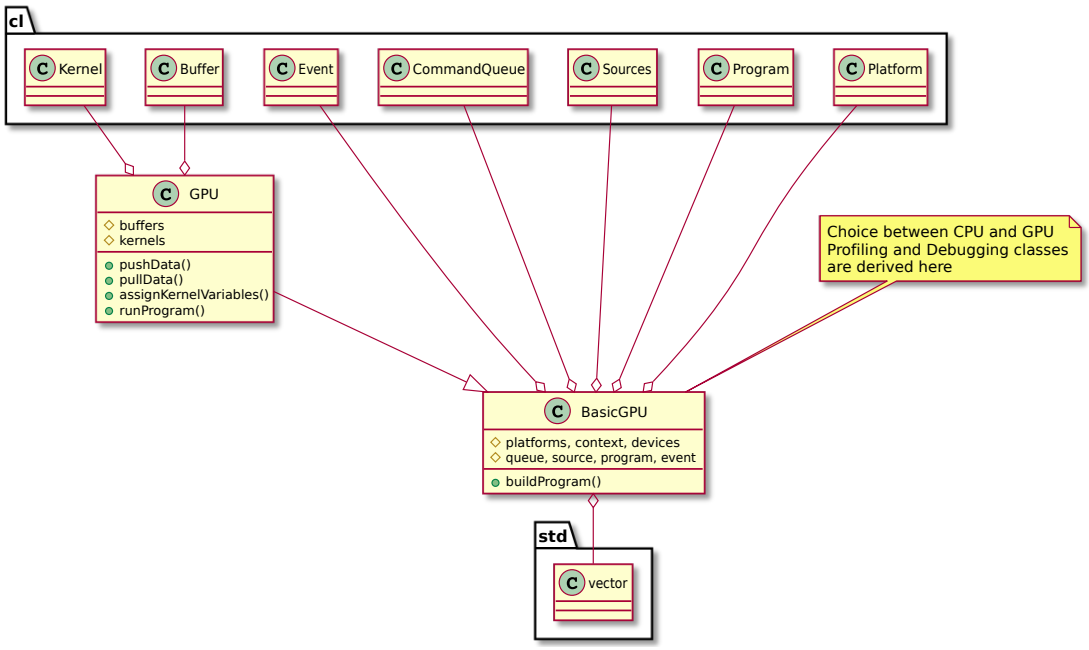


Figure 5.5.: Using the C++ implementation of OpenCL the classes *BasicGPU* and *GPU* are used to control the execution stream. The internal member variables of `GPU` and `BasicGPU` are protected (denoted by yellow diamond) as they are used only for the device management and should not be accessed from outside the classes. Public methods (green circle) allow for a controlled access of the buffers and kernels.

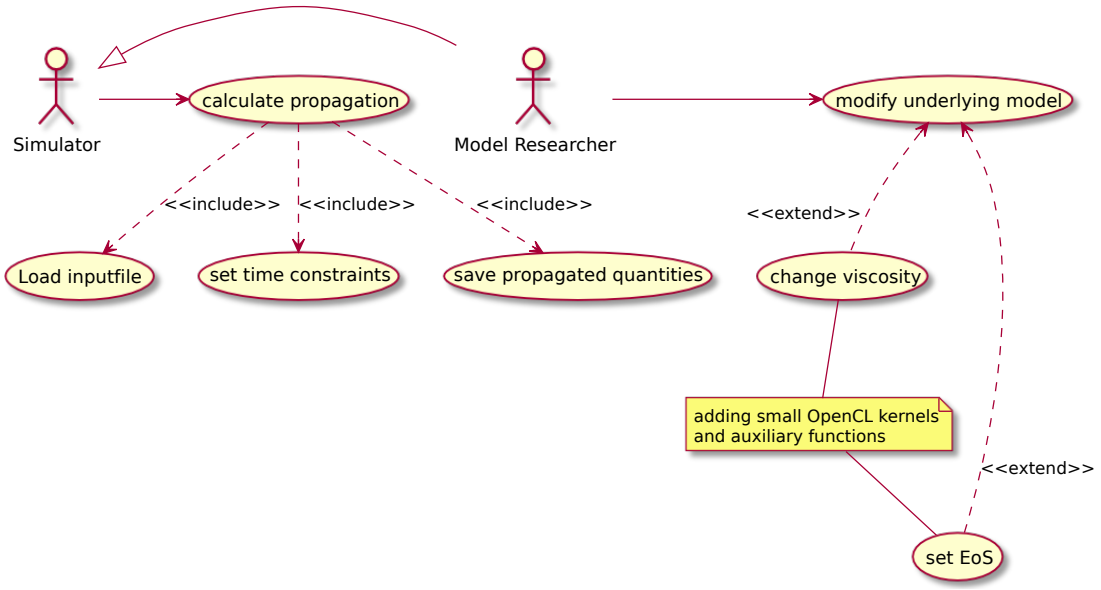


Figure 5.6.: The first implementation of OpenCL-SHASTA aims on two different levels of usage: the OpenCL-SHASTA included in the UrQMD simulation framework is designed for current state hydro+cascade simulations. Additionally OpenCL-SHASTA allows for an extensive model research by adding kernels to the OpenCL code base.

```
Grid->setstep(step);
Vector->setstep(step);
for (int k = 0; k < size; ++k) {
  for (int i = 0; i < size; ++i) {
    position = k*size*size + i;
    Grid->setoffset(position);
    Vector->setoffset(position);
    halfstep(Grid, Vector, EOS, Data);
  }
}
```

Listing 6: The control member variables of the `Oriented` and `GridClass` objects are configured according to the propagation direction before passing the objects to the halfstep function. Within the halfstep function no further difference between the propagation direction is necessary.

5.1.3. Testing

The first step from the FORTRAN 77 version of SHASTA to a C++ hybrid version was accompanied by a continuous *acceptance test*. The FORTRAN subroutines have been refactored in C++ and gradually substituted as symbols on the linker level. Therefore it was possible to cope very early with common pitfalls: line versus column addressing, start indices at zero or one etc.. Although these are not automated *unit tests*, a fine granularity of testing could be preserved, as only small subroutines had to be changed at a time. The resulting data was checked manually with feedback of participating physicists. Very soon the differences between C++ and FORTRAN in the handling of mathematical expressions showed their effects in slight variations of the numerical results. Unfortunately the creation of suitable unit tests needs a significant knowledge of the underlying physics. Only with this knowledge one can estimate the sustainable level of numerical approximations in comparison with the errors introduced automatically by the physics model used.

5.1.4. Algorithm Design

The usage of GPUs had been of great interest to the field of high energy physics even before state-of-the-art programming frameworks, like CUDA or OpenCL, have been developed [19]. In order to harvest the best performance of GPUs, one has to understand the architectural constraints of these devices explained in section 1.4. This implies often a very different approach compared to classical CPU programming. The presented SHASTA implementation is designed in OpenCL and fitted to an AMD Radeon HD 5870 GPU. Therefore certain optimisations are also different [17] to typical optimisations carried out on NVIDIA GPUs.

Although designed for this special kind of GPU the implementation still shows significant speed-up on CPU-only systems. Here additional optimisation, respecting caching and memory layout, e.g. interleaving the three dimensional grid variables, would mitigate the losses due to cache misses when propagating in y or z direction and allow for further significant accelerations.

The OpenCL-SHASTA consists of a C++-part, managing the memory allocation and enqueueing of the kernels.

The approach to parallelisation on GPUs must bear in mind, that the running program at the end consists not of a few parallel threads, but merely thousands of concurrent execution streams. Yet most of this streams are computing *almost* the same, which ranks this approach as a SPMD approach. In order to organise these thousands of execution streams the first, and arguably most important steps, are the *problem decomposition* and *domain decomposition*. Both points are somewhat entangled, still to a first order they can be handled separately.

Problem Decomposition

Firstly the data flow of the algorithm has to be analysed. Not only the physical quantities, that are resulting from the algorithm description, but also the steps in between and intermediate results of the computation are important. In figure 5.7 the dependencies within each time step are illustrated. Before each quantity can be calculated, all the quantities pointed to, have to be computed. The data flow, i.e. the spacial and causal dependencies of variables, determines the constraints to any possible parallel computation. The SHASTA performs the propagation of the energy-momentum-tensor and the net baryon current. Therefore five physical quantities ($\mathcal{E}$, $\boldsymbol{\mathcal{M}}$, and $\mathcal{N}$) are propagated (equations (4.38), (4.39), and (4.40)). The full propagation of each quantity is done by the subsequent steps from equations (4.52) and (4.53) for the pure geometric transport to equation (4.56) for the phoenical antiflux and equations (4.58) and (4.59) for the flux corrected quantity. Subsequently relativistic corrections have to be applied, as well as the calculation of the thermodynamic quantities in their eigenframe.

As illustrated in figure 5.8, the propagation of each quantity, including the anti-diffusion and flux corrector, is calculated independently. As the relativistic corrector needs the propagated state of all quantities (see also figure 5.7), the execution of the relativistic corrector and the calculation of the equation of state is scheduled after all five kernels, propagating the quantities independently, have finished computation. This is realised by the orchestrating method `runProgram()` of the class `GPU` (listing 10).

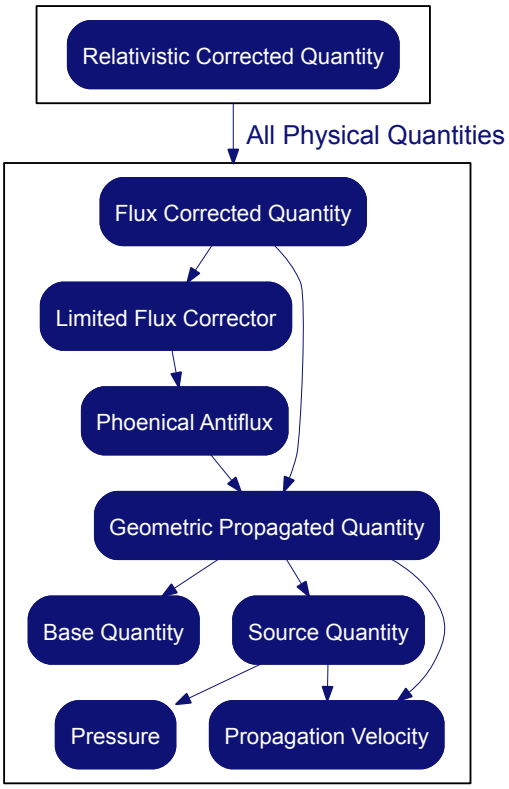


Figure 5.7.: Variable dependencies in data flow, time and space indices are omitted. Each arrow reads as *depends on*.

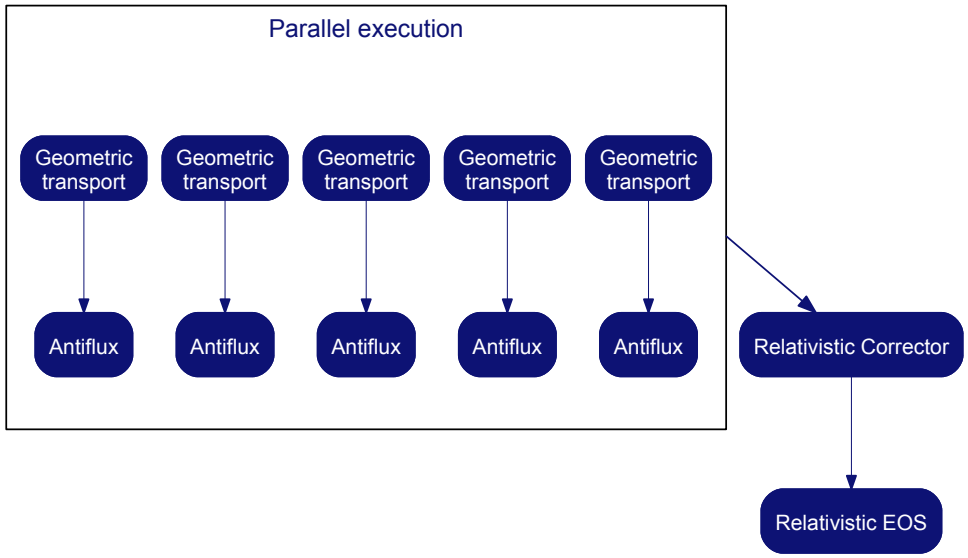


Figure 5.8.: Problem decomposition and data flow in OpenCL-SHASTA.

Another possible decomposition we have investigated, is using the kernels firstly to calculate all the geometric propagated quantities (equation (4.53)) in parallel and subsequently applying new kernels for the anti-diffusion step. Thereby one kernel can calculate more than one geometric propagated quantity (e.g. all the momenta together) which favours the usage of the vector units of GPUs and modern CPUs. However, this approach makes extensive usage of the global memory, by storing intermediate results to the global memory and reloading it in the anti-diffusion kernels later. Thus, different kernels have to be enqueued to serialise the task.

Domain Decomposition

For every grid based algorithm the domain decomposition is often suggested by the grid structure. Though, the exact mapping of the number of work-items to the grid size depends on hardware and algorithm type. The current implementation of OpenCL-SHASTA uses one kernel per physical quantity and hence one work-item per quantity for each of the eight million grid cells. This choice is suitable for GPUs, as it fits their hardware design with the aim of calculating a multitude of voxels in parallel. Optimisations aimed mainly

Figure 5.9.: The neighbourhood used to compute the propagated quantity in cell j (black) spans seven different cells. To compute the flux corrected value for cell j the geometric transport for cells $(j-2)\ldots(j+2)$ (grey) is needed.

on multi-core CPUs would do more efficiently using work-items responsible for more than one cell or more than one physical quantity per cell. Depending on the algorithm type, the dependencies within each grid cell might lead to even more work-items per grid cell on GPUs. The $(3+1)$-dimensional problem of relativistic hydrodynamics is perfectly suitable to the three dimensional `NDRange` arguments in OpenCL. In order to implement the $(3+1)$-dimensional propagation of the energy-momentum-tensor, we use a dimension split approach. Therefore each work-item needs for its calculation only a one dimensional neighbourhood. The size of the 1-D differential stencil still depends on the applied scheme, nevertheless this is a significant reduction of the buffer size needed.

In this implementation each work-item computes independently the geometric flux of all neighbouring cells in order to calculate its flux corrector and save the flux corrected value. In figure 5.9 the needed neighbourhood of each cell is illustrated: to compute the flux corrected value for cell j (black) each work-item has to calculate the geometric propagated quantity in cells $(j-2)\ldots(j+2)$ (grey). In order to calculate all (uncorrected) propagated quantities, additionally the cells $j-3$ and $j+3$ have to be used. This is done in the `for`-loop serially and buffered in the `flux[]`-variable. However, the loop increases the computational workload of each kernel significantly, as each geometric flux has to be computed five times more often than in the serial implementation.

An additional overhead is caused by the calculation of both limited antifluxes, i.e. A_j and A_{j-1} from equation (4.58) in the variables `ea` and `eb` of listing 9, hence this step doubles the workload compared to the serial implementation. Although the alternative problem decomposition avoids the multiple calculation of geometric fluxes and antifluxes, its overall performance proofed to be lower due to the additional memory usage. This is because the additional computations allow less each work-item to compute the propagated quantity independently from all other work-items, and thus no serialisation e.g. between the geometric propagation and the anti-diffusion step is needed.

For each quantity a specialised kernel (similar to listing 9) is enqueued. To use the full capacity of a GPU, the kernel size must be chosen carefully. If the kernel's active working set is too big, only few work-items can run concurrently, as they must share the available memory per compute unit. Not only the neighbourhood (figure 5.9) of the propagated quantity is necessary, but additionally each work-item needs to access a similar neighbourhood of velocity, pressure, and different control variables. On the

other hand, if too many kernels are needed to compute the propagation, the increased number of enqueued kernels imposes an additional orchestrating overhead. The use of mid-weight kernels, computing all needed fluxes of a neighbourhood but propagating only one quantity per kernel, allows for an efficient usage of the underlying hardware (all processing elements in parallel) without overly increasing the orchestration overhead.

Branching Free Execution Flow

We have designed the components of OpenCL-SHASTA in a modular way and implemented different kernels and auxiliary functions. This allows shorter development cycles and ensures good validation of the algorithm. Additionally the kernels and auxiliary functions can be substituted even at run time leaving the choice of different antiflux functions, different source terms, and even different equations of state. In listing 9 the constant `diff` allows a fine controlled application of numerical viscosity. The value of the constant as well as the selection of more sophisticated or faster anti-diffusion routines are controlled by metaprogramming. As the kernels are compiled and loaded onto the GPU at run-time the choices do not infer additional branchings, which would slow the execution down. Nevertheless a free selection of the desired numerical hydrodynamics realisation is possible. Therefore a wide variety of calculations can be carried out by the suggested implementation without the need of complicated branching patterns within the computational relevant parts of the program.

Let us stress the importance of avoiding (unnecessary) branching in GPU programs: if the number of work-items per branch falls under the wavefront length, the execution stream is serialised. (As stated in [17] up to a loss of 30% of the execution speed.) To avoid branchings we have designed specialised kernels for all quantities in OpenCL-SHASTA. The kernels vary according to their propagated quantity, source terms and propagation direction. For example the kernels responsible for the momenta parallel to the propagation direction have an additional source term, whereas momenta perpendicular to the propagation direction are transported source free. Since complicated indexing methods not only imply branchings but often a very inefficient memory access, we use a number of halo cells to avoid it. Therefore the cells at the border of the grid in the finite difference scheme are handled equivalently to the inner cells, whilst special correction steps assure the desired behaviour by controlling the data in the halo cells. The kernel operates only within the boundaries of the inner grid, limited by grid size (GS) and halo size (HS). For stability reasons the code has been implemented based on a half-step method. The different half steps are implemented without a branching control structure, instead additional kernels change underlying control variables like the Courant-Friedrich-Lewy number. We use a special double buffering scheme to spare expensive memory copies. Instead of copying onto different grids, we have designed to each kernel call an adjunct kernel call, which reverses the used buffers. The adjunct kernel calls are orchestrated in

the launcher method (listing 10) of the GPU-class.

All different governing equations have been implemented in different kernels in order to stay clear of any branches within the execution flow. The orchestration of this set of 30 different kernel calls, is done by the execution method on the host (listing 10). Accordingly, the GPU can start the computation, while further kernels are still enqueued into the command queue. The control flow is almost completely managed by the C++ methods at the host side.

Memory Aware 3-D Calculation

Due to the dimension split scheme each operation is executed only in one dimension at a time. This reduces the amount of registers needed for each kernel drastically, as only 1-D differential stencils (figure 5.9) have to be calculated. This reduces the needed memory size to a third compared to a full 3-D stencil implementation. The propagation direction follows a fixed permutation of the three axes.[2] After the initial copy of all needed quantities to the private memory in local scope, the kernel (listing 9) does not need any access to global memory for its calculations. Contrary to the classical FORTRAN implementation no differentials have to be stored in extra data structures. Intermediate results, like the geometric flux, anti-diffusion, and flux-limiter can be held in registers. The functions yielding the necessary geometric factors (`qpt()` and `qmt()`) and the antiflux (`antiflux()`) are inlined functions. They are calculated exactly when needed (figure 5.7) and need not to be calculated in advance. (The former mentioned alternative problem decomposition (section 5.1.4) needs additional global memory for the geometric fluxes, antifluxes, and flux limiters.)

According to the permutation scheme only the propagation speed $v_{\parallel}$ parallel to the actual propagation direction is calculated beforehand, which limits the global memory usage to only one field for the propagation speed. This approach underlines again the *recompute instead of memory lookup* paradigm which holds for various applications on many-core architectures [35]. The global memory footprint is determined by the seven quantities residing on a 200^3 cell grid. As only single precision is needed to represent these quantities, the total memory consumption has been further reduced: including the double buffering memory scheme, the total consumption is less than 500 MB. This allows to run the code even on commodity GPUs. By limiting the working set to a minimum and concentrating on recompute instead of expensive memory lookups, the remaining memory can be used to increase the grid size, enhance precision or to hold more complicated (tabled) equations of state, or even to enhance SHASTA with the necessary tables to calculate viscous hydrodynamics [46, 49].

[2] We found this approach more stable than a Monte Carlo approach.

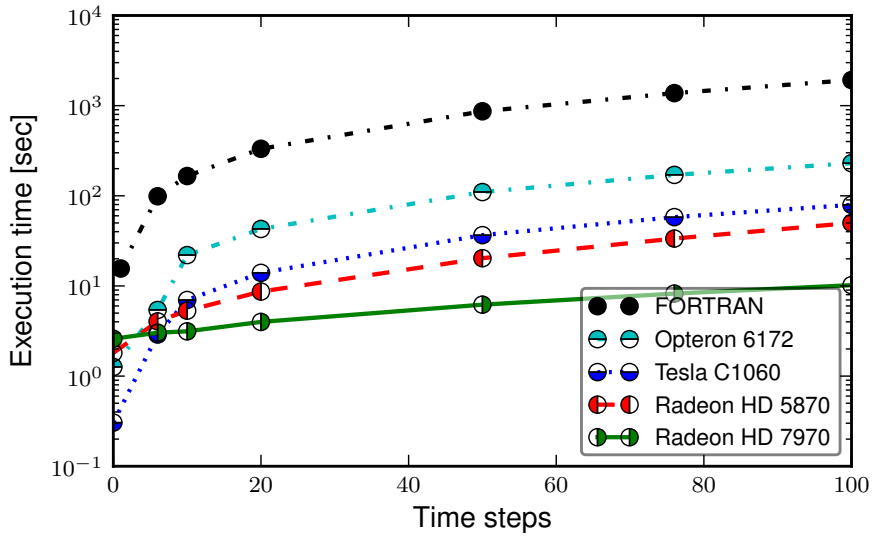


Figure 5.10.: Total execution time for the expanding ball of $r = 2$ fm with constant energy density. The exact same OpenCL code is used on all devices. For comparison the execution time of the FORTRAN implementation is shown too [52, 57].

5.2. Results

The OpenCL version has been tested on NVIDIA Tesla C1060 GPUs, AMD Radeon HD 5870 GPUs, AMD Opteron 6172 processors (24 core), and on AMD Radeon HD 7970 GPUs. The most extensive testing was done on the LOEWE-CSC with the AMD Radeon HD 5870 GPU.

Test cases included classic test problems, spheres of different metrics, and realistic initial conditions generated by UrQMD. For realistic cases the physics simulation time is on the order of 10 – 20 fm/c which transforms to 200 time steps (equalling to 16 fm/c in the current setup).

The measured accelerations depend slightly on the geometry of the input. The best acceleration is achieved for the realistic UrQMD input files. Here the overall computing time for a physics running time of $t = 8$ fm/c for an Au+Au collision is reduced to less than 30 seconds on the AMD Radeon HD 5870 GPU and less than 10 seconds on the AMD Radeon HD 7970 GPU. Compared to the classic FORTRAN implementation [52, 57] which needs 1 hour and 15 minutes, we find an acceleration of more than a factor of 160 on the AMD Radeon HD 5870 and more than a factor of 460 on the AMD Radeon HD 7970.

In figure 5.10 and figure 5.11 the total execution time for two different initial geometries is depicted. Figure 5.10 shows the total computation time for the spherical setup (*ball*, $\|\cdot\|_2$) of radius 2 fm on different devices. Figure 5.11 shows the comparison between the

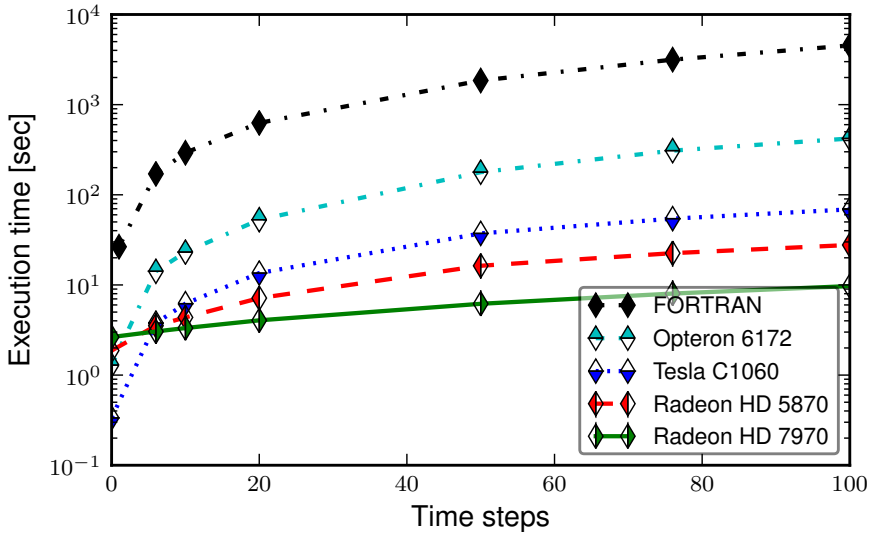


Figure 5.11.: Total execution time for an Au+Au collision with $\sqrt{s_{NN}} = 200$ GeV. The exact same OpenCL code is used on all devices. For comparison the execution time of the FORTRAN implementation is shown too [52, 57].

FORTRAN and the OpenCL implementation on CPU/GPUs for the UrQMD initial configuration of an Au+Au collision at $\sqrt{s_{NN}} = 200$ GeV with impact parameter b = 7 fm.

Figures 5.12 and 5.13 summarise the increasing speed-ups of the OpenCL-SHASTA execution on different devices. The difference between the geometries is caused by the better distribution of computations to work-items and finally processing elements. In the spherical symmetric case (denoted as *ball*) the initial geometry is concentrated in the centre of the grid. Due to the domain decomposition scheme, mapping grid cells to work-items, at the beginning all the workload is concentrated on few work-items. During the execution more cells are filled with non-zero values. As each quantity in each cell is computed separately in a work-item, a sparse grid is not very efficient, while a full grid makes best use of all the processing elements of a GPU.

Even without a GPU at hand the OpenCL implementation provides a significant speed-up on CPUs. As shown in figure 5.14, the code scales well up to 24 cores with a parallel efficiency of $E = \frac{20}{24} = 0.8\bar{3}$. However, the OpenCL implementation for one core is slower than the classic serial FORTRAN implementation. Hence the total acceleration for 100 time steps on the AMD Opteron 6172 processor (24 cores), is by a factor of ten compared to the classical implementation (single-core). Additionally, it is notable that the classical implementation is simply not executable in parallel. The memory consumption of the

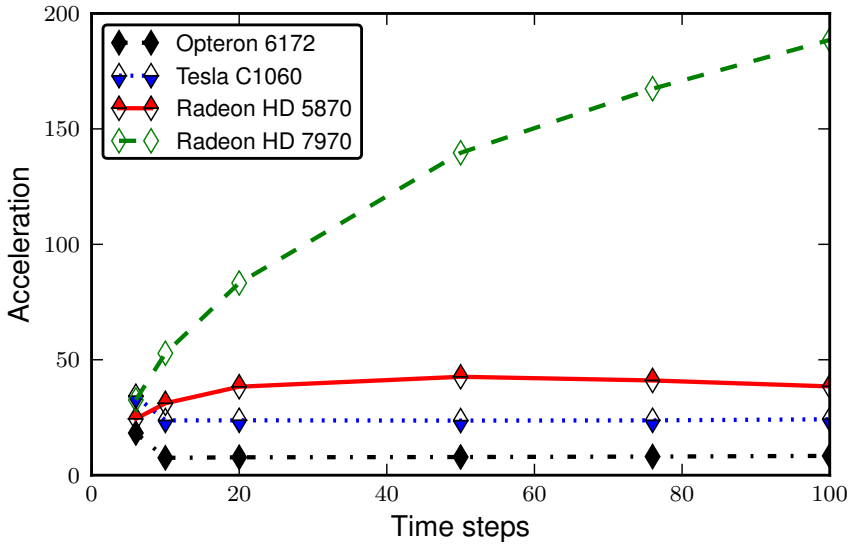


Figure 5.12.: The acceleration of the execution speed on different devices for the expanding ball of $r = 2$ fm with constant energy density. Here again the exact same code is executed on CPU and GPU.

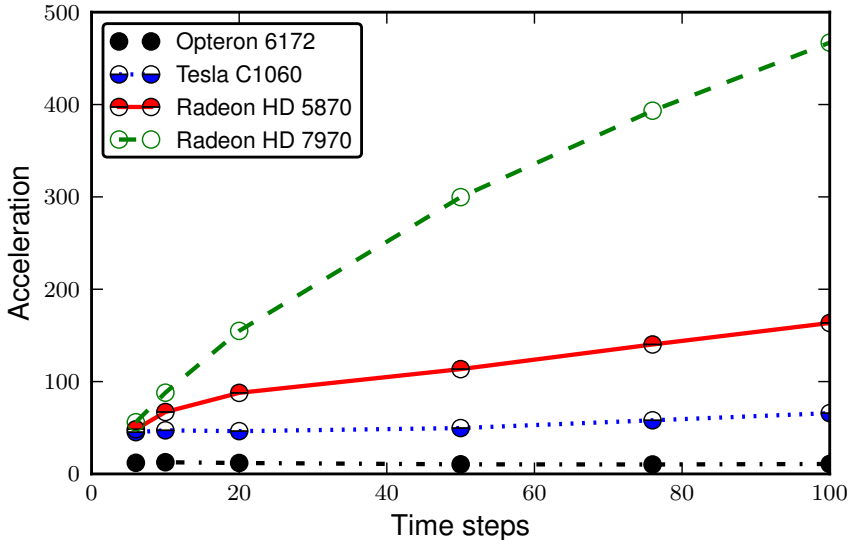
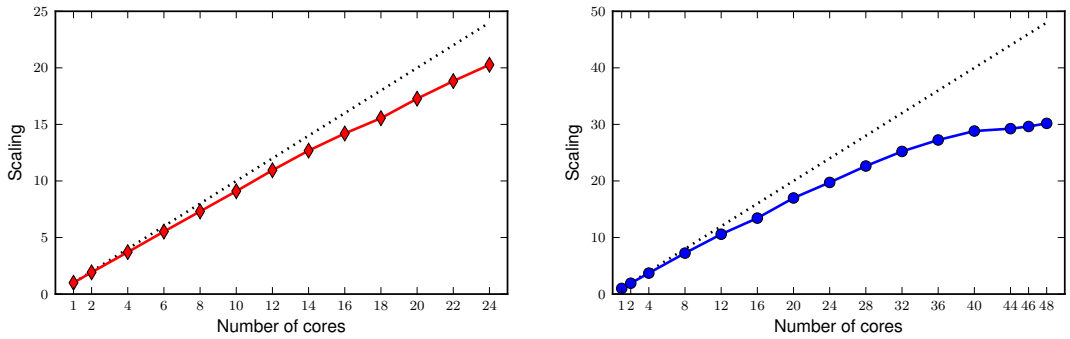


Figure 5.13.: The acceleration of the execution speed on different devices for an Au+Au collision with $\sqrt{s_{NN}} = 200$ GeV. Here again the exact same code is executed on CPU and GPU.



(a) Scaling on a 24 core CPU

(b) Scaling on a 48 core CPU

Figure 5.14.: The scaling of OpenCL-SHASTA on the Opteron 6172 processor with 24 and 48 cores. The scaling is measured for the expansion of a spherical symmetric system for ten time steps.

standard implementation is more than five times higher than the consumption of the OpenCL implementation, which limits the number of parallel instances significantly.

Over the above the exact same OpenCL implementation is executed on GPUs and CPUs. Though, the program design allows to choose appropriate kernels and environment variables for their execution, depending on the provided architecture (see section 5.1.4 and 5.1.4), enabling further speed-ups on CPUs.

In figure 5.15 the average time consumption for the computation of one time step with different initial geometries on the AMD Radeon HD 5870 is shown. We observe the initial cost of the memory transfer to the GPU, which is compensated after a small number of time steps. Nevertheless time consumption increases again later, especially for the spherical symmetric initial geometry. One observes the impact of the complex antiflux function on the average execution time. Without the complex antiflux no increase can be observed and the acceleration, due filling of the grid, takes fully place.

The latter increase of time consumption is dominant for the ball geometry case. In figure 5.16 only a slight increase of the execution time for the Au+Au collision can be observed on all devices. However, for the expansion of the ball this effect is strongest on the AMD Radeon HD 5870 and the NVIDIA C1060, as shown in figure 5.17.

In figure 5.18 the average time consumption of the `antiflux` function for the ball geometry is isolated. In SHASTA the antiflux is corrected by a flux limiter. This flux limiter is calculated by a search of maxima and minima of the surrounding cells and fluxes

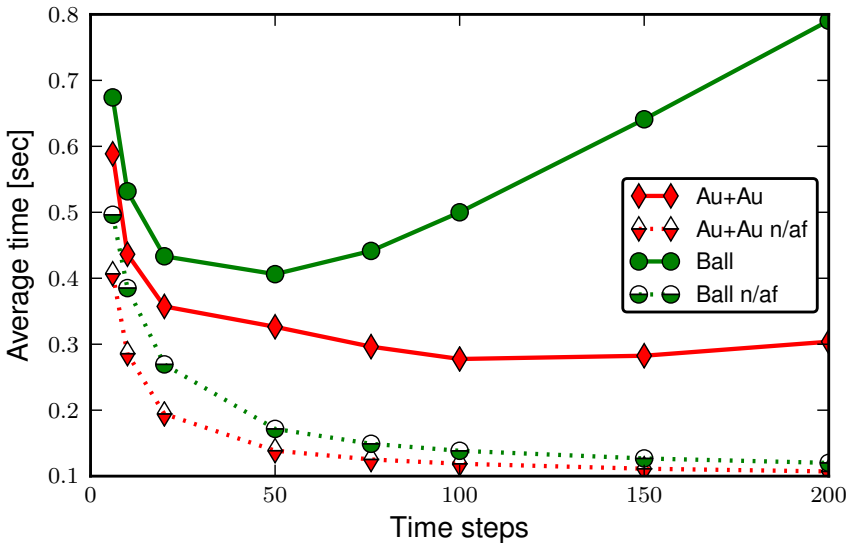


Figure 5.15.: Average time consumption for one time step in the expansion of a spherical symmetric system (*ball*) and for an Au+Au collision on the AMD Radeon HD 5870 GPU. The *n/af* curves show calculations without antiflux.

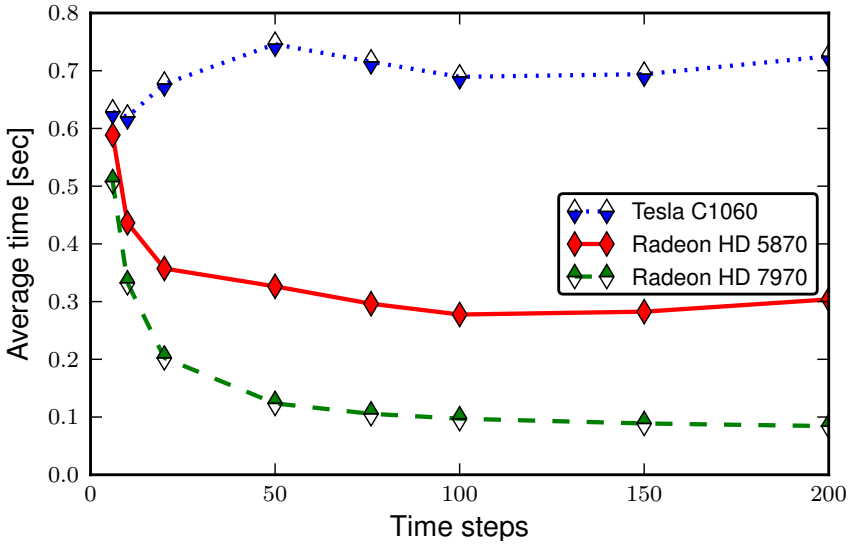


Figure 5.16.: Average time consumption for one time step in the expansion of an Au+Au collision with $\sqrt{s_{NN}} = 200$ GeV measured different GPUs.

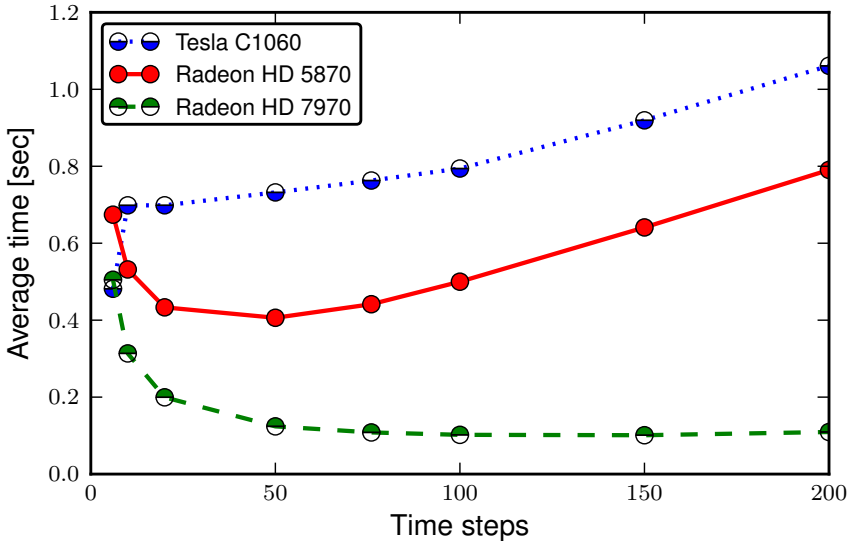


Figure 5.17.: Average time consumption for one time step in the expansion of a ball of $r = 2$ fm with constant energy density measured on different GPUs.

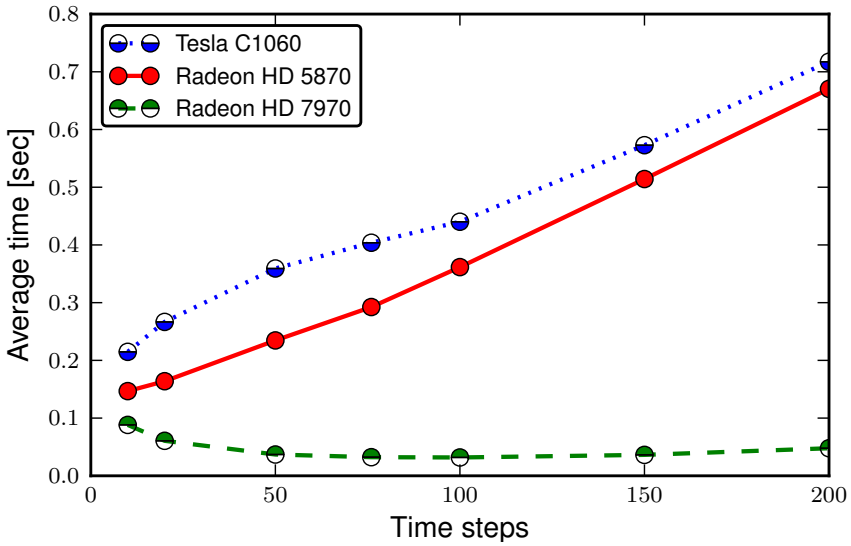


Figure 5.18.: Average time consumption of the pure antiflux calculation per time step in the expansion of a spherical symmetric system on different devices.

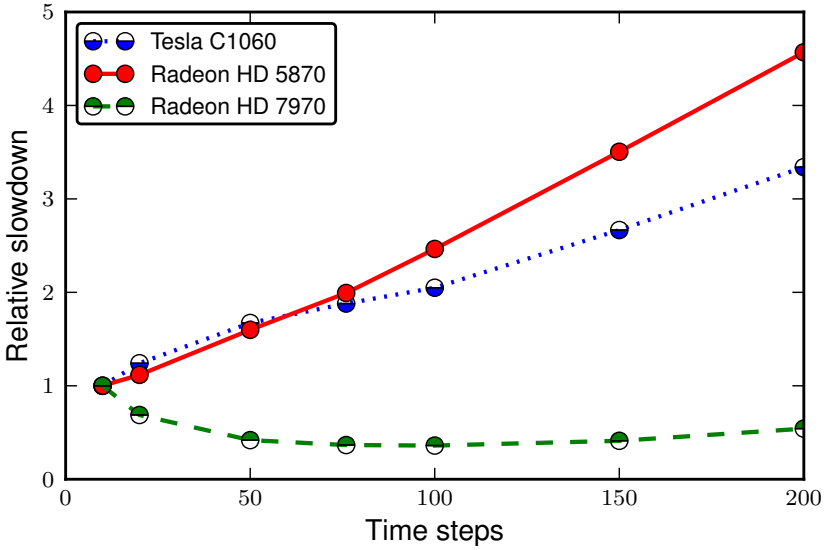


Figure 5.19.: Normalised average time consumption of the pure antiflux computation per time step in the expansion of a spherical symmetric system. The average time is normalised to the time consumption for ten time steps.

Name	Tesla C1060	Radeon HD 5870	Radeon HD 7970
Compute Units	30	20	32
Processing Elements	240	1600	2048
Peak Performance [GFLOP/s]	933	2720	3789
Wavefront length	32	64	16

Table 5.1.: Parameters of the different GPUs in the experimental setup.

towards this cells (see equation (4.58)). In this calculation branching is inherent and takes also place within different wavefronts. Therefore all branches within these wavefronts are calculated serially by the device and the correct results are gained by masking the wrong branches out. Hence the execution time is increased significantly, when the flux limiter is not uniform.

In figure 5.19 the normalised time consumption of the isolated `antiflux` function is shown. The average time consumption is normalised to the average time consumption for ten time steps, to compensate for the overall performance differences of the used GPUs. One can observe the different slowdowns for the used devices, which correspond to the wavefront sizes of the GPUs. The strongest slowdown of the antiflux calculation is measured on the AMD Radeon HD 5870 with its wavefront size of 64. The NVIDIA C1060 has a wavefront size of 32 and therefore the slowdown is smaller.

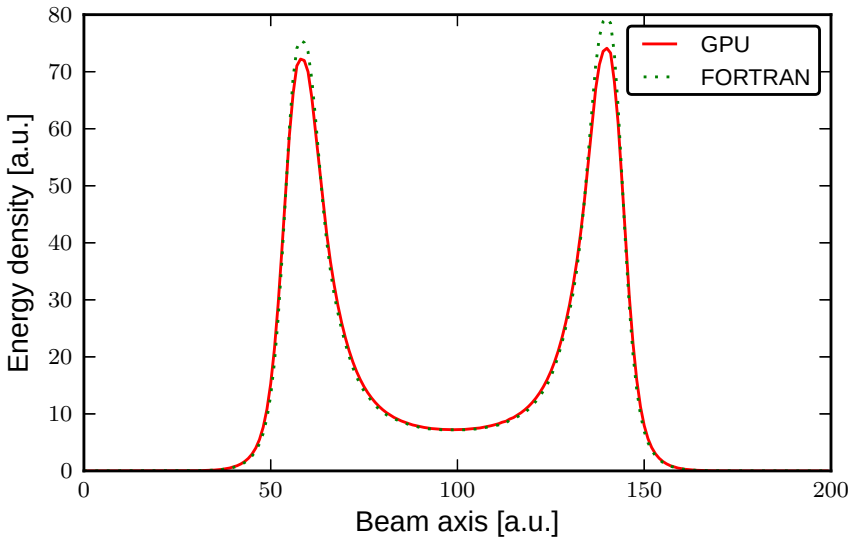


Figure 5.20.: Comparison between FORTRAN and OpenCL propagation for realistic initial conditions of an Au+Au collision with $\sqrt{s_{NN}} = 200$ GeV and impact parameter $b = 7$ fm at $t = 8$ fm/c provided by UrQMD. (The asymmetry present in both implementations is due to fluctuations in the initial state.)

The slowdown effect is compensated almost completely on the AMD Radeon HD 7970 which provides a wavefront size of 16 and a more modern scheduler, see table 5.1.

Finally figure 5.20 shows the direct comparison between the present *single precision* implementation (full line) and the standard FORTRAN *double precision* implementation (dotted line). For the realistic initial setup of a $\sqrt{s_{NN}} = 200$ GeV Au+Au collision provided by UrQMD we find only minor differences between both implementations. This differences originate in the different ordering of the calculation of the thermodynamic quantities and not in the single precision usage. The serial C++ implementation allows for a simple switching between single and double precision usage. The single precision and corresponding double precision calculation for a $\sqrt{s_{NN}} = 200$ GeV Au+Au collision differ only negligibly.

5.3. Summary and Outlook

On current hardware, like the AMD Radeon HD 5870, or the AMD Radeon HD 7970 GPU, double precision calculations come with a slowdown of a factor of five or four respectively[3]. Additionally a full double precision approach doubles the memory consumption. We conclude: a single precision implementation allows for fast calculations, as well as the enhancements of the underlying physics model, e.g. by adding calculations of viscosity [49, 46]. If numerical instabilities occur, e.g. in the calculation of the Lorentz-boost γ we suggest the usage of mixed precision implementations.

The OpenCL-SHASTA has been designed to work on commodity GPUs, nowadays present in almost every computer. The OpenCL implementation allows for the usage of GPUs, accelerators, as well as classical multi-core processors. Therefore OpenCL-SHASTA can be included in bigger frameworks, that need to be executed on a variety of different architectures. Due to the tremendous speed-up it resolves the problem of a computationally demanding hydrodynamic phase in hybrid models, like UrQMD, and allows for better statistics, stability analyses [24], and unprecedented event-by-event simulations.

[3] A special setup is needed to use double precision as well as the increased memory consumption. These setups are highly hardware dependent and future hardware may be more suited to double precision calculations.

6. Conclusion

The redesign of physics frameworks, like UrQMD, not only provides better execution time and statistics, but—under the light of hardware development—poses the only possibility to use these frameworks in the future. The lessons that can be learned do not only concern the redesign of software. New simulation frameworks should not only include the current state of physics, but also the best practices of software design and the knowledge of the underlying hardware. Firstly research groups should agree upon use-cases, design criteria, and tests before the actual coding begins. Thus, the whole process of redesigning a whole framework can be evaded at all.

Under the light of the complete redesign of UrQMD the possibilities of new quantum mechanical algorithms were tested. The prospective usage of a direct massively parallel calculation of Feynman-Path-Integrals is of high interest. It was explored by using a rapid-prototyped parallel implementation. The usage of Python as glue code with OpenCL for the parallel execution allowed for a rapid testing of new physics ideas. In addition to the test of the predictive power the usability on modern hardware was demonstrated.

The UrQMD framework was analysed and profiled. The stability of the simulation towards variations of the experimental parameters was verified by a massively parallel execution, applying metaprogramming techniques. Following the consequences of Amdahl's law, the most time consuming phase of the UrQMD hybrid model, the hydrodynamic evolution, was redesigned entirely in order to harvest the benefits of massively parallelisation on modern many-core architectures.

Starting with the mathematical description of the underlying physics, independent variables, due to the superposition principle, were separated. In the subsequent analysis of the SHASTA the dependency of additional intermediate variables was analysed which opened further ways of parallelisation. As the extend and benefit of parallelisation depends strongly on the underlying hardware, not all possible parallel calculations were also implemented in parallel. A massively parallel branching on current GPUs is not beneficial at all. Therefore, it was evaded in the final implementation. In contrast to classic sequential programming techniques, the best approach in the parallel implementation was to calculate all quantities needed for further computation directly and almost never rely on a memory look-up.

Explicit caching of complex calculations, e.g. saving differential quotients in extra arrays, had been beneficial in former architectures. Modern hardware architectures, however, offer hierarchical cache structures and have a comparatively high latency for a

non-cached memory access. Therefore, recalculations of even complex functions come at a much lower cost than memory look-ups.

The presented parallel implementation of SHASTA is explicitly optimised for many-core architectures, such as modern GPUs. However, the usage of an vendor independent language, namely OpenCL, allows for the application to run also on multi-core CPUs with a significant speedup. By an acceleration of up to a factor of 160 on the LOEWE-CSC and up to a factor of 460 on additional test systems, a completely new range of event-by-event analysis of UrQMD-simulations in a hybrid-hydro approach can now be executed. Additionally the stability, which has been proven for the non hydro-hybrid execution, can now be tested.

The optimised OpenCL-SHASTA is re-integrated in the UrQMD framework for immediate event-by-event analyses. Furthermore it is devised as an integral part of the *Dynamical Description of Heavy Ion Reactions at FAIR* project, which works on a novel transport model for the description of the impending experiments at the FAIR facility. Influenced by the analyses of UrQMD provided in this thesis it aims at a full object oriented design and modular approach.

However, the methods shown in the previous chapters are applicable for other heavy ion simulation models too: OpenCL-SHASTA has been chosen to provide the fundamental hydrodynamics part of the *Computational Jet Tomography* (CJET) project [55, 56] at the *Oak Ridge Leadership Computing Facility* in the *Oak Ridge National Lab.*

Because of its modular design it is easily integrated in the CJET project. Due to its formulation in OpenCL it could be executed directly on TITAN [2], the world's most powerful super computer [3].

A. Kernel Performances

Kernel Name	FLOPs	Time HD 5870 [ms]	Time HD 7970 [ms]
`Generic_X`	256	2.8	1.7
`Impulse_X`	268	5.2	1.9
`Energy_X`	271	5.2	2.0
`Generic_Y`	268	2.8	1.7
`Impulse_Y`	280	5.2	1.9
`Energy_Y`	283	5.2	2.0
`Generic_Z`	268	4.0	2.8
`Impulse_Z`	280	6.2	3.8
`Energy_Z`	283	5.8	3.8

Table A.1.: Floating point operations and initial execution time of the propagation kernels. The execution time is profiled on the targeted platforms AMD Radeon HD 5870 and HD 7970.

Kernel Name	Performance on Radeon HD 5870	Performance on Radeon HD 7970
`Generic_X`	740 GFLOP/s	1189 GFLOP/s
`Impulse_X`	410 GFLOP/s	1140 GFLOP/s
`Energy_X`	420 GFLOP/s	1080 GFLOP/s
`Generic_Y`	776 GFLOP/s	1228 GFLOP/s
`Impulse_Y`	432 GFLOP/s	1195 GFLOP/s
`Energy_Y`	438 GFLOP/s	1122 GFLOP/s
`Generic_Z`	540 GFLOP/s	761 GFLOP/s
`Impulse_Z`	363 GFLOP/s	594 GFLOP/s
`Energy_Z`	389 GFLOP/s	595 GFLOP/s

Table A.2.: Approximate (initial) performance of the profiled propagation kernels.

B. Listings

B.1. SHASTA Implementation Details

Listing 7: Within the `readeos3()`-routine the ambiguity between EoS 4 and EoS 5 is resolved.

```
    subroutine readeos3()
    real*8 t,mu,e,p,n,s,mstar,lam,mus
    real*8 ptab(0:2000,0:400),ttab(0:2000,0:400),lamtab(0:200,0:239)
    real*8 mutab(0:2000,0:400),stab(0:2000,0:400),msttab(0:2000,0:400)
    real*8 ptab2(0:200,0:200),ttab2(0:200,0:200)
    real*8 mutab2(0:200,0:200),stab2(0:200,0:200),msttab2(0:200,0:200)
    real*8 mustab(0:2000,0:400),mustab2(0:200,0:200)
    real*8 cstab(0:2000,0:400),cstab2(0:200,0:200)
    real*8 ptab3(0:200,0:200),ttab3(0:200,0:200)
    real*8 mutab3(0:200,0:200),stab3(0:200,0:200),msttab3(0:200,0:200)
    real*8 mustab3(0:200,0:200),cstab3(0:200,0:200)
    integer in, ie, j, eos

    common /eqofst/ eos,stabil,anti
    common /eos/ ptab,ttab,mutab,stab,lamtab,ptab2,ttab2,mutab2,stab2,
    mustab,mustab2,cstab,cstab2,cstab3,ptab3,ttab3,mutab3,stab3,mustab3

    open(unit=51, file='eosfiles/chiraleos.dat')
    open(unit=52, file='eosfiles/chiralsmall.dat')
    open(unit=56, file='eosfiles/chiralmini.dat')

    do 1109 in = 0,400,1
       j = 51
       do 1110 ie = 0,2000,1
          read(j,7777) t,mu,e,p,n,s,mstar,lam
          ptab(ie,in) = p
          ttab(ie,in) = t
          mutab(ie,in) = mu
          stab(ie,in) = s
          msttab(ie,in) = mstar
          cstab(ie,in) = lam
110    continue
109 continue
    close(51)
```

```
      do 1209 in = 0,200,1
         j = 52
         do 1210 ie = 0,200,1
            read(j,7777) t,mu,e,p,n,s,mstar,lam
            ptab2(ie,in) = p
            ttab2(ie,in) = t
            mutab2(ie,in) = mu
            stab2(ie,in) = s
            msttab2(ie,in) = mstar
            cstab2(ie,in) = lam
210      continue
209   continue
      close(52)

      do 1509 in = 0,200,1
         j = 56
         do 1510 ie = 0,200,1
            read(j,7777) t,mu,e,p,n,s,mstar,lam
            ptab3(ie,in) = p
            ttab3(ie,in) = t
            mutab3(ie,in) = mu
            stab3(ie,in) = s
            msttab3(ie,in) = mstar
            cstab3(ie,in) = lam
510      continue
509   continue
      close(56)

      if(eos.eq.5) then
         open(unit=53, file='eosfiles/hadgas_eos.dat')
         open(unit=54, file='eosfiles/hg_eos_small.dat')
         open(unit=55, file='eosfiles/hg_eos_mini.dat')

         do 1669 in = 0,400,1
            j = 53
            do 1616 ie = 0,2000,1
               read(j,7777) t,mu,e,p,n,s,mus,lam
               ttab(ie,in) = t
               mutab(ie,in) = mu
               mustab(ie,in) = mus
616         continue
669      continue
         close(53)

         do 1769 in = 0,200,1
            j = 54
            do 1716 ie = 0,200,1
```

```
            read(j,7777) t,mu,e,p,n,s,mus,lam
            ttab2(ie,in) = t
            mutab2(ie,in) = mu
            mustab2(ie,in) = mus
716      continue
769   continue
      close(54)

      do 1869 in = 0,200,1
         j = 55
         do 1816 ie = 0,200,1
            read(j,7777) t,mu,e,p,n,s,mus,lam
            ttab3(ie,in) = t
            mutab3(ie,in) = mu
            mustab3(ie,in) = mus
816      continue
869   continue
      close(55)
   endif
   return

777 format(2(1x,f8.3),6(1x,e15.7))

   end
```

B.2. C++-SHASTA Implementation Details

Listing 8: The abstract `EqofState`-class avoids branching patterns within the thermodynamical functions.

```
class EqofState{
  protected:
  typedef float BIGSIZE[DOUBLE_EOS_SIZE][BIG_EOS_SIZE];
  typedef float SMALLSIZE[SMALL_EOS_SIZE][SMALL_EOS_SIZE];
  BIGSIZE ttab, mutab, ptab, stab, mustab, cstab;
  SMALLSIZE ttab2, mutab2, ptab2, stab2, mustab2, cstab2;
  SMALLSIZE ttab3, mutab3, ptab3, stab3, mustab3, cstab3;
  float lamtab[240][201];
  public:
  EqofState(){
    for(int j = 0; j < BIG_EOS_SIZE; ++j)
    for (int i = 0; i < DOUBLE_EOS_SIZE; ++i) {
      ttab[i][j] = 0.f;
      mutab[i][j] = 0.f;
      ptab[i][j] = 0.f;
      stab[i][j] = 0.f;
      mustab[i][j] = 0.f;
      cstab[i][j] = 0.f;
    }
    for(int j = 0; j < SMALL_EOS_SIZE; ++j)
    for (int i = 0; i < SMALL_EOS_SIZE; ++i) {
      ttab2[i][j] = 0.f;
      mutab2[i][j] = 0.f;
      ptab2[i][j] = 0.f;
      stab2[i][j] = 0.f;
      mustab2[i][j] = 0.f;
      cstab2[i][j] = 0.f;
    }
    for(int j = 0; j < SMALL_EOS_SIZE; ++j)
    for (int i = 0; i < SMALL_EOS_SIZE; ++i) {
      ttab3[i][j] = 0.f;
      mutab3[i][j] = 0.f;
      ptab3[i][j] = 0.f;
      stab3[i][j] = 0.f;
      mustab3[i][j] = 0.f;
      cstab3[i][j] = 0.f;
    }
    for(int j = 0; j < 201; ++j)
    for (int i = 0; i < 240; ++i)
    lamtab[i][j] = 0.f;
  };
  virtual float chem(float e, float n)=0;
  virtual float schem(float e, float n)=0 ;
```

```
    virtual float temp(float e, float n)=0 ;
    virtual float entro(float e, float n)=0 ;
    virtual float lambda(float e, float n)=0 ;
    virtual float press(float e, float n)=0 ;
};
```

B.3. OpenCL-SHASTA Implementation Details

Listing 9: A generic kernel propagating source free quantities in x-direction.

```
__kernel void generic_X( __global float* In, __global float* Out,
                         __global float* v, __global float* outcfl)
uint idx = get_global_id(0);
uint idy = get_global_id(1);
uint idz = get_global_id(2);
uint myid = idx + idy * DY + idz * DZ;
if ( (idx >= HS) && (idy >= HS) && (idz >= HS) && (idx < GS - HS)
      && (idy < GS - HS) && (idz < GS - HS) ){
  const float cfl = *outcfl ;
  const float diff = 1.0f;
  __private float basis[7] = {In[myid-3], In[myid-2], In[myid-1],
                              In[myid], In[myid+1], In[myid+2],
                              In[myid+3]};
  __private float flux[5];
  __private float velocity[7]= {v[myid-3] ,v[myid-2], v[myid-1],
                                v[myid], v[myid+1], v[myid+2],
                                v[myid+3]};
  for (short i = 0; i < 5; ++i){
    const float qpt = qp( velocity, i+1, cfl );
    const float qmt = qm( velocity, i+1, cfl );
    flux[i] = 0.5f * qpt*qpt * (basis[i+2]-basis[i+1])
            - 0.5f * qmt*qmt * (basis[i+1]-basis[i]) + (qpt+qmt)
            * basis[i+1];
  }
  const float ea = antiflux(flux, basis, 0, diff);
  const float eb = antiflux(flux, basis, -1, diff);
  Out[myid] = flux[2] - ea + eb;
} else Out[myid] = In[myid];
```

Listing 10: A part of the enqueueing routine.

```
const int branch[] = {0,1,2, 0,2,1, 2,0,1, 2,1,0, 1,2,0, 1,0,2};
for (int round = 0; round < steps; ++round) {
  i = branch[round % 18];
  if ( i == 0 ) {
    queue.enqueueNDRangeKernel(untangT, cl::NullRange,
                               cl::NDRange(GS,GS,GS), cl::NullRange,
                               NULL, NULL);
    queue.enqueueNDRangeKernel(half_cfl, cl::NullRange,
                               cl::NDRange(1,0,0), cl::NullRange,
                               NULL, NULL);
    queue.enqueueNDRangeKernel(propX_e, cl::NullRange,
                               cl::NDRange(GS,GS,GS), cl::NullRange,
                               NULL, NULL);
    queue.enqueueNDRangeKernel(prop_parallel_mx, cl::NullRange,
                               cl::NDRange(GS,GS,GS), cl::NullRange,
                               NULL, NULL);
    queue.enqueueNDRangeKernel(propX_my, cl::NullRange,
                               cl::NDRange(GS,GS,GS), cl::NullRange,
                               NULL, NULL);
                             }
```

B.4. Worldline Implementation Details

Listing 11: The loop-methods in the Python prototype recursively generate the midpoints of loop-archs and directly compute the calculation of a predicate (e.g. hits object) on the loop-points.

```
def _loop(depth, start=zero, end=None, variance=1.0,
                         gaussdist=random.gauss, sample=[],
                         observer=None):
  """
  internal function
  """
  if not end:
      end = start

  if depth == 0:
      return [ f(observer,start) for f in sample]

  mid = gauss(midpoint(start,end),variance,gaussdist=gaussdist)
  if not observer:
      observer = midpoint(mid,start)

  firsthalf  = _loop(depth-1, start, mid, variance=0.5*variance,
                             gaussdist=gaussdist, sample=sample,
                             observer=observer)
  secondhalf = _loop(depth-1, mid,   end, variance=0.5*variance,
                             gaussdist=gaussdist, sample=sample,
                             observer=observer)
  return [ left.union(right) for (right,left) in
          zip(firsthalf,secondhalf) ]
```

Listing 12: The analogous code in Scheme uses explicit typing of variables and operators, allowing the *Gambit-C* compiler a more aggressive optimisation than the Python interpreter.

```
(define (loop depth start
              #!key (end '()) (s 1.0) (length 1.0) (sample '()))
  (declare (fixnum depth)
           (flonum length s))
  (define (walk depth start end s observer)
    (declare (fixnum depth)
             (flonum s))
    (if (fx= 0 depth)
        (map (lambda (f) (f observer start)) sample)
        (let* ((mid (random-gauss-vector (midpoint start end) s))
               (obs (if (null? observer) (midpoint start mid) observer))
               (firsthalf (walk (fx- depth 1) start mid
                                (fl* 1/sqrt2s)  obs))
               (secondhalf (walk (fx- depth 1) mid end
                                 (fl* 1/sqrt2 s) obs)))
          (map oset-union firsthalf secondhalf))))
  (map (lambda (oset) (oset-scale oset length))
       (walk depth start (if (null? end) start end) (fl* s length)
       '())))
```

Listing 13: The OpenCL code is reduced to a pointwise checking of the predicate (here loop hitting the parallel plates). The loops are generated in the host-code and subsequently uploaded to the GPU. Host and device can compute in parallel, i.e. during the (partial) energy computation on the device, the host is able to generate more loops, in order to provide better statistics.

```
float loop_ener(__global float2* Loop, float uplatey, float dplatey) {
  float sigma = Loop[0].x;
  bool cutA = false;
  bool cutB = false;
  for (int i = 1; i < LENGTH; ++i) {
    cutA = cutA || cut_uplate(Loop[i].y, uplatey);
    cutB = cutB || cut_dplate(Loop[i].y, dplatey);
  }
  float part = 0.f;
  if (cutA && cutB) {
    part = 1.f/(sigma*sigma*sigma*sigma);
  }
  return part;
}
```

Listing 14: The `loop_ener()` method calculates the scaling intervals used for the semi-analytic evaluation of the Feynman-integral. It is tailored to the specific plate-plate geometry. The usage of the special OpenCL vector type `float2` allows the GPU to make better usage of the underlying hardware within the processing elements. The function `energy()` is inlined and provides a scaling according to the distribution function of the loop length.

```
float loop_ener(__global float2* Loop, float uplatey, float dplatey){
  // first analyze upper plate
  // Intervals are of the form (0, inf) to (inf, inf)
  float min_upper, min_lower;
  if ( (uplatey >= 0.f) && (0.f >= dplatey) ) {
    min_upper = INF;
    min_lower = INF;
    for (int i = 1; i < LENGTH; ++i) {
      const float POINT = Loop[i].y;
      if (POINT < 0) min_lower = min(min_lower, dplatey / POINT);
      if (POINT > 0) min_upper = min(min_upper, uplatey / POINT);
    }
  }
  if (dplatey > 0.f){
    min_upper = INF;
    min_lower = 0;
    //barycentre is within hyperspace {x<=d}
    //=> Loop intersects allways D
    for (int i = 1; i < LENGTH; ++i) {
      const float POINT = Loop[i].y;
      if (POINT > 0) min_upper = min(min_upper, uplatey / POINT);
    }
  }
  if (uplatey < 0.f){
    min_upper = 0;
    //barycentre is within hyperspace {x>=u}
    //=> Loop intersects allways U
    min_lower = INF;
    for (int i = 1; i < LENGTH; ++i) {
      const float POINT = Loop[i].y;
      if (POINT < 0) min_lower = min(min_lower, dplatey / POINT);
    }
  }
  float a = max(min_lower, min_upper);
  return( energy(a) );
}
```

Listing 15: A kernel tailored to the parallel plate geometry. It is responsible for a tile of the whole geometry. It calculates the contribution of the current set of loops to all points within its domain.

```
__kernel void calc_energy(__global float2* Loops,
__global float* platey,  __global uint* rank, __global float* Forces){
  const int gid1 = get_global_id(0);
  const int gid2 = get_global_id(1);
  const float2 OF = (float2) (LLIMIT + PIXELS * L * (*rank % TILES),
  ULIMIT + PIXELS* L * (*rank / TILES));
  // offset for tile
  const float2 POS = (float2)(gid1 * L, gid2 * L) + OF;
  // global position within tile
  __private float myEnergie = 0.f;
  float uplatey = *platey - POS.y;
  float dplatey = (-1) * *platey - POS.y;
  const float dy = DX *(0.5f) ;//central differential quotient
  for (ulong run = 0; run < NUMLOOPS; ++run) {
    myEnergie +=  loop_ener(&(Loops[run*(LENGTH)]), uplatey+dy,
                            dplatey)
    - loop_ener(&(Loops[run*(LENGTH)]), uplatey - dy, dplatey);
  }
  Forces[gid1 + PIXELS * gid2] = myEnergie/(NUMLOOPS);
}
```

C. Summary

Summary

Ultrarelativistic Quantum Molecular Dynamics is a physics model to describe the transport, collision, scattering, and decay of nuclear particles. It allows to simulate complex processes, like particle collisions in the LHC and the upcoming FAIR experiments. The physics needed to describe this kind of reaction is multi-faceted. Simulation frameworks, such as the Frankfurt UrQMD framework use therefore different ideas, models and algorithms ranging from relativistic mechanics, realised by linear equation solvers for particle trajectories, up to quantum mechanical effects, implemented by Monte Carlo approaches depending on pseudo-random number generators.

The UrQMD framework has been in use for nearly 20 years since its first development. Over this years different approaches have been implemented, used to generate data and predictions, and thereafter refined respecting new experimental results or rejected due to clear falsification by the experiments.

In this period computing aspects, the design of code, and the efficiency of computation have been—at best—minor points of interest. The widely common maxim

> *"If the software is not fast enough, it is sufficient to wait only for the next generation of computers to make it work."*

has in many places delayed the use of modern hardware capabilities, because the software has just been considered a means to an end. As in many big software systems *code-entropy* has grown over years, making parts of the source inaccessible without a prior deep research. Changes within the code base can lead to unexpected side-effects, therefore parts of the source are declared as *terra prohibita* to newcomers. Comments advise researchers to leave parts of the source unaltered and therefore prohibit any major change or introduction of new ideas.

Nowadays an additional sorrow is due to the fact that the run time of the framework does not diminish any more with new hardware generations, contrary to the above cited maxim. The time of *spending Moore's dividend*[1] with *free lunches* is over. While Moore's law seems still unbroken, it is only unbroken in its original formulation: the number of transistors in integrated circuits doubles every two years. Though, the velocity or

[1] James Larus, Microsoft Research

frequency of operation of those integrated circuits does not increase significantly any more. At the current state, it is not yet due to physical constraints: current CPUs are produced with 22 nm structures but research has provided 3 nm transistors [41]. The clock rate of current CPUs ranging between 2 and 3 GHz is still significantly lower as the records (around 8 GHz) realised even with standard CPUs[2]. It is mostly due to economic reasons current development in computing hardware is mainly parallelism oriented.

Especially for scientific applications often a high order of parallelisation can be achieved due to the superposition principle. The need of computing power and the existence of highly parallel problems in physics has lead early to the usage of vector processors up to the adventurous use of GPUs, by transforming the physics algorithms to graphics handling and a back transforming after the computation on the GPUs, see e.g. [19].

In this thesis I show how modern design criteria and algorithm redesign is applied to physics frameworks. The redesign with a special emphasise on many-core architectures, allows for significant improvement of the execution speed. In order to harvest all possibilities of modern hardware, algorithms have to respect both: the design of the hardware and the nature of the physics behind the model. The first chapter describes the physics of UrQMD. As only a thorough understanding of the physics allows for major changes, the gained information about the structure of UrQMD allowed for a deep analysis of the model as such, which culminated in an unprecedented stability analysis of the whole model [24]. The computational approach and the results of the numerical analysis are described in chapter 3. By applying metaprogramming techniques, the UrQMD source could be modified, compiled and executed in a massively parallel setup. The resulting simulation data of all parallel UrQMD instances were hereafter gathered and analysed. Hence UrQMD could be proven of high stability to the uncertainty of experimental data, which is in light of the upcoming experiments at the FAIR facility of uttermost importance.

Chapter 2 describes an implementation of the Worldline Method. A numerical method for the calculation of quantum fluctuations, e.g. the Casimir effect. The Casimir effect is a quantum mechanical effect, which is e.g. responsible for an attracting force between two perfectly conducting plates in an absolute vacuum. This effect, once only a curiosity in theoretical physics, is today experimentally measured and an effect to be considered in the construction of nano mechanical devices. This poses a serious pressure onto the search of fast computation possibilities. The efficient simulation of quantum mechanical effects is of crucial importance to effective models. The computation of the Casimir effect has allowed to test a new method directly without interfering into the existing UrQMD framework. Due to rapid prototyping we could perform simulations with scalar photons and test the predictive power of this approach. Additionally different code bases have been used to investigate the possibility of a massively parallel calculation, which was then performed on the LOEWE-CSC.

[2]The record (dating back to 01.11.11) is held by Andre Yang with approximately 8.71 GHz.

The most time consuming part of UrQMD is a newly introduced relativistic hydrodynamic phase. Following Amdahl's law it is a natural choice to start here the parallelisation. Hence, chapter 4 explains the principles of hydrodynamics and the distinctive features of the relativistic hydrodynamics. The algorithm used to simulate the hydrodynamic evolution, SHASTA, is described in section 4.4. As the sequential form of SHASTA is successfully applied in various simulation frameworks for heavy ion collisions its possible parallelisation is analysed. In chapter 5 two different implementations of SHASTA are presented. The first one is a sequential implementation, which mimics the FORTRAN implementation currently used in UrQMD as closely as possible. However, the C++ implementation is a standalone simulation program following the object oriented design criteria, the structure of the program is presented in different UML diagrams. By applying a more concise design and evading unnecessary memory copies, the execution time could be reduced to the half of the FORTRAN version's execution time. The usage of memory could be reduced by 80% compared to the memory needed in the original version.

The second implementation concentrates fully on the usage of many-core architectures. Therefore its implementation is in OpenCL, which allows to harvest the compute power of different architectures by various vendors. The implementation is fully oriented on a massively parallel processing and deviates significantly from the classical implementation. Contrary to the sequential implementation, it follows the recalculate instead of memory look-up paradigm. By this means the execution speed could be accelerated up to a factor of 460 on GPUs. This allows now for extremely fast event-to-event simulations, which enable completely new approaches and unprecedented analyses.

Zusammenfassung

Das UrQMD Modell

Ultrarelativistische Quantenmolekulardynamik ist ein so genanntes effektives Modell zur Beschreibung von Transport, Kollision und Zerfall von Hadronen. Es dient zum Beispiel zur Simulation der komplexen Prozesse, die im LHC stattfinden und in den FAIR Experimenten erwartet werden. Zur Beschreibung dieser Prozesse werden verschiedene Bereiche der Physik herangezogen. Angefangen bei den Bewegungsgleichungen der relativistischen Physik, die mit linearen Gleichungssystemlösern berechnet werden, bis hin zu Modellen aus der Quantenphysik, die mit Zufallszahlengeneratoren im Monte Carlo Ansatz modelliert werden.

Das UrQMD System wird seit fast 20 Jahren ständig genutzt und weiterentwickelt. Immer wieder wurden in dieser Zeit neue Ansätze aus der theoretischen Physik getestet und, falls die Vorhersagen mit den experimentellen Daten übereinstimmten, weiter genutzt, oder falls die Vorhersagen den Daten widersprachen, verworfen.

Das Programm selbst war dabei als Mittel zum Zweck zu verstehen. Moderne Paradigmen zur Klarheit und Struktur des Programms wurde zwar nicht völlig ignoriert, standen aber auf der Prioritätenliste weit unten. Die Frage der Laufzeit oder Fragen zur Effizienz der Algorithmen wurden, wie häufig, durch die lange gültige Maxime:

> „Wenn das Programm nicht schnell genug läuft, muss man nur auf die nächste Prozessorgeneration warten."

beantwortet. Das Problem zu langer Laufzeit erledigte sich sozusagen von selbst. Wie üblich bei großen Softwaresystemen, ist die *Code-Entropie* seit Jahren ständig gewachsen. Teile des Quelltextes sind zu unzugänglich und gleichen mit ihren Nebeneffekten wahren Minenfeldern: jede Änderung in diesen Zeilen hat unvorhersehbare Konsequenzen im Gesamtablauf.

Die Entwicklung der Hardware in den letzten Jahren hat mittlerweile auch obige Maxime überholt. Zwar gilt das Moorsche Gesetz, siehe Abbildung C.1, heute immer noch, aber es gilt in seiner ursprünglichen Formulierung, die Anzahl der Transistoren betreffend, und nicht in der abgeleiteten Formulierung, die Taktrate betreffend. Die Rechenmöglichkeit heutiger Prozessoren ergibt sich nicht mehr durch die hohe Taktrate, sondern vor allem durch die Möglichkeit viele Rechnungen parallel durchzuführen.

Die Entwicklung der Prozessoren ist dabei noch nicht am Ende des physikalisch Möglichen angekommen. Moderne Prozessoren nutzen zur Zeit dieser Arbeit 22 nm Strukturkonstanten und werden mit etwa 2–3 GHz betrieben, erste Prototypen können im Labor bereits mit 3 nm Struktur hergestellt werden [41] und selbst „Heimprozessoren" werden mit bis zu 8 GHz betrieben[3]. Der momentane Stand stellt jedoch ein Optimum in ökono-

[3]Der momentane Rekord in der „Übertaktungszene" wird von A. Yang mit ca. 8,71 GHz gehalten.

mischen Sinne dar und es bleibt zu erwarten, dass sich der Trend zu mehr Parallelisierung auch noch die nächsten Jahre fortsetzt.

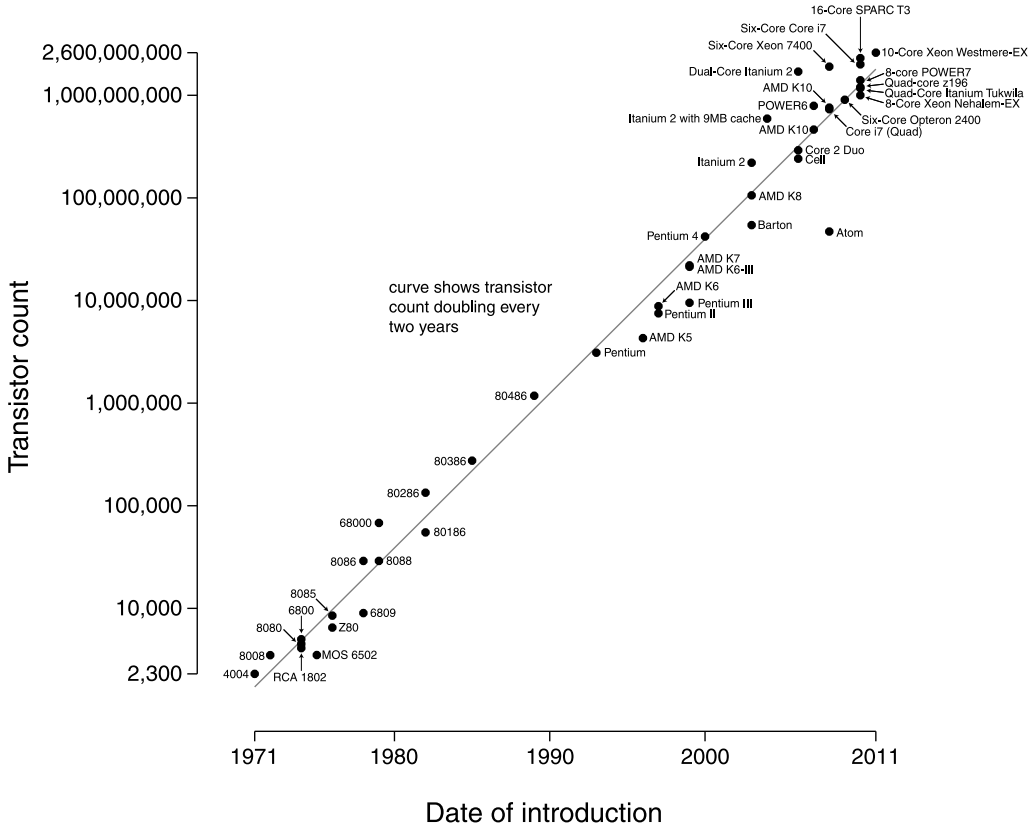


Abbildung C.1.: Das Moorsche Gesetz in seiner Originalfassung sagt eine Verdopplung der Transistoranzahl auf einer IC innerhalb zweier Jahre voraus. (Abbildung von [66].)

Gerade wissenschaftlichen Simulationen profitieren stark von der effizienten Nutzung moderner Hardware. Durch das Superpositionsprinzip sind viele Probleme der Naturwissenschaften in sich schon parallel gestellt. Nichtsdestotrotz ist die klassische Herangehensweise in wissenschaftlichen Programmen ungeeignet die Leistungsfähigkeit moderner Hardware zu nutzen. Gerade im Hinblick auf parallele Berechnungen hat sich eine besondere Möglichkeit durch die Nutzung von Grafikkarten ergeben. Diese Karten wurden in erster

Linie entwickelt um Computerspielern möglichst realistische visuelle Eindrücke darzustellen. Hierzu ist vor allem das massiv parallele Berechnen vieler Pixel notwendig. Die Pixel unterliegen dabei häufig identischer oder fast identischer Operationen, weshalb Grafikkarten gerade in diesem Gebiet mit enormer Rechenkraft hervortun. Zudem ist der Markt für Grafikkarten einem massiven Evolutions- und Preisdruck ausgesetzt, was die Möglichkeiten zur kostengünstigen parallelen Berechnung massiv befördert hat.

In dieser Arbeit zeige ich, wie ein Physik Framework (UrQMD) von einem Redesign unter Berücksichtigung moderner Ansätze aus Software Design und paralleler Algorithmen profitiert. Die hier aufgezeigten Methoden sind dabei nicht allein auf UrQMD anwendbar, sondern im wesentlichen für jeden wissenschaftlichen Code, der sich mit Transport, Zerfall und Kollision von Teilchen beschäftigt.

Die Physik in UrQMD

Gerade im Hinblick auf Parallelisierung finden sich mittlerweile einige Softwarepakete zur Molekulardynamik, die massiv parallel rechnen und sogar die Rechenkraft von Grafikkarten nutzen. Die Ansätze aus diesen Softwarepaketen lassen sich jedoch nicht ohne Weiteres auf UrQMD übertragen. Dies liegt vor allem daran, dass die zugrunde liegende Physik in der Molekulardynamik, wie sie in der theoretischen Chemie und den Materialwissenschaften genutzt wird, auf der Elektrodynamik basiert. Die, von vielen als *erfolgreichste* physikalische Theorie angesehene, Quantenelektrodynamik beschreibt das Verhalten von elektrisch geladenen Teilchen sowie elektrische und magnetische Felder. Prinzipiell genügt die Quantenelektrodynamik vollständig zur Beschreibung aller chemischen Prozesse. Für größere Systeme sind jedoch Simulationen auf der Basis der Quantenelektrodynamik aufgrund der kombinatorischen Komplexität zur Zeit nicht möglich. Zusätzlich folgen aus den räumlichen Dimensionen der betrachteten Systemen weitere Unterschiede. Während die Molekulardynamik Moleküle und Atome modelliert, muss UrQMD inneratomare Prozesse berücksichtigen. Hierzu muss UrQMD prinzipiell die Kräfte zwischen Quarks modellieren. Die zu Grunde liegende Quantenchromodynamik ist jedoch in wesentlichen Punkten verschieden von der Quantenelektrodynamik. Zum einen gibt es in der Chromodynamik mehr Austauschteilchen, die so genannten Gluonen, zum anderen zeigen diese farbgeladenen Teilchen eine wesentlich kompliziertere Dynamik. Resultierend aus diesen Unterschieden benötigt man auch im Experiment enorme Energiedichten, weshalb zusätzlich relativistische Effekte zu berücksichtigen sind. So wie in der Molekulardynamik werden auch in der Ultrarelativistischen Quantenmolekulardynamik nur sehr kleine Systeme direkt mit der Quantenelektrodynamik oder der Quantenchromodynamik simuliert. Die im Einsatz befindlichen effektiven Modelle weisen aber, da die zugrunde liegenden Kräfte so unterschiedlich sind, substantiell verschiedene Effekte auf, die zu vollkommen unterschiedlichen Algorithmen führen.

Die Berechnung des Casimir Effekts

Im Rahmen eines Forschungsaufenthalts an der Universität Southampton entstand ein Softwarepaket zur Berechnung des Casimir-Effekts auf Grafikkarten. Der Casimir-Effekt ist ein rein Quantenmechanischer Effekt und sollte unter Verwendung von *rapid prototyping* untersucht werden. In der einfachsten Konfiguration, zweier paralleler Platten im Vakuum, ergibt sich eine anziehende Kraft. In der Worldline-Methode erfolgt die Berechnung der resultierenden Kräfte durch geschlossene Schleifen im Vakuum, deren geometrische Eigenschaften die Gewichtung in einer Monte-Carlo Integration ergeben, vergleiche Abbildung C.2. Im Rahmen des Forschungsaufenthalts wurde ein speicher-

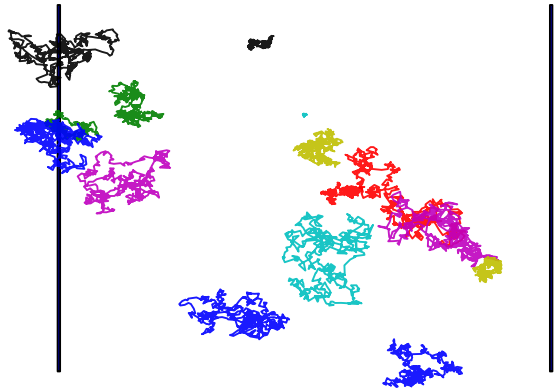
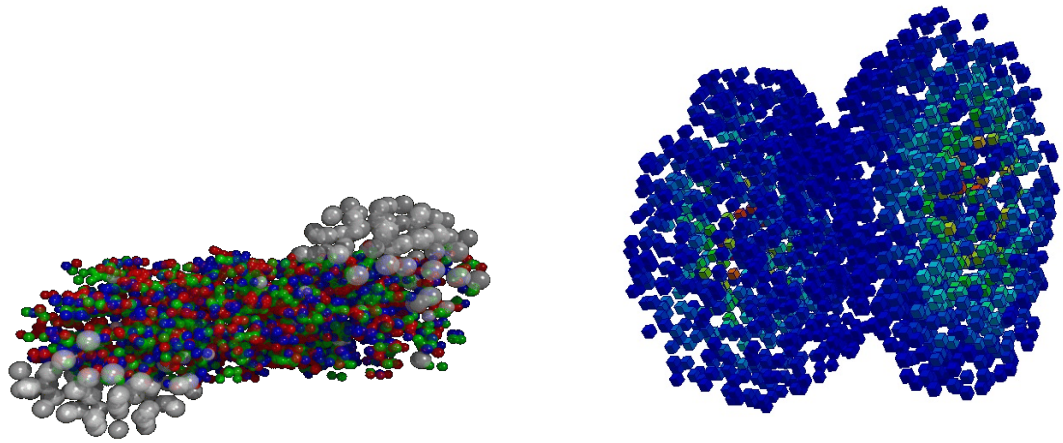


Abbildung C.2.: Geometrische Veranschaulichung der Simulation des Casimir-Effekts zwischen zwei parallelen perfekt leitenden Platten im Vakuum. Zur Simulation werden geschlossene Schleifen unterschiedlicher Größe (verschiedene Farben) durch Brownsche Bewegung modelliert. Die Schnitte zwischen den modellierten Platten (schwarz) und den geschlossenen Schleifen bestimmen die Stärke des Effekts.

sparsamer Algorithmus für die Worldline-Methode entwickelt. Dazu konnte mit Hilfe eines Python Prototyps sowohl eine generelle Scheme Implementierung, als auch eine spezialisierte OpenCL Implementierung für Grafikkarten entwickelt werden. Dieser Ansatz erlaubt die Analyse des physikalischen Modells und seiner zugehörigen Algorithmen gerade auch im Hinblick auf ihre Parallelisierbarkeit.



(a) Teilchen (Abbildung von [43]).

(b) Volumen Elemente.

Abbildung C.3.: Nach einer initialen Phase schaltet UrQMD von Teilchen-Freiheitsgraden auf Flüssigkeits-Freiheitsgrade um.

Die Modell-Stabilität von UrQMD

Ausgehend von der grundlegenden Analyse von UrQMD wurde unter Hilfe von Metaprogrammierung ein Framework geschaffen, das die zugrunde liegende Physik in UrQMD im Rahmen von experimentellen Daten validiert. Die hierbei erzeugten variierten *Versionen* von UrQMD konnten auf dem LOEWE-CSC massiv parallel genutzt werden. Die resultierenden Simulationsdaten konnten analysiert werden, um das Verhalten des Transportmodells unter Variation der Teilchendaten zu Untersuchen. Die hierbei angewandte Methode ist prinzipiell auf andere Transportmodelle ausdehnbar und erlaubt deren Validierung. Als Ergebnis konnte erstmals ein Transportmodell auf seine Stabilität untersucht werden. Das UrQMD Modell weist, selbst für ein Szenario bei dem alle Parameter einen korrelierten systematischen Fehler haben, ein lineares Fehlerverhalten auf, weshalb der maximale systematische Fehler auf weniger als 20% abgeschätzt werden kann.

Hydrodynamik in UrQMD

Eine der neuesten Erweiterungen von UrQMD stellt die Einbeziehung einer so genannten hydrodynamischen Phase dar. Nach einer initialen Kollisionsberechnung, wird das Verhalten von Atomkernen als Flüssigkeit modelliert, wie in Abbildung C.3 dargestellt. Gegenüber klassischer Hydrodynamik müssen hier aber relativistische Effekte berücksichtigt werden.

Dieses Modell ist, obgleich ausgezeichnet zur Beschreibung von Effekten wie des *elliptischen Flusses*, leider enorm zeitaufwendig. Mit Fluid Dynamik benötigt UrQMD bis zu

mehreren Stunden zur Berechnung eines einzigen *Events*; um statistische Fehler auszuschließen werden aber zehntausende Events benötigt. Durch genaue Analyse der zugrunde liegenden Physik der relativistischen Fluid Dynamik sowie des verwendeten sequentiellen Algorithmus, konnte ein massiv paralleler Algorithmus für Grafikkarten entwickelt werden.

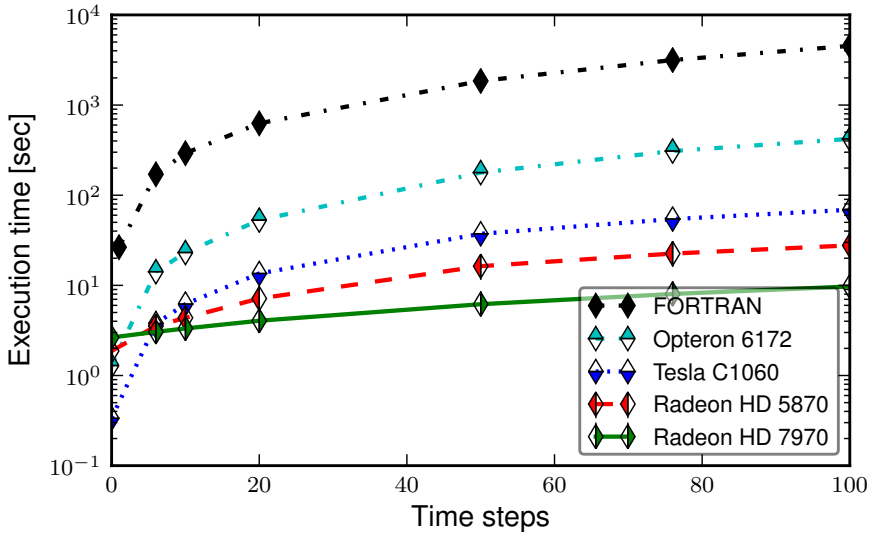


Abbildung C.4.: Laufzeit einer Au+Au Kollision mit $\sqrt{s_{NN}} = 200$ GeV. Auf allen Grafikkarten wurde der gleiche OpenCL Code, ohne spezifische Optimierungen, ausgeführt. Zum Vergleich ist die Laufzeit der klassischen FORTRAN Implementation [52, 57] mit angegeben.

Obwohl es mittlerweile eine starke Forschung über Fluid Dynamik auf Grafikkarten gibt, ist der im Rahmen dieser Arbeit entwickelte OpenCL-SHASTA, der erste Algorithmus auf GPUs für die Art von Fluid Dynamik, die in der Schwerionenforschung benötigt wird. Die Berechnungen mit OpenCL-SHASTA sind im Vergleich mit der klassischen FORTRAN Implementation um den Faktor 460 schneller, siehe Abbildung C.4.

Bibliography

[1] *AMD accelerated parallel processing OpenCL*, tech. rep., Advanced Micro Devices, Inc., December 2012.

[2] *Introducing TITAN*, March 2013.

[3] *Top 500 Supercomputer Sites, November 2012*, March 2013.

[4] K. Aehlig, H. Dietert, T. Fischbacher, and J. Gerhard, *Casimir Forces via Worldline Numerics: Method Improvements and Potential Engineering Applications*, (2011).

[5] J. Aichelin and H. Stöcker, *Quantum molecular dynamics - a novel approach to n-body correlations in heavy ion collisions*, Phys. Lett. B, 176 (1986), pp. 14 – 19.

[6] B. J. Alder and T. E. Wainwright, *Phase transition for a hard sphere system*, J. Chem. Phys., 27 (1957), pp. 1208 – 1209.

[7] M. Bach, V. Lindenstruth, O. Philipsen, and C. Pinke, *Lattice QCD based on OpenCL*, (2012).

[8] S. Bass, M. Belkacem, M. Bleicher, M. Brandstetter, L. Bravina, et al., *Microscopic models for ultrarelativistic heavy ion collisions*, PrPNP, 41 (1998), pp. 255 – 369.

[9] S. Belen'kji and L. Landau, *Hydrodynamic theory of multiple production of particles*, Il Nuovo Cimento (1955-1965), 3 (1956), pp. 15 – 31. 10.1007/BF02745507.

[10] M. Bleicher, E. Zabrodin, C. Spieles, S. Bass, C. Ernst, et al., *Relativistic hadron hadron collisions in the ultrarelativistic quantum molecular dynamics model*, J. Phys. G, 25 (1999), pp. 1859 – 1896.

[11] D. Book, J. Boris, and K. Hain, *Flux-corrected transport ii: Generalizations of the method*, J. Comput. Phys., 18 (1975), pp. 248 – 283.

[12] J. P. Boris and D. L. Book, *Flux-corrected transport*, J. Comput. Phys., 135 (1997), pp. 172 – 186.

[13] H. B. G. Casimir, *On the attraction between two perfectly conducting plates*, Proc. R. Neth. Acad. Arts Sci., 51 (1948), pp. 793 – 795.

[14] A. J. Chorin and J. E. Marsden, *A Mathematical Introduction to Fluid Mechanics*, Springer, Berlin, Heidelberg, 3rd ed. 1993. corr. 4th printing ed., 1993.

[15] A. Corrigan, F. Camelli, R. Löhner, and F. Mut, *Semi-automatic porting of a large-scale fortran cfd code to gpus*, Int. J. Numer. Methods Fluids, 69 (2012), pp. 314 – 331.

[16] L. Csernai, *Introduction to relativistic heavy ion collisions*, Wiley, Chichester New York, 1994.

[17] M. Daga, T. Scogland, and W.-c. Feng, *Architecture-aware mapping and optimization on a 1600-core gpu*, in Proceedings of the 2011 IEEE 17th International Conference on Parallel and Distributed Systems, ICPADS '11, Washington, DC, USA, 2011, IEEE Computer Society, pp. 316 – 323.

[18] E. W. Dijkstra, *Letters to the editor: go to statement considered harmful*, Commun. ACM, 11 (1968), pp. 147 – 148. EWD 215.

[19] G. I. Egri, Z. Fodor, C. Hoelbling, S. D. Katz, D. Nogradi, et al., *Lattice QCD as a video game*, Comput. Phys. Commun., 177 (2007).

[20] R. Feynman, A. Hibbs, and D. Styer, *Quantum Mechanics and Path Integrals: Emended Edition*, Dover Books on Physics, Dover Publications, 2010.

[21] R. P. Feynman, R. B. Leighton, and M. Sands, *The Feynman Lectures on Physics*, Pearson/Addison-Wesley, 2nd revised edition ed., 1963.

[22] M. S. Friedrichs, P. Eastman, V. Vaidyanathan, M. Houston, S. Legrand, A. L. Beberg, D. L. Ensign, C. M. Bruns, and V. S. Pande, *Accelerating molecular dynamic simulation on graphics processing units*, J. Comput. Chem., 30 (2009), pp. 864 – 872.

[23] B. Gaster, D. R. Kaeli, and L. Howes, *Heterogeneous Computing with OpenCL*, Elsevier, Amsterdam, 2011.

[24] J. Gerhard, B. Bauchle, V. Lindenstruth, and M. Bleicher, *How stable are transport model results to changes of resonance parameters? A UrQMD model study*, Phys. Rev. C, 85 (2012), p. 044912.

[25] J. Gerhard, V. Lindenstruth, and M. Bleicher, *Relativistic Hydrodynamics on Graphic Cards*, Comput. Phys. Commun., 184 (2013), pp. 311 – 319.

[26] H. Gies and K. Klingmüller, *Quantum energies with worldline numerics*, J. Phys. A, 39 (2006), pp. 6415 – 6421.

[27] H. Gies and K. Klingmüller, *Worldline algorithms for Casimir configurations*, Phys. Rev. D, 74 (2006), p. 045002.

[28] H. Gies and K. Langfeld, *Loops and Loop Clouds - A Numerical Approach to the Worldline Formalism in QED*, Internat. J. Modern Phys. A, 17 (2002), pp. 966 – 976.

[29] H. Gies, K. Langfeld, and L. Moyaerts, *Casimir effect on the worldline*, JHEP, 6 (2003), p. 18.

[30] H. Gies, J. Sánchez-Guillén, and R. A. Vázquez, *Quantum effective actions from nonperturbative worldline dynamics*, JHEP, 0508 (2005).

[31] S. Gorbunov et al., *ALICE HLT high speed tracking on GPU*, IEEE Trans. Nucl. Sci., 58 (2011), pp. 1845 – 1851.

[32] M. J. Harvey and G. De Fabritiis, *A survey of computational molecular science using graphics processing units*, WIREs Comput. Mol. Sci., 2 (2012), pp. 734 – 742.

[33] M. P. Hertzberg, R. L. Jaffe, M. Kardar, and A. Scardicchio, *Casimir forces in a piston geometry at zero and finite temperatures*, Phys. Rev. D, 76 (2007), p. 045016.

[34] P. Huovinen and P. Petreczky, *QCD Equation of State and Hadron Resonance Gas*, Nucl. Phys. A, 837 (2010), pp. 26 – 53.

[35] S. Kalcher and V. Lindenstruth, *Accelerating galois field arithmetic for reed-solomon erasure codes in storage applications*, CLUSTER, (2011), pp. 290 – 298.

[36] M. Karplus and J. A. Mccammon, *Molecular dynamics simulations of biomolecules*, Nature Struct. Biol., 9 (2002), pp. 646 – 652.

[37] Khronos Group, *The OpenCL Specification*, 1.2 ed., 11 11. Revision 15.

[38] M. L. Klein and W. Shinoda, *Large-scale molecular dynamics simulations of self-assembling systems*, Science, 321 (2008), pp. 798 – 800.

[39] A. Klöckner, N. Pinto, Y. Lee, B. Catanzaro, P. Ivanov, and A. Fasih, *PyCUDA and PyOpenCL: A Scripting-Based Approach to GPU Run-Time Code Generation*, Tech. Rep. 2009-40, Scientific Computing Group, Brown University, Providence, RI, USA, Nov. 2009.

[40] D. Kuzmin, R. Löhner, and S. Turek, *Flux-corrected Transport: Principles, Algorithms, And Applications*, Scientific Computation, Springer, 2005.

[41] H. Lee, L.-E. Yu, S.-W. Ryu, J.-W. Han, K. Jeon, D.-Y. Jang, K.-H. Kim, J. Lee, J.-H. Kim, S. C. Jeon, G. S. Lee, J. S. Oh, Y. C. Park, W. H. Bae, H. M. Lee, J. M. Yang, J. J. Yoo, S. I. Kim, and Y.-K. Choi, *Sub-5nm all-around gate finfet for ultimate scaling*, in VLSI Technology, 2006. Digest of Technical Papers. 2006 Symposium on, 2006, pp. 58 – 59.

[42] H. Lesch and G. Birk, *Theoretische Hydrodynamik*, Universitäts-Sternwarte München, skriptum ed., 2007.

[43] The UrQMD Collaboration, *The urqmd model*, July 2012.

[44] A. P. McCauley, A. W. Rodriguez, J. D. Joannopoulos, and S. G. Johnson, *Casimir forces in the time domain: Applications*, Phys. Rev. A, 81 (2010), p. 012119.

[45] U. Mohideen and A. Roy, *Precision measurement of the casimir force from* 0.1 *to* $0.9\mu m$, Phys. Rev. Lett., 81 (1998), pp. 4549 – 4552.

[46] E. Molnar, H. Niemi, and D. Rischke, *Numerical tests of causal relativistic dissipative fluid dynamics*, Eur. J. Phys, C65 (2010), pp. 615 – 635.

[47] L. Moyaerts, K. Langfeld, and H. Gies, *Worldline Approach To The Casimir Effect*, (2003).

[48] K. Nakamura and P. D. Group, *Review of particle physics*, J. Phys. G, 37 (2010), p. 075021.

[49] H. Niemi, G. S. Denicol, P. Huovinen, E. Molnar, and D. H. Rischke, *Influence of the shear viscosity of the quark-gluon plasma on elliptic flow in ultrarelativistic heavy-ion collisions*, Phys. Rev. Lett., 106 (2011), p. 212302.

[50] J.-Y. Ollitrault, *Relativistic hydrodynamics for heavy-ion collisions*, Eur. J. Phys, 29 (2008), pp. 275 – 302.

[51] H. Petersen, *UrQMD organization map.* private communication, 2012.

[52] H. Petersen, J. Steinheimer, G. Burau, M. Bleicher, and H. Stöcker, *Fully integrated transport approach to heavy ion reactions with an intermediate hydrodynamic stage*, Phys. Rev. C, 78 (2008), p. 044901.

[53] Portland Group, *CUDA Fortran, Programming Guide and Reference*, 12.6 ed., 06 12. First Release 2010.

[54] S. J. Rahi, A. W. Rodriguez, T. Emig, R. L. Jaffe, S. G. Johnson, and M. Kardar, *Nonmonotonic effects of parallel sidewalls on Casimir forces between cylinders*, Phys. Rev. A, 77 (2008), p. 030101.

[55] D. K. F. Read, *Computational Jet Tomography*, March 2013.

[56] ——, *OpenCL SHASTA*, March 2013.

[57] D. H. Rischke, S. Bernard, and J. A. Maruhn, *Relativistic hydrodynamics for heavy ion collisions. 1. General aspects and expansion into vacuum*, Nucl. Phys. A, 595 (1995), pp. 346 – 382.

[58] D. H. Rischke, Y. Pursun, and J. A. Maruhn, *Relativistic hydrodynamics for heavy ion collisions. 2. Compression of nuclear matter and the phase transition to the quark - gluon plasma*, Nucl. Phys. A, 595 (1995), pp. 383 – 408.

[59] A. W. Rodriguez, A. P. McCauley, J. D. Joannopoulos, and S. G. Johnson, *Casimir forces in the time domain: I. Theory*, (2009).

[60] O. Rosenberg, B. R. Gaster, B. Zheng, and I. Lipov, *Opencl static c++ kernel language extension*, tech. rep., Advanced Micro Devices, Inc., 12 2011.

[61] T. Sjöstrand, *The lund monte carlo for jet fragmentation and e+e- physics - jetset version 6.2*, Comput. Phys. Commun., 39 (1986), pp. 347 – 407.

[62] H. Stöcker and W. Greiner, *High energy heavy ion collisions – probing the equation of state of highly excited hardronic matter*, Phys. Rep., 137 (1986), pp. 277 – 392.

[63] F. G. Tinetti, M. Méndez, M. A. López, J. C. Labraga, and P. G. Cajaraville, *Update and Restructure Legacy Code for (or Before) Parallel Processing.*, in Proceedings of the 2011 International Conference on Parallel and Distributed Processing Techniques and Applications, July 2011, pp. 652 – 658.

[64] I. Ufimtsev and T. Martinez, *Graphical processing units for quantum chemistry*, CiSE, 10 (2008), pp. 26 – 34.

[65] J. van Meel, A. Arnold, D. Frenkel, S. Portegies Zwart, , and R. Belleman, *Harvesting graphics power for md simulations*, Mol. Simulat., 34 (2008), pp. 259 – 266.

[66] Wgsimon, *Transistor count and moore's law - 2011.svg*, May 2011. Creative Commons Attribution-Share Alike 3.0 Unported license.

Index