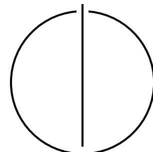
SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Evaluating Single Precision for the Finite-Volume Solver in the Hyperbolic PDE Engine ExaHyPE2

Cedric Dietermann



SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Evaluating Single Precision for the Finite-Volume Solver in the Hyperbolic PDE Engine ExaHyPE2

Untersuchung von Single-Precision für den Finite-Volumen-Löser in der Hyperbolische-PDE-Engine ExaHyPE2

Author:	Cedric Dietermann
Examiner:	Prof. Dr. Michael Georg Bader
Supervisor:	M.Sc. Marc Marot-Lassauzaie
Submission Date:	17.02.2025

I confirm that this bachelor's thesis is my own work and I have documented all sources and material used.

Munich, 17.02.2025 Cedric Dietermann

Abstract

ExaHyPE 2 is a generic hyperbolic Partial Differential Equation (PDE) software Engine designed to solve user defined specific hyperbolic PDEs. Typical applications are solid and fluid dynamics problems in geophysical applications like seismology.

ExaHyPE 2 is built on top of Peano 4, a software tool providing modeling for the physical simulation domains by Cartesian grids as well as their storage and traversal using an adaptive spacetree concept. One implemented way to realize the numerics is the Finite-Volume Solver, specifically the Rusanov Solver, which is implemented using floating-point numbers of double precision (i. e. 64 bits).

In this thesis the existing implementation was extended for use of single-precision (i. e. 32 bits) or half-precision (i. e. 16 bits) floating-point numbers, and simulation results based on the new lower-precision datatypes were evaluated by comparing it to the existing double-precision implementation. For the evaluated scenarios the double and single-precision implementations show similar computing accuracy and performance, while the half-precision datatypes are both less precise and performant.

Contents

Abstract **iii**

1 Introduction **1**

2 Related Work **3**

2.1 Theoretical background . . . 3

2.1.1 Solid and Fluid Dynamics . . . 3

2.1.2 Elastic Wave Equation and Shallow Water Equations . . . 4

2.2 Discretization Methods . . . 5

2.2.1 Overview . . . 5

2.2.2 Finite Volume Method for Solving Wave Equations . . . 7

2.3 Peano 4 . . . 8

2.3.1 Architecture . . . 8

2.3.2 Spacetree traversal approach . . . 9

2.4 ExaHyPE 2 . . . 11

3 Scenarios **14**

3.1 Elastic Planar Waves . . . 14

3.2 Euler Gaussian Bell . . . 17

3.3 SWE Resting Lake . . . 17

4 Implementation **19**

4.1 Types of Floating-Point Numbers . . . 19

4.2 Code Adaptation . . . 20

5 Evaluation **24**

5.1 Hypotheses and Hardware . . . 24

5.2 Accuracy . . . 25

5.2.1 Method . . . 25

5.2.2 Elastic Planar Wave . . . 26

5.2.3 Euler Gaussian Bell . . . 28

5.2.4 SWE Resting Lake . . . 31

5.3 Performance . . . 32

5.3.1 Method . . . 32

5.3.2 Runtime Comparison . . . 33

5.3.3 VTune Analysis for the SWE Resting Lake scenario . . . 34

6 Conclusion **36**

Abbreviations **37**

List of Figures **38**

List of Tables **39**

Bibliography **40**

1 Introduction

Constant advances in hardware, algorithms and software for high performance computing have drastically pushed boundaries for numerical simulation of real-world problems and applications. Lately, high-performance computing has entered the so-called exa-scale era, denoting that fastest supercomputers can operate at a speed of executing 10^{18} floating-point operations per second[1]. These advances for example enable research of large-scale geophysical problems such as earthquake or tsunami wave analysis increasingly to rely on numerical computing.

General Continuum Mechanics (CM) concepts help to describe these geophysical problems and cover fields including solid mechanics, fluid mechanics, thermodynamics, and heat transfer. For many practical problems in CM, simulations are needed if a given problem cannot be investigated by experiments, e. g. if the problem area is just too large, some quantities cannot be measured, can only be measured with great effort, or intrusive measuring techniques often distort the results.

General computational methods in Continuum Mechanics cover the numerical study of flow and deformation effects in solids and fluids based on the principles of mass, momentum, and energy conservation. Mathematically, CM problems usually are modeled by complex systems of Partial Differential Equations (PDEs) which are generally non-linear, but can often be at least partially linearized. Their solutions cannot be reached analytically unless the problem is highly simplified – which usually is not possible with keeping necessary accuracy - but require numerical computation. For example, earthquakes are modeled as elastic waves as either seismic so-called P-waves (Primary, compressional waves) that can travel through both solids and fluids and travel fast, or S-waves (Secondary, shear waves) that can travel only through solids and cause more damage due to their transverse motion, while for tsunamis shallow-water wave models are used (or in special cases, acoustic wave models).

Simulation capabilities in terms of computable problem size and computation speed increased due to advances in computing hardware and software and costs per floating point operation decreased [12], while cost and capabilities on practical experimenting cannot keep up.

However, numerical modeling and computation is necessarily subject to approximation and require trade-offs between accuracy, computation time, computing power and storage requirements.

Reasons for inaccuracies include:

[1]The exa-scale barrier was first surpassed in May 2022 [19]

- Modeling: In simulations, individual phenomena are described using models. These models are not a perfect representation of reality but reduce the complexity of the problem being described to such an extent that a solution becomes achievable.
- Discretization: For the simulation, the implemented differential equations are converted into difference equations in space and time, e. g. on a finite grid. This process can result in a loss of information and accuracy. The solution is approximated through an iterative process, which can also lead to deviations.
- Numerics: Finally, the simulation result is numerically computed with finite numbers based on a computer system's numerical capability potentially introducing additional inaccuracies or (in worst cases) even leading to non-convergence of the mathematical algorithm not even yielding any result.

Thus, the results of the numerical simulations must be critically examined for plausibility. [7]

This work focuses on the study of the last mentioned inaccuracy effect, namely how the use of different precision types of floating-point numbers impacts the simulation results with respect to mainly accuracy versus computing effort (i. e. required computing time and storage).

Generally, use of lower-resolution numbers (i. e. of single or even so-called "half" precision) can be expected to yield faster computation times at the expense of lower accuracy of the numerical result versus the reference "real" result/reality as compared to respective results with use of higher-resolution numbers (i. e. of double precision). [15]

To study the impact of lower-resolution numbers, this thesis uses and extends the open-source simulation software tools Peano 4 and ExaHyPE 2 written in C++ and Python. ExaHyPE 2 is an open-source engine to solve hyperbolic PDEs. It is built on top of and integrated with Peano 4, which provides the storage and traversal of a simulated domain based on a spacetree concept (see Chapter 2).

The thesis work extends the available double-precision implementation of ExaHyPE 2 and Peano 4 to include use of single-precision as well as half-precision number types (Chapter 4).

The effect of the different precision-type usage is evaluated (see Chapter 5) based on simulations of given reference wave scenarios described in Chapter 3: A planar wave in an elastic medium, a two-dimensional Euler Gaussian bell wave, and a Shallow Water Equations-based resting-lake scenario.

2 Related Work

The present thesis project has been carried out based on pre-existing related work, namely

- commonly used Finite Volume Method simulation [6] (Section 2.2) solving general PDE systems for wave propagation problems (Section 2.1), as well as
- software framework/packages
  - Peano 4 [26] (Section 2.3) and
  - ExaHyPE 2 [5] (Section 2.4).

2.1 Theoretical background

2.1.1 Solid and Fluid Dynamics

Many physical phenomena and problems are described and investigated by Continuum Mechanics methods. Conservation laws for mass, momentum and energy are foundational to solid mechanics, fluid dynamics, and wave propagation. Mathematically they are expressed in terms of Partial Differential Equations (PDEs). Especially important are hyberbolic PDEs describing wave propagation. If a disturbance is made in the initial data of a hyperbolic differential equation, the disturbance is not "seen" by every point of space at once, instead the disturbance (wave) travels with a finite propagation speed along the characteristics of the equation (while for elliptic PDEs, the disturbance effects would be seen by all points in the domain at once). [2]

Table 2.1 summarizes a high-level overview on the basic concepts with the basic equations listed as well.

Criteria	Continuum Mechanics (CM) & Theory of Elasticity	Computational Fluid Dynamics (CFD)
Scope	Solids & some fluids: Deformation, stress, strain, waves	Fluids (liquids & gases): Flow behavior, turbulence, aerodynamics, multiphase flows
Governing Equations	**General wave equation / Cauchy's equation of motion:** Describe motion and deformation in a continuum	**Navier-Stokes Equations:** Govern the motion of viscous fluid flows based on mass, momentum, and energy conservation
Special Case	**Elastic Wave Equation (EWE):** Special case for wave propagation in elastic materials	**Shallow Water Equations (SWE):** Special case for thin-layer fluid flows under hydrostatic approximation

Table 2.1: Theory overview [8, 33, 2]

2.1.2 Elastic Wave Equation and Shallow Water Equations

The Elastic Wave Equation (EWE) is used to model the propagation of elastic waves, such as seismic waves or sound waves in solids. For the EWE the material is assumed elastic and isotropic for simplicity, and the model assumes linear elasticity, where strain is proportional to stress. The EWE is only valid for elastic media. Simplified models, such as the acoustic wave equation, may not be applicable for shear waves.

The EWE is most useful for modeling seismic wave propagation in geophysics and sound wave propagation in materials for non-destructive testing (e. g. ultrasonic testing). It is not suitable for non-elastic materials/plastic deformation. For fluids, the EWE only models P-wave behavior but is not applicable for S-waves.

The Shallow Water Equations (SWE) are primarily applicable in shallow water regions, such as coastal areas, rivers, and for flood modeling. Key assumptions in the SWE include the assumption that the fluid layer is shallow, meaning the depth is much smaller than the horizontal dimensions. The equations also neglect vertical accelerations and assume a hydrostatic pressure distribution.

The SWE are particularly effective in predicting wave behavior in shallow regions where gravity waves dominate and are not suitable for modeling behavior in deep water or high-frequency waves.

Both the SWE and the EWE describe wave propagation in different media. The SWE model surface waves in a fluid layer, governed by the velocity and height of the fluid, while the EWE models elastic waves in solids, governed by the displacement and stress in the material. Both are hyperbolic equations, meaning they describe wave-like phenomena that propagate disturbances through a continuous medium.

The SWE is concerned with gravity waves, which are dominated by gravitational forces, while the EWE handles elastic waves, which are dominated by material elasticity.

In summary, SWE are best suited for modeling tsunami waves and shallow water

dynamics, in regions with shallow depths, while EWE are used for modeling seismic wave propagation in both solids and fluids, including both compressional and shear waves. [8, 33, 2]

2.2 Discretization Methods

As it is very difficult to solve the resulting PDEs in a continuum, it is necessary to discretize the continuous problem to be able to solve it numerically. To do this there are three common methods:

- Finite Difference
- Finite Element
- Finite Volume

Following, first an overview on these is given following literature, e.g. [6, 7] chapters 2.6, 3 and 4, resp., and [32]. After the general overview of all three methods a more detailed explanation of the Finite Volume (FV) method used in this thesis is given.

2.2.1 Overview

Finite Difference (FD) method is the oldest of the three methods traced back to L. Euler in the 18th century [7]. It uses a usually structured point grid of the solution domain with conservation equations in their differential form. The latter are approximated at each grid point resulting in a single algebraic equation at each node containing the unknown value at this and neighboring grid nodes. First and second derivatives of the variables are approximated using Taylor series expansion and polynomial fitting. Also, FD may be used to interpolate between grid nodes to calculate values at locations lying between several grid nodes. Although FD is simple and effective on structured grids, it is restricted to simple geometries and therefore limited for most real-world problems.

Next, **Finite Element (FE)** methods discretize the domain into discrete volumes or elements. Depending on the domain, these elements mostly are of triangular or quadrangular shape (for 2D domains) and tetrahedra or hexahedra (for 3D), respectively.

In contrast to FD and Finite Volume, FE uses equations multiplied by a weight function prior to integration over the entire domain. To ensure continuity across elements' boundaries the simple FE both solution functions and weight functions are of linear shape within an element. Contrary to FD, FE methods are capable of simulating and solving complex geometries also allowing for further grid refinement by subdividing existing elements. However, FE methods suffer from unstructured grids leading to not well structured matrices.

Finally, **Finite Volume (FV)** methods are similar to FE as the domain is discretized using a finite number of elements, called control volumes in context of FV, each

containing a computational node in the center of the control volume (CV) at which all variables' values are to be calculated. [7]

Usually, these nodes are positioned in the center of the control volume, i. e. the volume is defined first and the node afterwards. However, it is also possible to first define the computational nodes and based on the grid of nodes define the control volumes. Both methods have advantages and disadvantages regarding accuracy with respect to the mean value of the volume or at the control volumes' surfaces. Similar to the equation described above for FD methods, also for FV each computational node is assigned an algebraic equation containing its value as well as the values of the neighboring nodes. In a next step the integral forms of the conservation equations are used and approximated by approximating the surface integral. Although the approach of FV methods is the easiest one among the presented approaches, as each term to be approximated has a clear physical meaning, it requires considerable implementation effort especially in 3D applications [7]. Nevertheless, FV methods are commonly used in CFD frameworks and toolboxes such as Peano/ExaHyPE.

Figure 2.1 gives an iconized graphical illustration of the discretization model for the three described methods as well as their basic mathematical discretization logic.

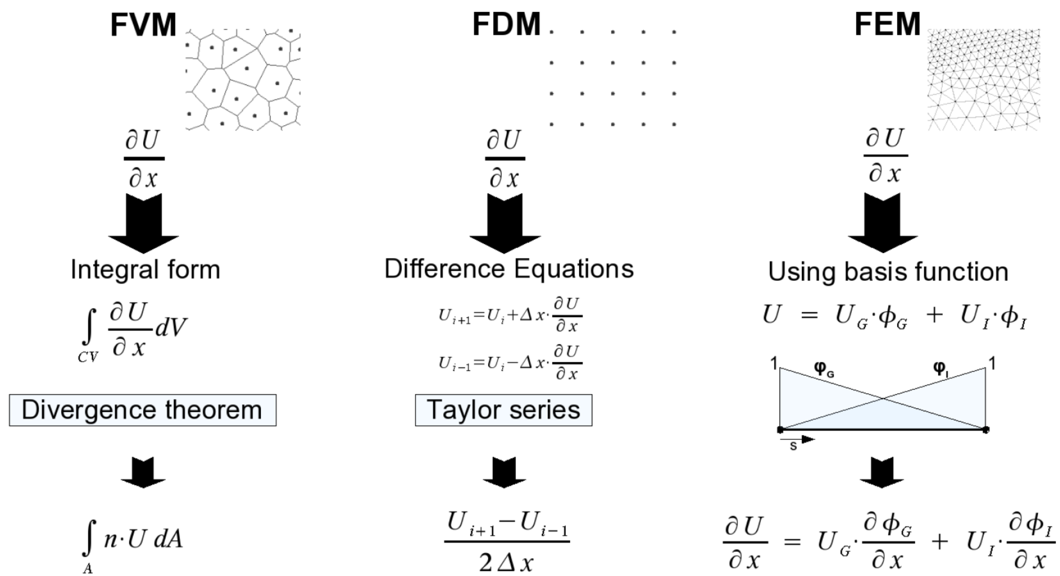


Figure 2.1: Discretization methods with their grids and equations [17]

2.2.2 Finite Volume Method for Solving Wave Equations

The FV method can be used for solving various types of wave equations, such as the Elastic Wave Equation (EWE) in solids and fluids, and the Shallow Water Equations (SWE). It is a conservative, robust, and flexible method for solving hyperbolic PDEs, making it suitable for a wide range of applications in computational geophysics, hydrodynamics, and engineering.
The principal steps for the FV method are [20, 22]:

- **Volume Discretization:** FV transforms the basic PDEs to be solved into algebraic equations by integrating over a control volume (CV) and applying Gauss's divergence theorem. The governing equation for a conserved quantity Q (e. g. velocity, water height, etc.) can be written as:

$$\frac{\partial Q}{\partial t} + \nabla \cdot \mathbf{F} = 0, \tag{2.1}$$

  where $\mathbf{F}$ is the flux vector. Applying the divergence theorem transforms the volume integral of the CV into a surface integral over its surface S:

$$\frac{\mathrm{d}}{\mathrm{d}t} \int_V Q \mathrm{d}V + \int_S \mathbf{F} \mathrm{d}S = 0. \tag{2.2}$$

  which is then approximated by discretizing over a set of Cartesian cells V_i.

- **Time Discretization:** Time integration can be performed using explicit or implicit schemes. Explicit schemes, e. g. Leapfrog or Runge-Kutta methods, are typically used in more stable cases. Implicit schemes, such as the Crank-Nicholson method, allow for larger time steps but require solving a system of equations at each time step, increasing computational complexity per step. E. g. a simple Euler method yields:

$$Q_i^{n+1} = Q_i^n - \frac{\Delta t}{V_i} \sum_{\mathrm{S}} \mathbf{F}_{\mathrm{S}} \cdot \mathbf{n}_{\mathrm{S}} A_{\mathrm{S}}. \tag{2.3}$$

  where A_{S} is the face surface area and $\mathbf{n}_{\mathrm{S}}$ is the outward normal.

- **Solver:** In this work a simple Rusanov-type Riemann solver is used approximating the numerical fluxes between cells by correcting the average of the neighboring cell's fluxes by a wave-speed depending factor with the maximum wave speed $\lambda_{\max}$ across the interface. [21]

$$\mathbf{F}_{\mathrm{Rusanov}} = \frac{1}{2}(\mathbf{F}_{\mathrm{in}} + \mathbf{F}_{\mathrm{out}}) - \frac{1}{2}\lambda_{\max}(Q_{\mathrm{out}} - Q_{\mathrm{in}}), \tag{2.4}$$

2.3 Peano 4

The Peano software, currently available in its 4th generation, is a framework for generating dynamically adaptive Cartesian grids for a wide range of scientific applications. It, as well as many of the subprojects that rely on it, is implemented in a mix of C++ and Python.

The framework is named after the Italian mathematician Giuseppe Peano who discovered its foundational space-filling curve of the same name. [35]. The development of Peano started originally in the early 2000s at the group of Prof. Tobias Weinzierl at TUM's Department of Computer Science and is available as open source repository. [24].

On top of Peano multiple extensions and applications have been implemented. One key example is the ExaHyPE solver engine for hyberbolic PDEs used in this work. ExaHyPE uses Peano's Adaptive Mesh Refinement (AMR) and has been closely integrated and merged into Peano in its second generation ExaHyPE 2 since 2019 (Section 2.4). [24]

2.3.1 Architecture

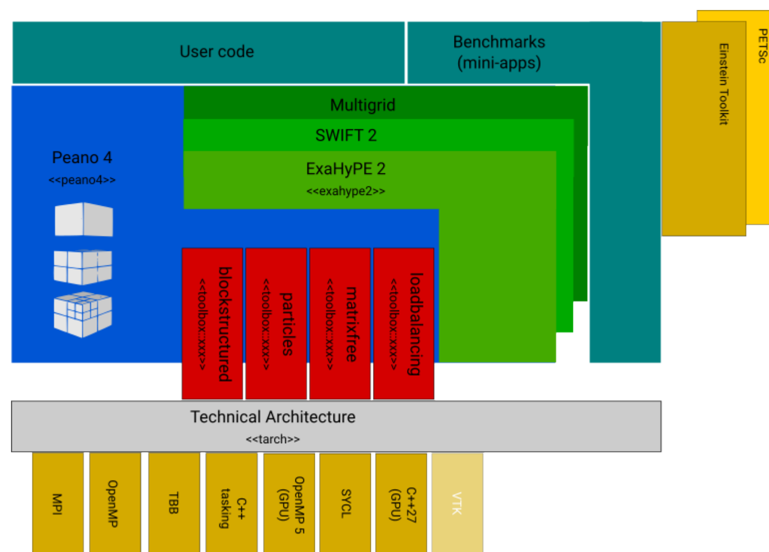


Figure 2.2: Peano's code architecture [23]

Peano is structured in layers providing components in static libraries implemented in C++ code. Its layer structure in turn is also reflected by all applications based on Peano (Figure 2.2).

Peano (with any extension) does not consist of static libraries but generates application-specific code. However, some of the applications built on top of Peano with all its

toolboxes and extensions provide "ready to use" options. Peano components are integrated in one repository and one code base.

Likewise, Peano's extensions are tools that enable users to write simulation code for a specific application domain. [23]

2.3.2 Spacetree traversal approach

Peano is based on the concept of spacetrees, a generalisation of the classical octree concept [1], that generate cascaded series of adaptive Cartesian grids [35].

A spacetree traversal means an element-wise traversal of the hierarchy of the adaptive Cartesian grids. Peano implements such a grid traversal together with a storage algorithm. It provides interfaces for applications performing operations (e. g. per element, per vertex) on the grid as well as those for adaptive load balancing, advanced geometry enhancements, etc.

There are multiple extensions of Peano for specific application domains. Examples of such extensions include those for Smoothed Particle Hydrodynamics (SPH; a rather particle- than mesh-based concept), multigrids or hyperbolic solvers such as ExaHyPE 2 which is used in the present work. [24]

The grid traversal, i. e. the sequence of stepping through the grid, is prescribed in the Peano code, i. e. the user (application) cannot or does not have to advise, command or change it. [35]

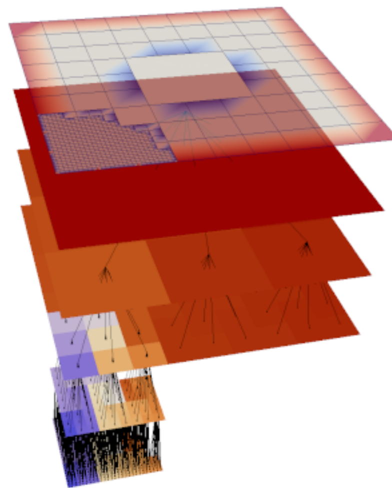


Figure 2.3: Spacetree traversal [25]

Peano traverses spacetrees top-down, i. e. from coarse to fine. This traversal is given by the Peano core code as an integral element.

Simulation traversals are executed in the following steps (Figure 2.4) [23]:

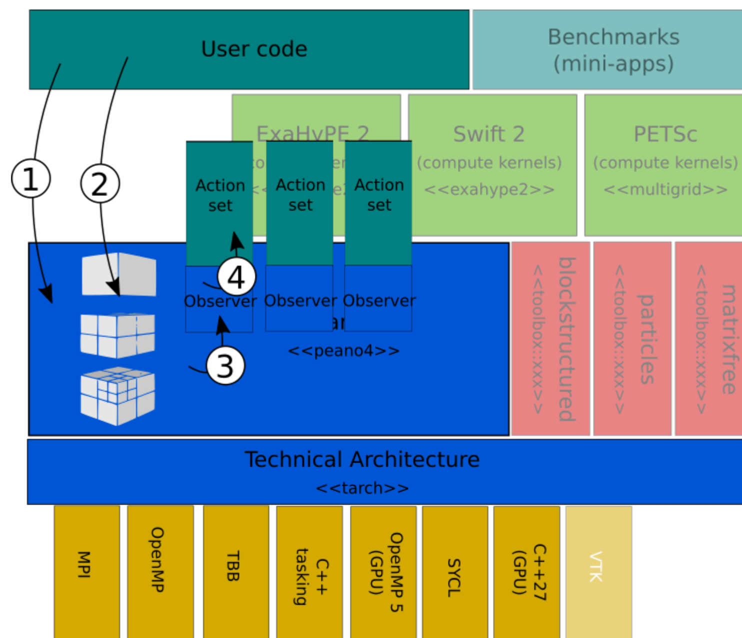


Figure 2.4: Flow of simulation (traversals) in Peano [23]

1. The main file hosted in the user code first creates a spacetree. (For parallel code also sets of spacetrees can exist from forkation).

2. The user's main program runs through its algorithmic steps. Each step might trigger a mesh traversal by calling Peano (`peano4::parallel::SpacetreeSet::traverse()`) to run through all spacetrees. Peano manages the traversal itself retaining control on all parameters incl. order and parallelization approach.

3. Users interact with the ongoing traversal by so-called Observers (`peano4::grid::TraversalObserver`). These classes provide binding points, such as when the core loads, enters or leaves a cell, at which code can be injected in order to interact with data on the mesh.

4. The user code passes the observer object into the traverse() function. The observer allows to execute instructions (e. g. visualising a cell or updating the finite-volume solution) upon specific events during the traversal (e. g. when entering a cell or starting a traversal of a tree) by calling a function from the user code or the supporting toolbox (depicted in red color in Figure 2.4). [23, 35]

To use Peano's functionality, typically users would not need to access/change Peano's low level C++ code but rather either use Peano's Python Application Programming Interface (API) or one of its extensions built on top of it (such as ExaHyPE) hiding Peano's internal details.

Figure 2.5: Peano's traversal path in AMR grid [36]

Peano models the computational space into a unit hypercube of dimension d. The domain is then iteratively divided further down into $k = 3$ equidistant parts along each axis, resulting in 3^d hypercubes („refinement"). For further refinement this step is repeated recursively (Figure 2.5). In cases where greater computing resources are necessary (e. g. for greater accuracy) patches (hypercubes) can be further refined down. This constitutes a so-called spacetree hierarchy, whose nodes at a depth n represent a patch which has been n times refined. [36]

Cells and vertices are unique by their level of refinement and spatial position. During run-time, Peano determines the order of traversal through the spacetree. A traversal visits each cell. The user can define exact functionality of actions performed in each cell during each traversal. [24]

2.4 ExaHyPE 2

ExaHyPE ("Exascale Hyperbolic PDE Engine") is an open-source software simulation engine for solving systems of first-order (both linear and non-linear) hyperbolic Partial Differential Equations (PDEs) [30]. The term "Exascale" denotes the capability to compute exaFLOPS i. e. 10^{18} 64-bit double-precision floating point operations per second.

Hyperbolic PDEs are typically derived from the conservation laws of physics and are useful in a wide range of application areas. Applications powered by ExaHyPE can be scaled to run on a wide range of computer hardware from laptops up to supercomputers using thousands of processor cores.

ExaHyPE provides robust numerical methods yielding high-accuracy solution capabilities for both linear and non-linear hyperbolic PDEs by increasing the simulation accuracy based on its dynamic Adaptive Mesh Refinement.

Users can parametrize the engine to the particular PDE to be solved. Therefore ExaHype 2/Peano allows to drastically reduce creation time for a complete simulation code for a new hyperbolic PDE from typically weeks down to hours. Typical application areas include seismology (used in this work) and astrophysics. [30]

ExaHyPE is built on top of Peano (Section 2.3) and is implemented in C++ and Python using Peano's Python API. As such ExaHyPE is also based on dynamically adaptive Cartesian meshes. It supports Finite Volume, Runge-Kutta, Discontinuous Galerkin (DG) and ADER-DG discretization models. ExaHyPE hides most of its technical and computer science aspects as well as most of the used numerical algorithms from the users: Instead, they insert user functions for their specific PDE problems into the engine entrusting the further modeling and computation to ExaHyPE.

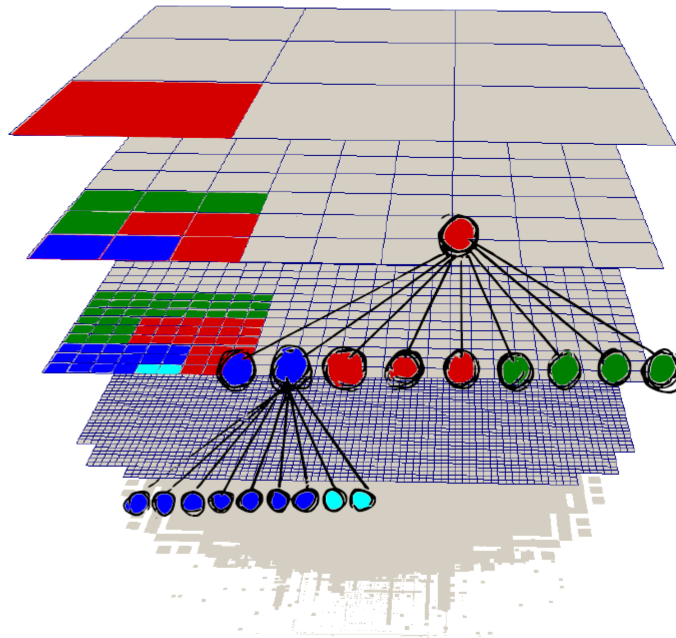


Figure 2.6: Top-down spacetree decomposition in Peano [36]

The architecture of ExaHyPE 2 is illustrated in Figure 2.7. ExaHyPE is a solver engine, leaving it to the user to create domain-specific code (turquoise color denotes files written by the user) as basis for specific simulation code generation. From the user input, ExaHyPE creates glue code, empty application-specific classes and core routines. These templates are filled by the user with application-specific PDE terms and parameters in C++, Python or Fortran.

Specifically for the FV method used in this work, during the run-time spacetree traversal, ExaHyPE ensures the computed values' boundary conditions between the current computed patch and its neighboring, i. e. face-connected, patches are satisfied by modeling them in a halo representing the neighboring cells around the current patch.

For this, ExaHyPE passes values of a patch of size $p \times p$ cells plus its halo of

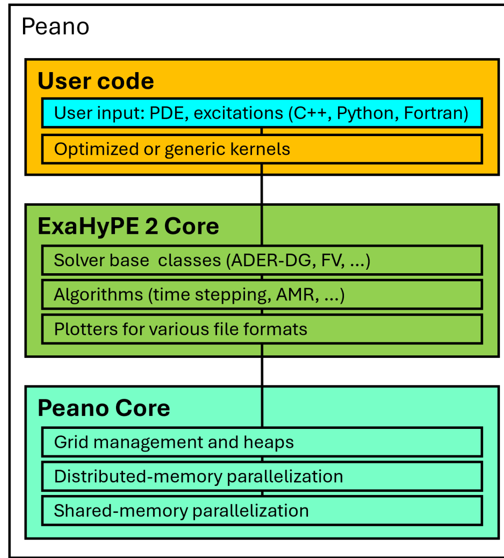


Figure 2.7: ExaHyPE2 architecture [30, 23, 5]

neighboring cells with the halo width N (so a total of $(p+2N)^d$ cells) into a kernel, where d is the spatial dimension, i. e. 2 or 3. ExaHyPE fills these halo values with the neighbor cells' valid data called Q_{in}. The kernel then returns $(p+2N)^d$ data with the new time step's values, called Q_{out}. An example for $p=7$, $N=2$ and $d=2$ is shown in Figure 2.8.

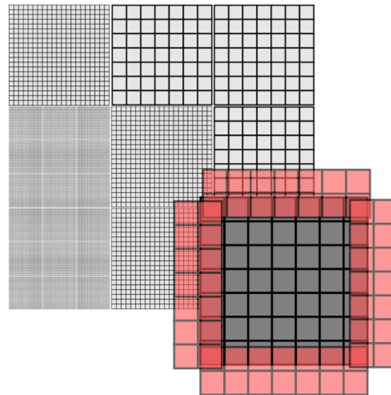


Figure 2.8: ExaHyPE's halo concept for boundary conditions for the FV solver [21]

3 Scenarios

In this chapter, an overview is given for the three basic (2D) test scenarios used in Chapter 5 for evaluating and comparing the results of the different implementations of the Finite Volume solver:

- Elastic planar wave (Section 3.1),
- 2D Euler Gaussian bell wave (Section 3.2),
- SWE Resting Lake (Section 3.3).

These scenarios were chosen selected based on several reasons.

An elastic harmonic planar wave has a simple waveform that remains stable in an ideal elastic medium. Likewise Euler Gaussian wave excitation theoretically also maintains a stable wave form in an ideal lossless medium, however its numerical solution tends to be more sensitive to numerical instabilities depending on the discretization method, solver schemes, and handling of numerical errors. A 2D shallow water scenario represents a well-balanced physical problem, and as such represents a good test case for the behavior of the numerical solution algorithm. Thus the three scenarios cover a meaningful range of problem types. Also, all three scenarios have been studied thoroughly (references see Sections 3.1-3.3). Last but not least, due to their simple symmetry/periodicity they allow for analytical solutions.

3.1 Elastic Planar Waves

Waves can propagate in elastic or non-elastic media. Table 3.1 shows the basic characteristics.

Waves propagate through an elastic medium depending on the nature of the medium and the initial wave excitation:

1. Longitudinal Waves (P-Waves): Particle motion is parallel to the wave propagation direction. These waves depend on the bulk modulus and density of the medium.
2. Transverse Waves (S-Waves): Particle motion is perpendicular to the wave propagation direction. These waves exist only in solids and depend on the shear modulus of the material.

Property	Elastic Waves	Non-Elastic Waves
Energy Loss	Minimal (ideally lossless), energy is conserved	Energy dissipates as heat/deformation
Medium Response	Returns to original shape	Undergoes permanent deformation
Propagation	Efficient, travels long distances	Attenuates quickly
Examples	Sound waves, seismic P- and S-waves, ultrasound	Shock waves, plastic deformation

Table 3.1: Comparison of Elastic and Non-Elastic Waves

Elastic planar waves propagate through an elastic medium (solid), with their wavefronts forming infinite, parallel planes. The deformation caused by these waves results in strain in the medium. Key to understanding their propagation is knowing the initial conditions and boundary conditions. The initial condition is governed by the material's Lamé parameters, λ and μ, and the initial deformation causing strain. [9, 18, 4]

In this scenario [27], based on that of [16], planar waves are translated through a periodic domain without deformation, returning to their initial conditions at regular intervals.

For a homogeneous, isotropic elastic medium, the displacement field $\mathbf{u}$ of an elastic planar (harmonic) wave is given by:

$$\mathbf{u} = \mathbf{A}e^{i(\mathbf{k}\cdot\mathbf{r}-\omega t)} \tag{3.5}$$

where

- $\mathbf{u}$: Displacement vector field representing the displacement of particles in the medium as a function of position and time.
- $\mathbf{A}$: Amplitude vector indicating the initial magnitude and direction of the displacement,
- i: Imaginary unit, where $i^2 = -1$,
- $\mathbf{k}$: Wave vector defining the direction of wave propagation and its magnitude, related to the wavelength l by $|\mathbf{k}| = 2\pi/l = 2\pi f/v$, with the wave speed v,
- $\mathbf{r}$: Position vector specifying a point in the medium
- ω: Angular frequency of the wave, related to the temporal frequency f by $\omega = 2\pi f$, and
- t: Time.

For longitudinal waves, the wave phase speed is:

$$v_p = \sqrt{\frac{\lambda + 2\mu}{\rho}} \tag{3.6}$$

For transverse waves, the wave phase speed is:

$$v_s = \sqrt{\frac{\mu}{\rho}} \tag{3.7}$$

where

- λ: First Lamé parameter, a material constant describing its compressibility,
- μ: Second Lamé parameter (also known as the shear modulus), representing the medium's resistance to shear deformation, and
- ρ: Mass density of the medium.

The 2D Elastic Wave Equation linearized for small deformations only is [27, 16]:

$$\partial_t \begin{bmatrix} u \\ v \\ \sigma_{xx} \\ \sigma_{yy} \\ \sigma_{xy} \end{bmatrix} + \nabla \cdot \mathbf{F} \begin{bmatrix} u \\ v \\ \sigma_{xx} \\ \sigma_{yy} \\ \sigma_{xy} \end{bmatrix} = \mathbf{0} \tag{3.8}$$

where

- u, v are the displacement velocities in x- and y-directions, resp.,
- σ_{xx} and σ_{yy} are the normal stress components in x- and y-directions, resp.,
- σ_{xy} is the shear stress component acting on the xy-plane, and
- $\mathbf{F} = (\mathbf{F_1}, \mathbf{F_2})$ is a tensor for the linear flux:

$$\mathbf{F_1} = \begin{bmatrix} 0 & 0 & 1/\rho & 0 & 0 \\ 0 & 0 & 0 & 0 & 1/\rho \\ (\lambda + 2\mu) & 0 & 0 & 0 & 0 \\ \lambda & 0 & 0 & 0 & 0 \\ 0 & \rho & 0 & 0 & 0 \end{bmatrix}, \quad \mathbf{F_2} = \begin{bmatrix} 0 & 0 & 0 & 0 & 1/\rho \\ 0 & 0 & 0 & 1/\rho & 0 \\ 0 & \lambda & 0 & 0 & 0 \\ 0 & (\lambda + 2\mu) & 0 & 0 & 0 \\ \mu & 0 & 0 & 0 & 0 \end{bmatrix} \tag{3.9}$$

The eigenvalues of the flux are $(-v_p, -v_s, 0, v_s, v_p)$.
For the simulations the values are chosen as $\lambda = -4.0$, $\mu = 4.0$ and $\rho = 1.0$.

3.2 Euler Gaussian Bell

The Euler equations describe the behavior of an inviscid fluid in the absence of both friction and heat conduction. A common test case is the Gaussian bell, which models a localized density perturbation propagating in a medium. The Gaussian bell propagates as a wave, maintaining its shape in a non-dispersive medium. However, numerical artifacts, such as artificial diffusion and dispersion errors, may distort the propagation of the Gaussian wave, even though the physical system itself is non-dispersive.

The Euler equations in two dimensions are [14, 28]:

$$\frac{\partial}{\partial t}\begin{bmatrix}\rho \\ \rho u_1 \\ \rho u_2 \\ \rho E_\mathrm{t}\end{bmatrix} + \frac{\partial}{\partial x}\begin{bmatrix}\rho u_1 \\ \rho {u_1}^2 + \\ \rho u_1 u_2 \\ (\rho E_\mathrm{t} + p)u_1\end{bmatrix} + \frac{\partial}{\partial y}\begin{bmatrix}\rho u_2 \\ \rho u_1 u_2 \\ \rho {u_2}^2 + p \\ (\rho E_\mathrm{t} + p)u_2\end{bmatrix} = \mathbf{0} \tag{3.10}$$

where ρ is the medium's mass density, $\mathbf{u} = (u_1(x,y,0), u_2(x,y,0))$ is the velocity vector, E_t is the total energy density and $p(x,y,0)$ is the pressure.

The initial condition is given by [14]:

$$\rho(x,y,0) = \rho_0 \left(1 + e^{-\frac{(x-x_0)^2+(y-y_0)^2}{2\sigma^2}}\right) \tag{3.11}$$

where ρ_0 is the background density, (x_0, y_0) is the initial peak location, and σ controls the width of the Gaussian profile. For the computations in Chapter 5 the values were used as $\rho_0 = 0.02$, $\sigma = 0.1$, $p = 1$ and $u_1 = u_2 = 1$. Also in this scenario a periodic domain is assumed.

The eigenvalues of the 2-dimensional Euler equation are [28]:

$$\begin{bmatrix}\lambda_1 \\ \lambda_2 \\ \lambda_3\end{bmatrix} = \begin{bmatrix}u - c \\ u \\ u + c\end{bmatrix} \tag{3.12}$$

where c is the wave propagation speed.

3.3 SWE Resting Lake

The Shallow Water Equations (SWE) model the motion of a fluid layer under gravity, assuming the horizontal length scale is much larger than the vertical depth. The lake-at-rest condition is a steady-state solution where the water remains still, satisfying the balance condition

$$h + b = \text{constant}, \tag{3.13}$$

where h is the water height above sea bottom and b is the bathymetry, i. e. the sea bottom's elevation. [3, 29]

The SWE system consists of mass and momentum conservation equations:

$$\frac{\partial}{\partial t}\begin{bmatrix} h \\ hu_1 \\ hu_2 \\ b \end{bmatrix} + \frac{\partial}{\partial x}\begin{bmatrix} hu_1 \\ hu_1{}^2 \\ hu_1 u_2 \\ 0 \end{bmatrix} + \frac{\partial}{\partial y}\begin{bmatrix} hu_2 \\ hu_1 u_2 \\ hu_2{}^2 \\ 0 \end{bmatrix} + hg\begin{bmatrix} 0 \\ \partial_x(b+h) \\ \partial_y(b+h) \\ 0 \end{bmatrix} = \mathbf{0} \tag{3.14}$$

with eigenvalues

$$\begin{bmatrix} \lambda_1 \\ \lambda_2 \\ \lambda_3 \end{bmatrix} = \begin{bmatrix} u + \sqrt{g(h+b)} \\ u \\ u - \sqrt{g(h+b)} \end{bmatrix} \tag{3.15}$$

where g is the gravitational acceleration, $\mathbf{u} = (u_1, u_2)$ is the velocity vector, and $u = |\mathbf{u}|$.

At rest, the velocity components vanish ($u = 0$), leading to a stable equilibrium. If perturbed, gravity acts as a restoring force, leading to wave-like oscillations.

The SWE formulation is used in modeling tsunamis, dam breaks, and other geophysical flows under shallow water conditions.

For the computations in Chapter 5 initially the domain is divided in half and the water height of one half is slightly higher (1.1 units) than the height in the other half (1.0 units).

4 Implementation

4.1 Types of Floating-Point Numbers

Floating-point numbers come in different precision levels, each with varying storage sizes, precision, and range.

Classically, there have been floating-point numbers of

- single (with binary 32-bit coding, also called "FP32") and
- double (with binary 64-bit coding, also called "FP64") precision.

Later so-called "half"-precision types of floating-point numbers have been introduced:

- IEEE standard 754's 2008 version [13] introduced half-precision 16-bit "float16" or "FP16" [10] [11].
- To overcome float16's main disadvantage of much narrower range while preserving storage advantage, a decade later half-precision type "bfloat16" (or "BF16", "brain floating-point") was introduced by the Google Brain group focusing on Artificial Intelligence [34].

Below is a comparison of FP64, FP32, FP16 and BF16 floating-point number types:

Precision Type	Storage Size	Exponent Bits	Mantissa Bits	Approx. Dec. Precision	Approx. Range
FP64	64 bits	11 bits	52 bits	15-17 digits	$-10^{308}... + 10^{308}$
FP32	32 bits	8 bits	23 bits	6-9 digits	$-10^{38}... + 10^{38}$
FP16	16 bits	5 bits	10 bits	3-4 digits	$-10^{5}... + 10^{4}$
BF16	16 bits	8 bits	7 bits	2-3 digits	$-10^{38}... + 10^{38}$

Table 4.1: Floating-point precision characteristics

The key differences between the different floating-point precision types are:

- **Precision:** FP64 has the highest precision ($\sim$15-17 decimal digits), followed by FP32 ($\sim$6-9 digits). BF16 has low precision ($\sim$2-3 digits) due to fewer mantissa bits, whereas FP16 is slightly better ($\sim$3-4 digits).

- **Range:** BF16 retains the exponent size of FP32, allowing for a large range but with reduced precision. FP16, however, has a much smaller exponent (5 bits), leading to a narrower range.
- **Performance & Use Cases** [31, 34]**:**
  - FP64 is used for high-precision applications like scientific computing and finance - at cost of higher memory and computing time compared to FP32.
  - FP32 is the standard for general computing, gaming, and deep learning.
  - FP16 is common in GPUs for deep learning inference and graphics, offering reduced memory usage and increased speed while providing much narrower range than FP32.
  - BF16 is optimized for machine learning (especially training on TPUs) since it maintains FP32's range but uses less memory.

4.2 Code Adaptation

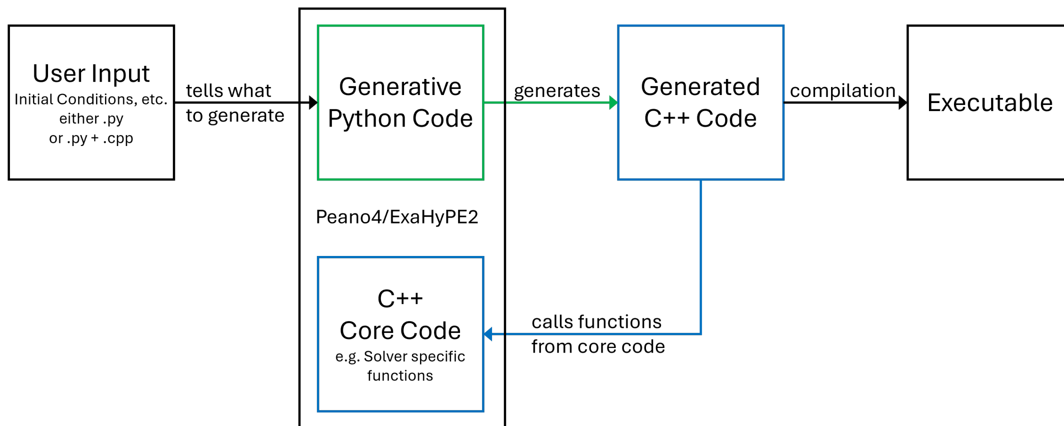


Figure 4.1: Code Structure

In the following section I explain how I changed and templated the code to extend the usage to 32-bit single-precision (C++ type `float`) and 16-bit half-precision (`_Float16` and `__bf16`) beyond previous 64-bit double-precision floating-point numbers (`double`). ExaHyPE 2 supports different FV solvers, but for this thesis I constrain the changes and outcomes to the Rusanov Global Adaptive FV solver which is a specialized Riemann solver (Subsection 2.2.2) using Rusanov fluxes and global adaptive time stepping, i. e. timestep sizes are chosen automatically based on the greatest wave speed over the domain.

As Figure 4.1 illustrates the code of the ExaHyPE 2 and Peano 4 environment is basically divided in two main parts. One part is the core code written in C++. The

second code part written in Python generates the final code depending on the input the user gives. The process of computing a specific scenario using ExaHyPE 2 and Peano 4 can be divided in the following steps:

1. The user defines a specific scenario describing an initial condition and other information like which solver to use and passes it to the Python part of the ExaHyPE 2 and Peano 4 code.

2. The Python code takes the given information and generates C++ code. It should be noted that there is no actual numerical computation executed in the Python part.

3. The generated C++ code calls functions from the C++ core code (function calls depend on the user input, e. g. the user specifying the Rusanov Global Adaptive FV solver leads to a function call of `RusanovGlobalAdaptiveFV::<function>()`).

4. The generated C++ code is compiled into an executable that computes the problem stated in the user's input.

Considering this structure of the code the implementation of the templated Rusanov Global Adaptive FV solver was executed in two phases:

- In a first phase the generated C++ code and the C++ core code shown in blue in Figure 4.1 were modified to validate the approach.

- In a second phase the generative Python code shown in green was adapted.

Phase 1 - Extension of C++ code parts

An initial implementation of both the solver and existing scenarios relying on said solver were available before I started, which I used to generate a functioning implementation in 64-bit double precision for the pre-defined Euler Gaussian Explosion scenario. The generated C++ code calls the relevant functions of ExaHyPE's Rusanov Global Adaptive FV solver and the Peano 4 traversal implemented in the C++ core code. At that moment everything is implemented in 64-bit double precision.

Next, I redefined all variables from `double*` to `float*`. From here I templated the used functions in the C++ core code as shown in the simple example below

```
template<typename T=double>
void clearHaloLayerAoS(
        const peano4::datamanagement::FaceMarker& marker,
        int numberOfDoFsPerAxisInPatch,
        int overlap,
        int unknowns,
```

```
        T* values
);
```

where the first line `template<typenameT=double>` tells that the function is templated and can take various datatypes instead of the `T`. In this specific example `T*` is then used as the datatype of the parameter `values` meaning that it is possible to call the function not only by

```
clearHaloLayerAoS<double>(const peano4::datamanagement::FaceMarker&
    exampleMarker, int exampleNumberOfDoFsPerAxisnPatch, int
    exampleOverlap, int exampleUnknowns, double* exampleValues)
```

as in the original version of the solver but also by

```
clearHaloLayerAoS<float>(const peano4::datamanagement::FaceMarker&
    exampleMarker, int exampleNumberOfDoFsPerAxisnPatch, int
    exampleOverlap, int exampleUnknowns, float* exampleValues)
```

or similarly using the datatypes `_Float16*` and `__bf16*` for the `exampleValues`.

After not only templating every relevant function in the C++ core code but also declaring it with the possible types as shown below for the example of `float` the generated code was compiled without errors and used the 32-bit single precision datatype `float` successfully for the first time.

```
template
void toolbox::blockstructured::clearHaloLayerAoS<float>(
        const peano4::datamanagement::FaceMarker& marker,
        int numberOfDoFsPerAxisInPatch,
        int overlap,
        int unknowns,
        float* value
);
```

Phase 2 - Extension of Python code generation

In a second phase, in order to be able to reproduce the generated code supporting the four different datatypes mentioned above I modified the generative Python code of ExaHyPE 2 and Peano 4. To allow the user to specify the used precision type I modified the code of the main Python file of the FV solvers (`FV.py`) as shown below adding the entry `PRECISION` to the dictionary. `PRECISION` takes the chosen precision type as a string that the user gives as a parameter to the instance of the Rusanov Global Adaptive FV solver.

```
class FV(object):
def __init__(
```

```
        self,
        precision: str,
        name,
        patch_size,
        overlap,
        unknowns,
        auxiliary_variables,
        min_volume_h,
        max_volume_h,
        plot_grid_properties,
        pde_terms_without_state: bool,
        kernel_namespace,
        baseline_action_set_descend_invocation_order=0,
):
self._precision = precision
# Other initialization code here

def _init_dictionary_with_default_parameters(self, d):
        d["PRECISION"] = self._precision
        # Other dictionary entries here
```

I also added the parameter `precision` to the `__init__` function of the subclasses of `FV`. This ensures that the precision parameter can be passed up along the class hierarchy to the top-level's class dictionary when calling the Rusanov Global Adaptive FV solver (on the lowest level).

Using this dictionary entry, every occurrence of the 64-bit double precision variables changed in the generated C++ code in the first phase of the code adaptation gets replaced by the adapted generative Python code and guarantees that the generated code uses the chosen precision type at every relevant occurrence as shown below:

```
[[maybe_unused]] const {{PRECISION}}* __restrict__ Q,
```

`{{PRECISION}}` gets replaced using the dictionary entry and therefore lets the generated code use the specified datatype.

The outcome of the explained implementation process is a templated version of the implemented Rusanov Global Adaptive FV solver that works with the 64-bit double-precision C++ datatype `double`, the 32-bit single-precision datatype `float` and the 16-bit half-precision datatypes `_Float16` and `__bf16`, respectively. The implementation allows the user to specify the precision type by adding it to the parameters given to the Rusanov FV solver.

The source code extended in this thesis work can be found at `https://gitlab.lrz.de/CedricDietermann/singleprecisionexahype2`.

5 Evaluation

5.1 Hypotheses and Hardware

In the following sections both the performance and the accuracy of the simulation results of the extended solver are discussed and compared to the original solver using double precision. Several effects can be expected when changing the precision types. Due to the difference in the approximated decimal precision of the four different datatypes introduced in Section 4.1 a difference in the errors of the results is expected. In particular FP64 should yield the best results followed by FP32, then the 16-bit half-precision types FP16 and BF16 are expected to have higher errors in the results. The use of lower-precision types should lead to a reduced computing time and lower storage use.

In order to compare both the accuracy and the performance of the different numerical precision types the code for the different scenarios described in Chapter 3 was compiled and run on the CoolMUC-4 Cluster of the LRZ. The program was executed on the partition cm4_tiny that uses Intel Sapphire Rapids (Xeon Platinum 8480+) nodes and therefore Peano was configured with the following command:

```
./configure --enable-exahype --enable-loadbalancing --enable-
    blockstructured --with-multithreading=omp --with-mpi=mpicxx CC=gcc
     CXX=g++ CXXFLAGS="-Ofast -std=c++23 -W -Wall -Wextra -fopenmp -g -
    funroll-loops -Ofast -g3 -march=sapphirerapids -mtune=
    sapphirerapids -static-libstdc++ -static -Wno-attributes=clang::"
```

The parameters `-march=sapphirerapids` and `-mtune=sapphirerapids` ensure that the code is optimized for the architecture of the used compute nodes of the partition cm_tiny.

Using `-std=c++23` additionally ensures that the 16-bit precision FP16 (`Float16`) and BF16 (`__bf16`) datatypes are supported in the code.

5.2 Accuracy

5.2.1 Method

In this chapter I compare the errors of the four different precision types. In order to see how the differences in errors between the precision types behave with different amounts of cells every scenario is simulated for double, single and both half precision types with different amounts of cells.

The following different amounts of cells are considered in detail:

- Elastic Planar Wave Scenario
  - 450 x 450 cells
  - 2835 x 2835 cells
- Euler Gaussian Bell Scenario
  - 135 x 135 cells
  - 450 x 450 cells
  - 810 x 810 cells
- SWE Resting Lake Scenario
  - 1350 x 1350 cells
  - 4455 x 4455 cells

The computing accuracy of the different implementations is presented as the error in the l^2-norm of the numerical solution with respect to the analytical solution. The l^2-norm [37] for a vector

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$$

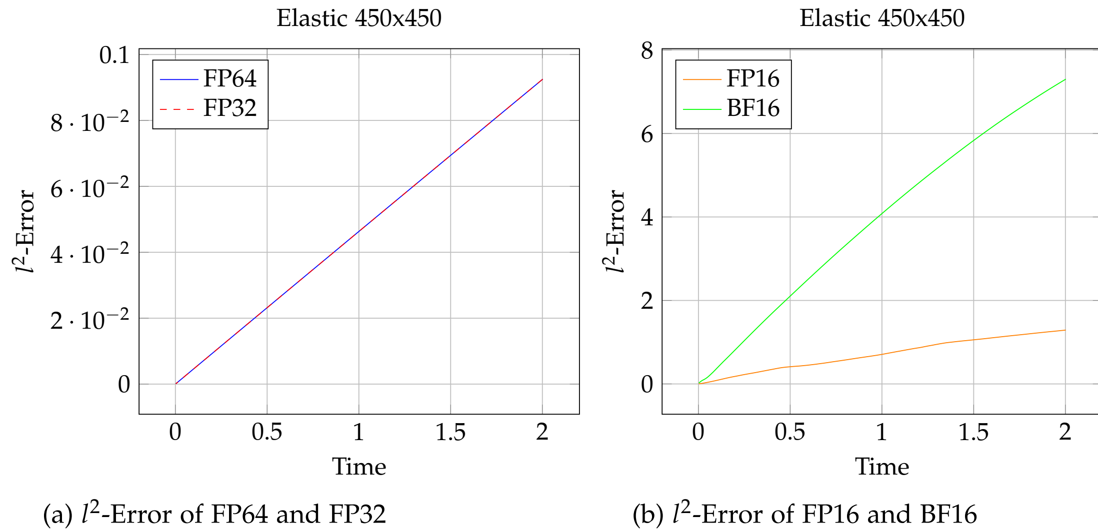
is defined by:

$$\|\mathbf{x}\| = \sqrt{\sum_{k=1}^{n} |x_k|^2} \tag{5.16}$$

5.2.2 Elastic Planar Wave

The Elastic Planar Wave scenario (Section 3.1) is evaluated for either 450 x 450 or 2835 x 2835 cells.

Comparison for 450 x 450 cells



(a) l^2-Error of FP64 and FP32

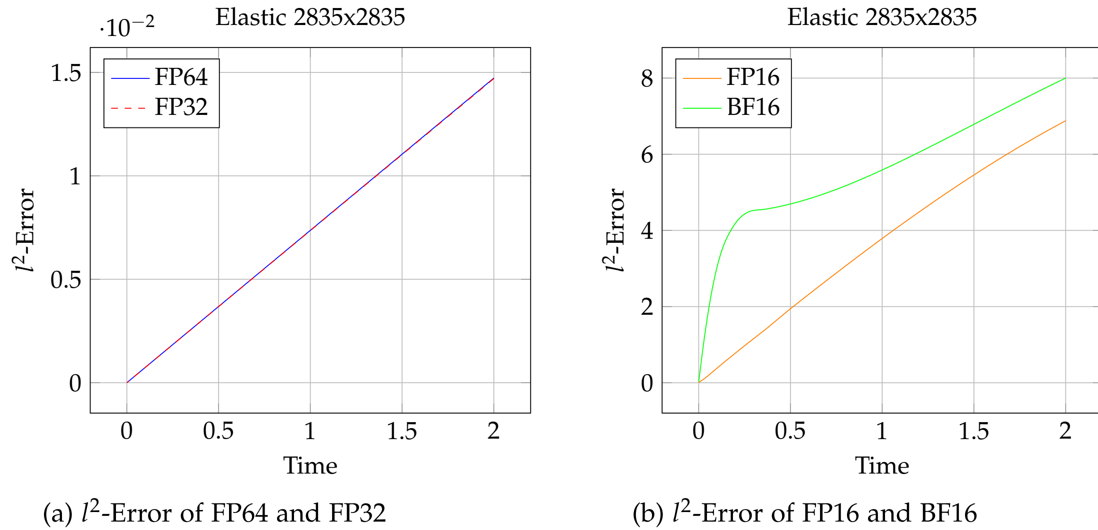
(b) l^2-Error of FP16 and BF16

Figure 5.1: l^2-Error of the Elastic Planar Wave scenario simulated with the finite volume method using 450 x 450 cells in FP64, FP32, FP16 and BF16 precision

Figure 5.1 shows the l^2-errors of FP64, FP32, BF16 and FP16 in case of a domain divided in 450 x 450 cells. FP64 and FP32 show almost the same l^2-errors, increasing linearly over time, i. e. the source of the errors is constant. However the l^2-errors of BF16 and FP16 are much higher, as expected, and they also show a significant difference between each other. While the l^2-error of BF16 shows a slightly decreasing gradient, the FP16 curve is more linear. Additionally, the l^2-error of the BF16 implementation is increasing much faster than the l^2-error of FP16.

In summary, the results of the Elastic Planar Wave scenario with 450 x 450 cells show that the FP32 precision type (like FP64) is sufficient for the computation, while both half-precision types lead to high l^2-errors, increasing over the simulated time.

Comparison for 2835 x 2835 cells



(a) l^2-Error of FP64 and FP32

(b) l^2-Error of FP16 and BF16

Figure 5.2: l^2-Error of the Elastic Planar Wave scenario simulated with the finite volume method using 2835 x 2835 cells in FP64, FP32, FP16 and BF16 precision

When dividing the domain in 2835 x 2835 cells the behavior for the FP64 and FP32 implementations remain roughly the same but the l^2-error is constantly higher than in case of 450 x 450 cells. Figure 5.2a) shows an increase in l^2-error over time but both precision types show the same behavior, meaning again that there is no significant difference between the two implementations.

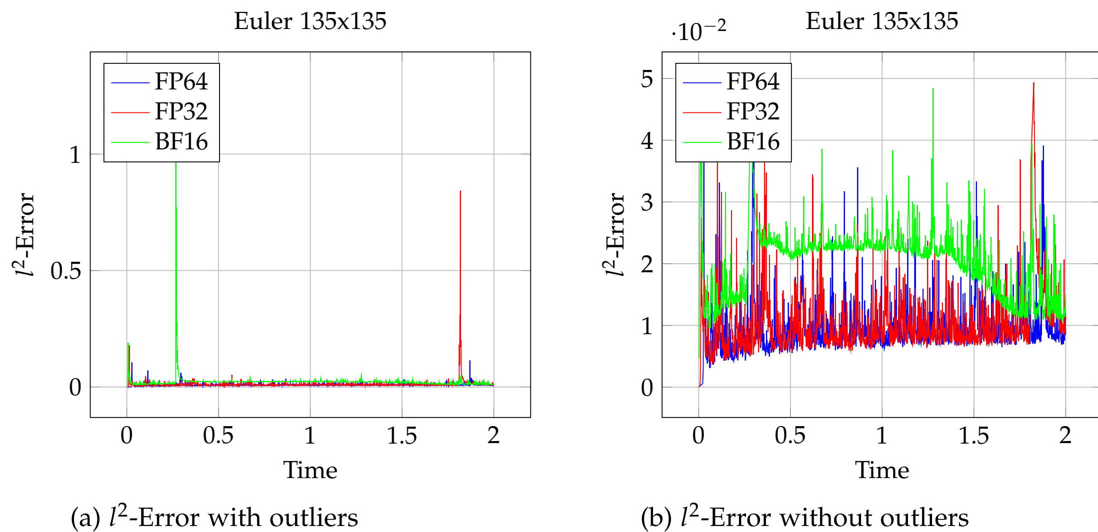
Figure 5.2b shows that the l^2-error of the FP16 implementation still evolves roughly linear. However, the BF16 implementation shows fast growing errors in the beginning and then behaves similar to the FP16 implementation, i. e. the errors grow steadily and more slowly. This behavior could arise because the BF16 implementation is not able to simulate the initial conditions stably and therefore quickly changes to another semi-stable state. Both implementations still show much higher l^2-errors than the FP64 and FP32 implementations.

This simulation confirms the results of the simulation with less cells, namely that there is no significant difference between the FP64 and FP32 precision types, while the half-precision types show much higher l^2-errors, i. e. considerably less accurate solution results.

5.2.3 Euler Gaussian Bell

Simulation of the Euler Gaussian Bell scenario (Section 3.2) showed that FP16 is not accurate enough to complete the computing over the whole simulating time. The computation stopped before reaching the end of the time and the l^2-errors resulted in NAN values (Not A Number). Therefore in the following only FP32, FP64 and BF16 cases are considered. For 810 x 810 cells BF16 is also not considered due to a significantly higher computing time that could not be reached with the used partition of the CoolMUC-4 (see Subsection 5.3.2).

Comparison for 135 x 135 cells



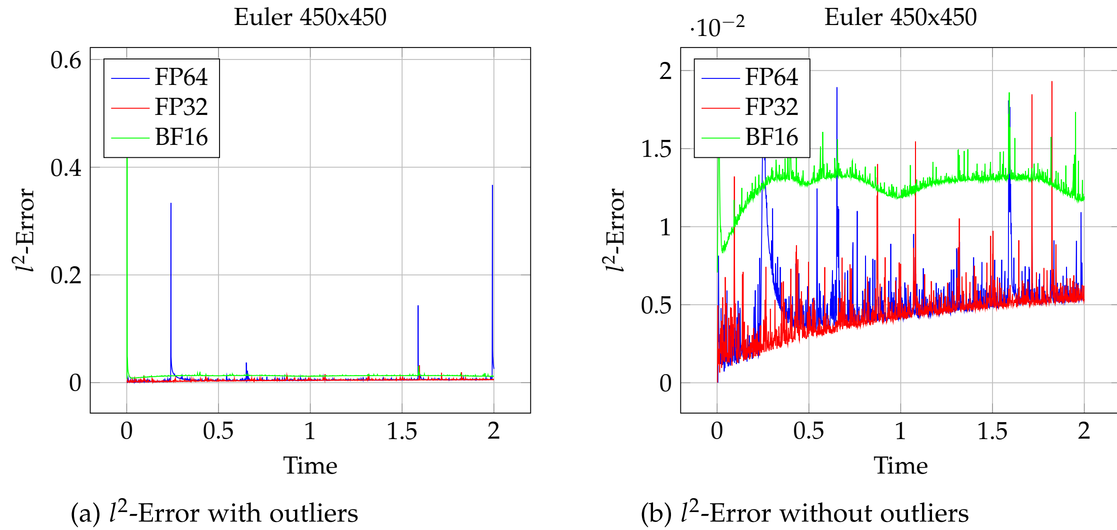
(a) l^2-Error with outliers

(b) l^2-Error without outliers

Figure 5.3: l^2-Error of the Euler Gaussian Bell scenario simulated with the finite volume method using 135 x 135 cells in FP64, FP32 and BF16 precision with outliers (a) and without outliers (b)

Figure 5.3 compares the l^2-error of the FP64, FP32 and BF16 implementations for 135 x 135 cells using the Euler Gaussian Bell scenario. Figure 5.3a shows outliers in the l^2-errors of the FP32 and BF16 implementations. Filtering the outliers in the results yields the error evolution shown in Figure 5.3b. It shows that the FP64 and FP32 implementations are similar in terms of the level of errors, while the BF16 implementation shows higher error values with a peak in the middle of the simulation. All three curves show a strongly oscillating behavior.

Comparison for 450 x 450 cells



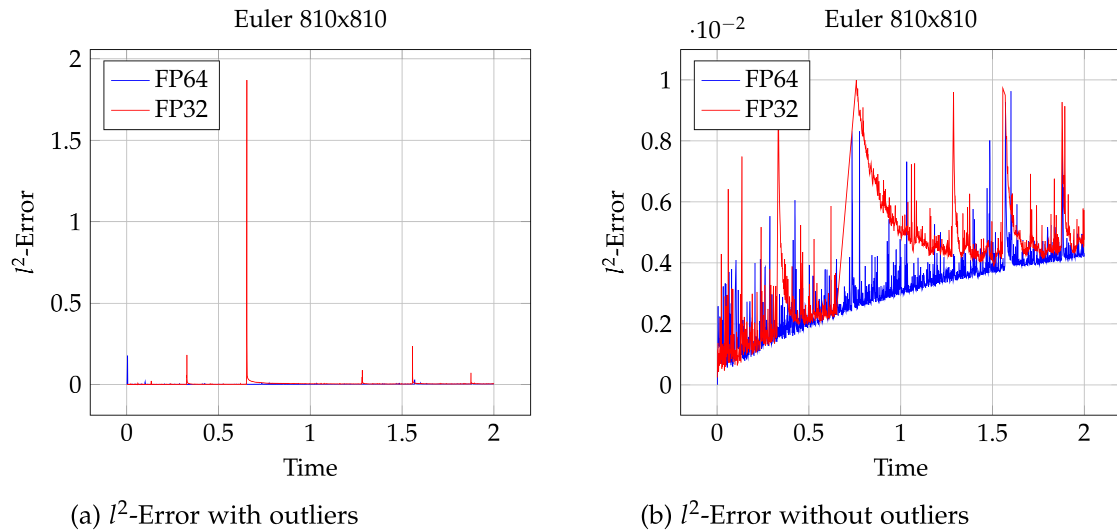
(a) l^2-Error with outliers

(b) l^2-Error without outliers

Figure 5.4: l^2-Error of the Euler Gaussian Bell scenario simulated with the finite volume method using 450 x 450 cells in FP64, FP32 and BF16 precision with outliers (a) and without outliers (b)

Figure 5.4 partially confirms the results of the simulation with 135 x 135 cells. While Figure 5.4a shows that there are still outliers, they now appear in FP64 and BF16 cases but not in the FP32 one anymore. Filtering the outliers, Figure 5.4b confirms that the BF16 implementation yields higher l^2-errors than the FP64 and FP32 implementations, that show similar behavior with increasing errors over time within the same level. All three implementations still show strongly oscillating behavior.

Comparison for 810 x 810 cells



(a) l^2-Error with outliers

(b) l^2-Error without outliers

Figure 5.5: l^2-Error of the Euler Gaussian Bell scenario simulated with the finite volume method using 810 x 810 cells in FP64 and FP32 precision with outliers (a) and without outliers (b)

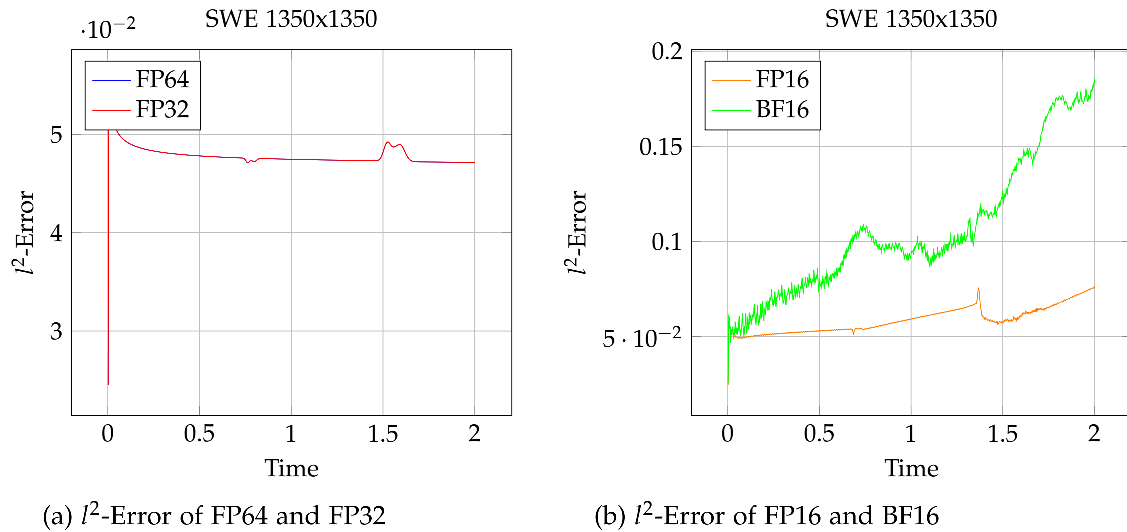
Increasing the amount of cells further to 810 x 810 yields the results shown in Figure 5.5. There are still outliers shown in Figure 5.5a, but they mostly occur in the FP32 implementation again, while the FP64 implementation only shows one smaller outlier in the beginning of the simulation. Again filtering the outliers Figure 5.5b shows a similar evolution of the l^2-error over time for both implementations, increasing over time with a slowly decreasing gradient. However the effect causing the highest outlier in the FP32 implementation at about 0.7 seconds still influences the curve, because the error values slowly decrease after the outlier and approaches the curve of the FP64 implementation.

In summary, the Euler Gaussian Bell scenario could also be computed using the FP32 precision type without significantly increasing the l^2-error. While the outliers shown could be caused by some rounding done in the analytical solution (used as reference in the l^2-error computation) they need further investigations.

5.2.4 SWE Resting Lake

The SWE Resting Lake scenario (Section 3.3) was simulated with 1350 x 1350 and 4455 x 4455 cells.

Comparison for 1350 x 1350 cells



(a) l^2-Error of FP64 and FP32

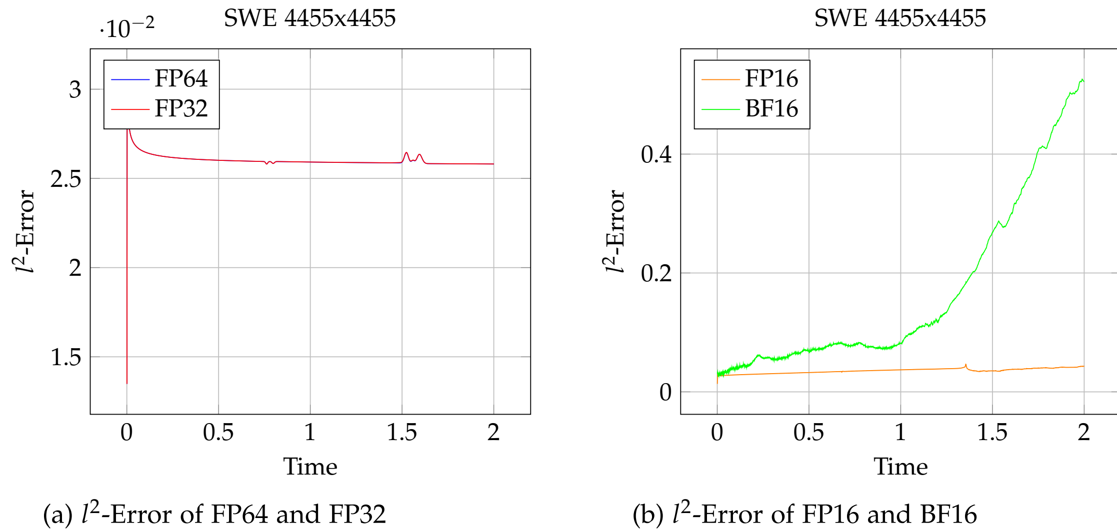
(b) l^2-Error of FP16 and BF16

Figure 5.6: l^2-Error of the SWE Resting Lake scenario simulated with the finite volume method using 1350 x 1350 cells in FP64, FP32, FP16 and BF16 precision

The SWE Resting Lake scenario shows similar behavior as the Elastic Planar Wave and Euler Gaussian Bell scenarios. As Figure 5.6a shows in case of 1350 x 1350 cells the FP64 and the FP32 implementation lead to the same computing accuracy. However, in this scenario there is no constant increase over time but the l^2-error sharply jumps at the very beginning and further remains at a roughly constant level. This can be explained by the immediately appearing waves in the beginning of this scenario. After these the scenario settles into a different steady state that is off by the amount of error that slowly decreases because bigger waves smooth out over time.

The BF16 and FP16 implementations show a higher l^2-error (Figure 5.6b). Confirming the results of the Elastic Planar Wave scenario the BF16 implementation shows higher l^2-errors over the simulated time period than the FP16 implementation. Both increase over the simulation time. This shows that no steady state is reached when using the half-precision datatypes. In contrary to the FP64 and FP32 implementations especially using the BF16 precision yields oscillating l^2-errors which might indicate an oscillating internal state.

Comparison for 4455 x 4455 cells



(a) l^2-Error of FP64 and FP32

(b) l^2-Error of FP16 and BF16

Figure 5.7: l^2-Error of the Euler Gaussian Bell scenario simulated with the finite volume method using 4455 x 4455 cells in FP64, FP32, FP16 and BF16 precision

Also with 4455 x 4455 cells, the SWE Resting Lake scenario still shows the same behavior for the FP64 and FP32 implementations (Figure 5.7a). While the order of magnitude of the l^2-error is still the same, the value is noticeably lower. The curve still shows the same behavior with a sharp jump in the beginning followed by a slow decrease.

Considering the half-precision types the results of the other simulations are confirmed, namely that the BF16 implementation leads to higher l^2-errors than the FP16 implementation (Figure 5.7b). However in this simulation the BF16 l^2-error shows a strongly ascending curve while the FP16 l^2-error still increases very slowly over time.

In summary, also for the SWE Resting Lake scenario the FP32 implementation is sufficient, while the half-precision types show much higher l^2-errors, i. e. do not lead to an accurate result.

5.3 Performance

5.3.1 Method

In this section the computation time of the scenarios is compared using the same amounts of cells as in Section 5.2. Additionally a VTune analysis for the SWE Resting Lake Scenario with 1350 x 1350 cells is carried out. In particular the VTune command `-collect system-overview` provides an overview including the CPU utilization, the memory usage and the time allocation.

5.3.2 Runtime Comparison

Elastic Planar Wave scenario

Amount of Cells	FP64	FP32	FP16	BF16
450 x 450	00:00:59	00:01:06	00:01:14	00:01:25
2835 x 2835	03:53:24	04:03:50	04:19:50	05:47:56

Table 5.1: Runtime Comparison for the Elastic Planar Wave scenario (hh:mm:ss)

The amount of time needed to simulate the Elastic Planar Wave scenario on the CoolMUC-4 is displayed in Table 5.1. Surprisingly the FP64 implementation needed less time to compute the simulated problem than the FP32 one. However the difference between those two precision types' required runtime is very small. The two half-precision cases FP16 and BF16 however needed significantly more time which can be attributed to the fact that these have to be emulated on the CoolMUC-4 because there is no native hardware support for them.

Euler Gaussian Bell scenario

Amount of Cells	FP64	FP32	FP16	BF16
135 x 135	00:00:54	00:00:57	00:00:54	00:01:43
450 x 450	00:20:15	00:20:20	N/A	00:40:27
810 x 810	02:01:52	02:07:23	N/A	timeout

Table 5.2: Runtime Comparison for the Euler Gaussian Bell scenario (hh:mm:ss)

Table 5.2 shows the runtime results for the Euler Gaussian Bell scenario. It confirms the results for the Elastic Planar Wave scenario regarding the FP64 and FP32 implementations as both show similar computing times for each amount of cells. Contrary to the Elastic Planar Wave scenario, the FP16 implementation for the Euler Gaussian Bell scenario shows a similar computing time for 135 x 135 cells as the FP64 and FP32 implementations. Using more cells (450 x 450 and 810 x 810) the FP16 simulation stopped unsuccessfully. The significantly higher computing time of the BF16 implementation shows the same issue of emulation as in the Elastic Planar Wave scenario. With the domain consisting of 810 x 810 cells the computation using BF16 could not be executed as the required computing time exceeds the time limits of the used partition of the CoolMUC-4, i. e. after 8 hours computing time only 0,38 seconds (i. e. 19% of the simulation) were simulated.

SWE Resting Lake scenario

Amount of Cells	FP64	FP32	FP16	BF16
1350 x 1350	00:07:43	00:07:38	00:08:34	00:12:33
4455 x 4455	04:15:48	04:29:39	05:03:56	07:25:33

Table 5.3: Runtime Comparison for the SWE Resting Lake scenario (hh:mm:ss)

The runtimes of the SWE Resting Lake scenario compared in Table 5.3 show the same characteristics as in the other scenarios. While the FP64 and FP32 implementations don't show much difference in computing time, BF16 and FP16 take considerably more time to compute the scenario.

Summary

Overall the analysis of the computing time shows that there is no significant difference between the double and single-precision implementations, while the half-precision implementations take considerably more time, while giving less accurate results.

5.3.3 VTune Analysis for the SWE Resting Lake scenario

In the following a VTune analysis of the SWE Resting Lake scenario with 1350 x 1350 cells is interpreted. Due to the fact that the half-precision types have to be emulated on the CoolMUC-4 and showed considerably worse performance when comparing the total computing time in Subsection 5.3.2, only the FP64 and FP32 implementations are taken into consideration.

Execution Time & CPU Usage

Metric	**FP64**	**FP32**
Elapsed Time (s)	469.613	462.863
Total CPU Time (s)	618.405	594.650
User CPU Time (s)	353.400	309.035
Unknown CPU Time (s)	265.005	285.615

Table 5.4: Comparison of Execution Time and CPU Usage

Table 5.4 shows that the FP32 float format is slightly faster than FP64, with a reduction of approximately 1.4% in elapsed time. Additionally, the total CPU time for FP32 is about 3.8% less than FP64, indicating a more efficient use of CPU resources. In terms

of user CPU time, FP32 spent 44 seconds, i. e. 12.5%, less than FP64, showing better performance in user mode. However, FP32 used 20 seconds, i. e. 7.5%, higher unknown CPU time compared to FP64.

Thread Count

Metric	FP64	FP32
Total Thread Count	2,126	1,850

Table 5.5: Thread Count Comparison

Table 5.5 shows that FP32 uses considerably fewer threads than FP64 (approximately 13% less). This result is somewhat surprising given native hardware support for both.

Memory & Bandwidth Utilization

Metric	FP64	FP32
Total DRAM Bandwidth (GB/sec)	0.151	0.163
Average Read Bandwidth (GB/sec)	0.078	0.083
Average Write Bandwidth (GB/sec)	0.073	0.079
Max Observed DRAM BW (GB/sec)	3.000	4.200
Average PCIe Bandwidth (MB/sec)	0.621	0.620

Table 5.6: Memory and Bandwidth Utilization

The FP32 precision type utilizes approximately 8% more total DRAM bandwidth compared to FP64, indicating a higher demand for memory resources. In terms of read bandwidth, FP32 reads slightly more data, with a marginal increase over FP64. Similarly, write bandwidth is also higher for FP32. The maximum observed DRAM bandwidth for FP32 is significantly higher, with a peak of 4.2 GB/sec compared to FP64's 3.0 GB/sec, suggesting more intensive memory usage during execution. Furthermore, the average PCIe bandwidth of the FP32 is similar to the FP64 version.

Summary

In summary, for the analyzed scenario FP64 and FP32 show rather similar results in terms of computing times, and resource utilization. All results remain within a ±15% range, most of them even below 10% deviation. Some of the comparison results seem rather unexpected but as they are both very close to one another this indicates that there is no significant difference in performance between the FP64 and FP32 implementations.

6 Conclusion

In this thesis I implemented a templated version of the pre-existing Rusanov Global Adaptive Finite Volume Solver in ExaHyPE 2 so that it supports not only 64-bit double precision (FP64) but also 32-bit single precision (FP32) and 16-bit half precision (FP16, BF16) datatypes. Analyzing the performance and the precision of the different implemented computing precisions led to some interesting results.

In the evaluated cases there is no significant difference in terms of accuracy and performance between the use of 64-bit and 32-bit floating point numbers. However using the 16-bit half-precision datatypes has a huge negative impact on the precision of the solver, even though it does not lead to an improvement in performance on the used hardware but on the contrary increases the computing time by quite a margin.

Future work should elaborate more on potential reasons leading to some of the unexpected results. Specifically the implementation could be extended to support other Finite Volume solvers to further evaluate the accuracy and performance over the different precision types. Additionally it could be analyzed more deeply if there are significant changes in memory and storage usage depending on the used precision type. With regard to performance, unexpected behaviors should be more thoroughly analyzed through additional profiling.

Abbreviations

AMR Adaptive Mesh Refinement

API Application Programming Interface

CFD Computational Fluid Dynamics

CM Continuum Mechanics

EWE Elastic Wave Equation

FD Finite Difference

FE Finite Element

FV Finite Volume

PDE Partial Differential Equation

SWE Shallow Water Equations

List of Figures

2.1 Discretization methods with their grids and equations 6
2.2 Peano's code architecture [23] 8
2.3 Spacetree traversal [25] 9
2.4 Flow of simulation (traversals) in Peano [23] 10
2.5 Peano's traversal path in AMR grid [36] 11
2.6 Top-down spacetree decomposition in Peano [36] 12
2.7 ExaHyPE2 architecture [30, 23, 5] 13
2.8 ExaHyPE's halo concept for boundary conditions for the FV solver [21] 13

4.1 Code Structure 20

5.1 l^2-Error of the Elastic Planar Wave scenario simulated with the finite volume method using 450 x 450 cells in FP64, FP32, FP16 and BF16 precision 26
5.2 l^2-Error of the Elastic Planar Wave scenario simulated with the finite volume method using 2835 x 2835 cells in FP64, FP32, FP16 and BF16 precision 27
5.3 l^2-Error of the Euler Gaussian Bell scenario simulated with the finite volume method using 135 x 135 cells in FP64, FP32 and BF16 precision with outliers (a) and without outliers (b) 28
5.4 l^2-Error of the Euler Gaussian Bell scenario simulated with the finite volume method using 450 x 450 cells in FP64, FP32 and BF16 precision with outliers (a) and without outliers (b) 29
5.5 l^2-Error of the Euler Gaussian Bell scenario simulated with the finite volume method using 810 x 810 cells in FP64 and FP32 precision with outliers (a) and without outliers (b) 30
5.6 l^2-Error of the SWE Resting Lake scenario simulated with the finite volume method using 1350 x 1350 cells in FP64, FP32, FP16 and BF16 precision 31
5.7 l^2-Error of the Euler Gaussian Bell scenario simulated with the finite volume method using 4455 x 4455 cells in FP64, FP32, FP16 and BF16 precision 32

List of Tables

2.1 Theory overview [8, 33, 2] 4

3.1 Comparison of Elastic and Non-Elastic Waves 15

4.1 Floating-point precision characteristics 19

5.1 Runtime Comparison for the Elastic Planar Wave scenario (hh:mm:ss) . 33
5.2 Runtime Comparison for the Euler Gaussian Bell scenario (hh:mm:ss) . 33
5.3 Runtime Comparison for the SWE Resting Lake scenario (hh:mm:ss) . . 34
5.4 Comparison of Execution Time and CPU Usage 34
5.5 Thread Count Comparison 35
5.6 Memory and Bandwidth Utilization 35

Bibliography

[1] S. Daum and A. Borrmann. "Efficient and robust octree generation for implementing topological queries for building information models." In: *Proc. of the EG-ICE Workshop on Intelligent Computing in Engineering*. 2012.

[2] J. L. Davis. *Wave Propagation in Solids and Fluids*. Springer, 1988.

[3] M. Dumbser and D. S. Balsara. "A new efficient formulation of the HLLEM Riemann solver for general conservative and non-conservative hyperbolic systems." In: *Journal of Computational Physics* 304 (2016), pp. 275–319.

[4] M. Dumbser and M. Käser. "An arbitrary high-order discontinuous Galerkin method for elastic waves on unstructured meshes — II. The three-dimensional isotropic case." In: *Geophysical Journal International* 167.1 (Oct. 2006), pp. 319–336.

[5] ExaHyPE Development Team, LRZ, TU Munich. *ExaHyPE 2 - Software Documentation*. LRZ, TU Munich. Accessed: 2025-02-01. 2025. URL: `https://hpcsoftware.pages.gitlab.lrz.de/Peano/d3/d82/page_exahype2_home.html`.

[6] J. H. Ferziger and M. Perić. *Numerische Strömungsmechanik*. Springer, 2008.

[7] J. H. Ferziger, M. Perić, and R. L. Street. *Computational Methods for Fluid Dynamics*. Fourth edition. Springer, 2020.

[8] P. García-Navarro, J. Murillo, J. Fernández-Pato, I. Echeverribar, and M. Morales-Hernández. "The shallow water equations and their application to realistic cases." In: *Environmental Fluid Mechanics* 19 (2019), pp. 1235–1252.

[9] J. G. Harris. *Linear Elastic Waves*. Cambridge University Press, Oct. 2001.

[10] N. J. Higham. *Half Precision Arithmetic: FP16 versus BFloat16*. Accessed: 2025-02-02. 2018. URL: `https://nhigham.com/2018/12/03/half-precision-arithmetic-fp16-versus-bfloat16/`.

[11] N. J. Higham, S. Pranesh, and M. Zounon. *Squeezing a Matrix into Half Precision, with an Application to Solving Linear Systems*. Tech. rep. 2018.30. MIMS Preprint. Manchester Institute for Mathematical Sciences, 2018.

[12] M. Hobbhahn and T. Besiroglu. *Trends in GPU Price-Performance*. Epoch AI. Accessed: 2025-02-12. 2022. URL: `https://epoch.ai/blog/trends-in-gpu-price-performance`.

[13] *IEEE Standard for Floating-Point Arithmetic*. Accessed: 2025-02-02. IEEE, 2008.

[14] M. Ioriatti, M. Dumbser, and R. Loubère. "A Staggered Semi-implicit Discontinuous Galerkin Scheme with a Posteriori Subcell Finite Volume Limiter for the Euler Equations of Gasdynamics." In: *Journal of Scientific Computing* 83.27 (2020).

[15] L. G. Jack Dongarra and N. J. Higham. "Numerical Algorithms for High-Performance Computational Science." In: *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 378.20190066 (2019).

[16] M. Käser and M. Dumbser. "An arbitrary high-order discontinuous Galerkin method for elastic waves on unstructured meshes – I. The two-dimensional isotropic case with external source terms." In: *Geophysical Journal International* 166.2 (2006), pp. 855–877.

[17] P. Milbradt and W. A. Abed. "Generalized Stabilization Techniques in Computational Fluid Dynamics." In: *ICHE 2008. Proceedings of the 8th International Conference on Hydro-Science and Engineering*. Ed. by S. S. Y. Wang. Nagoya, Japan: Nagoya Hydraulic Research Institute for River Basin Management, Sept. 2008.

[18] G. Müller. *Theory of Elastic Waves*. Potsdam, Frankfurt, Hamburg, 2007.

[19] Oak Ridge National Laboratory. *Frontier Supercomputer Debuts as World's Fastest, Breaking Exascale Barrier*. Accessed: 2025-02-10. 2022. URL: `https://www.ornl.gov/news/frontier-supercomputer-debuts-worlds-fastest-breaking-exascale-barrier`.

[20] OpenFOAM community. *OpenFOAM - Finite volume*. CFD-online.com. Accessed: 2025-02-11. 2025. URL: `https://openfoam.cfd-online.com/Wiki/Finite_volume`.

[21] Peano Development Team. *ExaHyPE 2 Numerics: Finite Volumes*. LRZ, TU Munich. Accessed: 2025-02-08. URL: `https://hpcsoftware.pages.gitlab.lrz.de/Peano/db/dfe/page_tutorials_exahype2_numerics_finite_volumes.html`.

[22] Peano Development Team. *Finite Volumes*. LRZ, TU Munich. Accessed: 2025-02-01. 2025. URL: `https://hpcsoftware.pages.gitlab.lrz.de/Peano/db/dfe/page_tutorials_exahype2_numerics_finite_volumes.html`.

[23] Peano Development Team. *Peano - Architecture*. LRZ, TU Munich. Accessed: 2025-02-01. 2025. URL: `https://hpcsoftware.pages.gitlab.lrz.de/Peano/db/d3f/page_architecture_home.html`.

[24] Peano Development Team. *Peano - Software Documentation*. LRZ, TU Munich. Accessed: 2025-02-01. 2025. URL: `https://hpcsoftware.pages.gitlab.lrz.de/Peano/index.html`.

[25] Peano Development Team. *Peano - Tutorial*. LRZ, TU Munich. Accessed: 2025-02-01. 2025. URL: `https://hpcsoftware.pages.gitlab.lrz.de/Peano/d3/d1a/tutorials_peano4_tutorials.html`.

[26] Peano Development Team. *Peano 4 - Software Documentation*. LRZ, TU Munich. Accessed: 2025-02-01. 2025. URL: https://hpcsoftware.pages.gitlab.lrz.de/Peano/dd/da2/page_peano4_home.html.

[27] Peano Development Team. *Tutorial - The Elastic Wave Equation*. LRZ, TU Munich. Accessed: 2025-02-03. 2025. URL: https://hpcsoftware.pages.gitlab.lrz.de/Peano/dc/da8/page_exahype2_tutorials_toyproblems_elastic.html.

[28] Peano Development Team. *Tutorial - The Euler Wave*. LRZ, TU Munich. Accessed: 2025-02-03. 2025. URL: https://hpcsoftware.pages.gitlab.lrz.de/Peano/dc/da8/page_exahype2_tutorials_toyproblems_euler.html.

[29] Peano Development Team. *Tutorial - The Shallow Water Equation*. LRZ, TU Munich. Accessed: 2025-02-03. 2025. URL: https://hpcsoftware.pages.gitlab.lrz.de/Peano/dc/da8/page_exahype2_tutorials_toyproblems_swe.html.

[30] A. Reinarz, D. E. Charrier, M. Bader, L. Bovard, M. Dumbser, K. Duru, F. Fambri, A.-A. Gabriel, J.-M. Gallard, S. Köppel, L. Krenz, L. Rannabauer, L. Rezzolla, P. Samfass, M. Tavelli, and T. Weinzierl. "ExaHyPE: An engine for parallel dynamically adaptive simulations of wave problems." In: *Computer Physics Communications* 254 (2020).

[31] SabrePC. *Explaining Floating Point Precision - What is FP32? What is FP64?* Accessed: 2025-02-08. Nov. 2023. URL: https://www.sabrepc.com/blog/deep-learning-and-ai/explaining-floating-point-precision-fp32-fp64.

[32] R. Schwarze. *CFD-Modellierung: Grundlagen und Anwendungen bei Strömungsprozessen*. Springer, 2013.

[33] SEG Wiki contributors. *Mathematical foundation of elastic wave propagation*. Society of Exploration Geophysicists. Accessed: 2025-02-11. URL: https://wiki.seg.org/wiki/Mathematical_foundation_of_elastic_wave_propagation.

[34] S. Wang and P. Kanwar. *BFloat16: The Secret to High Performance on Cloud TPUs*. Google. Accessed: 2025-02-02. 2019. URL: https://cloud.google.com/blog/products/ai-machine-learning/bfloat16-the-secret-to-high-performance-on-cloud-tpus.

[35] T. Weinzierl. *A Framework for Parallel PDE Solvers on Multiscale Adaptive Cartesian Grids*. Jan. 2009.

[36] T. Weinzierl. "The Peano Software—Parallel, Automaton-based, Dynamically Adaptive Grid Traversals." In: *ACM Trans. Math. Softw.* 45.2 (Apr. 2019).

[37] E. W. Weisstein. *L^2-Norm*. MathWorld–A Wolfram Web Resource. Accessed: 2025-02-14. URL: https://mathworld.wolfram.com/L2-Norm.html.