Technische Universität München
TUM School of Computation, Information and Technology

Adapting and Optimizing High Order Seismic Simulations for GPU-based Supercomputers

Ravil Dorozhinskii

Vollständiger Abdruck der von der TUM School of Computation, Information and Technology der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitz: Prof. Dr. Rüdiger Westermann

Prüfende der Dissertation:

1. Prof. Dr. Michael Georg Bader
2. Prof. Dr. Harald Köstler

Die Dissertation wurde am 19.12.2023 bei der Technischen Universität München eingereicht und durch die TUM School of Computation, Information and Technology am 23.05.2024 angenommen.

Acknowledgements

Firstly, I thank my advisor, Univ.-Prof. Dr. Michael Bader, for his constant support, mentoring, and guidance throughout my research. This project gave me an excellent opportunity to expand my knowledge in several orthogonal directions and meet many wonderful and inspiring people on my way.

My special thanks go to Lukas Krenz, Sebastian Wolf, Carsten Uphoff, and Thomas Ulrich, with whom I have worked on SeisSol all these years. I thank them for those countless hours of discussions and debates from which I learned a lot, especially at the beginning of my research. Thanks to David Schneller, Mario Marc Marot-Lassauzaie, and Mario Wille for their feedback on the drafts of this manuscript, and I wish them all the best with their research.

I am grateful to Gonzalo Brito Gadeschi and Patrick Atkinson from Nvidia for sharing their expert knowledge and experience in heterogeneous computing, which resulted in a very productive collaboration between academia and industry. It is worth mentioning that the results of some experiments presented in this work were conducted by Gonzalo and Patrick on Nvidia's Selene supercomputer. Additionally, I would like to thank Piero Lanicara from the CINECA supercomputing center and Arnau Folch from the Spanish National Research Council for their guidance, management, and support over the entire ChEESE project, without which this research may not be possible.

Thanks to my family for their continuous support, patience, and encouragement. I also thank Hasan Ashraf for listening to my endless ideas and complaints regarding my research work and for his assistance in proofreading some of my manuscripts.

Lastly, I would like to thank Nehil Danış for her constant care and support.

Abstract

An in-depth study of a single extreme-scale earthquake event may involve multiple highly resolved 3D numerical simulations, which may require immense supercomputing power. In the past, most supercomputers were predominantly CPU-based machines, but this trend has changed in recent years. GPU-accelerated heterogeneous supercomputers are gradually replacing traditional homogeneous systems. This necessitates changes in software design and algorithms to efficiently utilize the power of GPU-based supercomputing systems.

The goal of this study is to adapt and optimize an open-source, highly tuned CPU-based scientific application designed for simulating seismic wave propagation and earthquake dynamics for leading distributed multi-GPU supercomputers. The application consists of multiple kernels and makes use of the discontinuous Galkerin method with cluster-wise ADER Local Time Stepping (LTS) algorithm, and source code generation. Due to the specifics of the software design, I address the source code and performance portability differently for each kernel. I investigate performance bottlenecks and tuning parameters of the adapted algorithms using the roofline model, as well as strong and weak scaling studies. I demonstrate my final results by performing simulations of three real production earthquake scenarios on the LUMI and Leonardo supercomputers.

In this study, I show how the original task decomposition needs to be changed to efficiently utilize massively parallel processors. I demonstrate that the GPU performance of the ADER-DG method can be significantly improved by fusing subsequent batched matrix multiplication kernels, which I implemented as a part of the source code generation process. I show that, on average, fusion results in 2-2.5x speed-up when comparing the GPU and CPU versions of the code on a single HPC GPU and a single 48-core AVX512 CPU system. I also compare the OpenMP and SYCL standards and conclude that the latter results in better source code and performance portability for data processing on GPUs.

In addition, I show that the strong scaling parallel efficiency of the LTS algorithm drops faster on distributed multi-GPU systems than on distributed-memory CPU machines. I conclude that, in general, the performance depends on the computational throughput of a computing device and the distribution of mesh elements between LTS clusters. I show that the GPU throughput rapidly drops starting at a particular problem size, whereas CPU performance stays almost flat within the entire test range. In the end, I share ideas on how the LTS algorithm can be improved and hope that it can help other researchers to continue the investigation and come up with an improved version of the algorithm.

Contents

1. Introduction

Seismology is the science of earthquakes and the propagation of waves through the Earth. An earthquake occurs due to the sudden release of the accumulated potential energy at a particular location inside the Earth. Usually, the energy gets accumulated due to elastic deformations caused by frictional forces resisting the movements of plates, which the Earth's lithosphere consists of, along their common boundaries - i.e., the faults. The released potential energy converts to seismic waves, which travel through the Earth and, eventually, some of them reach the Earth's surface, causing ground shaking.

Ground shaking by itself is not as dangerous for the Earth's inhabitants as the consequences of it, especially when it occurs around densely populated areas like cities. The shaking can 1) cause destruction of buildings, 2) trigger tsunamis, liquefaction, landslides, etc., 3) lead to a fire in the case of broken power or gas supply lines, 4) contaminate the water in the case of damaged sewage systems, or 5) result in various combinations of the abovementioned events. Consequently, it may result in severe injuries, fatalities, substantial economic losses, and even ecological catastrophes. Earthquakes of different magnitudes happen more frequently than one might imagine. For example, Daniell and Vervaeck in [21] listed 91 damaging earthquakes that occurred worldwide only in 2010, ranging from M_w 4.3 to M_w 8.8.

Currently, seismologists cannot precisely predict locations, magnitudes, and the exact timing of earthquakes. However, in-depth studies of past events can help to minimize the consequences of upcoming quakes. For example, the results of such studies can contribute to the designs of hazard maps, evacuation and rescue plans, early warning systems, etc. An analysis of the history of seismic activities, fault zones, and ground characteristics at a particular region can assist engineers in city planning, designing buildings and infrastructure, selecting suitable construction materials, etc. Moreover, such studies advance scientific knowledge and lead to a better understanding of fault behavior and earthquake patterns, which help to anticipate earthquakes probabilistically. For this, seismologists need proper tools to conduct such studies.

There exist many approaches for modeling earthquakes, which can be divided into two broad categories: kinematic and dynamic ones. Kinematic models are based on solving data-driven inversion problems, which aim to closely fit observational data. Such models result in a vast free parameter space, which must be constrained to obtain physically plausible results. Dynamic models aim to reproduce the yielding and sliding processes of the rupture during an earthquake; they help to reveal how earthquakes start, evolve, and stop. In contrast to kinematic models, dynamic ones consider mechanical interactions between the rupture processes and seismic waves, which result in solving complex, coupled,

non-linear problems; the propagating rupture generates seismic waves, which travel through the Earth, causing other parts of the fault to rupture. Therefore, dynamic models are well-suited for in-depth research and studies of earthquakes. Both approaches involve complex and computationally intensive numerical simulations, often requiring supercomputing power.

In May 2022, the first exascale computer, Frontier[1], was put into service, which opened a new era in High Performance Computing (HPC). During acceptance tests, the aggregated performance measured on the supercomputer exceeded 1.1 DP-EFLOP/s while running the High-Performance Linpack (HPL) benchmark, making it the current world record holder. The engineers managed to achieved such outstanding results by incorporating the latest advances in heterogeneous computing technologies. Each Frontier node is equipped with four *AMD* MI250x GPUs. The peak performance of each accelerator is approximately 47.9 DP-TFLOP/s. Thus, the theoretical peak of each supercomputing node is close to 192 DP-TFLOP/s.

Heterogeneous computing refers to data processing on a system consisting of multiple kinds of processing units, which usually operate on different sets of instructions. Such systems typically contain a single host processor, which controls and manages computational or service tasks, and several accelerating devices, which execute them. In general, CPUs, GPUs, FPGAs, or a custom-designed ASIC circuits can be used as accelerators. However, nowadays, the most commonly used ones are GPUs designed for general-purpose programming.

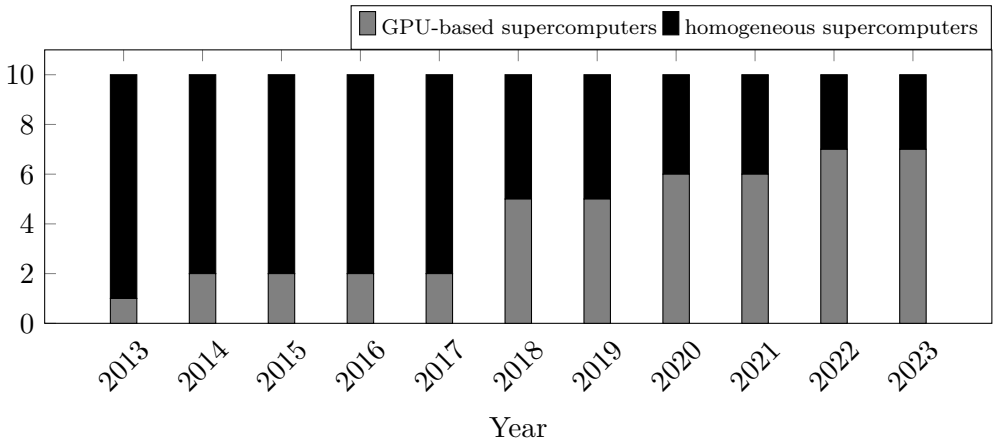


Figure 1.1.: Evolution of the top 10 faster supercomputers in the world according to the TOP500 list over the last ten years.

The number of GPU-based supercomputers has noticeably increased over the last decade. Fig. 1.1 shows how distributed multi-GPU systems have been gradually replacing the leading homogeneous CPU-based supercomputers since 2013. Today, seven out of ten of the most powerful supercomputers are GPU-accelerated, whereas ten years ago, the ratio was only one to nine. By extrapolating the data, one can assume that the number of heterogeneous computing systems will keep growing. If the trend remains and no

[1] https://www.olcf.ornl.gov/frontier/

particular measures are taken, many important scientific applications and libraries will not be able to take advantage of the top tier of supercomputers in a few years. Thus, it becomes worth investing time, effort, and resources in adapting HPC applications for heterogeneous computing environments. In some cases, it is not possible to determine in advance whether a particular scientific application, which usually consists of many different algorithms, will perform well on accelerators or not. Sometimes, a certain answer can be obtained only after exhaustive research, which may involve considerable changes in the source code of an application.

This work sits at the intersection of two scientific domains: dynamic earthquake modeling and heterogeneous computing. The former is represented by the open-source software *SeisSol*[2], designed for simulating seismic wave phenomena and earthquake dynamics. The latter concerns only GPU computing platforms. At this moment, it is unclear whether GPUs will preserve their leadership in heterogeneous computing or whether a new type of accelerator will replace them in the future. However, the heterogeneous programming principles will likely stay the same.

SeisSol operates on unstructured tetrahedral meshes, which are well-suited for discretizing geometrically complex 3D structures, for example, high-resolution topography, complex fault networks, curved subduction interfaces, etc. The main computational core of the application is based on the Discontinuous Galerkin (DG) method and the Arbitrary high-order DERivatives (ADER) explicit time integration scheme. *SeisSol* allows users to set almost an arbitrary convergence order for spatial and time numerical integrations, which can be used to obtain either fast approximations or more accurate numerical results for a given problem. The application features a cluster-wise Local Time Stepping (LTS) scheme, which reduces redundant computations and, thus, results in algorithmic speed-up. *SeisSol* allows users to set heterogeneous material properties for each part of a computational domain and provides a wide range of models: 1) elastic and viscoelastic wave propagation, 2) off-fault plasticity, 3) elastic-acoustic coupling, 4) kinematic point sources, 5) dynamic rupture, and 6) multiple friction laws of rocks. *SeisSol* is a homogeneous HPC application that efficiently utilizes SIMD units, cache hierarchy, multithreading, asynchronous communication, and parallel I/O features. The application scales up to several thousands of CPU nodes, reaching 40-50% of the peak CPU performance on each running process (for example, see [117, 116, 68, 123]).

In the last ten years, several outstanding results have been achieved with *SeisSol*. For example, in 2014, Heinecke et al. [50] presented simulations of the 1992 Landers earthquake, which involved a complex fault system consisting of multiple overlapping fault segments. The simulations revealed highly detailed rupture evolution and ground motion at frequencies up to 10 Hz. The work included a full-machine run on the Stampede supercomputer - i.e., the 7th fastest supercomputer in 2014 according to the TOP500 list - resulting in about 2 DP-PFLOP/s, which was equal to approximately 23.4% of the peak machine performance. As mentioned by the authors, the numerical results obtained during their study helped them to gain a better understanding of the rupture transferring mechanisms between adjacent fault segments during the earthquake.

[2] https://seissol.org

Another example is the simulation of the 2004 Sumatra-Andaman earthquake, as presented by Uphoff et al. in their 2017 work [117]. According to the authors, it was one of the largest and longest dynamic rupture simulations at the time. The simulation took approximately 13.9 hours and utilized all 3072 nodes of the SuperMUC Phase 2 supercomputer, which was ranked as the 28th most powerful machine in the world in 2016, according to the TOP500 list. This full-machine run resulted in 0.94 DP-PFLOPS - i.e., approximately 35% of the HPL benchmark performance. The authors said that the high-resolution seafloor displacement obtained during that simulation served as the input data for their follow-up research focused on studying time-dependent tsunami generation and propagation caused by the earthquake.

Another example worth mentioning is the simulation of the 2018 Palu, Sulawesi earthquake and tsunami presented by Krenz et al. in 2021 [68]. The authors contributed to *SeisSol* by adding ocean dynamics, elastic-acoustic coupling, and a gravitational free-surface boundary condition. The extensions allowed the authors to develop a fully-coupled scenario, which revealed the dynamics of the entire tsunami-genesis in a single run. As noted by the authors, such simulations open up possibilities to fundamentally improve the understanding of earthquake-tsunami interaction in its full complexity. The final simulation included a half-billion element mesh and was executed on 3072 nodes of the SuperMUC-NG supercomputer - i.e., the 13th faster supercomputer in 2020 according to the TOP500 list - resulting in approximately 3 DP-PFLOP/s.

The mentioned examples should convince the reader that *SeisSol* is an important and very promising application in the field of computational seismology. This study has three primary goals: 1) to adapt *SeisSol* to GPU computing platforms, 2) to explore possible performance bottlenecks and tuning parameters, and 3) to evaluate the whole application's performance on distributed multi-GPU computing systems using real production earthquake scenarios. This work has one major constraint: any software changes must not deteriorate the original high CPU performance. Thus, performance portability is a very important concern in this work. *SeisSol* has a huge configuration space, which needs to be constrained to make the research feasible. Therefore, in this study, I consider wave propagation through elastic isotropic materials, as well as dynamic rupture and off-fault plasticity models.

SeisSol uses the *YATeTo*[3] DSL for generating highly efficient micro-kernels for computational sub-tasks. In *SeisSol*, a sub-task is an element local computation, which can consist of multiple tensor operations - i.e., a tensor expression. Data processing of an element can follow one of many execution paths - e.g., due to different boundary conditions. Thus, a CPU task can consist of different kinds of sub-tasks. The original software design delegates 1) sub-task scheduling and 2) dispatching data to a correct set of micro-kernels to the host application - i.e., *SeisSol*. The computations resulting from the ADER-DG method involve operations on small-sized tensors. Thus, data processing of a single sub-task on a GPU cannot fully utilize all its available hardware resources - e.g., streaming multiprocessors. Therefore, finding an optimal task decomposition for GPUs is the first and most important step of this study.

[3] https://github.com/SeisSol/yateto

There exist many publications investigating various GPU implementations of the DG method applied to different systems of hyperbolic PDEs - e.g., different variants of 1) compressible and incompressible Navier-Stokes equations [104, 39, 127, 58, 65, 15], 2) shallow water equations [41, 122, 126], 3) wave equations [89, 88, 16, 87], and 4) Maxwell's equations [66, 43, 7]. Algorithmically, the method boils down to element local computations, which operate on small-sized tensors. In this work, the maximum tensor rank equals 2 due to the structure of the underlying system of PDEs of the elastic wave propagation problem; thus, my primary focus is on sequences of small matrix multiplications. In contrast to the above-listed works, where the authors had opportunities to invest time to manually tune their GPU kernels, my solutions must be integrated into the code generation step, which is a key part of *SeisSol*'s software design. This approach can have a potential advantage because it can naturally address the portability aspects. By and large, the goal of this part of the study is to find and implement an optimal code-generation technique to achieve high-performance solutions for the ADER-DG method, considering the vast configuration space of matrix multiplication kernels - e.g., data types, matrix sizes, etc.

According to the literature review performed during this study, there exist just a few publications related to the strong scaling behavior of the LTS algorithm on distributed multi-GPU systems. The most comprehensive results were obtained by Rietmann in [98, 97] and in his PhD thesis [96]. The author observed that the average GPU performance dropped rapidly during scaling. Rietmann suggests that it happened due to a small number of elements in the finest mesh refinement level, which could not keep the GPUs adequately busy to mask the overhead of setting up and launching GPU tasks [97]. In this work, I dive deeper and attempt to establish a dependency between the computational throughput of a processing unit, which, in general, depends on the problem size and the distribution of mesh elements between LTS clusters.

This thesis is structured as follows. In Chapter 2, I introduce the governing equations for elastic wave propagation, dynamic rupture, and off-fault plasticity models. The discussion also contains a brief explanation of the mathematical representation of kinematic point sources. In Chapter 3, I explain the ADER-DG method, the LTS algorithm, and numerical implementations of the most relevant boundary conditions. Chapter 4 describes the state of *SeisSol* prior to this study. It includes several important discussions, such as 1) data memory layout, 2) code generation, 3) multithreading, and 4) the implementation of message-passing for distributed-memory computing systems. In Chapter 5, I explain the most relevant details of modern GPU hardware architectures and list the most popular GPU programming models in the field of HPC at the moment of writing. At the end, I explain the mechanism involved in launching GPU tasks, which can help to understand the associated overheads.

Starting from Chapter 6, I describe main contributions this work. In Chapter 6, I explain the GPU implementation of *SeisSol*'s wave propagation solver in detail. The discussion includes topics such as 1) memory management, 2) changes in task decomposition, 3) GPU code generation, 4) concurrent and graph-based GPU execution, 5) changes in message-passing, 6) extensions in mesh partitioning, 7) portability, etc. The most prominent and worth-mentioning discussions are the implementation of the fused GEMM kernels and the influence of clustering on the strong scaling performance of the LTS algorithm. The

chapter also contains roofline model analysis of generated GPU kernels, as well as strong and weak scaling studies. In Chapters 7 and 8, I explain the key details of the GPU implementations of the dynamic rupture solver and the return-mapping algorithm. The latter is used for modeling the plastic behavior of the material in the wave propagation domain. Both chapters discuss decompositions of GPU tasks, explain how source code portability was addressed, and present verifications of numerical results obtained with the GPU version of *SeisSol* using several earthquake benchmark scenarios. Chapter 9 presents numerical simulations of three production earthquake scenarios conducted on the LUMI and Leonardo supercomputers, rated as the 3rd and the 4th most powerful supercomputers in 2023 according to the TOP500 list, respectively. Both supercomputers are distributed multi-GPU systems. LUMI is accelerated with the *AMD* MI250x GPUs, whereas Leonardo uses custom-built *Nvidia* A100 graphics cards. In this chapter, I reproduce the 2023 Kahramanmaraş earthquake, the 2019 Ridgecrest earthquake, and the 2018 Palu, Sulawesi earthquake and tsunami using the artifacts of several scientific publications.

Finally, in Chapter 10, I summarize the key results of this thesis and share my ideas, which can be used as a basis for follow-up research and studies.

2. Governing Equations for Earthquake Modeling

Earthquake modeling is a coupled problem. The energy released during local fault cracking is converted to seismic waves which start traveling along the fault, causing other parts of the fault to crack. The process repeats until the total energy released by an earthquake completely dissipates, and thus no further local fault cracking occurs. In this chapter, I establish the main governing equations required for earthquake modeling.

I only consider the elastic wave propagation in isotropic materials in this work. In Section 2.1, I derive a linear hyperbolic system of PDEs that describes the propagation process. In Section 2.2, I focus on dynamic rupture modeling and mathematically describe it using Coulomb's model of friction. The sections also contains examples of the friction laws widely used in computational seismology. The chapter proceeds with a discussion of kinematic point sources commonly used as an alternative to the friction-based approach (see Section 2.3). Finally, I describe the off-fault plasticity model in Section 2.4, which takes into account inelastic energy dissipation. As mentioned by Dunham et al. in [33], this limits unreasonably high slip velocities on the fault and thus results in more accurate earthquake simulations.

2.1. Elastic Wave Propagation

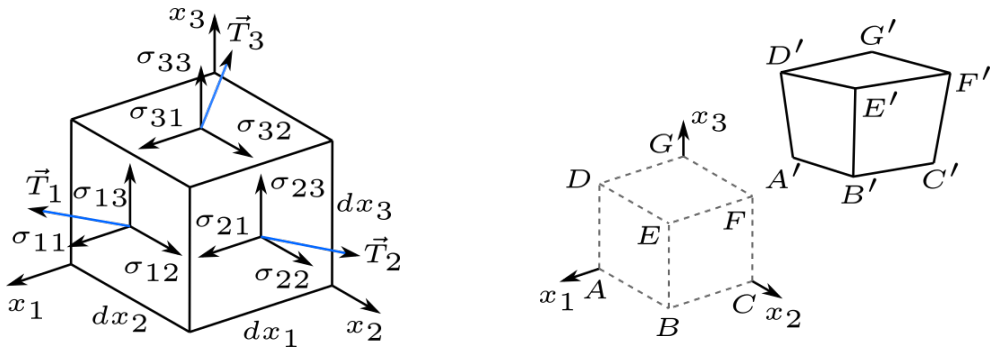


Figure 2.1.: Components σ_{ij} of the stress tensor resulting from the decomposition of $\vec{T}_1$, $\vec{T}_2$ and $\vec{T}_2$ forces (on the left) and the displacement of an infinitesimal cubic element caused by deformation and a rigid body movement (on the right).

Elastic wave propagation is governed by the time derivative of Hooke's law (Eq. 2.1) and the dynamic relationship between accelerations and stresses (Eq. 2.4) - i.e., Newton's second law.

$$\frac{\partial \sigma_{ij}}{\partial t} = C_{ijkl} \frac{\partial \epsilon_{kl}}{\partial t} \tag{2.1}$$

where C_{ijkl} is the stiffness tensor; σ_{ij} are the normal and sheer components of the stress tensor $\boldsymbol{\sigma}$; ϵ_{kl} are the normal and sheer components of the strain tensor $\boldsymbol{\epsilon}$ - i.e., displacements of particles in a solid body relative to a reference point.

The stiffness tensor C_{ijkl} takes a compact form for isotropic materials [79], namely

$$C_{ijkl} = \lambda \delta_{ij} \delta_{kl} + \mu \left(\delta_{ik} \delta_{jl} + \delta_{il} \delta_{jk} \right) \tag{2.2}$$

where δ_{ij} is the Kronecker delta function which is equal to 1 if $i = j$, and 0 otherwise; λ and μ are Lamé parameters. In the general case, λ, μ and ρ are functions of space and greater than zero at any point.

Therefore, Eq. 2.1 for an isotropic material can be written as

$$\frac{\partial \sigma_{ij}}{\partial t} = \lambda \delta_{ij} \frac{\partial \epsilon_{kk}}{\partial t} + 2\mu \frac{\partial \epsilon_{ij}}{\partial t} \tag{2.3}$$

Newton's second law written in the differential form is given by

$$\rho \frac{\partial u_j}{\partial t} = \frac{\partial \sigma_{ij}}{\partial x_i} \tag{2.4}$$

where ρ is the material density; u_j is a particle velocity which is equal the a rate of the infinitesimal particle displacement d_j along x_j direction i.e., ($u_j = \frac{\partial d_j}{\partial t}$).

According to the theory of continuum mechanics (e.g., see [80]), the relation between strains ϵ_{kl} and particle displacements d_i is given by

$$\epsilon_{kl} = \frac{1}{2} \left(\frac{\partial d_l}{\partial x_k} + \frac{\partial d_k}{\partial x_l} \right) \therefore \frac{\partial \epsilon_{kl}}{\partial t} = \frac{1}{2} \left(\frac{\partial}{\partial t} \frac{\partial d_l}{\partial x_k} + \frac{\partial}{\partial t} \frac{\partial d_k}{\partial x_l} \right) \tag{2.5}$$

Assuming continuity of the second partial derivatives of d_j with respect to both x_i and t, Schwarz's theorem allows us to re-write Eq. 2.5 as follows

$$\frac{\partial \epsilon_{kl}}{\partial t} = \frac{1}{2} \left(\frac{\partial}{\partial x_k} \frac{\partial d_l}{\partial t} + \frac{\partial}{\partial x_l} \frac{\partial d_k}{\partial t} \right) = \frac{1}{2} \left(\frac{\partial u_l}{\partial x_k} + \frac{\partial u_k}{\partial x_l} \right) \tag{2.6}$$

Considering the symmetry of stress components (i.e., $\sigma_{ij} = \sigma_{ji}$) resulted from Newton's third law, the elastic wave propagation problem can be written as a system of partial

differential equations, namely:

$$\begin{cases} \frac{\partial \sigma_{11}}{\partial t} - (\lambda+2\mu)\frac{\partial u_1}{\partial x_1} - \lambda\frac{\partial u_2}{\partial x_2} - \lambda\frac{\partial u_3}{\partial x_3} = s_1 \\ \frac{\partial \sigma_{22}}{\partial t} - \lambda\frac{\partial u_1}{\partial x_1} - (\lambda+2\mu)\frac{\partial u_2}{\partial x_2} - \lambda\frac{\partial u_3}{\partial x_3} = s_2 \\ \frac{\partial \sigma_{33}}{\partial t} - \lambda\frac{\partial u_1}{\partial x_1} - \lambda\frac{\partial u_2}{\partial x_2} - (\lambda+2\mu)\frac{\partial u_3}{\partial x_3} = s_3 \\ \frac{\partial \sigma_{12}}{\partial t} - \mu(\frac{\partial u_2}{\partial x_1} + \frac{\partial u_1}{\partial x_2}) = s_4 \\ \frac{\partial \sigma_{23}}{\partial t} - \mu(\frac{\partial u_2}{\partial x_3} + \frac{\partial u_3}{\partial x_2}) = s_5 \\ \frac{\partial \sigma_{13}}{\partial t} - \mu(\frac{\partial u_1}{\partial x_3} + \frac{\partial u_3}{\partial x_1}) = s_6 \\ \rho\frac{\partial u_1}{\partial t} - \frac{\partial \sigma_{11}}{\partial x_1} - \frac{\partial \sigma_{12}}{\partial x_2} - \frac{\partial \sigma_{13}}{\partial x_3} = s_7 \\ \rho\frac{\partial u_2}{\partial t} - \frac{\partial \sigma_{12}}{\partial x_1} - \frac{\partial \sigma_{22}}{\partial x_2} - \frac{\partial \sigma_{23}}{\partial x_3} = s_8 \\ \rho\frac{\partial u_3}{\partial t} - \frac{\partial \sigma_{13}}{\partial x_1} - \frac{\partial \sigma_{23}}{\partial x_2} - \frac{\partial \sigma_{33}}{\partial x_3} = s_9 \end{cases} \tag{2.7}$$

where $s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_8, s_9$ are right-hand sides of the corresponding equations, added for generality.

System 2.8 can be also written as a linear PDE system in the first-order formulation

$$\frac{\partial Q_p}{\partial t} + A_{pq}\frac{\partial Q_q}{\partial x_1} + B_{pq}\frac{\partial Q_q}{\partial x_2} + C_{pq}\frac{\partial Q_q}{\partial x_3} = S_p \tag{2.8}$$

where $Q = Q(\boldsymbol{x}, t) = (\sigma_{11}, \sigma_{22}, \sigma_{33}, \sigma_{12}, \sigma_{23}, \sigma_{13}, u_1, u_2, u_3)$ is a vector of unknowns at a point $\boldsymbol{x} = (x_1, x_2, x_3) \in \mathbb{R}^3$ and time $t \in \mathbb{R}$; $S_p = S_p(\boldsymbol{x}, t)$ is the external point source. $A_{pq} = A_{pq}(\boldsymbol{x})$, $B_{pq} = B_{pq}(\boldsymbol{x})$ and $C_{pq} = C_{pq}(\boldsymbol{x})$ are the space-dependent Jacobian matrices and are given by

$$A_{qp} = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & -(\lambda+2\mu) & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -\lambda & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -\lambda & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -\mu & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -\mu \\ -\frac{1}{\rho} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -\frac{1}{\rho} & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -\frac{1}{\rho} & 0 & 0 & 0 \end{pmatrix}$$

$$B_{qp} = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & -\lambda & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -(\lambda+2\mu) & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -\lambda & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -\mu & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -\mu \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -\frac{1}{\rho} & 0 & 0 & 0 & 0 & 0 \\ 0 & -\frac{1}{\rho} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -\frac{1}{\rho} & 0 & 0 & 0 & 0 \end{pmatrix} \tag{2.9}$$

$$C_{qp} = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -\lambda \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -\lambda \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -(\lambda+2\mu) \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -\mu & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -\mu & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -\frac{1}{\rho} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -\frac{1}{\rho} & 0 & 0 & 0 & 0 \\ 0 & 0 & -\frac{1}{\rho} & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Eigenvalue decomposition of A_{pq}, B_{pq} and C_{pq} shows that all eigenvalues (κ_i) are real numbers (see Eq. 2.10) and thus System 2.8 is hyperbolic.

$$\kappa_1 = -c_p,\ \kappa_2 = -c_s,\ \kappa_3 = -c_s,\ \kappa_4 = 0,\ \kappa_5 = 0,\ \kappa_6 = 0,\ \kappa_7 = c_s,\ \kappa_8 = c_s,\ \kappa_9 = c_p \tag{2.10}$$

where $c_p = \sqrt{\frac{\lambda+2\mu}{\rho}}$ and $c_s = \sqrt{\frac{\mu}{\rho}}$ can be considered as propagation velocities.

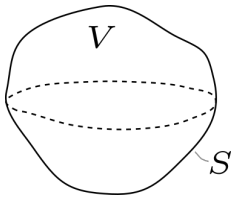


Figure 2.2.: Control Volume V with its boundaries S.

The weak form solution inside a control volume (see Fig. 2.2) can be obtained by multiplying Eq. 2.8 by test functions $\Phi_k = \Phi_k(\boldsymbol{x})$ and integrating in space. A particular choice of the test functions is explained in Section 3.3.

$$\int_V \Phi_k \frac{\partial Q_p}{\partial t}\, dV + \int_V \Phi_k \left(A_{pq}\frac{\partial Q_q}{\partial x_1} + B_{pq}\frac{\partial Q_q}{\partial x_2} + C_{pq}\frac{\partial Q_q}{\partial x_3} \right) dV = \int_V \Phi_k S_p\, dV \tag{2.11}$$

Assuming that Jacobin matrices A_{pq}, B_{pq} and C_{pq} do not change within the control volume, the second term of Eq. 2.11 can be integrated by parts as follows

$$\begin{aligned} &\int_V \left(\frac{\partial}{\partial x_1}(\Phi_k A_{pq} Q_q) + \frac{\partial}{\partial x_2}(\Phi_k B_{pq} Q_q) + \frac{\partial}{\partial x_3}(\Phi_k C_{pq} Q_q) \right) dV \\ &- \int_V \left(\frac{\partial \Phi_k}{\partial x_1} A_{pq} Q_q + \frac{\partial \Phi_k}{\partial x_2} B_{pq} Q_q + \frac{\partial \Phi_k}{\partial x_3} C_{pq} Q_q \right) dV \end{aligned} \tag{2.12}$$

The first term of Eq. 2.12 can be written in the vector notation using $\vec{i}$, $\vec{j}$, $\vec{k}$ standard basis vectors and vector differential operator $\vec{\nabla}$:

$$\int_V \vec{\nabla} \cdot \left[\Phi_k \left(A_{pq}\vec{i} + B_{pq}\vec{j} + C_{pq}\vec{k} \right) Q_q \right] dV = \int_V \vec{\nabla} \cdot \vec{\mathbb{G}}\, dV \tag{2.13}$$

Using Gauss's theorem, the volume integral in Eq. 2.13 can be replaced with the surface one

$$\int_V \vec{\nabla} \cdot \vec{\mathbb{G}}\, dV = \int_S \Phi_k \left(A_{pq}\vec{i} + B_{pq}\vec{j} + C_{pq}\vec{k} \right) Q_q \cdot \vec{n}\, dS = \int_S \Phi_k F_p\, dS \tag{2.14}$$

where $\vec{n}$ is a unit vector orthogonal to dS; F_p is a numerical flux function.

A more convenient form of the weak solution of Eq. 2.8 can be obtained by combining Eq. 2.11, Eq. 2.12 and Eq. 2.14.

$$\begin{aligned} &\int_V \Phi_k \frac{\partial Q_p}{\partial t}\, dV + \int_S \Phi_k F_p\, dS - \\ &- \int_V \left(\frac{\partial \Phi_k}{\partial x_1} A_{pq} Q_q + \frac{\partial \Phi_k}{\partial x_2} B_{pq} Q_q + \frac{\partial \Phi_k}{\partial x_3} C_{pq} Q_q \right) dV = \int_V \Phi_k S_p\, dV \end{aligned} \tag{2.15}$$

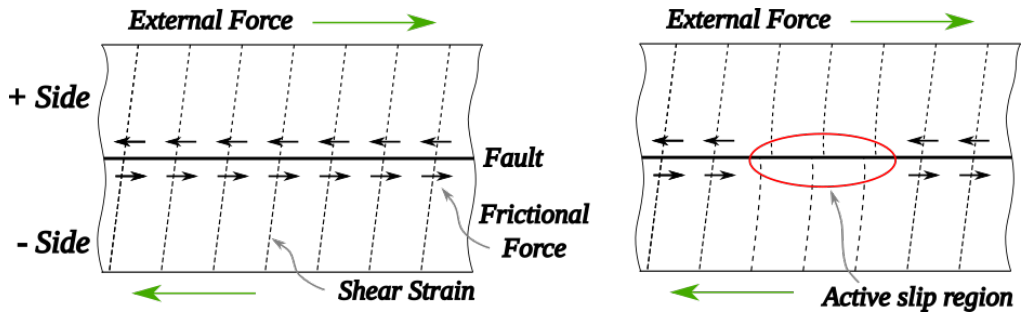


Figure 2.3.: Locked and unlocked states of the fault.

2.2. Dynamic Rupture Process

Faults are fractures in a volume of rock located in Earth's crust. Compressional and thus frictional forces lock two sides of the Earth and prevent them from a relative movement. However, sudden slip of rocks along the fault occurs when external forces (e.g., resulted form tectonic plates movements) exceed static friction. A rapid slip leads to an almost instant release of mechanical energy, which transforms into seismic waves.

Sliding is a point-local process and it is determined by relations between shear traction $\boldsymbol{\tau}$, fault strength τ_s and slip rate vector $\Delta\boldsymbol{u}$, which is the time derivative of the corresponding slip vector $\Delta\boldsymbol{d}$. The slip vector is the difference between fault-tangential material displacements on each side (i.e., $\boldsymbol{d}^+$ and $\boldsymbol{d}^-$).

$$\Delta\boldsymbol{u} = \Delta\boldsymbol{d}'_t = (\boldsymbol{d}^+ - \boldsymbol{d}^-)'_t \tag{2.16}$$

According to Coulomb's model of friction, the shear traction $\boldsymbol{\tau}$ is bounded by the fault strength τ_s as follows:

$$\boldsymbol{\tau} \leq \tau_s = max(0, -\mu_f \sigma_n) \tag{2.17}$$

where σ_n is a normal stress at point $\boldsymbol{x}$; $\mu_f \geq 0$ is a non-constant friction coefficient.

The model also assumes that the slip is changing only when the shear traction reaches the level of the fault strength which can be mathematically expressed as

$$(|\boldsymbol{\tau}| - \tau_s)|\Delta\boldsymbol{u}| = 0 \tag{2.18}$$

Moreover, the directions of the slip rate and the shear traction must be antiparallel. This can be written as

$$\Delta\boldsymbol{u}|\boldsymbol{\tau}| + |\Delta\boldsymbol{u}|\boldsymbol{\tau} = 0 \tag{2.19}$$

Without loss of generality, Eq. 2.19 can be rewritten using Eq. 2.18 as follows

$$\Delta\boldsymbol{u}\ \tau_s + |\Delta\boldsymbol{u}|\boldsymbol{\tau} = 0 \tag{2.20}$$

Analyses of laboratory experiments of rock friction show that coefficient μ_f correlates with the slip rate magnitude (see [101]) and can be modeled by a system of differential

algebraic equations such as

$$\begin{cases} \mu_f = f(U, \psi) \\ \frac{d\psi}{dt} = g(U, \psi) \end{cases} \tag{2.21}$$

where U is the symbol representing the slip rate magnitude (i.e., $|\Delta \boldsymbol{u}|$), which I use for simplicity.

For example, Eq. 2.22 shows a mathematical representation of a so-called Aging friction law [84, 76]. Fig. 2.4 shows a dependency between friction coefficient μ_f and the slip displacement governed by Eq. 2.22 given a specific slip velocity profile and friction parameters.

$$\begin{cases} \mu_f = a \sinh^{-1}\left[\frac{U}{2U_0} \exp\left(\frac{f_0 + b \ln(U_0 \psi / L)}{a}\right)\right] \\ \frac{d\psi}{dt} = 1 - \frac{U\psi}{L} \end{cases} \tag{2.22}$$

where a is the frictional evolution coefficient; b is the frictional state coefficient; L is the characteristic slip scale; U_0 is the reference slip velocity; f_0 is the reference friction coefficient.

Another example of friction is shown in Eq. 2.23 which is based on the Linear Slip-Weakening (LSW) law [56, 23]. Because of a more simple form of $g(U, \psi)$, this friction law is computationally less costly in comparison to the Aging law.

$$\begin{cases} \mu_f = C - \left(\mu_s - \frac{\mu_s - \mu_d}{d_c}\right) \min(\psi, d_c) \\ \frac{d\psi}{dt} = U \end{cases} \tag{2.23}$$

where C is the cohesion coefficient; μ_s and μ_d are the static and dynamic friction coefficients, respectively; and d_c is the slip-weakening critical distance.

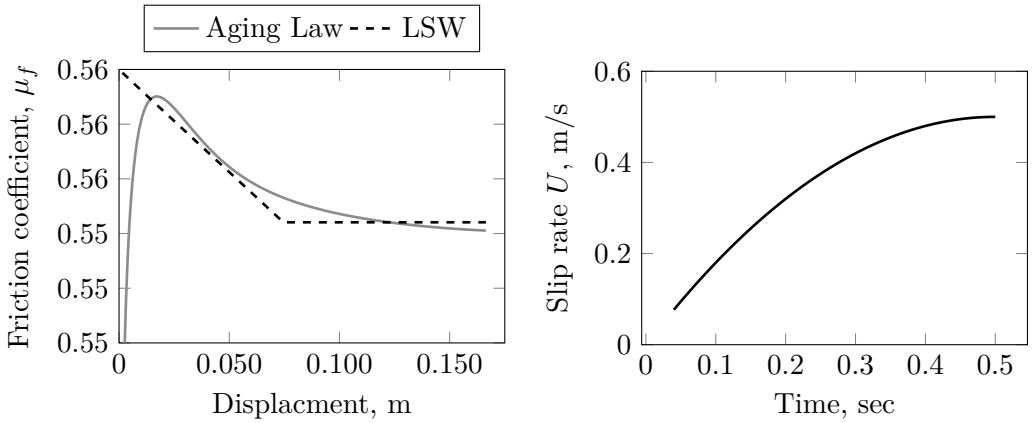


Figure 2.4.: Friction coefficients μ_f (on the left) computed according to 1) the Aging law (see Eq. 2.22) using: $a = 0.0085$; $b = 0.012$; $L = 0.01$; $U_0 = 1e-6$; $f_0 = 0.6$ and 2) the LSW (see Eq. 2.23) using: $C = 0.56$, $\mu_s = 0.5589$, $\mu_d = 0.5225$ and $d_c = 0.075$. The used velocity profile is shown on the right.

2.3. Kinematic Point Sources

Alternatively to Section 2.2, an earthquake at the hypocenter can be modeled using the moment tensor rate functions $m(t)$ which stem from the generalized force couples, shown in Fig. 2.5. In practice, the most commonly used approach is to decompose the rate functions into a time-invariant moment tensor M and one source time function. This approach leads to a significantly fewer number of parameters for modeling while adequately preserving an earthquake process [120]. The decomposition can be obtained by solving a nonlinear inverse problem using the recorded displacement seismograms of an earthquake. It is worth pointing out that this topic is beyond the scope of this thesis, and it is assumed that the decomposition is given in advance.

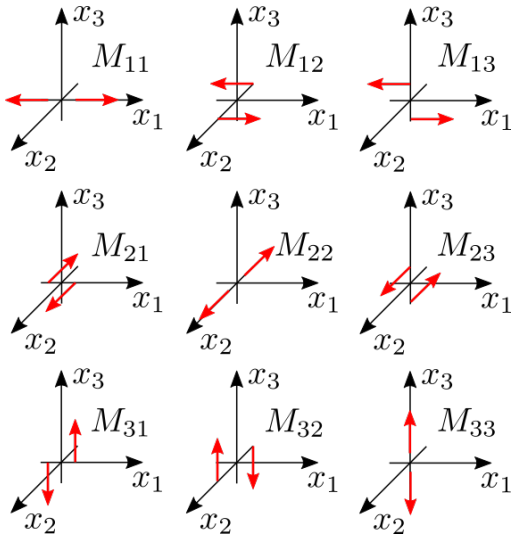


Figure 2.5.: Force couples representing the moment tensor.

The moment tensor rate functions can be written as in [62], namely:

$$m(t) = R_{pr}^{T} M_{rs} R_{sq} \cdot \mu \cdot A \cdot s(t) \tag{2.24}$$

where R is the transformation matrix from the fault-aligned system (defined by the strike, dip, and rake angles) to the global coordinate system; μ is the rigidity of the material; A is the area of the fault plane; $s(t)$ is the source time function of the slip rate.

As shown below, the Dirac Delta distribution can be used to embed the moment tensor into the right-hand side of Eq. 2.7.

$$S_p = s_p(t) \cdot \delta(\boldsymbol{x} - \boldsymbol{x_s}) \tag{2.25}$$

where $\boldsymbol{x_s}$ is the earthquake hypocenter location; $s_p(t)$ are the time-dependent source components such that

$$s_p(t) = \left[m_{11}(t), m_{22}(t), m_{33}(t), m_{12}(t), m_{23}(t), m_{13}(t), 0, 0, 0\right]^T \tag{2.26}$$

where the symmetry of the moment tensor is preserved.

2.4. Off-fault Plasticity Model

In contrast to elasticity, plasticity is associated with permanent and thus irreversible deformations in solid bodies [14]. Transitioning from the elastic to plastic material state is known as the yielding process - i.e., significant material stretching occurring at almost constant stress. Wollherr, Gabriel, and Uphoff in [125] showed that off-fault plastic yielding leads to more realistic dynamic rupture modeling in natural fault systems. This results in limiting unreasonably high slip velocities around faults [6] and thus effects rupture speed and style [31, 40].

Determining yielding criteria using just σ_{ij} is complicated because values of the stress tensor $\boldsymbol{\sigma}$ depend on the coordinate system orientation. Therefore, the plasticity analysis is commonly performed using the stress tensor invariants because, by definition, they are independent from the coordinate system orientation. The invariants are given by

$$\begin{aligned} I_1 &= \mathrm{tr}(\boldsymbol{\sigma}) \\ I_2 &= \frac{1}{2}\left(\sigma_{ii}\sigma_{jj} - \sigma_{ij}\sigma_{ji}\right) = \frac{1}{2}\left(\sigma_{ij} - \delta_{ij}\sigma_m\right)\left(\sigma_{ji} - \delta_{ji}\sigma_m\right) \\ I_3 &= \det\left(\boldsymbol{\sigma}\right) \end{aligned} \tag{2.27}$$

where σ_m is the mean stress and equal to $\frac{1}{3}\sum_{i=1}^{3}\sigma_{ii}$.

Transitioning from elastic to plastic states is defined by a so-called yield function F which, in general, depends on the stress tensor invariants. $F(\boldsymbol{\sigma}) = 0$ determines the yield surface which is usually convex in the 3-dimensional space and is defined by the eigenvectors of $\boldsymbol{\sigma}$, also known as principal stresses. Stress states on the yield surface become plastic and thus the material becomes affected by irreversible deformations. The total strain ϵ^{Σ} can be represented as the sum of the elastic ϵ and plastic ϵ^p strain components, where the increment of the latter can be determined by the flow rule and the plastic potential function g [79]. The flow rule and $g(\boldsymbol{\sigma})$ establish a relation between the plastic strain and stresses and thus to close the system of equations.

Among many proposed yielding functions, the most suitable for modeling frictional materials (e.g., soil and rocks) is Drucker-Prager plasticity criterion [4], where the yield and plastic potential functions are defined by

$$\begin{aligned} F(\boldsymbol{\sigma}) &= \sqrt{I_2} + \frac{1}{3}I_1 \sin\phi - C\cos(\phi) \\ g(\boldsymbol{\sigma}) &= \sqrt{I_2} \end{aligned} \tag{2.28}$$

where C is the cohesion; ϕ is the internal friction angle of the material.

The flow rule determines the plastic strain increment as [125]

$$d\epsilon_{ij}^{p} = \frac{1}{2\mu}\frac{\partial g(\boldsymbol{\sigma})}{\partial \sigma_{ij}} = \frac{1}{2\mu}\frac{\sigma_{ij} - \delta_{ij}\sigma_m}{2\sqrt{I_2}} \tag{2.29}$$

Due to known ill-posedness of the aforementioned plasticity model, viscoplastic relaxation is widely used to regularize the numerical implementation of plastic yielding in dynamic rupture simulations [125]. In contrast to Eq. 2.29, the strain increment is given by

$$d\epsilon_{ij}^{p} = \frac{1}{2\mu T_v}\left(\sigma_{ij} - P_{ij}(\boldsymbol{\sigma})\right) \tag{2.30}$$

where T_v is the plastic relaxation time; $P_{ij}(\boldsymbol{\sigma})$ is the adjusted stress state such that

$$P_{ij}(\boldsymbol{\sigma}) = \begin{cases} \frac{\tau_c}{\sqrt{I_2}}\sigma_{ij} + (1 - \frac{\tau_c}{\sqrt{I_2}})\delta_{ij}\sigma_m, & \text{if } F(\boldsymbol{\sigma}) = 0 \\ \sigma_{ij}, & F(\boldsymbol{\sigma}) < 0 \end{cases} \tag{2.31}$$

As shown by Dunne and Petrinic in [34], the stress state after an infinitesimal time increment (i.e., at $t + dt$) can be implicitly found using the radial return method which is based on Hook's law assuming no volumetric changes occur due to plastic yielding (i.e., $d\epsilon_{kk}^{p} = 0$). In the following, all quantities except for $d\epsilon_{kk}$ are considered to be at time $t + dt$.

$$\begin{aligned} \sigma_{ij} &= \lambda\delta_{ij}\left(\epsilon_{kk} + d\epsilon_{kk}\right) + 2\mu\left(\epsilon_{ij} + d\epsilon_{ij}\right) \\ &= \lambda\delta_{ij}\left(\epsilon_{kk} + d\epsilon_{kk}^{\Sigma} - d\epsilon_{kk}^{p}\right) + 2\mu\left(\epsilon_{ij} + d\epsilon_{ij}^{\Sigma} - d\epsilon_{ij}^{p}\right) \\ &= \lambda\delta_{ij}\left(\epsilon_{kk} + d\epsilon_{kk}^{\Sigma}\right) + 2\mu\left(\epsilon_{ij} + d\epsilon_{ij}^{\Sigma}\right) - 2\mu\, d\epsilon_{ij}^{p} \\ &= \underbrace{\sigma_{ij}^{trial}}_{\text{elastic predictor}} - \underbrace{2\mu\, d\epsilon_{ij}^{p}}_{\text{plastic corrector}} \end{aligned} \tag{2.32}$$

The trial stress σ_{ij}^{trial} is evaluated assuming purely elastic material properties. The plastic corrector term imposes $F(\boldsymbol{\sigma}) \leq 0$. Wollherr, Gabriel, and Uphoff in [125] discussed that $d\epsilon_{ij}^{p}$ at $t + dt$ could be evaluated by integrating the stress increment $d\sigma_{ij}$ arising from Eq. 2.32 assuming that P_{ij} and T_v are constant between t and $t + dt$. They showed that, in that case, the plastic strain increment was given by

$$d\epsilon_{ij}^{p} = \frac{1}{2\mu}\left(1 - f^{*}\right)\left(\sigma_{ij}^{trial} - \delta_{ij}\sigma_{m}^{trial}\right) \tag{2.33}$$

where f^{*} is the adjustment factor (see Eq. 2.34).

$$f^{*} = (1 - e^{-\frac{\Delta t}{T_v}})\frac{\tau_c}{\sqrt{I_2}} + e^{\frac{-\Delta t}{T_v}} \tag{2.34}$$

where Δt is the time step width arising from numerical time-integration.

3. Discontinuous Galerkin Method in SeisSol

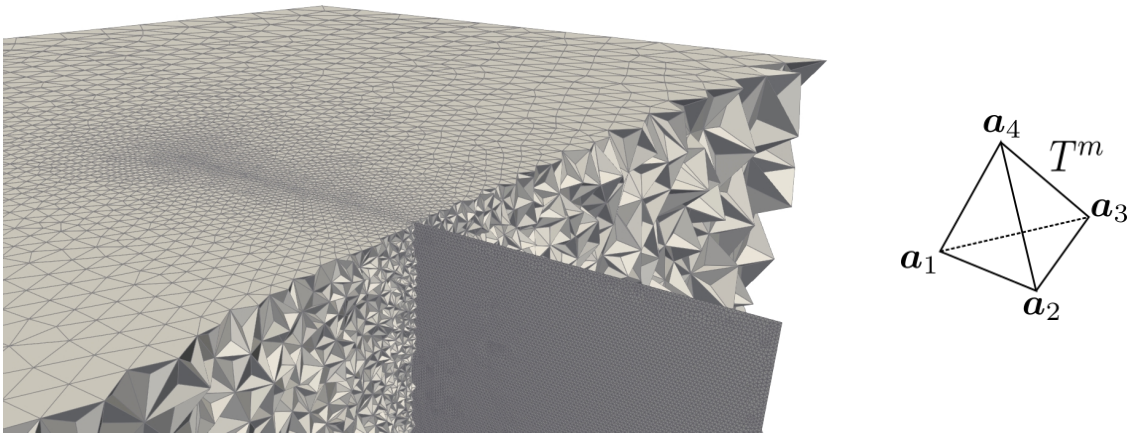


Figure 3.1.: Exemplary tetrahedral mesh comprising the wave propagation domain and a vertical fault plane on the left and a tetrahedral element T^m on the right.

The numerical solution of Eq. 2.8 inside the computational domain Ω can be found in the weak form using a conforming mesh consisting of tetrahedral elements T such that $\Omega = \bigcup_m T^m$ where m is a unique element index.

$$\begin{aligned} \int_{T^m} \Phi_k \frac{\partial Q_p}{\partial t}\, dV + \overbrace{\int_{\delta T^m} \Phi_k F_p^h\, dS}^{\text{flux term}} - \\ - \int_{T^m} \left(\frac{\partial \Phi_k}{\partial x} A_{pq} Q_q + \frac{\partial \Phi_k}{\partial y} B_{pq} Q_q + \frac{\partial \Phi_k}{\partial z} C_{pq} Q_q \right) dV = \int_T \Phi_k S_p\, dV \end{aligned} \tag{3.1}$$

where δT^m are boundaries of a tetrahedron T^m; F_p^h is the flux of Q_p through the h-th face of a tetrahedron.

In this work, the Discontinuous Galerkin (DG) method is used. The method defines a solution of the weak form inside each tetrahedral mesh element as a linear combination of basis functions given by piecewise polynomials. In contrast to the Finite Elements method (also known as the Continuous Galerkin method), the DG approach does not require the global numerical solution to be prescribed by a continuous function over the entire domain; the local solutions are allowed to be discontinuous on elements' boundaries. The discontinuities are resolved using numerical fluxes, which require a careful design to ensure numerical accuracy.

This chapter starts with a discussion of the numerical fluxes and their implementations in *SeisSol* (see Section 3.1). In Section 3.2, I explain how the weak form shown in Eq. 3.1 is mapped from the global to the reference-element coordinate system. This approach is common for both the Finite Elements and Discontinuous Galerkin methods and it leads to many simplifications and precomputations for the final numerical scheme. In Section 3.3, I apply the basis functions to the weak form and thus obtain the semidiscrete formulation of Eq. 3.1. Section 3.4 and Section 3.5 explain details of the numerical time integration, namely: the Arbitrary high-order DERivatives and Local Time Stepping schemes. Finally, in Section 3.6, I show how the most important boundary conditions, required for earthquake modeling, are implemented as numerical fluxes in *SeisSol*.

3.1. Numerical Fluxes

The DG formulation assumes that the solutions Q_p at element boundaries of adjacent tetrahedral elements and, thus, F_p^h may be discontinuous. Therefore, the integral in Eq. 2.14 requires some approximations.

Figure 3.2.: Rotated faced-aligned coordinate system on the left. Discontinuities of a solution between two adjacent elements T^m and T^{m_j} on the right.

Taking into account the rotational invariance of system Eq. 2.8 (see [114]) caused by the isotropic material properties, the problems can be conveniently solved in the corresponding face-aligned coordinate system considering the wave propagation only along the face-normal direction - i.e., x_1'-direction. It is assumed that linear transformations T_{pq} of the solution vector from the global (Q_p) to the faced-aligned ($\hat{Q}_q$) coordinate systems are given for each face of a tetrahedral element (see Eq. 3.2) and they are given by

$$Q_p^h = T_{pq}^h(\boldsymbol{n})\hat{Q}_q^h \tag{3.2}$$

where $\boldsymbol{n}$ is a face normal vector. For convenience of the follow-up discussion, I omit the tensor notation that has been established so far.

Therefore, the following Initial Value Problem (IVP) needs to be solved for each tetrahedron face neglecting any source terms at element boundaries (see [110]).

$$\begin{aligned} \frac{\partial \hat{Q}(x_1',t)}{\partial t} + A^{\pm}\frac{\partial \hat{Q}(x_1',t)}{\partial x_1'} &= 0 \\ \hat{Q}(x_1',t_0) = \mathring{\hat{Q}}(x_1') &= \begin{cases} \hat{Q}^+, & \text{if } x_1' > 0 \\ \hat{Q}^-, & \text{if } x_1' < 0 \end{cases} \\ A^{\pm} &= \begin{cases} A^+, & \text{if } x_1' > 0 \\ A^-, & \text{if } x_1' < 0 \end{cases} \end{aligned} \tag{3.3}$$

where $\mathring{\hat{Q}}(x_1')$ is the initial solution at time t_0; A^- and A^+ are the Jacobian matrices on the left "$-$" and right "$+$" sides of a boundary, respectively.

The eigendecomposition analysis of the Jacobian matrices $A^{\pm}$, discussed in Section 2.1, shows that the eigenvalues ($\kappa_i^{\pm}$) are real and the corresponding eigenvectors ($r_i^{\pm}$) are linearly independent. The corresponding right-eigenvector matrices of both the left and right sides are given by

$$R^{\pm} = \begin{pmatrix} \lambda^{\pm}+2\mu^{\pm} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \lambda^{\pm}+2\mu^{\pm} \\ \lambda^{\pm} & 0 & 0 & 0 & 1 & 0 & 0 & 0 & \lambda^{\pm} \\ \lambda^{\pm} & 0 & 0 & 0 & 0 & 1 & 0 & 0 & \lambda^{\pm} \\ 0 & \mu^{\pm} & 0 & 0 & 0 & 0 & 0 & \mu^{\pm} & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \mu^{\pm} & 0 & 0 & 0 & \mu^{\pm} & 0 & 0 \\ c_p^{\pm} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -c_p^{\pm} \\ 0 & c_s^{\pm} & 0 & 0 & 0 & 0 & 0 & -c_s^{\pm} & 0 \\ 0 & 0 & c_s^{\pm} & 0 & 0 & 0 & -c_s^{\pm} & 0 & 0 \end{pmatrix} \tag{3.4}$$

This determines that the matrix is diagonalizable and, thus, System 3.3 can be decoupled into n independent advection equations. Following [74], the solution can be written as a linear combination of the right eigenvectors as

$$\hat{Q}(\vec{0},t) = \hat{Q}^- + \sum_{i:\kappa_i<0} W_i = \hat{Q}^- + \sum_{i:\kappa_i<0} \alpha_i r_i \tag{3.5}$$

where W_i is a jump in solution $\hat{Q}(0,t)$ caused by a change in the solution of the i-th advection equation; α_i is the strength of the i-th wave.

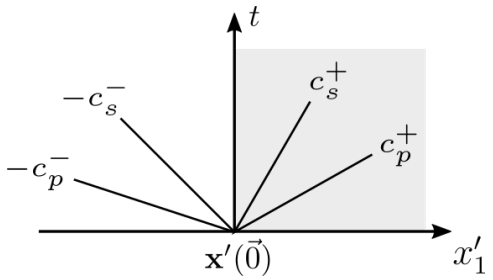


Figure 3.3.: Solution structure of the Riemann problem for Eq. 3.3.

To correctly approximate fluxes between adjacent elements with different material properties (see Fig. 3.3), jumps W_i across each wave must be associated with eigenvectors related

to the appropriate materials. For this purpose, matrix R^{-+} is assembled. It combines eigenvectors from left-going (negative) and right-going (positive) waves.

$$R^{-+} = \begin{pmatrix} \lambda^- + 2\mu^- & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \lambda^+ + 2\mu^+ \\ \lambda^- & 0 & 0 & 0 & 1 & 0 & 0 & 0 & \lambda^+ \\ \lambda^- & 0 & 0 & 0 & 0 & 1 & 0 & 0 & \lambda^+ \\ 0 & \mu^- & 0 & 0 & 0 & 0 & 0 & \mu^+ & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \mu^- & 0 & 0 & 0 & \mu^+ & 0 & 0 \\ c_p^- & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -c_p^+ \\ 0 & c_s^- & 0 & 0 & 0 & 0 & 0 & -c_s^+ & 0 \\ 0 & 0 & c_s^- & 0 & 0 & 0 & -c_s^+ & 0 & 0 \end{pmatrix} \tag{3.6}$$

Following LeVeque et al. (see [74]), the wave strengths are give by

$$\alpha = (\alpha_1, \alpha_2, \ldots, \alpha_n)^T = \left(R^{-+}\right)^{-1} \left(\hat{Q}^+ - \hat{Q}^-\right) \tag{3.7}$$

To simplify the subsequent derivations, I introduce an auxiliary diagonal matrix X where the i-th diagonal entry is equal to 1 if eigenvalue κ_i is less than zero; otherwise, it is set to 0. Matrix X acts as a mask that helps to remove a non-trivial summation indexing in Eq. 3.5. According to Eq. 2.10, only the first three eigenvectors must be considered while applying Eq. 3.5 because their associated eigenvalues are less than zero. Therefore,

$$X = diag(1, 1, 1, 0, 0, 0, 0, 0, 0) \tag{3.8}$$

The application of X and Eq. 3.7 to Eq. 3.5 results in

$$\begin{aligned} \hat{Q}(\vec{0}, t) &= \hat{Q}^- + \left[r_1^-, r_2^-, r_3^-, 0, \ldots, 0\right] \alpha \\ &= \hat{Q}^- + R^{-+} X \left(R^{-+}\right)^{-1} \left(\hat{Q}^+ - \hat{Q}^-\right) \\ &= \left(I - R^{-+} X \left(R^{-+}\right)^{-1}\right) \hat{Q}^- + R^{-+} X \left(R^{-+}\right)^{-1} \hat{Q}^+ \end{aligned} \tag{3.9}$$

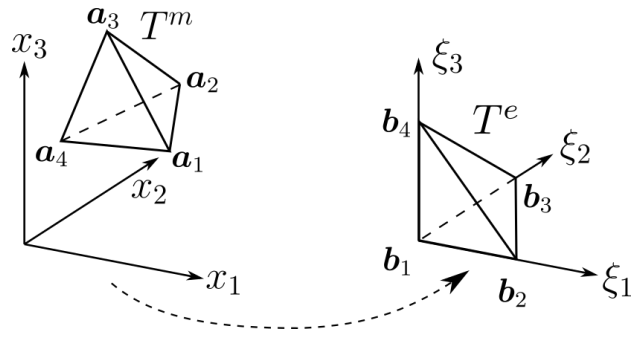
Following Eq. 2.14, the flux in the faced-aligned coordinate system can be obtained by multiplying Eq. 3.9 by matrix A from the left. This yields

$$\hat{F} = \left(A - AR^{-+} X \left(R^{-+}\right)^{-1}\right) \hat{Q}^- + AR^{-+} X \left(R^{-+}\right)^{-1} \hat{Q}^+ \tag{3.10}$$

The flux in the global coordinate system is given by

$$F = T\hat{F} = \underbrace{T \left(A - AR^{-+} X \left(R^{-+}\right)^{-1}\right) T^{-1}}_{\mathcal{A}^-} Q^- + \underbrace{TAR^{-+} X \left(R^{-+}\right)^{-1} T^{-1}}_{\mathcal{A}^+} Q^+ \tag{3.11}$$

where $\mathcal{A}^-$ and $\mathcal{A}^+$ are so-called flux solvers which can be pre-computed for all faces of each tetrahedron.



Points			Face
1	3	2	1
1	2	4	2
1	4	3	3
2	3	4	4

Figure 3.4.: Mapping tetrahedron T^m to the reference canonical element with vertices $(0,0,0)$, $(1,0,0)$, $(0,1,0)$ and $(0,0,1)$ on the left; and face numbering of a tetrahedron on the right.

3.2. Reference Element

Unknowns of Eq. 2.15 can be efficiently found inside the canonical reference element T_E^n (see Fig. 3.4). This approach is common for the Finite Element and DG frameworks and leads to many simplifications and pre-computations. It expresses physical coordinates x_1, x_2, x_3 as functions of the reference coordinates - i.e., $x_1(\xi_1,\xi_2,\xi_3)$, $x_2(\xi_1,\xi_2,\xi_3)$, $x_3(\xi_1,\xi_2,\xi_3)$. The barycentric mapping is given by

$$\boldsymbol{x}(\xi_1,\xi_2,\xi_3) = (1-\xi_1-\xi_2-\xi_3)\boldsymbol{a_1} + \xi_1\boldsymbol{a_2} + \xi_2\boldsymbol{a_3} + \xi_3\boldsymbol{a_4} \tag{3.12}$$

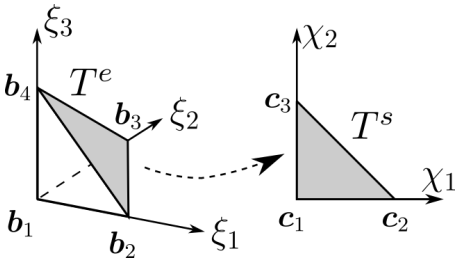
The change of variables from x_1, x_2, x_3 to ξ_1, ξ_2, ξ_3 entails the change of the differential operator (shown in Eq. 3.13) and thus differentials in volume integrals i.e., $dx_1\,dx_2\,dx_3 = |J|\,d\xi_1\;d\xi_2\;d\xi_3$, where J is the Jacobian of the corresponding mapping function (see Eq. 3.12).

$$\begin{bmatrix} \frac{\partial}{\partial x_1} \\ \frac{\partial}{\partial x_2} \\ \frac{\partial}{\partial x_3} \end{bmatrix} = \begin{bmatrix} \frac{\partial \xi_1}{\partial x_1} & \frac{\partial \xi_2}{\partial x_1} & \frac{\partial \xi_3}{\partial x_1} \\ \frac{\partial \xi_1}{\partial x_2} & \frac{\partial \xi_2}{\partial x_2} & \frac{\partial \xi_3}{\partial x_2} \\ \frac{\partial \xi_1}{\partial x_3} & \frac{\partial \xi_2}{\partial x_3} & \frac{\partial \xi_3}{\partial x_3} \end{bmatrix} \begin{bmatrix} \frac{\partial}{\partial \xi_1} \\ \frac{\partial}{\partial \xi_2} \\ \frac{\partial}{\partial \xi_3} \end{bmatrix} \tag{3.13}$$

Additional mapping is required for evaluating surface integrals - i.e., from surfaces of T^e to the canonical triangle T^s. In contrast to the previous case, this entails a map from 3D to 2D spaces, as shown in Fig. 3.5.

The mesh generation requirements enforce counter-clockwise vertex ordering; therefore, vertices of the adjacent canonical triangles do not match. This leads to several possible orientations of the triangles denoted by a parameter h and shown in Fig. 3.6.

To simplify the surface integration, $\tilde{\chi_2}$, $\tilde{\chi_1}$ coordinates within the canonical triangle of a neighboring element can be expressed using coordinates of the canonical triangle in question (i.e., χ_2, χ_1) as given in Table 3.1.



Face	Mapping
1	$\boldsymbol{\xi}^1(\chi_2,\chi_1) = \chi_1\boldsymbol{b}_2 + \chi_2\boldsymbol{b}_3$
2	$\boldsymbol{\xi}^2(\chi_2,\chi_1) = \chi_2\boldsymbol{b}_2 + \chi_1\boldsymbol{b}_4$
3	$\boldsymbol{\xi}^3(\chi_2,\chi_1) = \chi_1\boldsymbol{b}_3 + \chi_2\boldsymbol{b}_4$
4	$\boldsymbol{\xi}^4(\chi_2,\chi_1) = \chi_1\boldsymbol{b}_4 + \chi_2\boldsymbol{b}_3 + (1-\chi_1-\chi_2)\boldsymbol{b}_2$

Figure 3.5.: Mapping the 4th face of the reference tetrahedron T^e to the reference canonical triangle with vertices $(0,0)$, $(1,0)$ and $(0,1)$.

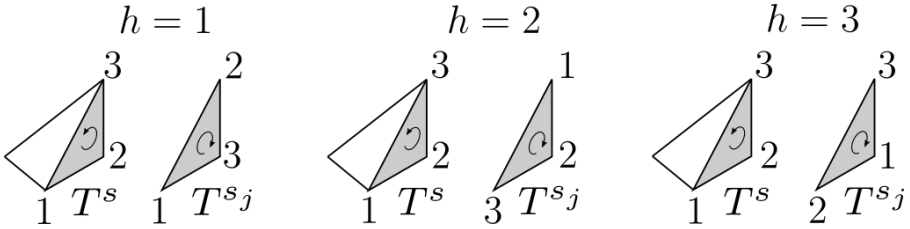


Figure 3.6.: Orientations of the adjacent canonical triangles.

Eq. 2.8, written in the reference element coordinate system with zero right-hand side, becomes

$$\begin{aligned}\frac{\partial Q_p}{\partial t} &+ A_{pq}\left(\frac{\partial \xi_1}{\partial x_1}\frac{\partial Q_q}{\partial \xi_1} + \frac{\partial \xi_2}{\partial x_1}\frac{\partial Q_q}{\partial \xi_2} + \frac{\partial \xi_3}{\partial x_1}\frac{\partial Q_q}{\partial \xi_3}\right) + \\ & B_{pq}\left(\frac{\partial \xi_1}{\partial x_2}\frac{\partial Q_q}{\partial \xi_1} + \frac{\partial \xi_2}{\partial x_2}\frac{\partial Q_q}{\partial \xi_2} + \frac{\partial \xi_3}{\partial x_2}\frac{\partial Q_q}{\partial \xi_3}\right) + \\ & C_{pq}\left(\frac{\partial \xi_1}{\partial x_3}\frac{\partial Q_q}{\partial \xi_1} + \frac{\partial \xi_2}{\partial x_3}\frac{\partial Q_q}{\partial \xi_2} + \frac{\partial \xi_3}{\partial x_3}\frac{\partial Q_q}{\partial \xi_3}\right) = S_p\end{aligned} \tag{3.14}$$

Re-arranging terms in Eq. 3.14 yields

$$\frac{\partial Q_p}{\partial t} + A^*_{pq}\frac{\partial Q_q}{\partial \xi_1} + B^*_{pq}\frac{\partial Q_q}{\partial \xi_2} + C^*_{pq}\frac{\partial Q_q}{\partial \xi_3} = S_p \tag{3.15}$$

where A^*_{pq}, B^*_{pq} and C^*_{pq} are defined as

$$\begin{aligned}A^*_{pq} &= A_{pq}\frac{\partial \xi_1}{\partial x_1} + B_{pq}\frac{\partial \xi_1}{\partial x_2} + C_{pq}\frac{\partial \xi_1}{\partial x_3} \\ B^*_{pq} &= A_{pq}\frac{\partial \xi_2}{\partial x_1} + B_{pq}\frac{\partial \xi_2}{\partial x_2} + C_{pq}\frac{\partial \xi_2}{\partial x_3} \\ C^*_{pq} &= A_{pq}\frac{\partial \xi_3}{\partial x_1} + B_{pq}\frac{\partial \xi_3}{\partial x_2} + C_{pq}\frac{\partial \xi_3}{\partial x_3}\end{aligned} \tag{3.16}$$

Table 3.1.: Transformations of χ_2-χ_1 coordinates of the canonical triangle to $\tilde{\chi_2}$-$\tilde{\chi_1}$ coordinates of the neighbor triangle for all possible orientations defined by h (see Fig. 3.6).

h	1	2	3
${\tilde{\chi_2}}^h(\chi_2,\chi_1)$	χ_1	$1-\chi_2-\chi_1$	χ_2
${\tilde{\chi_1}}^h(\chi_2,\chi_1)$	χ_2	χ_1	$1-\chi_2-\chi_1$

3.3. Basis Functions

Unknowns $Q_p(\boldsymbol{\xi},t)$ within the reference tetrahedron can be approximated using a linear combination of space-dependent basis functions $\Psi_l(\boldsymbol{\xi})$ multiplied by the corresponding time-dependent coefficients $Q_{lp}(t)$ - i.e., Degrees Of Freedom (DOF).

$$Q_p(\boldsymbol{\xi},t) = Q_{pl}(t)\Psi_l(\boldsymbol{\xi}) \tag{3.17}$$

The accuracy of the DG method depends on a choice of basis functions. In *SeisSol*, the orthogonal, hierarchical basis functions based on the Jacobi polynomials are used. The details about constructing this polynomial basis are given in [13]. The required number of basis functions $\mathcal{B}$ and the maximum polynomial degree $\mathcal{N}$ of the basis depends on the desired convergence order $\mathcal{O}$ of the scheme and are given by

$$\mathcal{O} = \mathcal{N}+1 \tag{3.18}$$

$$\mathcal{B} = \frac{1}{6}\mathcal{O}(\mathcal{O}+1)(\mathcal{O}+2) \tag{3.19}$$

After inserting the flux solvers derived in Section 3.1 and mapping Eq. 2.15 to the reference element coordinate system, the weak solution can be written as

$$\begin{aligned}
|J|\frac{\partial Q^m_{pl}}{\partial t}\int_{T^e}\Phi_k(\boldsymbol{\xi})\Psi_l(\boldsymbol{\xi})\,d\xi_1 d\xi_2 d\xi_3 + \\
+\sum_{j=1}^{4}|S_j|\mathcal{A}^{-,m}_{pq}Q^m_{ql}\int_{\delta T^{e_j}}\Phi_k(\boldsymbol{\xi}^j(\boldsymbol{\chi}))\Psi_l(\boldsymbol{\xi}^j(\boldsymbol{\chi}))\,d\chi_1 d\chi_2 + \\
\sum_{j=1}^{4}|S_j|\mathcal{A}^{+,m}_{pq}Q^{m_j}_{ql}\int_{\delta T^{e_j}}\Phi_k(\boldsymbol{\xi}^j(\boldsymbol{\chi}))\Psi_l(\boldsymbol{\xi}^i(\tilde{\boldsymbol{\chi}}^h(\boldsymbol{\chi})))\,d\chi_1 d\chi_2 - \\
|J|A^{*,m}_{pq}Q^m_{ql}\int_{T^e}\frac{\partial\Phi_k(\boldsymbol{\xi})}{\partial\xi_1}\Psi_l(\boldsymbol{\xi})\,d\xi_1 d\xi_2 d\xi_3 - \\
|J|B^{*,m}_{pq}Q^m_{ql}\int_{T^e}\frac{\partial\Phi_k(\boldsymbol{\xi})}{\partial\xi_2}\Psi_l(\boldsymbol{\xi})\,d\xi_1 d\xi_2 d\xi_3 - \\
|J|C^{*,m}_{pq}Q^m_{ql}\int_{T^e}\frac{\partial\Phi_k(\boldsymbol{\xi})}{\partial\xi_3}\Psi_l(\boldsymbol{\xi})\,d\xi_1 d\xi_2 d\xi_3 = |J|\int_{T^e}\Phi_k(\boldsymbol{\xi})S_p\,d\xi_1 d\xi_2 d\xi_3
\end{aligned} \tag{3.20}$$

where δT^e denotes boundaries of the reference element; S_j is the determinant of the space Jacobian matrix resulting from mapping the j-th face of element T^m from the global to the reference triangle coordinate systems; i is the face number of element T^{m_j} adjacent to

the j-th face of element T^m; h is the orientation number of the i-th face of element T^{m_j} relatively to the j-th face of element T^m; and $\boldsymbol{\xi}_s$ is the location of a point source within T^e.

As mentioned in Section 2.1, the kinematic point source model, shown in Eq. 2.25, results in a special form of the right-hand side. Therefore, the properties of the Dirac Delta distribution need to be considered while performing the volume integration in Eq. 3.20.

$$\int_{T^e} \Phi_k(\boldsymbol{\xi}) S_p(t,\boldsymbol{\xi})\, d\xi_1 d\xi_2 d\xi_3 = \sum_s s_{ps}(t) \int_{T^e} \Phi_k(\boldsymbol{\xi})\, \delta(\boldsymbol{\xi}-\boldsymbol{\xi}_s)\, d\xi_1 d\xi_2 d\xi_3 = s_{ps}(t)\, \Phi_k(\boldsymbol{\xi}_s) \tag{3.21}$$

where $s_{ps}(t)$ are time-dependent components of the point sources which happen to be inside the T^m element - i.e., $\boldsymbol{\xi}_s \in T^e$.

Following the Galerkin approach [53], test functions Φ are chosen from the same set of basis functions Ψ. This leads to the following element-local semi-discrete formulation.

$$\begin{aligned} &\frac{\partial Q^m_{pl}}{\partial t}|J|M_{kl} + \sum_{j=1}^{4} |S_j| \mathcal{A}^{-,m}_{pq} Q^m_{ql} \mathcal{F}^{-,j}_{kl} + \sum_{j=1}^{4} |S_j| \mathcal{A}^{+,m}_{pq} Q^{m_j}_{ql} \mathcal{F}^{+,jih}_{kl} - \\ &|J| A^{*,m}_{pq} Q^m_{ql} K^1_{kl} - |J| B^{*,m}_{pq} Q^m_{ql} K^2_{kl} - |J| C^{*,m}_{pq} Q^m_{ql} K^3_{kl} = |J| s_{ps} \Psi_k(\boldsymbol{\xi}_s) \end{aligned} \tag{3.22}$$

where M_{pr}, K^i_{kl}, $\mathcal{F}^{-,j}_{kl}$ and $\mathcal{F}^{+,jih}_{kl}$ are mass, stiffness, and flux matrices arising from the following integrals.

$$\begin{aligned} M_{kl} &= \int_{T^e} \Psi_k(\boldsymbol{\xi}) \Psi_l(\boldsymbol{\xi})\, d\xi_1 d\xi_2 d\xi_3 \\ \mathcal{F}^{-,j}_{kl} &= \int_{T^{e_j}} \Psi_k(\boldsymbol{\xi}^j(\boldsymbol{\chi})) \Psi_l(\boldsymbol{\xi}^j(\boldsymbol{\chi}))\, d\chi_1 d\chi_2 \\ \mathcal{F}^{+,jih}_{kl} &= \int_{\delta T^{e_j}} \Psi_k(\boldsymbol{\xi}^j(\boldsymbol{\chi})) \Psi_l(\boldsymbol{\xi}^i(\tilde{\boldsymbol{\chi}}^h(\boldsymbol{\chi})))\, d\chi_1 d\chi_2 \\ K^i_{kl} &= \frac{\partial \Psi_k(\boldsymbol{\xi})}{\partial \xi_i} \Psi_l(\boldsymbol{\xi})\, d\xi_1 d\xi_2 d\xi_3 \end{aligned} \tag{3.23}$$

It is worth mentioning that all integrals, shown in Eq. 3.23, can be calculated beforehand using computer algebra software systems like *Maple* [32].

3.4. ADER

The numerical time-integration of Eq. 3.22 can be performed using one of the Runge–Kutta schemes with the same convergence order as the one selected for the spatial discretization. However, as mentioned by Dumbser and Käser in [32], the computational efficiency of Runge–Kutta methods decreases when the convergence order becomes greater than 5 due to a drastic increase of intermediate Runge–Kutta stages. The alternative approach

for numerical time integration is the Arbitrary high-order DERivatives (ADER) scheme. Because the Cauchy–Kowalevski procedure applied to a linear hyperbolic system can express time derivatives with spatial derivatives, a Taylor expansion of DOFs in time (shown in Eq. 3.24) is used to compute high-order predictors for explicit time integration.

$$Q_{pl}(t_0, t) = \sum_{i=0}^{\mathcal{O}-1} \frac{(t-t_0)^i}{i!} \frac{\partial^i Q_{pl}(t_0)}{\partial t^i} \tag{3.24}$$

where t_0 is the point of expansion.

The second time derivative of the vector of unknowns can be written as

$$\begin{aligned} \frac{\partial Q_p^2}{\partial t^2} &= -A_{pq}\frac{\partial}{\partial t}\frac{\partial Q_q}{\partial x_1} - B_{pq}\frac{\partial}{\partial t}\frac{\partial Q_q}{\partial x_2} - C_{pq}\frac{\partial}{\partial t}\frac{\partial Q_q}{\partial x_3} + \frac{\partial s_{ps}(t)}{\partial t}\delta(\boldsymbol{x}-\boldsymbol{x}_s) \\ &= -A_{pq}\frac{\partial}{\partial x_1}\frac{\partial Q_q}{\partial t} - B_{pq}\frac{\partial}{\partial x_2}\frac{\partial Q_q}{\partial t} - C_{pq}\frac{\partial}{\partial x_3}\frac{\partial Q_q}{\partial t} + \frac{s_{ps}(t)}{t}\delta(\boldsymbol{x}-\boldsymbol{x}_s) \end{aligned} \tag{3.25}$$

Assuming that Q and $s(t)$ are the i-th times differentiable, the i-th time derivative can be recursively expressed as

$$\frac{\partial^i Q_p}{\partial t^i} = -A_{pq}\frac{\partial}{\partial x_1}\frac{\partial^{i-1} Q_q}{\partial t^{i-1}} - B_{pq}\frac{\partial}{\partial x_2}\frac{\partial^{i-1} Q_q}{\partial t^{i-1}} - C_{pq}\frac{\partial}{\partial x_3}\frac{\partial^{i-1} Q_q}{\partial t^{i-1}} + \frac{\partial^{i-1} s_{ps}(t)}{\partial t^{i-1}}\delta(\boldsymbol{x}-\boldsymbol{x}_s) \tag{3.26}$$

Similar to Eq. 3.17, the time derivatives are represented as a linear combination of basis functions multiplied by time-dependent coefficients $\mathcal{D}^i_{lp}$.

$$\frac{\partial Q_p^i(\boldsymbol{\xi}, t)}{\partial t^i} = \mathcal{D}^i_{pl}(t)\Psi_l(\boldsymbol{\xi}) \tag{3.27}$$

where, by definition, $\partial^0 Q_p(t)/\partial t^0 = Q_p(t) = \mathcal{D}^0_{pl}(t)\Psi_l(\boldsymbol{\xi})$ and, thus, $Q_{pl}(t) = \mathcal{D}^0_{pl}(t)$ according to Eq. 3.17.

$\mathcal{D}^i_{lp}$ coefficients can be recovered by inserting Eq. 3.27 into Eq. 3.26 and projecting the result onto each basis function. The aforementioned procedure in the reference element coordinate system yields

$$\begin{aligned} |J| D^{i,m}_{pl} \int_{T^e} \Psi_k(\boldsymbol{\xi})\Psi_l(\boldsymbol{\xi}) d\xi_1 d\xi_2 d\xi_3 = -|J| A^m_{pq} D^{i-1,m}_{ql} \int_{T^e} \Psi_k(\boldsymbol{\xi}) \frac{\partial \Psi_l(\boldsymbol{\xi})}{\partial \xi_1} d\xi_1 d\xi_2 d\xi_3 - \\ -|J| B^m_{pq} D^{i-1,m}_{ql} \int_{T^e} \Psi_k(\boldsymbol{\xi}) \frac{\partial \Psi_l(\boldsymbol{\xi})}{\partial \xi_2} d\xi_1 d\xi_2 d\xi_3 - |J| C^m_{pq} D^{i-1,m}_{ql} \int_{T^e} \Psi_k(\boldsymbol{\xi}) \frac{\partial \Psi_l(\boldsymbol{\xi})}{\partial \xi_3} d\xi_1 d\xi_2 d\xi_3 + \\ +|J| \frac{\partial^{i-1} s_{ps}(t)}{\partial t^{i-1}} \Psi_k(\boldsymbol{\xi}_s) \end{aligned} \tag{3.28}$$

The time-dependent part of a single point source can be represented in-time basis over $[t, t+\Delta t]$. Similar to [61], the classical Legendre polynomials are chosen as basis functions which are orthogonal by definition.

$$s_p(t) = S_{pl}\theta_l(t) \tag{3.29}$$

Time-independent coefficients $\mathcal{S}_{pl}$ can be retrieved by projecting Eq. 3.29 onto each basis function $\theta_i(t)$.

$$\int_t^{t+\Delta t} s_p(t)\theta_i(t)dt = S_{pl}\underbrace{\int_t^{t+\Delta t}\theta_l(t)\theta_i(t)dt}_{M^{\theta}_{li}} \tag{3.30}$$

Gaussian quadrature rules allow us to evaluate the left-hand side of Eq. 3.30 and, thus, obtain the coefficients

$$S_{pl} = \left(\int_t^{t+\Delta t} s_p(t)\theta_i(t)dt\right)\left(M^{\theta}_{li}\right)^{-1} = \left(\sum_{j=1}^{\mathcal{O}}\omega_j s_p(\tau_j)\theta_i(\tau_j)\right)\left(M^{\theta}_{li}\right)^{-1} \tag{3.31}$$

where $\tau_j \in [t, t+\Delta t]$ are the Gaussian integration points; ω_j are the corresponding weights.

Therefore, the last term of Eq. 3.28 can be written as

$$\frac{\partial^{i-1}s_{ps}(t)}{\partial t^{i-1}}\Psi_k(\boldsymbol{\xi}_s) = S_{psl}\Theta^{i-1}_l(t)\Psi_k(\boldsymbol{\xi}_s) \tag{3.32}$$

where Θ^{i-1}_l is the $(i-1)$-th time derivative of θ_l, which can be computed beforehand, similar to Eq. 3.23.

Finally, the time derivatives of DOFs can be expressed as follows

$$\mathcal{D}^{i,m}_{pl} = \left(S_{psl}\Theta^{i-1}_l\Psi_k(\boldsymbol{\xi}_s) - A^m_{pq}\mathcal{D}^{i-1,m}_{ql}(K^1_{lk})^T - B^m_{pq}\mathcal{D}^{i-1,m}_{ql}(K^2_{lk})^T - C^m_{pq}\mathcal{D}^{i-1,m}_{ql}(K^3_{lk})^T\right)M^{-1}_{kl} \tag{3.33}$$

High-order predictors $\mathcal{T}_{pl}$, also called integrated DOFs, can be obtained by integrating Eq. 3.24 in time over $[t, t+\Delta t]$ such that $t_0 < t$ and $\Delta t > 0$.

$$\begin{gathered}\mathcal{T}^m_{pl}(t_0,t,\Delta t) = \int_t^{t+\Delta t} Q^m_{pl}(t,t_0)dt = \\ = \int_t^{t+\Delta t}\sum_{i=0}^{\mathcal{O}-1}\frac{(t-t_0)^i}{i!}\frac{\partial^i Q^m_{pl}(t_0)}{\partial t^i}dt = \sum_{i=0}^{\mathcal{O}-1}\frac{(t+\Delta t-t_0)^{i+1}-(t-t_0)^{i+1}}{(i+1)!}\mathcal{D}^{i,m}_{pl}(t_0)\end{gathered} \tag{3.34}$$

The discrete form of the underlying system of PDEs can be obtained by integrating Eq. 3.22 in time over $[t, t+\Delta t]$, replacing DOFs integrals with Eq. 3.34, and applying a Gaussian quadrature rule for evaluating the right-hand side. Therefore,

$$\begin{gathered}\left[(Q^m_{pl})^{t+\Delta t} - (Q^m_{pl})^t\right]|J|M_{kl} + \\ \sum_{j=1}^{4}|S_j|\mathcal{A}^{-,m}_{pq}\mathcal{T}^m_{ql}(t,t,\Delta t)\mathcal{F}^{-,j}_{kl} + \sum_{j=1}^{4}|S_j|\mathcal{A}^{+,m}_{pq}\mathcal{T}^{m_j}_{ql}(t,t,\Delta t)\mathcal{F}^{+,jih}_{kl} - \\ |J|A^{*,m}_{pq}\mathcal{T}^m_{ql}(t,t,\Delta t)K^1_{kl} - |J|B^{*,m}_{pq}\mathcal{T}^m_{ql}(t,t,\Delta t)K^2_{kl} \\ -|J|C^{*,m}_{pq}\mathcal{T}^m_{ql}(t,t,\Delta t)K^3_{kl} = |J|\left(\sum_{j=1}^{\mathcal{O}}\omega_j s_{ps}(\tau_j)\right)\Psi_k(\boldsymbol{\xi}_s)\end{gathered} \tag{3.35}$$

In contrast to the Runge–Kutta approach, Eq. 3.35 performs high-order time integration in a single step.

3.5. Local Time Stepping

The explicit time-integration scheme in Eq. 3.35 is known to be conditionally stable. In the Finite Difference framework, it is possible to derive a necessary condition for convergence and, thus, stability of a linear hyperbolic problem using the von Neumann stability analysis. This is known as the Courant–Friedrichs–Lewy (CFL) condition, which gives the upper bound for Δt regarding the advection velocity and the element size. For the ADER-DG scheme applied to Eq. 2.7, the CFL condition for element T^m is given by

$$\Delta t^m < \frac{1}{2\mathcal{N}+1}\frac{d^m}{c_p^m} \tag{3.36}$$

where d^m is the in-sphere diameter of the tetrahedron T^m; c_p is its local seismic P-wave velocity (see Eq. 2.10); $\mathcal{N}$ is the maximum polynomial degree of the basis functions.

To ensure convergence of a numerical solution within the entire domain Ω, the smallest Δt^m must be used in Eq. 3.35. This approach implies that all mesh elements are updated at the same rate - i.e., Global Time Stepping (GTS).

$$\Delta t^{GTS} = \Delta t_{min} = \min_{\forall T^m \in \Omega}(\Delta t^m) \tag{3.37}$$

where t^m denotes the time step width of the m-th tetrahedron.

GTS is suitable for numerical simulations which involve uniform meshes and isotropic media - i.e., where all elements have more or less the same size and similar material properties. However, many geophysical applications contain a limited number of zones of interest (e.g., fault planes, surface topography, areas around kinematic point sources, sedimentary basins, etc.) which may require high mesh resolutions (see Fig. 3.1 as an example). The stability criteria inside such zones determine the global time step width Δt^{GTS} which is typically much smaller than the element-local ones in the bulk of a domain. Therefore, GTS may result in excessive computations and, thus, in high computational efforts. Alternatively, the Local Time Stepping (LTS) scheme updates each element with its optimal time step width, which results in fewer overall computations. However, good approximations of numerical fluxes are required to implement LTS. Fortunately, the ADER-DG scheme, which involves solving the Generalized Riemann Problems (GRP) at the element interfaces, allows us to evaluate flux contributions from neighboring elements using their time derivatives of DOFs.

A straightforward, element-wise implementation of the LTS often results in low performance on parallel computers due to serialized computations resulting from numerical time integration. Therefore, a cluster-wise implementation of the LTS is commonly used in practice. Given an adjacent time-cluster update ratio $r \in \mathbb{N}^+$, element T^m is defined to

belongs to a time-cluster C_l if the following condition holds

$$\Delta t^m \in \left[r^l \cdot \Delta t_{min}, r^{l+1} \cdot \Delta t_{min} \right) \tag{3.38}$$

Therefore, the total number of time-clusters L is given by

$$L = \left\lceil \log_r \left(\frac{\max(\Delta t^m)}{\Delta t_{min}} \right) \right\rceil \tag{3.39}$$

Moreover, as shown in [13], an efficient cluster-wise LTS implementation requires to limit the dependencies between time-clusters to a single time-level. This restricts a face neighbor element T^{m_j} of a tetrahedron $T^m \in C_l$ to belong to a direct neighboring time-cluster, namely: $T^{m_j} \in C_{l-1}$ (if $l > 0$), or $T^{m_j} \in C_l$, or $T^{m_j} \in C_{l+1}$ (if $l < L - 1$). As mentioned by Breuer in [13], this constraint simplifies the LTS algorithm and reduces the number of possible sends and receives on distributed-memory computers.

The LTS algorithm can be illustrated using an element T^m and its neighbor T^{m_i}, which belong to neighboring time-clusters C_l and C_{l+1}, respectively. Let's start with time-iteration n when both T^m and T^{m_i} elements are in sync i.e., $t^n \mid \Delta t^n_m = \Delta t^n_{m_i}$. Time-integrated DOFs of element T^m must update element T^{m_i} according to Eq. 3.35. These can be obtained after r time-iterations of element T^m as follows

$$\mathcal{T}^m_{pl}(t^n, t^n, \Delta t^n_{m_i}) = \sum_{i=0}^{r-1} \mathcal{T}^m_{pl}(t^n + i \cdot \Delta t^n_m, t^n + i \cdot \Delta t^n_m, \Delta t^n_m) \tag{3.40}$$

Similarly, element T^m needs the time-integrated DOFs of element T^{m_i} to perform its time-updates. However, given derivatives of T^{m_i}, the element T^m can evaluate them by itself using Eq. 3.34 and thus perform its local time-update according to Eq. 3.35.

To summarize, the algorithm contains the following steps, namely:

1. At a sync-time t^n, element T^{m_i} saves its DOFs time-derivatives in a buffer $\mathcal{B}_{k_i}$, while element T^m zeros its associated buffer $\mathcal{B}_k$.

2. r time-updates of elements T^m are performed using time-derivatives of element T^{m_i} stored in $\mathcal{B}_{k_i}$, while time-integrated DOFs of element T^m are getting accumulated in $\mathcal{B}_k$ at the end of each time-update.

3. Element T^{m_i} performs its time-update from t^n to $t^n + \Delta t^n_{m_i}$ using time-integrated DOFs of element T^m accumulated in $\mathcal{B}_k$.

Therefore, the LTS algorithm enforces data dependencies during the numerical time integration, i.e., elements of a time-cluster C_i must be updated before the ones belonging to a time-cluster C_j if $i < j$. This also entails extra memory consumption due to allocating

additional memory buffers - e.g., $\mathcal{B}_{k_i}$ and $\mathcal{B}_{k_i}$. Moreover, splitting elements into time-clusters shrinks parallel regions, which may negatively affect the hardware performance of some computing devices designed for massive parallelism - e.g., GPUs. Nevertheless, the algorithm reduces redundant computations and, thus, the overall time required to obtain the final numerical solution.

3.6. Boundary Conditions

Setting correct boundary conditions is not trivial in the DG method. In contrast to the Finite Difference and Finite Element methods, there is no direct control over values at the nodes along the domain boundaries. Therefore, boundary conditions must be imposed via numerical fluxes, which can be found by designing and solving inverse Riemann problems. In the following, the most important boundary conditions relevant to this study are discussed.

3.6.1. Absorbing Boundaries

The absorbing boundary condition is used to model the wave propagation in the direction of a surface-normal without reflections - i.e., without incoming waves. The flux form, given in Eq. 3.11, has already been split into outgoing and incoming waves, where the latter are influenced by the state only inside the neighboring element. The simplest approach in simulating absorbing boundaries is to omit the second term in Eq. 3.11 from the right-hand side. This yields

$$F_p = T_{pj} \left(A_{jr} - A_{jn} R^-_{nm} X_{ml} \left(R^-_{lr} \right)^{-1} \right) T^{-1}_{rq} Q^-_q \tag{3.41}$$

However, as noticed by Dumbser and Käser and Hermann (in [32] and [52], respectively), this approach still results in some reflections if waves do not perfectly hit a boundary perpendicularly. Alternative and more sophisticated techniques (e.g., Perfectly Matched Layer, see [11]) exist and can be used to eliminate such phenomena, but at the same time, it entails performing some additional computations.

3.6.2. Free-Surface Boundaries

The free-surface boundary condition models a contact surface between the wave propagation domain and the void. This requires the normal and shear stresses at the surface boundary to be zero. The solution of the inverse Riemann problem can be found if one assumes that each boundary element p has a virtually extrapolated state Q^v_q outside the computational domain. The state Q^v_q is the same as Q_p except for σ_{11}, σ_{12}, and σ_{13}

components which are taken with the same magnitudes but with the opposite signs. This can be expressed as follows in the mathematical notation

$$Q_q^v = diag\,(-1,1,1,-1,1,-1,1,1,1)\,Q_p = \Gamma_{qp} Q_p \tag{3.42}$$

Therefore, the flux at a free-surface boundary is given by

$$F_p = T_{pj}\left(A_{jr} - A_{jn}R_{nm}^{-}X_{ml}\left(R_{lr}^{-}\right)^{-1}\right)T_{rq}^{-1}Q_q^{-} + T_{pj}A_{jn}R_{nm}^{-}X_{ml}\left(R_{lr}^{-}\right)^{-1}T_{rs}^{-1}\Gamma_{sq}Q_q^{-} \tag{3.43}$$

3.6.3. Dynamic Rupture

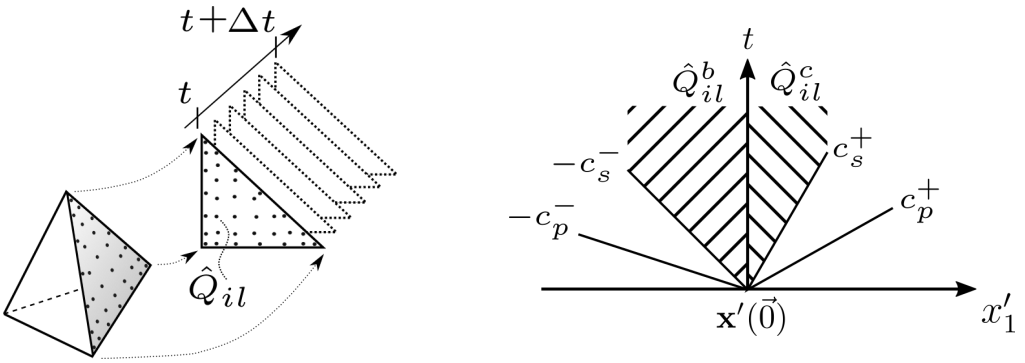


Figure 3.7.: 2D Gaussian points on the dynamic rupture interface (on the left), and the solution structure of the Riemann problem for Eq. 3.3 (on the right).

Rupture processes can also be considered as a special boundary condition. As discussed in Section 2.2, the process is point-local; therefore, the flux term needs to be numerically integrated using some quadrature rules as proposed by de la Puente, Ampuero, and Käser in [24].

$$\hat{F}_{pk} = A_{pr}\int_t^{t+\Delta t}\int_S \Phi_k \hat{Q}_r\; dS\; dt \approx A_{pr}\sum_{l=1}^{\mathcal{O}}\sum_{i=1}^{(\mathcal{O}+1)^2}\omega_i^S\omega_l^T\Phi_k(\boldsymbol{\chi}_i)\hat{Q}_{r,il}(\boldsymbol{\chi}_i,t_l) \tag{3.44}$$

where ω_l^T and ω_i^S are the temporal and spatial weights required for the Gaussian integration; $\boldsymbol{\chi}_i$ is a spatial Gaussian point written in the canonical reference triangle coordinate system; t_l is a temporal Gaussian point located within $[t, t+dt]$ interval.

First of all, the solution structure of the Riemann solver must be revisited to find $\hat{Q}_{r,il}(\boldsymbol{\chi}_i,t_l)$ because, in contrast to Section 3.1, it must be assumed that states $\hat{Q}^b_{r,il}(\boldsymbol{\chi}_i,t_l)$ and $\hat{Q}^c_{r,il}(\boldsymbol{\chi}_i,t_l)$ are not necessarily equal due to a possible slip of the material along a rupture surface. To find them, $\hat{Q}^+_{r,il}(\boldsymbol{\chi}_i,t_l)$ and $\hat{Q}^-_{r,il}(\boldsymbol{\chi}_i,t_l)$ approximations are required, which can be obtained by projecting their DOFs time-evolutions (i.e., $\mathcal{T}^{\pm}_{rm}(t,t,t_l)$) from

the modal basis to the chosen Gaussian points (see Fig. 3.7).

$$\hat{Q}^{-}_{r,il}(\boldsymbol{\chi}_i, \tau_l) = T^{-1}_{rp}(\boldsymbol{n})\mathcal{T}^{-}_{pm}(t,t,t_l)\Phi_m(\boldsymbol{\xi}^f(\boldsymbol{\chi}_i)) \tag{3.45}$$

$$\hat{Q}^{+}_{r,il}(\boldsymbol{\chi}_i, \tau_l) = T^{-1}_{rp}(\boldsymbol{n})\mathcal{T}^{+}_{pm}(t,t,t_l)\Phi_m(\boldsymbol{\xi}^g(\tilde{\boldsymbol{\chi}}^h(\boldsymbol{\chi}_i))) \tag{3.46}$$

where $\boldsymbol{n}$ is the normal of an element rupture face; f and g are the numbers of the rupture faces within the elements adjacent to "$-$" and "$+$" sides, respectively; h is the orientation number of the rupture face on the "$+$" side relative to the one on the "$-$" side.

In the following, I deliberately omit r, il-indices for simplicity - i.e., $\hat{Q}^{\pm} = \hat{Q}^{\pm}_{r,il}(\boldsymbol{\chi}_i, t_l)$. Additionally, I re-scale right-eigenvalues column-wise using Υ diagonal matrix as shown in Eq. 3.48. This helps to simplify derivations while preserving the correctness of the analysis because this transformation does not affect directions of the right-eigenvectors.

$$\Upsilon = diag\left(\frac{1}{\rho^-(c_p^-)^2}, \frac{1}{\rho^-(c_s^-)^2}, \frac{1}{\rho^-(c_s^-)^2}, 1, 1, 1, \frac{1}{\rho^+(c_s^+)^2}, \frac{1}{\rho^+(c_s^+)^2}, \frac{1}{\rho^+(c_p^+)^2}\right) \tag{3.47}$$

$$\mathcal{R} = R^{-+}\Upsilon = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ \frac{\lambda^-}{\lambda^-+2\mu^-} & 0 & 0 & 0 & 1 & 0 & 0 & 0 & \frac{\lambda^+}{\lambda^++2\mu^+} \\ \frac{\lambda^-}{\lambda^-+2\mu^-} & 0 & 0 & 0 & 0 & 1 & 0 & 0 & \frac{\lambda^+}{\lambda^++2\mu^+} \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ \frac{1}{Z_1^-} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -\frac{1}{Z_1^+} \\ 0 & \frac{1}{Z_2^-} & 0 & 0 & 0 & 0 & 0 & -\frac{1}{Z_2^+} & 0 \\ 0 & 0 & \frac{1}{Z_3^-} & 0 & 0 & 0 & -\frac{1}{Z_3^+} & 0 & 0 \end{pmatrix} \tag{3.48}$$

where Z_1^-, Z_2^-, Z_3^- and Z_1^+, Z_2^+, Z_3^+ are called impedances which are equal to

$$\begin{aligned} Z_1^- &= \rho c_p^-, \; Z_1^+ = \rho c_p^+ \\ Z_2^- &= \rho c_s^-, \; Z_2^+ = \rho c_s^+ \\ Z_3^- &= \rho c_s^-, \; Z_3^+ = \rho c_s^+ \end{aligned} \tag{3.49}$$

$\hat{Q}^b$ and $\hat{Q}^c$ states can be found by using the jump condition properties of a solution of the Riemann problem applied to the wave propagation problem (shown in Eq. 2.7). This is shown as follows

$$\hat{Q}^b - \hat{Q}^- = \alpha_1\mathfrak{r}_1^- + \alpha_2\mathfrak{r}_2^- + \alpha_3\mathfrak{r}_3^- \tag{3.50}$$

$$\hat{Q}^+ - \hat{Q}^c = \alpha_7\mathfrak{r}_7^+ + \alpha_8\mathfrak{r}_8^+ + \alpha_9\mathfrak{r}_9^+ \tag{3.51}$$

where $\mathfrak{r}_i^{\pm}$ is the i-th column vector of $\mathcal{R}$.

Eq. 3.50 and the structure of $\mathcal{R}$ can be used to find values of α_1, α_2, and α_3 via components highlighted in red in Eq. 3.48. Similarly, one can obtain α_7, α_8, α_9 using Eq. 3.51 via

components highlighted in blue. Moreover, we are looking for a solution where stresses across a fault are continuous and thus $\hat{\sigma}^b_{1j} = \hat{\sigma}^c_{1j}$, which I denoted as $\hat{\sigma}^m_j$ for simplicity.

$$\begin{aligned} \hat{\sigma}^m_1 - \hat{\sigma}^-_{11} = \alpha_1,\ \hat{\sigma}^+_{11} - \hat{\sigma}^m_1 = \alpha_9 \\ \hat{\sigma}^m_2 - \hat{\sigma}^-_{12} = \alpha_2,\ \hat{\sigma}^+_{12} - \hat{\sigma}^m_2 = \alpha_8 \\ \hat{\sigma}^m_3 - \hat{\sigma}^-_{13} = \alpha_3,\ \hat{\sigma}^+_{13} - \hat{\sigma}^m_3 = \alpha_7 \end{aligned} \tag{3.52}$$

The sum of Eq. 3.50 and Eq. 3.51 together is given by

$$[\![\hat{Q}]\!] - [\![\delta\hat{Q}]\!] = \alpha_1 \mathfrak{r}^-_1 + \alpha_2 \mathfrak{r}^-_2 + \alpha_3 \mathfrak{r}^-_3 + \alpha_7 \mathfrak{r}^+_7 + \alpha_8 \mathfrak{r}^+_8 + \alpha_9 \mathfrak{r}^+_9 \tag{3.53}$$

where the following variables are introduced: $[\![\hat{Q}]\!] = \hat{Q}^+ - \hat{Q}^-$ and $[\![\delta\hat{Q}]\!] = \hat{Q}^c - \hat{Q}^b$.

$$[\![\hat{Q}]\!] = \begin{pmatrix} \hat{\sigma}^+_{11} - \hat{\sigma}^-_{11} \\ \hat{\sigma}^+_{22} - \hat{\sigma}^-_{22} \\ \hat{\sigma}^+_{33} - \hat{\sigma}^-_{33} \\ \hat{\sigma}^+_{12} - \hat{\sigma}^-_{12} \\ \hat{\sigma}^+_{23} - \hat{\sigma}^-_{23} \\ \hat{\sigma}^+_{13} - \hat{\sigma}^-_{13} \\ \hat{u}^+_1 - \hat{u}^-_1 \\ \hat{u}^+_2 - \hat{u}^-_2 \\ \hat{u}^+_3 - \hat{u}^-_3 \end{pmatrix} = \begin{pmatrix} [\![\hat{\sigma}_{11}]\!] \\ [\![\hat{\sigma}_{22}]\!] \\ [\![\hat{\sigma}_{33}]\!] \\ [\![\hat{\sigma}_{12}]\!] \\ [\![\hat{\sigma}_{23}]\!] \\ [\![\hat{\sigma}_{13}]\!] \\ [\![\hat{u}_1]\!] \\ [\![\hat{u}_2]\!] \\ [\![\hat{u}_3]\!] \end{pmatrix},\ [\![\delta\hat{Q}]\!] = \begin{pmatrix} 0 \\ \hat{\sigma}^c_{22} - \hat{\sigma}^b_{22} \\ \hat{\sigma}^c_{33} - \hat{\sigma}^b_{33} \\ 0 \\ \hat{\sigma}^c_{23} - \hat{\sigma}^b_{23} \\ 0 \\ \hat{u}^c_1 - \hat{u}^b_1 \\ \hat{u}^c_2 - \hat{u}^b_2 \\ \hat{u}^c_3 - \hat{u}^b_3 \end{pmatrix} = \begin{pmatrix} 0 \\ [\![\delta\hat{\sigma}_{22}]\!] \\ [\![\delta\hat{\sigma}_{33}]\!] \\ 0 \\ [\![\delta\hat{\sigma}_{23}]\!] \\ 0 \\ [\![\delta\hat{u}_1]\!] \\ [\![\delta\hat{u}_2]\!] \\ [\![\delta\hat{u}_3]\!] \end{pmatrix} \tag{3.54}$$

The substitution of Eq. 3.52 to Eq. 3.53 yields

$$[\![\hat{u}_i]\!] - [\![\delta\hat{u}_i]\!] = \frac{\alpha_i}{Z^-_i} - \frac{\alpha_{9-i+1}}{Z^+_i} = \frac{\hat{\sigma}^m_i - \hat{\sigma}^-_{1i}}{Z^-_i} - \frac{\hat{\sigma}^+_{1i} - \hat{\sigma}^m_i}{Z^+_i} \tag{3.55}$$

where $i \in (1, 2, 3)$.

Re-arranging the terms in Eq. 3.55 allows to express $\hat{\sigma}^m_i$ as follows

$$\hat{\sigma}^m_i = \eta_i \left([\![\hat{u}_i]\!] + \frac{\hat{\sigma}^+_{1i}}{Z^+_i} + \frac{\hat{\sigma}^-_{1i}}{Z^-_i} \right) - \eta_i [\![\delta\hat{u}_i]\!] \tag{3.56}$$

where $\eta_i = \frac{Z^+_i Z^-_i}{Z^+_i + Z^-_i}$.

In this model, fault opening cannot occur; thus, particle velocities from "+" and "−" sides must be equal - i.e., $[\![\delta\hat{u}_1]\!] = 0$. Therefore, $\hat{\sigma}^m_1$ can be directly computed as follows

$$\hat{\sigma}^m_1 = \eta_i \left([\![\hat{u}_1]\!] + \frac{\hat{\sigma}^+_{11}}{Z^+_1} + \frac{\hat{\sigma}^-_{11}}{Z^-_1} \right) \tag{3.57}$$

Therefore, one can evaluate the fault strength using Eq. 2.17 as follows

$$\tau_s = max(0, -\mu_f \hat{\sigma}^m_1) \tag{3.58}$$

where $\mu_f = \mu_{f,il}$.

The slip rate $\Delta \boldsymbol{u}$ discussed in Eq. 2.16 also equals the sum of the $[\![\delta \hat{u}_2]\!]$ and $[\![\delta \hat{u}_3]\!]$ vectors. Therefore, the absolute value of the slip rate is given by

$$\Delta \boldsymbol{u} = \sqrt{[\![\delta \hat{u}_2]\!]^2 + [\![\delta \hat{u}_3]\!]^2} \tag{3.59}$$

The antiparallel property of the shear stress and the slip rate of the Coulomb model can be exploited as the closing equation for the inverse Riemann problem. The projection of the vector Eq. 2.20 onto 2- and 3-directions in the fault-aligned system results in

$$\begin{aligned} [\![\delta \hat{u}_2]\!]\tau_s - |\Delta \boldsymbol{u}|\hat{\sigma}_2^m = 0 \\ [\![\delta \hat{u}_3]\!]\tau_s - |\Delta \boldsymbol{u}|\hat{\sigma}_3^m = 0 \end{aligned} \tag{3.60}$$

Plugging Eq. 3.60 into Eq. 3.56 yields

$$[\![\delta \hat{u}_i]\!] = \frac{|\Delta \boldsymbol{u}|\eta_i}{\tau_s + \eta_i|\Delta \boldsymbol{u}|} \left([\![\hat{u}_i]\!] + \frac{\hat{\sigma}_{1i}^+}{Z_i^+} + \frac{\hat{\sigma}_{1i}^-}{Z_i^-} \right) = \frac{|\Delta \boldsymbol{u}|}{\tau_s + \eta_i|\Delta \boldsymbol{u}|}\theta_i \tag{3.61}$$

where $i \in (2, 3)$

Combining Eq. 3.61 and Eq. 3.59 allows to evaluate the absolute value of the slip rate.

$$\tau_s + \eta_s|\Delta \boldsymbol{u}| = \sqrt{\theta_2^2 + \theta_3^2} \tag{3.62}$$

where $\eta_s = \eta_2 = \eta_3$ by definition (see, Eq. 3.49 and Eq. 3.57).

According to Eq. 3.58, the fault strength is a function of normal stress $\hat{\sigma}_1^m$ and the friction coefficient μ_f, where the latter is determined by a particular friction law. As shown in Eq. 2.21, the friction coefficient is a function of slip rate magnitude $|\Delta \boldsymbol{u}|$ and state variable ψ. Therefore, Eq. 3.62 is generally non-linear and may require the Newton-Raphson algorithm to be solved regarding the slip rate magnitude. However, in some special cases - e.g., the Linear Slip Weakening (see Eq. 2.23) - the friction coefficient is a function of only the state variable, which is defined as an ordinary differential equation for which the value of the slip rate magnitude can be taken from the previous time step. In such cases, the slip rate magnitude can be found as follows

$$|\Delta \boldsymbol{u}| = \frac{\sqrt{\theta_2^2 + \theta_3^2} - \tau_s}{\eta_s} \tag{3.63}$$

The solution of Eq. 3.62 must be used to evaluate $[\![\delta \hat{u}_2]\!]$ and $[\![\delta \hat{u}_3]\!]$ using Eq. 3.61; thus, $\hat{\sigma}_2^m$ and $\hat{\sigma}_3^m$ can be computed according to Eq. 3.56.

Using Eq. 3.50, Eq. 3.51, and Eq. 3.52, the velocity components on both sides of the interface are given by

$$\begin{aligned} \hat{u}_i^b - \hat{u}_i^- = \frac{\alpha_i}{Z_i^-} \therefore \hat{u}_i^b = \hat{u}_i^- + \frac{\hat{\sigma}_i^m - \hat{\sigma}_{1i}^-}{Z_i^-} \\ \hat{u}_i^+ - \hat{u}_i^c = \frac{\alpha_{9-i+1}}{Z_i^-} \therefore \hat{u}_i^c = \hat{u}_i^+ + \frac{\hat{\sigma}_i^m - \hat{\sigma}_{1i}^+}{Z_i^-} \end{aligned} \tag{3.64}$$

At this point, $\hat{Q}^b_{il}$ and $\hat{Q}^c_{il}$ vectors can be assembled and applied to Eq. 3.44 to compute fluxes for adjacent elements in the face-aligned coordinate system.

4. HPC Concepts in SeisSol

It is important to discuss the state of the CPU implementation of *SeisSol* to better understand the problems faced during this research, as well as implementational details, achievements, and contributions. Therefore, this chapter lists the main HPC features adapted in *SeisSol* prior to this work, which result in performing highly-efficient and scalable earthquake simulations on distributed-memory CPU-based supercomputers.

Like many HPC applications, *SeisSol* had been originally designed and tuned for traditional multicore and manycore x86 platforms [50]. It fully exploits the main capabilities of vector processing units of modern x86 CPUs. *SeisSol* scales well on manycore shared-memory systems because 1) coarse granularities of CPU tasks resulted from the element-wise updates of Eq. 3.35, 2) abundant caching caused by small matrix sizes and intelligent data pre-fetching, and 3) little synchronizations between CPU threads.

Among various distributed-memory programming models, e.g., Chapel, UPC, X10, etc., *SeisSol* utilizes the Message Passing Interface (MPI), which is the most popular one in the field of scientific computing. Because asynchronous message-passing is well overlapped with computations, *SeisSol* scales well on distributed-memory machines up to ≈ 8000 CPU nodes (for example, see [13] or [114]).

In Section 4.1, I discuss *SeisSol*'s main macro-kernels and the adapted memory layout. In Section 4.1, I explain the main aspects of the code generation established in *SeisSol*, resulting in 1) flexibility in implementing various wave propagation models and 2) high single CPU core performance. Section 4.3 describes the original *SeisSol*'s task decomposition and the shared-memory multiprocessing of sub-tasks, which are implemented with *OpenMP*. In Section 4.4, I discuss the LTS-specific tetrahedral mesh partitioning and the main aspects of MPI-based parallelization.

4.1. Data Layout and Macro Kernels

SeisSol is based on the column-major layout for storing multidimensional arrays where the leading dimension is preferred to be the longest one. Taking into account that the size of polynomial basis $\mathcal{B}$, dictated by the maximum polynomial degree $\mathcal{N}$, is much larger than the number of unknown physical quantities - i.e., 9 for the elastic wave propagation - *pl*-indices of DOFs in Eq. 3.17 must to be swapped. This entails the appropriate changes in Eq. 3.35 and Eq. 3.33, which can be achieved by matrix transpositions of their left- and right-hand sides.

Eq. 3.35 can be split into entirely element-local and data-dependent parts, which can be written as follows

$$\left(Q_{lp}^{m}\right)^{t+\Delta t,*} = \left(Q_{lp}^{m}\right)^{t} - I_{surf}^{local}\left(\mathcal{T}_{lp}^{m}\right) + I_{vol}\left(\mathcal{T}_{lp}^{m}\right) + I_{src} \tag{4.1}$$

$$\left(Q_{lp}^{m}\right)^{t+\Delta t} = \left(Q_{lp}^{m}\right)^{t+\Delta t,*} - I_{surf}^{nghb}\left(\mathcal{T}_{lp}^{m_j}\right) \tag{4.2}$$

where I_{surf}^{local}, I_{surf}^{nghb}, I_{vol} and I_{src} are *local surface*, *neighbor surface*, *volume* and *source terms* integrals, which constitute the main computational macro-kernels in *SeisSol*; $\mathcal{T}_{lp}^{m}$ is the time-integrated DOFs of the m-the tetrahedron (see Eq. 3.34).

Uphoff et al. in [117] proposed to decompose the flux matrices (i.e., $\mathcal{F}_{tk}^{+,jih}$ and $\mathcal{F}_{tk}^{-,j}$) by representing functions compositions involved into the corresponding integrals using two-dimensional basis $\tilde{\Psi}_1 \dots \tilde{\Psi}_{\tilde{\mathcal{O}}}$, where $\tilde{\mathcal{O}} = \frac{1}{2}(\mathcal{O})(\mathcal{O}+1)$. The authors show that this approach 1) leads to fewer floating-point operations using an optimal matrix multiplication ordering and 2) reduces data eviction from low-level CPU caches. The decomposition is given by

$$\mathcal{F}_{kl}^{-,j} = R_{km}^{j} \cdot \int_{\delta T^{e_j}} \tilde{\Psi}_m(\boldsymbol{\chi})\tilde{\Psi}_n(\boldsymbol{\chi})\, d\chi_1 d\chi_2 \cdot R_{nl}^{j} = R_{km}^{j} f_{mn}^{-} R_{nl}^{j} \tag{4.3}$$

$$\mathcal{F}_{kl}^{+,jih} = R_{km}^{j} \cdot \int_{\delta T^{e_j}} \tilde{\Psi}_m(\boldsymbol{\chi})\tilde{\Psi}_n(\tilde{\boldsymbol{\chi}}^{h}(\boldsymbol{\chi}))\, d\chi_1 d\chi_2 \cdot R_{nl}^{i} = R_{km}^{j} f_{mn}^{+,h} R_{nl}^{i} \tag{4.4}$$

where R_{km}^{j} and R_{ln}^{i} matrices are projections from the three-dimensional basis onto the two-dimensional one for j and i faces, respectively. The details regarding the projection matrices are given in [117].

Therefore, the main macro-kernels are given by

$$\begin{aligned}
I_{surf}^{local}\left(\mathcal{T}_{lp}^{m}\right) &= \sum_{j=1}^{4} |S_j| \tilde{R}_{lm}^{j} \tilde{f}_{nt}^{-,j} \mathcal{T}_{tq}^{m} \mathcal{A}_{qp}^{-,m} \\
I_{surf}^{nghb}\left(\mathcal{T}_{lp}^{m_j}\right) &= \sum_{j=1}^{4} |S_j| \tilde{R}_{lm}^{i} f_{nm}^{+,h} R_{mt}^{j} \mathcal{T}_{tq}^{m_j} \mathcal{A}_{qp}^{+,m} \\
I_{vol}\left(\mathcal{T}_{lp}^{m}\right) &= -\tilde{K}_{lt}^{1} \mathcal{T}_{tq}^{m} A_{qp}^{*,m} - \tilde{K}_{lt}^{2} \mathcal{T}_{tq}^{m} B_{qp}^{*,m} - \tilde{K}_{lt}^{3} \mathcal{T}_{tq}^{m} C_{qp}^{*,m} \\
I_{src} &= M_{lk}^{-1} \left(\Psi_k(\boldsymbol{\xi}_s)\right)^{T} \sum_{j=1}^{\mathcal{O}} \omega_j s_{sp}(\tau_j)
\end{aligned} \tag{4.5}$$

where $\tilde{R}_{lm}^{j} = \frac{1}{|J|} M_{lk}^{-1} R_{kn}^{j}$; $\tilde{f}_{nt}^{-,j} = f_{nm}^{-,j} R_{mt}^{j}$; and $\tilde{K}_{lt}^{i} = M_{lk}^{-1} K_{tk}^{i}$, which can be pre-computed in advance.

Similarly, Eq. 3.33 needs to be transposed and thus can be written as follows

$$\mathcal{D}_{lp}^{m,i} = M_{lk}^{-1} \left(\Psi_k(\boldsymbol{\xi}_s)\right)^{T} \Theta_{l}^{m,i-1} S_{lps} - \hat{K}_{lt}^{1} \mathcal{D}_{tq}^{m,i-1} A_{qp}^{m} - \hat{K}_{lt}^{2} \mathcal{D}_{tq}^{m,i-1} B_{qp}^{m} - \hat{K}_{lt}^{3} \mathcal{D}_{tq}^{m,i-1} C_{qp}^{m} \tag{4.6}$$

where $\hat{K}_{lt}^{i} = M_{lk}^{-1} (K_{kt}^{i})^{T}$.

Table 4.1.: Matrix sizes for typical convergence orders.

Matrices	**Convergence Order**			
	4	**5**	**6**	**7**
Q, $\mathcal{D}$, $\mathcal{T}$	20×9	35×9	56×9	84×9
M, $\hat{K}^1$, $\hat{K}^2$, $\hat{K}^3$, $\tilde{K}^1$, $\tilde{K}^2$, $\tilde{K}^3$	20×20	35×35	56×56	84×84
A^*, B^*, C^*, $\mathcal{A}^-$, $\mathcal{A}^+$, $\hat{T}$	9×9	9×9	9×9	9×9
$\mathcal{V}^f$, $\mathcal{V}^{g,h}$	25×20	36×35	49×56	64×84
$\tilde{R}$	10×20	15×35	21×56	28×84
$\tilde{f}^{-,j}$	20×10	35×15	56×21	84×28
$f^{+,h}$	10×10	15×15	21×21	28×28

Therefore, Eq. 4.6 and Eq. 3.34 constitute a so-called *ader* macro-kernel which is given by

$$I_{ader}\left(Q_{lp}^{m}\right) = \sum_{i=0}^{\mathcal{O}-1} \frac{(t+\Delta t-t_0)^{i+1}-(t-t_0)^{i+1}}{(i+1)!}\mathcal{D}_{lp}^{m,i} \tag{4.7}$$

where $\mathcal{D}_{lp}^{m,0}$ is equal to Q_{lp}^{m} by definition.

Eq. 3.45 and Eq. 3.46 define so-called *interpolation* macro-kernels, which are used to project a volumetric solution of element m or its neighbor m_j to the f-th face of tetrahedron m.

$$\begin{aligned} I_{inter}^{local}\left(\mathcal{T}_{lp}^{m}\right) &= \mathcal{V}_{lk}^{f}\mathcal{T}_{kp}^{m}\hat{T}_{pr}^{m} \\ I_{inter}^{nghb}\left(\mathcal{T}_{lp}^{m_j}\right) &= \mathcal{V}_{lk}^{g,h}\mathcal{T}_{kp}^{m_j}\hat{T}_{pr}^{m} \end{aligned} \tag{4.8}$$

where $\mathcal{V}_{lk}^{f} = \left(\Phi_m(\boldsymbol{\xi}^f(\boldsymbol{\chi}_i))\right)^T$ and $\mathcal{V}_{lk}^{g,h} = \left(\Phi_m(\boldsymbol{\xi}^g(\tilde{\boldsymbol{\chi}}^h(\boldsymbol{\chi}_i)))\right)^T$ are the Vandermonde matrices; $\hat{T}_{pr}^{m}$ is equal to $\left(T_{rp}^{-1}(\boldsymbol{n})\right)^T$ and pre-computed in advance for each face of element m.

As a reference, Table 4.1 shows matrix sizes for the most commonly used convergence orders in *SeisSol*. In practice, columns of many matrices are padded to a size multiple of the vector register length of the underlying CPU architecture. Together with the memory alignment, it allows *SeisSol* to utilize more efficient (i.e., low latency) *load* and *store* vector instructions.

4.2. Code Generation

The macro-kernels, discussed in Section 4.1, mainly involve matrix multiplications and additions in the context of elastic wave propagation. It is worth mentioning that the kernels do not impose any particular matrix multiplications orders. Thus, the naïve

(i.e., right-to-left) multiplications may be sub-optimal regarding computational efficiency. Moreover, other wave propagation models (e.g., viscoelastic one) may result in operating on higher-order tensors. Some tensors/matrices may be vastly sparse, which needs to be considered while implementing a computer-efficient ADER-DG scheme.

Uphoff and Bader addressed these problems in [116] while working on Yet Another Tensor Toolbox (*YATeTo*) which is a *Python*-based Domain Specific Language (DSL) for the discontinuous Galerkin methods. The language operates on scalar and tensor data types and supports primary tensor operations, namely: 1) a linear combination of tensors and 2) a tensor product. *YATeTo* automatically performs a tensor contraction when a tensor product contains repeating (dummy) indices. The main objective of the language is 1) to provide flexibility and convenience for scientists while working on implementing numerical schemes stemming from DG-like methods and 2) to generate a high-performance code for a single CPU core. In the following, I describe the main concepts of the language and its compiler, which is necessary for understanding the main contributions of this thesis.

In *YATeTo*, a tensor is defined by 1) a unique name, 2) its shape, and 3) its sparsity pattern. *YATeTo*'s compiler front-end is based on the overloaded "+" and "*" *Python*-operators, and thus the precedence and associativity of *YATeTo*'s operators are inherited from *Python*. The language grammar does not support parentheses, meaning a user cannot enforce a particular order of operations within a tensor expression. Finding an optimal tensor ordering that results in minimal floating-point operations, also known as *Strength Reduction*, is an NP-complete problem, which can be deduced from [72]. Therefore, delegating this problem to the compiler is one of the main design decisions of the *YATeTo* DSL. Firstly, an optimal solution found by the compiler, based on the algorithm described by Lam, Sadayappan, and Wenger in [72], results in a faster and, thus, more energy-efficient implementation of a tensor expression. Secondly, it simplifies writing *YATeTo* programs and reduces HPC software development time.

YATeTo can reduce the formal tensor sizes by excluding zero blocks which may appear during a tensor chain evaluation. This idea is based on computing EQuivalent SParsity Patterns (EQSPPs), which were initially introduced in [115] for matrices and later extended for tensors in [116]. As a result, a formal size of a tensor, which is a dense block, encompasses only those elements which guarantee to generate non-zero results during an operation where the tensor gets involved. This reduces the required floating-point operations and, thus, leads to a more efficient implementation.

As discussed in [105] and similar works, a tensor inner product can be implemented in many different ways, e.g., *Transpose-Transpose-GEMM-Transpose*, *GEMM-like Tensor-Tensor Multiplication*, *Loops-over-GEMMs*, etc. The performance of a particular algorithm depends on the shapes of the involved tensors, and, in general, it is hard to predict the best algorithm in advance. *Tensor Contraction Code Generator*[1] contains several algorithms for implementing an inner tensor product and uses a combination of a cost model and an autotuning technique to find the best candidate. It is worth pointing out that almost all algorithms, except the naïve nested loops approach, are designed to utilize a GEneral

[1] https://github.com/HPAC/tccg

Matrix Multiply (GEMM) as a micro-kernel. The GEMM kernel can be taken from the BLAS library supplied by a hardware vendor (e.g., Intel-MLK, OpenBLAS, BLIS, cuBLAS, rocBLAS, etc) to achieve the maximum performance.

Uphoff and Bader argued in [116] that *Loops-over-GEMMs* (LoG) algorithm is best suited to the requirements of the domain for which *YATeTo* was designed. They list the following reasons based on the fact that small tensors, which are typically arisen from the DG methods, fit into the CPU caches. Firstly, the sparsity of the involved tensors can be fully and trivially exploited because the algorithm does not require splitting tensors into patches and to perform/optimize packing, re-shaping and unpacking subroutines for them. Secondly, the absence of such operations reduces the overall number of *load* and *store* CPU instructions, making the algorithm to be more compute-intensive. Lastly, the code generation for *LoG* becomes trivial once the looping dimensions and their orders are found. However, the last condition is, in fact, a complex problem because many possible configurations are possible. *YATeTo* employs dynamic programming to find the one which is estimated to result in the minimal run time.

The compilation process starts by building an Abstract Syntax Tree (AST) for a tensor expression. Afterward, *YATeTo* performs a semantic analysis by checking whether the indices and shapes of the involved tensors constitute a valid tensor expression. Then, EQSPPs are computed for each tensor. At this point, the AST is quite shallow because no concrete evaluation order is imposed. This is done during *Strength Reduction*, which follows immediately. This step results in a deep AST consisting of only *tensor-assignment*, *scaled-tensor-sum*, *tensor-product* and *tensor-contraction* (called as *IndexSum*) nodes. Then, *YATeTo* identifies sub-trees that represent inner tensor products and maps them to *LoG* nodes. The first compilation stage finishes by computing optimal looping dimensions and loops ordering for each *LoG* node. Afterward, the AST is transformed into a linearized Intermediate Representation (IR), which resembles a three-address code. This representation is more convenient for finding and optimizing temporary storage required for saving intermediate results while evaluating the whole tensor expression. During the last stage, *YATeTo* traverses the IR from top to bottom and generates C++ code in memory, which later, together with the initialization code and other auxiliary data structures, is written to text files.

Code generation for the *LoG* IR instruction is split into two steps. During the first one, *YATeTo* generates C++ code for the outer for-loops if they exist. Otherwise, the step is omitted. The second step can be performed in three different ways, namely: 1) *YATeTo* can insert a function call to the GEMM subroutine taken from an optimized BLAS library; 2) *YATeTo* can invoke a hardware-aware code generator (if available), e.g., LIBXSMM [51] or PSpaMM [102]; and 3) *YATeTo* can generate C++ code by itself. The last two variants allow *YATeTo* to generate code for *Dense*×*Dense*, *Dense*×*Sparse*, and *Sparse*×*Dense* matrix multiplications.

Memory alignment and data padding, discussed in Section 4.1, leads to a better vectorization of GEMM micro-kernels. Intelligent data pre-fetching of small matrices into low-level caches keeps the vector-processing units busy. As a result, the resultant arithmetic inten-

sity of the generated tensor expressions is quite high. As shown in [117] and [123], *SeisSol* can reach up to ≈ 40-50% of the peak CPU performance of modern x86 processors.

As it can be seen from the discussion, *YATeTo*'s software design is CPU-centric. It mainly focuses on binary tensor operations where the core of computations is built upon GEMMs. Multithreading is delegated to the host application - e.g., *SeisSol*. Apart from explicit data pre-fetching, no other explicit data manipulations are done - i.e., the generated code relies on the hardware for moving data within the cache hierarchy.

4.3. Multithreading

By definition, a task is a unit of work for a computing device to execute. The term is ambiguous, and the interpretation depends on a specific algorithm. In *SeisSol*, an LTS-task is defined as a time-update of elements within a time-cluster. Splitting Eq. 3.35 into Eq. 4.1 and Eq. 4.2 generates two data parallel in-order tasks for a time-cluster. A sub-task is determined as an execution of one or more tensor expressions defined by the macro-kernels for a single mesh element. *SeisSol* employs the traditional *OpenMP* parallel programming model for distributing sub-tasks between CPU cores within a shared-memory system.

In general, sub-tasks are not equal. Firstly, a face of an element can belong to 1 out of 48 different flux matrices $\mathcal{F}^{+,jih}$ which have different sparsity patterns and thus may entail a slightly different number of floating-point operations within I^{nghb}_{surf} macro-kernel (see Algorithm 1). Secondly, an element face can belong to a boundary condition requiring a different numerical treatment. Therefore, each sub-task may have a different execution control flow determined by constraints imposed on a mesh element. Nevertheless, *SeisSol* uses static *OpenMP* work scheduling, showing a little load imbalance between CPU cores in practice.

Each LTS-task contains two parallel fork-join regions. The overheads related to threads creations and synchronizations are negligible because 1) sub-tasks in *SeisSol* are quite coarse-grained, and 2) each *OpenMP* thread is assigned to independently process a sub-set of elements on a single CPU core. As shown in Fig. 4.1, *SeisSol*'s parallel performance scales almost linearly on the shared-memory systems relative to the number of CPU cores.

SeisSol-proxy is a set of benchmarks that contains the main *SeisSol*'s macro-kernels. The proxy implements a single LTS time-cluster, which can also be viewed as the GTS scheme. The proxy can run individual macro-kernels as well as combinations of them. Results, shown in Fig. 4.1, were obtained by running all macro-kernels in the order given by Eq. 4.1 and Eq. 4.2.

Algorithm 1 CPU implementation of the Neighbor Surface Integral - i.e., I_{surf}^{nghb}

procedure ComputeNSI($LtsLayer$,$PreComputedData$)
  $M \leftarrow LtsLayer.getClusterSize()$
  for m **from** 1 **to** M **do in parallel**
    $Q_{lp}^{m} \leftarrow LtsLayer.getDOFs(m)$
    $\mathcal{A}_{qp}^{+,m} \leftarrow LtsLayer.getFluxSolver(m)$
    $ElementInfo \leftarrow LtsLayer.getElementInfo(m)$
    for j **from** 1 **to** 4 **do**
      $FaceKind \leftarrow ElementInfo.getFaceKind(j)$
      if $FaceKind == Regular$ **or** $FaceKind == Periodic$ **then**
        $h, i \leftarrow ElementInfo.getParameters(j)$
        $\tilde{R}_{lm}^{i}, f_{nm}^{+,h}, R_{mt}^{j} \leftarrow PreComputedData.getFluxMatrices(i, h, j)$
        $m_j \leftarrow ElementInfo.getNeighbourElementIndex(j)$
        $\mathcal{T}_{tq}^{m_j} \leftarrow LtsLayer.getIntegratedDOFs(m_j)$
        $Q_{lp}^{m} = Q_{lp}^{m} - \tilde{R}_{lm}^{i} \cdot f_{nm}^{+,h} \cdot R_{mt}^{j} \cdot \mathcal{T}_{tq}^{m_j} \cdot \mathcal{A}_{qp}^{+,m}$ ▷ Generated Code
      end if
    end for
  done
end procedure

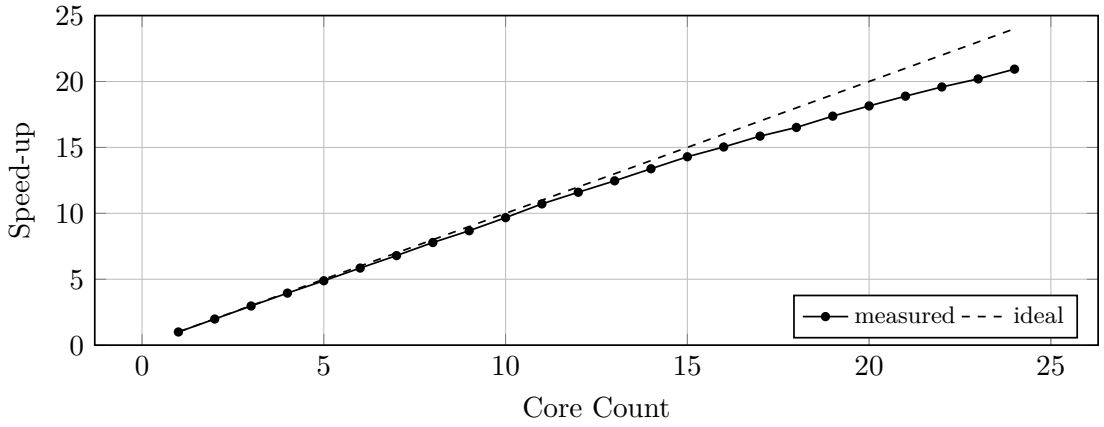


Figure 4.1.: Strong-scaling parallel efficiency of *SeisSol*-proxy on a single AMD EPYC 7402 CPU. The result were obtained with: $\mathcal{O} = 6$, double-precision, 96000 elements.

4.4. Distributed-Memory Computing

Simulations of extreme-scale complex earthquake events involve highly resolved 3D geometries, which result in processing dozens or hundreds of millions of mesh elements per time step. Such simulations can be performed only on distributed-memory machines which overcome the memory capacity limits of modern single-board computers. Moreover, the increased data processing power reduces time-to-solution.

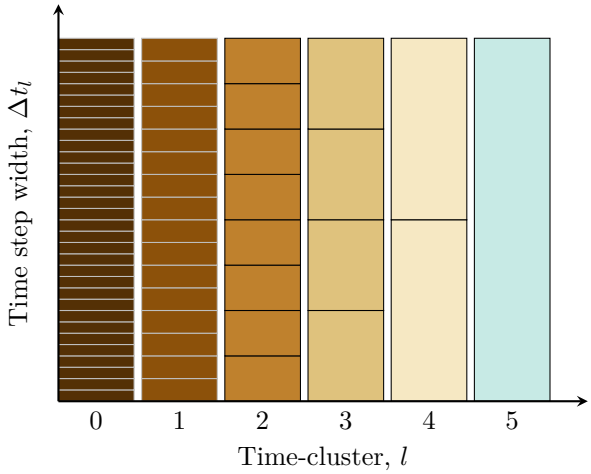


Figure 4.2.: Cluster-wise time stepping scheme with the update ratio equal to 2.

Distributed-memory computing for the ADER-DG and similar methods is not straightforward because such numerical schemes involve data dependencies between elements and their neighbors (see Eq. 4.2). In such circumstances, data exchange between computational resources connected over a network is unavoidable. Therefore, intelligent mesh partitioning is required to reduce overheads caused by communication between computers (nodes). It is also necessary to alter the sub-tasks scheduling and intelligently use asynchronous communication between nodes to overlap the data exchange with computations in order to achieve better utilization of computational resources. Moreover, computational work needs to be more or less equally distributed between nodes to minimize their load imbalance and, thus, to increase the efficiency of parallel computations.

In *SeisSol*, a unit of work is defined as a time-update of an element. The LTS scheme implies that some elements must be updated more frequently than others. Given an update ratio r and the total number of LTS time-clusters L (see Section 3.5), the update frequency w of an element m belonging to cluster l is given by

$$w_m = r^{L-l-1} \tag{4.9}$$

It is useful to reason about update frequencies relative to the time step width of cluster $L-1$. In this context, frequency w_m is equal to the amount of work required to update an element within a time interval $[t, t + \Delta t_{L-1}]$ (see Fig. 4.2). This metric is independent of the simulation time imposed by an earthquake model, can be easily evaluated, and thus is helpful for partitioning.

An unstructured mesh can be considered as an undirected graph whose vertices are located in the elements centers, and whose edges connect elements with their direct neighbors. Such mesh representation allows one to find a distribution of elements between nodes by solving the graph partitioning problem. The problem belongs the NP-hard complexity class. *SeisSol* uses the *ParMetis* [60] library, which partitions a graph using a multilevel k-way algorithm, whose complexity is linear with respect to the number of vertices in the

graph [59]. *ParMetis* operates on weighted graphs, i.e., when vertices and edges have their own associated k weights, where $k \in \mathbb{N}^+$. The algorithm successively coarsens a graph until only a small number of vertices can be trivially distributed between N parts. Then, the algorithm successively performs a graph refinement until it reaches the original graph. During both steps, the algorithm utilizes different heuristics to balance vertex weights between partitions while minimizing the edge-cuts, determined by edge-weights, at the same time.

In *SeisSol*, single-constraint graph partitioning is used. The update frequencies w_m are considered as vertex weights. This approach only aims to distribute the computational work equally between computing resources. *SeisSol* does not impose any particular constraints on vertex edges; thus, all edge-weights are equal to 1. As mentioned by Breuer, Heinecke, and Bader in [12], "this approach is simple but proves efficiency at scale up to 100 million elements".

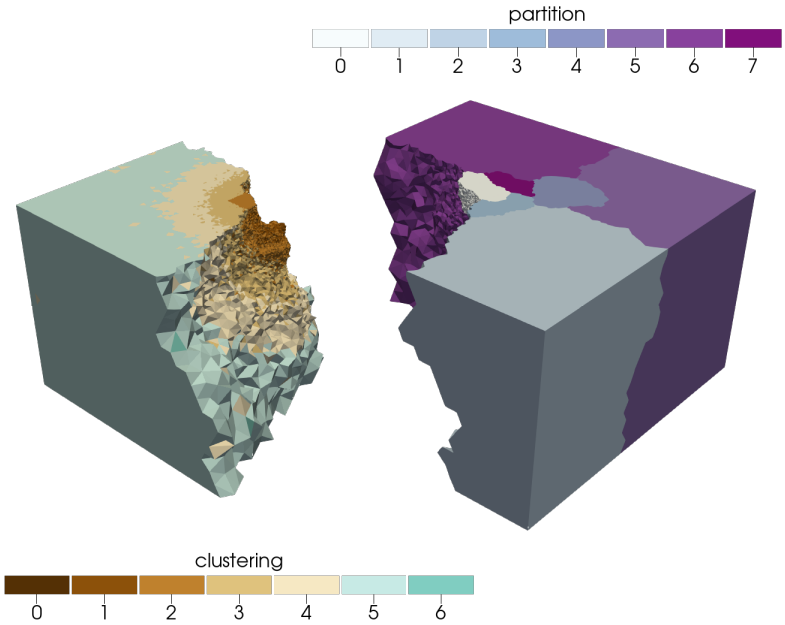


Figure 4.3.: Example of time-clustering and mesh partitioning in *SeisSol*.

Fig. 4.3 shows an example of partitioning a mesh, consisting of approximately 2.6 million elements, into 8 sub-domains. The mesh is locally refined around a single kinematic point source located approximately in the middle of the domain. Partition 1 is shifted on the left on purpose to show time-clustering within a sub-domain. The computational work is almost equally distributed between sub-domains, according to Fig. 4.4. The maximal work imbalance, computed based on the data shown in Fig. 4.4, is about 0.1%.

Fig. 4.4 shows the resultant partitioning of time-clusters between sub-domains. In contrast to a good work balance, one can observe that elements of time-clusters are not equally distributed. Moreover, some of them are missing in some sub-domains. This results in a complex communication pattern between nodes.

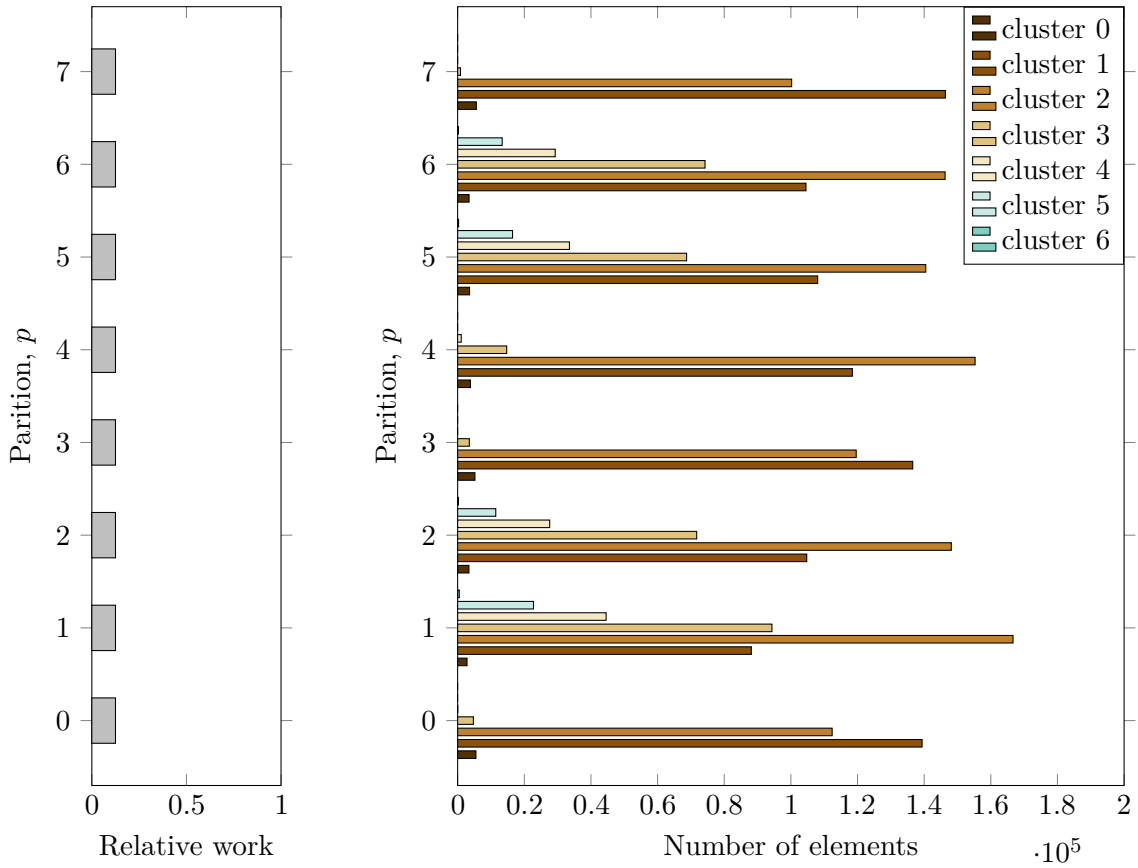


Figure 4.4.: Relative distribution of work between 8 partitions of the mesh shown in Fig. 4.3, on the left. The corresponding distribution of elements between time-clusters in each sub-domain, on the right.

The data processing speed of compute-units may vary between nodes, e.g., due to differences in processors clocking. In this context, even a distributed-memory CPU-based computer can be considered as a heterogeneous system. *SeisSol* uses topological weights for mesh partitioning to assign less work to slower nodes. The weights are computed by measuring the elapsed time $\hat{t}_p$ of *SeisSol*-proxy running on each compute-unit p. The proxy executes the macro-kernels defined by Eq. 4.1 with a pre-defined number of elements. The computational speed $\hat{s}_p$ can be estimated as the inverse of $\hat{t}_p$. Therefore, a topological weight $\hat{w^c}_p$ can be considered as a fraction of the total data processing speed of a distributed-memory computer and can be written as

$$\hat{w^c}_p = \frac{\hat{s}_p}{\sum_{i=0}^{P-1} \hat{s}_i} \tag{4.10}$$

where P is the total number of the involved compute units.

In each partition p, *SeisSol* splits each LTS time-cluster l into three layers, namely: 1) *interior* layer contains elements which are completely independent of communication; 2) *ghost* layer contains replicas of the face-neighboring elements residing in adjacent partitions m (i.e., $m \neq p$); and 3) *copy* layer which contains elements, residing in partition

p, which have face-neighbors located in neighboring partitions. Elements in *copy* and *ghost* layers are sorted with respect to the partition numbers with which they are supposed to communicate. These results in so-called communication regions, which form contiguous blocks of data to be exchanged between computational units.

An update of each layer can be treated as an LTS sub-task which needs to be intelligently scheduled. The scheduling is determined dynamically at run-time according to the LTS constraints discussed in Section 3.5. The processes exchange DOFs and their time derivatives over regions inside *copy* and *ghost* layers using asynchronous point-to-point communication. *SeisSol* prioritizes time-updates of *copy* layers. Firstly, *SeisSol* finds ready-to-schedule *copy* layers, updates them, and asynchronously sends the required data to the corresponding neighboring processes. Then, *SeisSol* processes ready-to-schedule *interior* layers. In between each sub-step, *SeisSol* checks for incoming messages, and if one arrives, *SeisSol* updates the corresponding *ghost* layer. *SeisSol*'s LTS sub-task scheduling is fully described in [13].

The MPI standard contains a sub-set of functions dedicated to non-blocking point-to-point communication between processes, e.g., MPI_Isend, MPI_Irecv, MPI_Test, MPI_Wait. The standard refers to "non-blocking" as a procedure that may return before a triggered operation completes, and before the user is allowed to reuse resources [86] e.g., message buffers. However, it does not enforce any particular implementation on the background processes, leaving it free for the library implementers. Some MPI libraries do not progress or complete a non-blocking data exchange, initiated between two processes, until the *test* or *wait* MPI functions are called by the user program. To circumvent this potential problem, *SeisSol* provides an option to spawn an auxiliary thread to repeatedly call MPI_Test for all ongoing point-to-point communications to progress the message exchange.

5. Graphical Processing Units

Traditionally, Central Processing Units (CPUs) were designed for a wide range of tasks, including: operating system, database, audio and video processing, interactive applications, real-time tasks, etc. Graphics Processing Units (GPUs) are specialized hardware, designed for massively parallel computing, and were originally invented as co-processors to accelerate graphics rendering and image processing. In the early days, offloading those tasks to GPUs considerably freed the CPU up and allowed it to focus on processing other workloads at the same time. Such approach was and still is quite important for accelerating Computer-Aided Design applications and computer games. However, the unique design and architecture of modern GPUs made them well-suited in other fields, such as scientific computing, machine learning, and data analysis.

Section 5.1 explains details and the main operational principles of modern GPU hardware components such as streaming multiprocessors, caches, global and shared memory, etc. In Section 5.2, I list the most popular GPU programming models and discuss similarities and differences between them. Section 5.3 briefly describes the mechanism involved in submitting GPU tasks to a device and its possible overheads.

5.1. Architectures

Many GPUs are discrete - i.e., separate extension cards installed in a motherboard as dedicated hardware components. Such GPUs have their own memory and thus separate address spaces with respect to the CPU (also known as the host). GPUs are usually connected to the host via high-speed data buses, for example, PCI Express, *Nvidia* NVLink, *AMD*'s Infinity Fabric, etc. The buses are used for transferring data before, after, or during data processing on GPUs (devices). The host is responsible for managing and controlling GPU activities and events. The host communicates with a device via a GPU driver. The driver is a part of an operating system, and provides a set of instructions and interfaces that enable the operating system to control a device. Traditionally, the interaction between a user program and the driver is non-blocking which allows the host and a device to work asynchronously.

The GPU design resembles a vector processor because it is highly optimized for performing the same operation on multiple data elements simultaneously. However, in contrast to the SIMD, GPUs use multiple threads to execute the same instruction on multiple data elements in parallel. The GPU hardware groups threads into so-called *warps* which are

getting executed together in lock-step. The warps are further grouped into Cooperative Thread Arrays (CTAs), known as *blocks* according to the *Nvidia* terminology.

Modern GPUs consist of many light-weight cores which are composed into Streaming Multiprocessors (SMs). A GPU core usually consists of arithmetic-logical and single-precision floating-point units. Some GPU models also have one or several double-precision floating-point and/or special function units per multiprocessor. Each SM contains its own L1 and instruction caches, a register file, a warp scheduler, a dispatch and multiple load/store units. Multiprocessors usually share the same L2 cache from which a memory read/write request is forwarded to the global GPU memory in case of an L2 cache miss.

GPU execution units are pipelined. To keep them busy and maximize throughput, a user program creates more threads than available GPU cores. This results in scheduling multiple blocks and thus warps to the same SM. The hardware divides warps into two groups, namely: 1) *ready-to-execute* warps and 2) pending ones. The pending warps are typically waiting for a completion of a long latency instruction - e.g., data-read. Once an active warp hits a long latency instruction, the GPU scheduler immediately saves the state of the running warp in memory, selects a ready-to-execute warp, loads its state from memory and kicks off its execution. The GPUs use registers for saving threads' states to make the context switching as fast as possible. Therefore, register files of modern GPUs are quite large containing up to 64 thousands 32-bit registers per multiprocessor. Having enough ready-to-execute warps in a queue allows the GPU hardware to partially or completely hide the long memory access latencies. However, the GPU hardware always puts restrictions on the maximum number of threads and blocks which can be simultaneously scheduled to a multiprocessor. Therefore, one of the programmer's optimization goals is to find such a block size that will maximize the number of ready-to-execute warps.

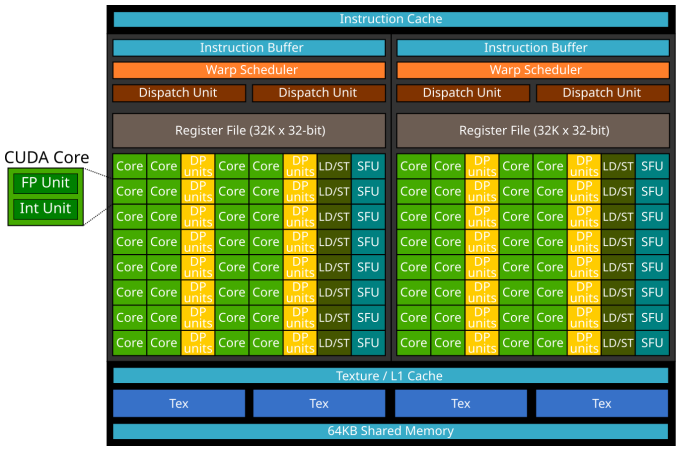


Figure 5.1.: *Nvidia* GP100 Streaming Multiprocessor. The image was drawn according to the description given in [91].

As an example, Fig. 5.1 shows the *Nvidia* GP100 Streaming Multiprocessor which consists of 2 compute-tiles. In total, the multiprocessor contains 64 CUDA cores, 32 double-precision units, 16 load/store units and 16 special function units. A CUDA core is

comprised of both single floating-point and arithmetic-logical units. *Nvidia* defines a warp as 32 consecutive threads. Therefore, all threads in a warp are going to be scheduled and executed either on the left or right compute-tile. This configuration of GP100 allows the multiprocessor to complete 64 (or 32, in the case of double-precision) instructions per cycle, assuming that the corresponding pipeline is completely full. *Nvidia* restricts the maximum block size to 1024 threads, the maximum number of simultaneously scheduled blocks per SM to 32, and the maximum simultaneously scheduled threads per SM to 2048 for the GP100 architecture.

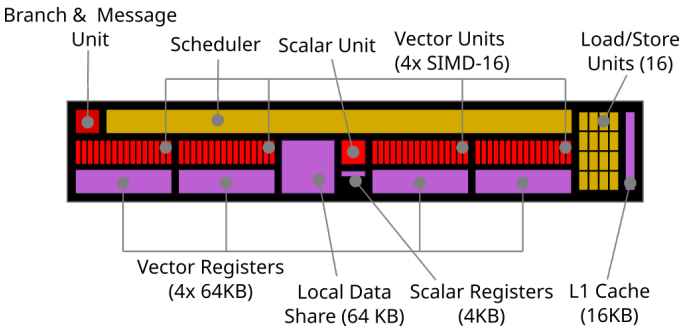


Figure 5.2.: *AMD* CDNA Compute Unit. The image was drawn according to the description given in [27].

Fig. 5.2 depicts the design of *AMD*'s CDNA Compute Unit (CU), which serves a similar purpose as *Nvidia*'s Streaming Multiprocessor. In contrast to *Nvidia*, *AMD* defines a warp as 64 consecutive threads, known as a wavefront. The hardware schedules each wavefront to one out of four 16-wide SIMD engines; thus, a warp instruction takes 4 (or 8) cycles for to complete. Therefore, a single CU can theoretically execute 64 (or 32) instructions per cycle. Another distinct feature of *AMD*'s CU is a scalar execution unit and its own register file. The unit is used for executing instructions which manipulate the data shared by all threads in the same wavefront - e.g., computing a base address for a load/store operation. *AMD* restricts the maximum block size to 1024 threads, the maximum number of simultaneously scheduled blocks per CU to 40, and the maximum simultaneously scheduled threads per CU to 2560 for the CDNA architecture.

Global memory of modern HPC GPUs is often made of several High Bandwidth Memory (HBM) chips and placed together with the multiprocessors and L2 cache on the same package which are connected through an interposer. A single HBM chip consists of several DRAM dies stacked on top of each other which are vertically interconnected. Each die has two channels where each one is connected to 16 memory banks. In total, a single HBM chip provides the 1024-bit wide data interface with the transfer rate up to 1 or 2 GT/s per pin (depending on the HBM generation). This high-density memory design results in low power consumption and provides significantly higher bandwidth in comparison to the traditional memory systems.

Each GPU thread distinguishes two spaces in global memory: 1) the one shared between all threads and 2) the local space, where the thread's private data is stored and cannot be accessed by other threads. A programmer is generally advised not to use thread's local memory because it entails long-latency memory access. However, sometimes it happens

implicitly; for example, the compiler can use the local space to store intermediate results which occupy too many registers - i.e., register spilling.

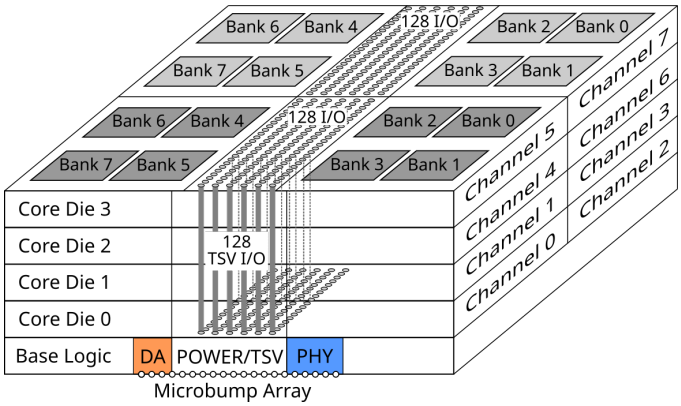


Figure 5.3.: High Bandwidth Memory architecture. The picture is taken from [73].

Memory access requests issued by a warp of threads are sent to the load/store units, from where it goes to the L1 cache. The access is called *coalesced* if all threads within a warp access data which fall within a single L1 data cache block. In this case, only a single memory access request will be generated and sent to L2 cache in the absence of the corresponding cache block. Modern GPUs also provide a scratchpad (also known as *shared*) memory, which is usually implemented as a part of L1 cache and is used by a CTA for data sharing or/and threads communication. It is a programmer's responsibility to explicitly load/store data and synchronize threads while working with shared memory. This gives a programmer an additional degree of freedom which can be used for further tuning and optimizations of GPU programs. This is important because accessing data from closer on-chip memory usually results in lower latency and, thus, higher performance. It is worth pointing out that L1 and L2 GPU caches are set-associative, whereas shared memory behaves as a direct-mapped cache [1].

L1 caches and, thus, shared memory are usually built from 128-byte blocks which are subdivided into four 32-byte sectors [1]. This sector size matches the DRAM atom size, i.e., the minimum data size, which can be served in a single memory access. Each block consists of 32-bit entries spread across 32 banks, which allows a warp of threads to read/write data in 1 (or 2) cycle in the case of a pipelined and coalesced memory access. A bank conflict happens when several warp threads access the same bank using different memory addresses. In this case, the access is serialized; thus, the request is served in multiple cycles. However, simultaneous data reading from the same address results in a broadcast - i.e., no serialization happens.

The GPU L1 caches are not coherent. Instead, they are usually implemented using the write-through cache policy. This results in a scalable hardware design and also reduces the hardware cost. This restricts threads from different blocks to directly exchange data between each other because the hardware logic may schedule the blocks to different SMs. The atomic operations through global memory can be used for synchronizing threads running in different blocks. In this case, the L2 cache, which is unified between all SMs,

can help to mitigate high costs of such operations because it usually implemented using the write-back cache policy.

To ease the GPU programming regarding copying data between the host and device, modern GPUs provide a unified virtual address space, also known as unified (managed) memory. The operating system and the hardware work together to ensure data coherency between two disjointed memory spaces. The implementation of this concept is based on the page-fault mechanism. When a fault happens during data access on one side, the operating system detects and invalidates the corresponding page on the other side and automatically copies data between the CPU and GPU. As one can expect, performance of the automatic data migration is lower in contrast to the explicit one (e.g., see [67]). A programmer can utilize memory pre-fetching or provide memory location hints to the driver in order to improve performance of the automatic data migration. However, there are scenarios where severe performance degradations are not avoidable; allocating more memory than available on a device, or accessing data from the same page from the CPU and GPU at the same time will result in page thrashing. Nevertheless, the use of the unified virtual address space is well-suited as the first step toward porting an existing software application to heterogeneous systems.

5.2. Programming models

The host and device belong to different Instruction Set Architectures (ISAs). Thus, the corresponding parts of the source code need to be identified, separated, compiled using either different compilers or compiler backends, and finally linked together. A computer program can be viewed as a set of global variables and functions at a low level regardless of the syntactic sugar of various high-level programming languages. The device functions are commonly called *kernels*. Some GPU programming models - e.g., CUDA, HIP - require a programmer to manually mark all device kernels using some sort of attributes (also known as decorators). The others - e.g., SYCL - hide this process using C++ classes and templates, making the source code portable for a wide range of accelerators.

GPU architectures, even from the same vendor, can vary significantly from generation to generation and thus their ISAs may not be forward- or backward-compatible. The problem is solved by introducing a new level of indirection via some stable intermediate code representations - e.g., *Nvidia*'s PTX, *AMD*'s IL, SPIR-V, etc. The intermediate code then needs to be processed by, usually a closed-source, optimizing assembler which emits the machine code for a target architecture. This step is usually delegated to a corresponding GPU driver and can be done during a program execution or ahead-of-time. The intermediate code can be generated by vendor-specific (e.g., *nvcc*, *hipcc*, *DPC++*) or open-source compilers such as *clang*, *gcc*, etc.

CUDA is a programming model developed by *Nvidia* for programming *Nvidia* GPUs. It stands for both a language, a runtime library, and an API, which aim to provide maximum control over the hardware. CUDA is a proprietary technology; thus, it is not bound to any

standard. Therefore, it allows CUDA developers to quickly extend the model as needed to help programmers to achieve high performance on *Nvidia* GPUs. The model also provide a set of high-performance libraries with the industry adopted interfaces, for example, *cuBLAS*, *cuSPARSE*, *cuDNN*, *cuTENSOR*, etc. User-friendly profiling tools, supplied as a part of the CUDA toolkit, make it easy for programmers to identify and optimize hot spots in their programs. The dominance of *Nvidia* hardware on the GPU market and access to general-purpose GPU programming, even for consumer graphics cards, made CUDA popular among programmers. Today, it is common to see the use of tuned CUDA implementations as the baselines when comparing various algorithms implemented with different parallel programming models. Nevertheless, CUDA lacks portability to GPUs from other vendors, - e.g., *AMD*, *Intel* - which is its major issue.

In 2016, *AMD* released a C++ runtime API and a language extension called HIP to address the portability issue between *Nvidia* and *AMD* GPUs. The API and the language closely resemble CUDA regarding its syntax and functionality. This makes it easy for experienced CUDA developers to switch to *AMD*'s ROCm software stack. HIP can be viewed as a software bridge between *Nvidia* and *AMD* GPUs, or as an attempt to establish a new GPU-centric programming standard based on the CUDA programming model.

OpenCL was one of the first parallel programming models, developed by the Kronos Group and expressed as an open standard, which tried to address the portability issue in a broad sense. The standard is based on a vendor-neutral approach to write high-performance code that can run on a wide range of accelerators, including CPUs, GPUs and FPGAs. This is achieved by generalizing and abstracting various computational devices and their memory systems. While OpenCL can provide high performance and portability for certain types of computations, it may not always be the most efficient for a given hardware configuration or an application. Moreover, because of a very high level of abstraction and low-level C interface, the OpenCL API is verbose, error-prone, and thus inconvenient for many programmers. To address this issue, the Kronos Group released the SYCL standard, i.e., a royalty-free, cross-platform abstraction layer that builds on the underlying concepts, portability and efficiency of OpenCL [63]. SYCL is based on the single-source approach and built on top C++17, which considerably improves programming productivity. The latest version of the standard (2020) introduced new features, making the model even more powerful and flexible for programmers, e.g., a unified shared memory concept, reduction algorithms, improved vectorization support, improved device selection mechanism. Some hardware design companies - e.g., *Intel* - decided to establish SYCL as the main high-level programming platform for their hardware.

Apart from OpenCL and SYCL, there were other attempts to address challenges related to developing portable applications for heterogeneous computing systems, e.g., Kokkos and RAJA developed at the Sandia National Laboratories and Lawrence Livermore National Laboratory, respectively. Kokkos and RAJA are C++ libraries that provide abstractions for expressing parallelism. While Kokkos abstracts a wide range of parallel computing scenarios, including shared-memory parallelism, distributed-memory parallelism, and GPU computing, Raja is focused on abstracting the shared-memory parallelism on CPUs and GPUs.

Another approach to provide portability for various heterogeneous computing systems is through language extensions, e.g., OpenMP and OpenACC. Both OpenMP and OpenACC are standards, which are based on *pragma* directives. Directives can be viewed as special instructions that pass auxiliary information to the compiler about how to process or interpret a follow-up statement. In most cases, programmers target *for*-loops for offloading, which usually contain regular parallelism and can be nested. During static analysis, a compiler decides how to map loop iterations to the execution units of the underlying hardware and whether or not to copy data from the host to a device, and vice versa, before and after loop execution. A programmer can influence the compiler's decision-making by providing additional, more specific directives. In contrast to OpenMP, OpenACC is considered to be a descriptive model [26] - i.e., the model describes how computations should be parallelized without enforcing how it should be done.

To summarize, there exist many ways to program GPUs. Some are more abstract and thus more portable. Certain algorithms expressed with these models may not perform well on some hardware configurations due to the absence of access to some hardware-specific low-level features. Other vendor-specific models may result in high performance but entail maintaining several versions of the same code - i.e., one per vendor - which puts extra pressure on developers. By and large, it is difficult to say, in advance, which model is well suited for a particular software application. Programmers are encouraged to try as many models as possible and select the best one based on their software requirements.

5.3. Kernel Launching Mechanism

The term "kernel launching overheads" appears too often in this thesis, and thus, it must be clarified. The term can be explained by examining the kernel launching mechanism. In this work, the Heterogeneous System Architecture industry standard is used for this purpose since it is publicly available and well-documented.

Like OpenCL, the HSA standard establishes the queue concept - i.e., the software interface between the host and a device. In HSA, a queue is defined as runtime-allocated, user-level accessible virtual memory of a certain size containing packets defined in the Architected Queuing Language (AQL) [121]. A packet can belong to one of several AQL types - e.g., kernel dispatch packet, agent dispatch packet, or vendor-specific packet. All packet types have a fixed and pre-defined binary format [38]. A user application can allocate a queue object using an HSA-compatible runtime system. The system uses the kernel-mode driver to initialize and register a queue with a packet processor. The processor is a part of a device, and it is responsible for detecting and scheduling kernels. It reads packets from a set of queues attached to the device, checks dependencies, and dispatches GPU tasks on available compute units [121]. The queue's virtual memory is a shared resource between the host system and the packet processor.

A queue object maintains a so-called write index, which is opaque to a user. When a user application wants to create a new packet to submit a new computational or service task

to a device, it requests the HSA runtime to provide the index. Internally, the runtime atomically increments the index by the AQL packet size and returns the previous value of the index back, to where the user application can place a new packet. The application adjusts the given space according to the task, appropriately filling all necessary packet fields. The fields include information about launch dimensions, address to the instruction code, address to kernel arguments, completion detection type, etc. According to the standard, memory for the kernel arguments must be allocated through the HSA runtime, which returns a pointer to a special memory type region. This memory type is read-only from the device's perspective. The user application copies the kernel arguments to the allocated memory space and assigns the pointer to the corresponding packet field. Once it is done, the application notifies the packet processor by signaling the queue doorbell through the runtime system. Once the signal is received, the packet processor starts dispatching the newly arrived packet to the computer units of the associated device. In contrast to other legacy systems (e.g., OpenCL), this design minimizes switching between user-mode drivers, kernel-mode drivers, and the operating system [121].

The packet processor views the queue's memory as a ring buffer, which has separate memory locations defining the write and read state information [121]. The main goal of the processor is to efficiently manage the queues on behalf of the device [121]. When a task, described by the associated AQL packet, is completed on a device, the processor signals back to the host, notifying it about the task completion.

The HSA standard only describes the main operational logic of the packet processor, leaving implementation details to hardware vendors. For example, the AMD CDNA2 architecture primarily distributes the logic between several hardware components: 1) the command processor and 2) asynchronous compute-engines. The command processor receives AQL packets and transforms them into computational tasks [29]. The four compute-engines consume the tasks by dispatching wavefronts to the compute units and managing the submitted tasks.

The hardware components are usually designed to hide the latency of the kernel launching mechanism. For example, this can be achieved by pipelining or internal buffering, implemented on a device at the hardware level. When a series of coarse-grained tasks are submitted in a row to an idle device, the user application will experience a delay approximately equal to the time of the first kernel launch. When some compute units complete processing their sub-tasks, the hardware (e.g., asynchronous compute-engines) is ready to make them busy again by assigning new ones from the same or a next task. However, when the workloads of submitted tasks become too small, it becomes challenging to completely hide the overheads. For example, the speed of the packet processor can limit the device's throughput. In this case, the compute units become idle and simply wait for a new task to get dispatched, resulting in low device utilization.

6. Implementation of Elastic Wave Propagation

SeisSol's wave propagation solver is based on the ADER-DG method, which generates most of the computational workload during an earthquake simulation. Thus, the computational efficiency of the method greatly impacts the entire application's performance. In contrast to many other works dedicated to accelerating the DG method with GPUs, *SeisSol*'s implementation of the method is tightly bundled with code generation, which is inherited from its highly optimized CPU design. In the beginning, the code generation approach significantly complicates and slows the GPU development process. But, in the end, it opens doors for many optimization opportunities. This chapter contains and explains the main contributions of this thesis.

The first and foremost step in optimizing applications for heterogeneous computing systems is to peel off long-latency GPU service tasks from the main computational loop (e.g., allocations and deallocations of the device memory). Ideally, all required data must be allocated and initialized on a device before entering the loop. In Section 6.1, I explain how it was achieved.

In Section 6.2, I discuss possible ways to perform task decomposition for the ADER-DG method. In *SeisSol*, the decomposition is mainly constrained by the adopted code generator. The outcome of this discussion highly reassembles the approaches discussed in other works. At the end of the section, I return to the data management and explain the code changes made in *SeisSol* to support new task decomposition.

In Section 6.3, I focus on generating high-performance GPU kernels, which result from discretizing the elastic wave propagation system of PDEs using the ADER-DG method. The discussion starts with a more straightforward approach, which was historically easier to implement - i.e., binary GEMM GPU kernels. Then, I describe a more advanced approach, which is considered one of the major contributions of this work - i.e., fused GEMM kernels. I discuss the advantages and disadvantages of both methods, compare them, and reveal challenges faced during this study. In this work, the efficiency of the generated kernels is shown on various HPC GPUs using the roofline model analysis. In the end, I revisit the flux matrix decomposition macro-kernels (i.e., Eq. 4.3 and Eq. 4.4), show the inefficiencies of their high-level implementations inherited from the original CPU design, and propose new variants that perform better on GPUs.

In Section 6.4, I investigate the concurrent execution of GPU tasks using multiple device streams and discuss some important implementation details. The section also shows how

the proposed changes affect the performance of the proxy application under low, medium, and high workloads.

In Section 6.5, I investigate *SeisSol*'s performance on distributed multi-GPU systems. The experiments in this part of the study were performed using different variants of the Layer Over Half-space (LOH.1) test scenario because this setup mainly involves only the wave propagation solver. I start the section listing majorly known GPU programming models regarding distributed multi-GPU systems and select the best-fitting one for *SeisSol*. Then, I describe the impact of the MPI buffer placements (i.e., on the device, host, and unified memory) on the strong scaling performance of the application. After that, I investigate the graph-based execution model (i.e., CUDA Graphs) and its impact on performance. Then, I demonstrate the influence of the LTS clustering on the strong scaling performance of *SeisSol* and present a simple mathematical model which highlights the main performance limiter. After that, I show how the LTS-specific mesh partitioning algorithm must be extended to accommodate *SeisSol*'s execution on distributed-memory environments subjected to limited memory resources per process. In the end of this section, I present the weak scaling study of *SeisSol* conducted on the LUMI supercomputer.

In Section 6.6, I focus on the portability aspects of *SeisSol*'s wave propagation solver. The code verification and convergence analysis are shown in Section 6.7. In Section 6.8, I summarize the main outcomes of this part of the study and conclude the chapter.

Due to a large configuration space of *SeisSol*, the results presented in this chapter were mainly obtained with the single-precision floating-point format, convergence order equal to 6, and LTS ratio equal to 2. This is done for convenience for comparing experiments shown in different chapters of the thesis. The results presented in this and the next chapters were obtained on the LUMI, Leonardo, and Selene supercomputers. At the moment of writing, LUMI is the third fastest supercomputer in the world, Leonardo is fourth, and Selene is ninth, according to the most recently published TOP500[1] list. The reader can find the hardware architecture details of each supercomputer mentioned above in Appendix A.1.

6.1. Memory Management

The GPU implementation of *SeisSol* is built on the assumption that all data required for computations and management fit into the device's global memory. This assumption is based on the experience gained from the previous work on optimizing *SeisSol* for *Intel*'s Xeon Phi co-processor [50], where 8 GB per device had already proven to be enough to hold the data required for a typical production earthquake scenario. Today, modern GPUs, for example, *Nvidia* A100 or *AMD* MI250x, are equipped with 40-128 GB of HBM2 onboard memory. Therefore, in this work, I allocate and initialize the necessary data on a device during the last initialization steps of *SeisSol*. Further data transfers in *SeisSol* are then mainly related to writing final and/or intermediate results to disks.

[1] https://www.top500.org/lists/top500/2023/06/

In this work, I use *unified memory* for DOFs and their time derivatives. This memory type was convenient at the beginning of the research - i.e., during the first attempts of porting *SeisSol* to GPUs. It helped to gradually port region by region to GPUs while leaving error-free computations on CPUs. This approach increased the productivity of the GPU code development at its early stage because no explicit memory copy operations were required. Today, it is still convenient to use unified memory for DOFs in some cases in *SeisSol* - e.g., to perform some non-trivial I/O operations, such as writing data captured by point receivers arbitrarily scattered within a computational domain. However, the use of this type of memory entails some overheads, which, as shown in Section 6.5.1, can degrade the performance of distributed multi-GPU systems. Therefore, in the future, it is worth considering switching to regular device memory for storing DOFs and their time derivatives; however, it may entail considerable code refactoring.

Like many software applications, *SeisSol* requires fast-access memory for storing temporary data - i.e., intermediate results. These data are usually a product of computations within or between macro-kernels. Traditionally, the stack portion of the program is used for these purposes. This approach results in programming flexibility and zero overheads with respect to memory allocation and deletion. Many heterogeneous systems do not support stack allocation in the context of a program execution. Therefore, memory for storing temporary data must be allocated dynamically on a device.

A naïve implementation of the aforementioned problem would result in frequent communications with the device driver and, thus, long latency between computations. This could be avoided by allocating all necessary memory only once - i.e., before entering the main computational loop. However, this approach would require a comprehensive program's static analysis to calculate the minimum amount of memory needed for handling intermediate results. Moreover, the analysis should also account for memory reuse inside the generated code. Therefore, this pre-processing step should be a part of the compilation process to provide programming flexibility.

In this work, this problem is solved differently; 1 GB of the device memory is allocated and reserved at the beginning of *SeisSol*'s execution by default. The resource management is delegated to a global data structure, which implements the LIFO policy - i.e., the stack. This helps to preserve the *new/delete* semantics and, thus, programming flexibility while avoiding long latency overheads because of simple pointer arithmetics. The 1GB limit is usually enough for many production scenarios. If necessary, it can be changed via an environment variable.

6.2. Task Decomposition for Massively Parallel Systems

The sub-task definition in the context of elastic wave propagation is determined in Section 4.3 - i.e., an execution of one or more tensor expressions defined by the macro-kernels for a single mesh element. As discussed in Section 4.2, the *YATeTo* DSL is used to generate the source code for each tensor expression. Sub-tasks can have different execution control flow

paths relative to each other - e.g., due to the presence or absence of boundary conditions or differences in numerical flux solvers between elements - making the task decomposition irregular (see Algorithm 1).

The size of the largest matrix multiplication determines the maximal parallelism inside a sub-task, which, according to Table 4.1, is too small for modern GPUs, which are built from several thousands of lightweight cores. Therefore, it is better to map each sub-task to a single SM to efficiently exploit data parallelism inside and between each SM. This task decomposition is similar to the ones used in other works focused on implementing and optimizing the DG method for GPUs - e.g., [66, 43, 104, 89, 39, 127, 41, 88, 16, 87, 122, 58, 7, 65, 126, 15]. Some authors (e.g., [41, 2, 58]) even proposed to combine data processing of several mesh elements into one sub-task to better balance the GPU occupancy and data movement.

Changes in control flows are managed by *SeisSol* because, as expected, *YATeTo*'s software design is mesh agnostic. Therefore, a direct map of *SeisSol*'s mixed sub-task execution model from CPUs to GPUs should contain both the generated code and the application logic. This approach is complicated because 1) it requires copying many non-trivial data structures to GPUs to implement branching, and 2) it becomes difficult to estimate computational resources (e.g., the number of threads per block, the shared memory size, etc.) required for optimal executions of thread blocks and, thus, optimal warp scheduling because an optimal block configuration for one sub-task can be sub-optimal for another.

Considering the problems listed above, a GPU task should be defined at least as an execution of a single tensor expression on multiple mesh elements. This completely solves the first problem because the code branching is delegated to the host application. Moreover, in this case, the task decomposition becomes regular and, thus, manageable regarding finding an optimal thread block configuration (see Algorithm 2 as an example). As a disadvantage, this approach splits each original CPU task into multiple ones. However, new GPU tasks are data-independent and, thus, can be executed concurrently on multiple GPU streams (see Section 6.4). Therefore, this task decomposition is chosen as the basis for this work.

It is necessary to group the required data into batches to supply them to each GPU task. The grouping is based on the control flow path to which a task belongs. This process can be done once per simulation because *SeisSol* is based on the static adaptive mesh refinement. Therefore, I introduce a new pre-processing step that records batches of all tensor expression operands along each possible execution path inside each LTS time-cluster layer during a dry-run of *SeisSol*. The batches are stored in 2-level hash tables and managed by the corresponding time-cluster layers. As shown in Algorithm 2, a key of the outer table encodes a branch condition inside a macro-kernel, and points to an entry that stores batches of all operands required for the execution path. A C-structure implements a key of the outer table. The structure contains four binary encoded fields: a macro-kernel name, a kernel type, a local face number (see Fig. 3.4), and the relation number of a neighboring element face. The first field is compulsory because it uniquely identifies a macro-kernel. The rest can be omitted if it is not needed for a particular case. Usually, the last two entries of the outer table key are used to record batches for macro-kernels

involved in the computations of fluxes or boundary conditions. Binary encoding enables the logical *AND* operation and, thus, helps to embed conditional statements inside the keys of the outer table. Keys of an inner table are integer encoded names of operands, which substitute their symbolic counterparts.

Algorithm 2 GPU implementation of the Neighbor Surface Integral - i.e., I^{nghb}_{surf}

procedure ComputeNSI(*Device*, *LtsLayer*, *PreComputedData*)
  $OuterTable \leftarrow LtsLayer.getBatchTable()$
  $M \leftarrow LtsLayer.getClusterSize()$
  for j **from** 1 **to** 4 **do**
    for f **from** 1 **to** 48 **do**
      $condition = Condition(FaceKind :: Regular, FaceKind :: Periodic)$
      $key \leftarrow Key(KenelName :: NeighborFlux, condition, j, f)$
      $InnerTable \leftarrow OuterTable[key]$
      if not $InnerTable.empty()$ **then** ▷ defines a GPU task
        $h \leftarrow f \mod 3$
        $i \leftarrow f \mod 12$
        $\tilde{R}^{i}_{lm}, f^{+,h}_{nm}, R^{j}_{mt} \leftarrow PreComputedData.getFluxMatrices(i, h, j)$
        $Q^{1:M}_{lp} \leftarrow InnerTable[InnerKey :: DOFs]$
        $\mathcal{T}^{1:M}_{tq} \leftarrow InnerTable[InnerKey :: IntegratedDOFs]$
        $\mathcal{A}^{+,1:M}_{qp} \leftarrow InnerTable[InnerKey :: FluxSolver]$
        $Q^{1:M}_{lp} = Q^{1:M}_{lp} - \tilde{R}^{i}_{lm} \cdot f^{+,h}_{nm} \cdot R^{j}_{mt} \cdot \mathcal{T}^{1:M}_{tq} \cdot \mathcal{A}^{+,1:M}_{qp}$ ▷ Generated Code
      end if
    end for
  end for
  $Device.wait()$
end procedure

6.3. Code Generation

As mentioned in Section 4.2, the source code for each tensor expression is generated using the *YATeTo* DSL in *SeisSol*. To preserve the overall software design (and, thus, its advantages) and to adapt *SeisSol* for GPU computing environments, the language needs to be extended to generate GPU code for the whole task - i.e., to operate on multiple data elements at once. To achieve this, I expand the DSL capabilities to generate tensor expressions for batched data. In my implementation, a batch can be given 1) as an array of pointers, 2) as a base pointer to stridden data, or 3) as a pointer to a single data element, for example, a pointer to a mass or stiffness matrix. A stride between adjacent data elements can be deduced from a tensor description given as an input to *YATeTo*.

A tensor expression consists of a sequence of binary tensor operations. Including an entire sequence into a single GPU kernel is desirable to obtain a high arithmetic intensity of the

generated code. However, it was challenging to achieve that, in practice, during the first attempt to port *SeisSol* to GPUs because 1) the required software changes were significant for the CPU-centric DSL, 2) the whole software requirements for a new version of *YATeTo* were not entirely determined, and 3) the outcome - i.e., application's GPU performance - was still to be discovered. In Section 6.2, I mentioned that finding an optimal thread block configuration was more manageable for a batched tensor expression because it resulted in a regular task decomposition. However, it may involve a complex analysis during code generation to account for the reuse of resources between operations - e.g., registers and shared memory. Moreover, long GPU kernels tend to result in register spilling, which can only be determined after compilation. To avoid spilling in favor of performance, a kernel may require splitting into multiple parts. Finding optimal split locations may lead to a large configuration space for an optimization problem that needs to be combined with auto-benchmarking.

The first step of this work is based on generating binary batched operations (see Section 6.3.1), which helps to avoid the problems mentioned above. In Section 6.3.2, I extend this approach by generating fused batched operations to obtain higher GPU performance. Because this work is constrained to elastic wave propagation, which operates on low-rank tensors (i.e., less or equal to 2), I restrict the code generation to only GEMMs and linear combinations of matrices. In *YATeTo*, I implemented the GPU code generation through backends - i.e., *GemmForge*[2] and *ChainForge*[3] - which are the key contributions of this work. This approach follows the single responsibility principle and results in reusing the contributions of Uphoff and Bader from [116] - e.g., EQSPP, operations ordering, etc.

6.3.1. GemmForge

In this work, a batched GEMM operation is defined as

$$C_e = \alpha \cdot Op(A_d) \cdot Op(B_f) + \beta \cdot C_e \tag{6.1}$$

where e is a batch index; d and f are batch indices equal to either 1 or e; $C_e \in \mathbb{R}^{m \times n}$; $A_d \in \mathbb{R}^{m \times k}$; $B_f \in \mathbb{R}^{k \times n}$; $Op(X)$ can be either X or X^T; α and β are scalar values. Indices d and f cannot be equal to 1 at the same time if $e \neq 1$ - i.e., at least one of them must be equal to e in this case. Following the discussion given in Section 4.1, the column-major matrix layout is only considered in this work.

Traditionally, hardware vendors supply their products with the software developing toolkits containing optimized and highly tuned HPC libraries, which include many commonly used scientific computing algorithms. For example, CUTLASS, MAGMA, and cuBLAS are extended implementations of the BLAS library designed by *Nvidia* to provide the full advantage of their GPUs. These and similar libraries (e.g., rocBLAS) support batched computations and usually deliver performance close to the peak. However, as shown in [83] and [35], some custom implementations can result in higher performance if some

[2] https://github.com/SeisSol/gemmforge
[3] https://github.com/SeisSol/chainforge

Listing 6.1: Example of operands descriptions in *GemmForge*.

```
mat_a = DenseMatrix(num_rows=64,
                    num_cols=56,
                    addressing="none",
                    bbox=[0, 0, 56, 56])

mat_b = DenseMatrix(num_rows=64,
                    num_cols=9,
                    addressing="strided",
                    bbox=[0, 0, 56, 9])

mat_c = DenseMatrix(num_rows=64,
                    num_cols=9,
                    bbox=[0, 0, 56, 9],
                    addressing="strided")

vm = vm_factory(backend="cuda",
                arch="sm_70",
                fp_type='float')

gen = GemmGenerator(vm)
gen.set(False, False, mat_a, mat_b, mat_c, alpha=1.1, beta=1.1)
gen.generate()
```

specific information - e.g., matrix structures or sizes - is known at compile time. To test whether that was applicable to the ADER-DG method in the context of *SeisSol*, I developed *GemmForge* - i.e., a *Python* library callable from *YATeTo* during the code generation phase.

GemmForge's input consists of operands descriptions required for the code generation. As shown in Listing 6.1, each matrix is described using 1) the number of rows and columns, 2) the formal size (see Section 4.2) specified as a bounding box, and 3) the batch addressing. The batch addressing specifies how a batch of data will be passed to a generated kernel at run-time: as an array of pointers, as a base pointer to stridden matrices, or as a pointer to a single matrix. This results in a more flexible and convenient interface to the batched GEMM kernels in contrast to libraries like cuBLAS or rocBLAS, where addressing of all operands must belong to the same type. *GemmForge* also generates kernel launchers that perform task decomposition - i.e., computing optimal thread block configurations based on the estimated shared memory consumption and number of registers per thread.

Inspired by [17], the main GEMM logic of *GemmForge* is based on performing a single matrix multiplication as a sum of outer products. Two or three consecutive warps, called a *team*, are required per multiplication for matrices typical for the ADER-DG method (see Table 4.1). The team size, m_a, equals the number of rows of the first operand aligned to the warp length. Any load or store operation of columns of A and C by the first m threads of a team, called active threads, leads to coalesced memory access because of the adopted column-major matrix layout. Each active thread allocates an array of registers of size n, initialized with zeros, to hold the intermediate results of a matrix product. If $Op(A_d)$

equals A_d^T, matrix A is loaded into shared memory using the transposition on-the-fly. In this case, the matrix is padded with zeros to avoid shared memory bank conflicts.

During each iteration of the algorithm, an active thread loads a corresponding element of a matrix A column from global device memory to a free register and computes n partial updates using a row of matrix B. The matrix product is complete after n steps but distributed row-wise between register arrays of active threads. Thus, n additional steps are required to move the data from the register file to the destination, during which scaling by α and β is performed. The algorithm is illustrated in Fig. 6.1.

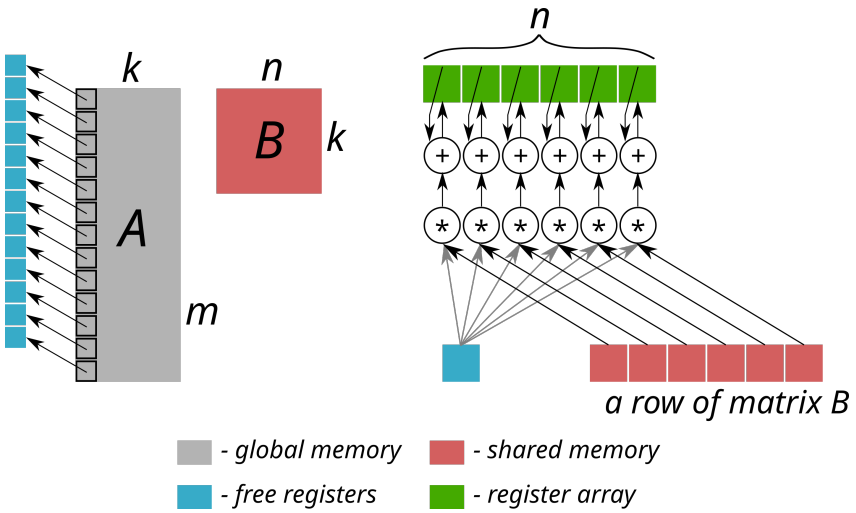


Figure 6.1.: Matrix multiplication scheme in *GemmForge* - i.e., GEMM as a sum of parallel outer products. Left: Coalesced memory read-access of matrix A. Right: Work of a single active GPU thread per iteration.

In this algorithm, all active threads must simultaneously load the same elements of a row of matrix B during each iteration. In [30], I propose to pre-load matrix B to shared memory at the beginning of the kernel execution to avoid un-coalesced memory access from global memory. In this case, no memory access serialization occurs because all threads read data from the same memory address and, thus, from the same shared memory bank. I will refer to this approach as *Type-1*.

GemmForge can generate a block of threads for several teams - i.e., for multiple matrix operations per block - which may increase the number of ready-to-schedule warps per SM. Therefore, a block can be two-dimensional if there are enough run-time resources. Given matrix descriptions, the required amount of shared memory can be evaluated precisely, whereas the exact number of registers per thread can be known only after compilation. In *GemmForge*, I estimate the latter using a heuristic - i.e., a sum of the register array length required for a matrix multiplication and an empirically derived constant specific for each supported GPU model. The shared memory size and the register count are combined with the hardware limits (see Section 5.1) to obtain an optimal number of matrix multiplications per block.

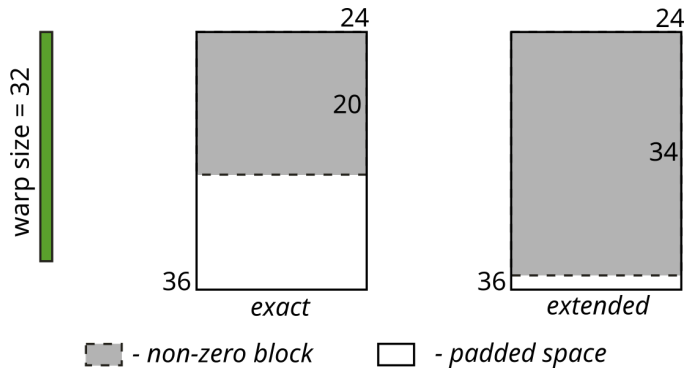


Figure 6.2.: Example of the shared memory loading strategies implemented in *GemmForge*. On the left, the *exact* strategy is chosen because it results in 24 warp-load operations (versus 27) and stores 480 matrix elements in shared memory (versus 864). On the right, *GemmForge* selects the *extended* strategy because it results in only 27 warp-load operations (versus 48) despite consuming extra shared memory space for 48 padded matrix elements.

GemmForge implements two strategies for loading matrices to shared memory, called *exact* and *extended*. While selecting a strategy, the main objective is to generate code with the least coalesced load operations using the maximum number of consecutive threads, which aims at minimizing latency. A particular choice depends on a formal matrix size and its padding. The *exact* strategy generates the code that loads only the bounding box of a matrix in a column-by-column manner. The *extended* strategy loads a matrix as it is - i.e., including padded rows. The selection logic is shown in Fig. 6.2.

Listing 6.2 shows the code generated according to the description given in Listing 6.1. In line 6, all thread teams compute the batch indices on which they will operate. In lines 11-13, each thread team selects the associated A, B, and C matrices using its batched index and the addressing type. *GemmForge* allows a user to optionally specify an extra matrix offset. This can be convenient in some scenarios when only a few continuous matrix columns are required for an operation (e.g., only velocity components of Q_{lp}) because an existing batch can be reused. The generated kernel also provides an option to specify a batch mask, which can be computed in advance based on some conditions known at run-time. This results in enabling/disabling specific batch indices - i.e., lines 8-10. The register and shared memory allocations are shown in lines 14 and 15. Lines 18-28 show the process of loading matrix B to shared memory. In this case, the entire matrix, including the padded rows, is loaded to shared memory in 9 interactions using 64 threads. The matrix multiplication is performed between lines 32 and 43, which follows the store operation shown between lines 46 and 54. The thread masks are used in both cases to enable the first 56 threads of a team - i.e., active threads.

As can be seen from Listing 6.2, all loops inside the kernel are completely unrolled. This results in a high number of machine instructions and may create pressure on the instruction cache. However, at the same time, this approach creates many opportunities for better

Listing 6.2: Generated batched GEMM kernel (56x56x9) for *Nvidia* sm70 model according to the description given in Listing 6.1.

```
__global__ void __launch_bounds__(64)
kernel_sgemm(const float* A, int A_extraOffset,
             const float* B, int B_extraOffset,
             float* C, int C_extraOffset,
             unsigned numElements, unsigned* flags) {
  unsigned batchID = (threadIdx.y + blockDim.y * blockIdx.x);
  if (batchID < numElements) {
    bool isFlagsProvided = (flags != nullptr);
    bool allowed = isFlagsProvided ? static_cast<bool>(flags[batchID]) : true;
    if (allowed) {
      const float * const __restrict__ glb_A = &A[A_extraOffset];
      const float * const __restrict__ glb_B = &B[batchID * 576 + B_extraOffset];
      float * const __restrict__ glb_C = &C[batchID * 576 + C_extraOffset];
      float reg0[9] = {0.0f, 0.0f, 0.0f, 0.0f, 0.0f, 0.0f, 0.0f, 0.0f, 0.0f};
      __shared__ __align__(8) float totalShrMem[568];
      float * localShrMem0 = &totalShrMem[568 * threadIdx.y];
      float* shrRegion0 = &localShrMem0[0];

      {
        // load B to shared memory
        #pragma unroll
        for (int i = 0; i < 8; ++i) {
          shrRegion0[threadIdx.x + i * 64] = glb_B[threadIdx.x + i * 64];
        }
        if (threadIdx.x < 56) {
          shrRegion0[threadIdx.x + 512] = glb_B[threadIdx.x + 512];
        }
      }
      __syncthreads();

      // perform GEMM
      if (threadIdx.x < 56) {
        float value;

                #pragma unroll
        for (int k = 0; k < 56; ++k) {
          value = glb_A[threadIdx.x + k * 64];

          #pragma unroll
          for (int n = 0; n < 9; ++n) {
            reg0[n] += value * shrRegion0[k + 64 * n];
          }
        }
      }

      // store results
      if (threadIdx.x < 56) {
        #pragma unroll
        for (int n = 0; n < 9; ++n) {
          glb_C[threadIdx.x + 64 * n] = 1.1f * reg0[n]
                                      + 1.1f * glb_C[threadIdx.x + 64 * n];
        }
      }
    }
  }
}
```

instruction scheduling inside the kernel, which *GemmForge* delegates to compiler backends of *nvcc*, *amdclang*, etc.

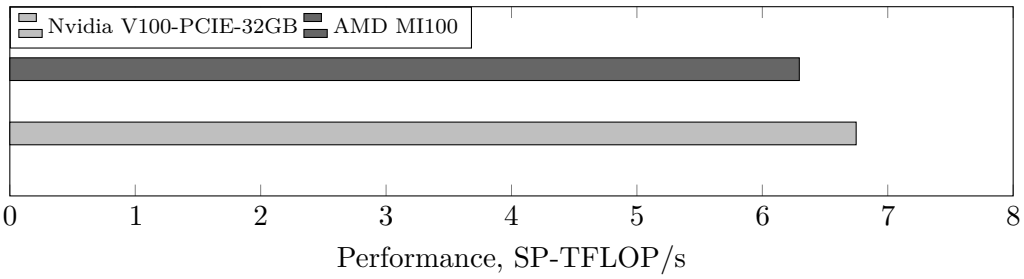


Figure 6.3.: Performance of the GEMM kernel generated according to Listing 6.1 on *Nvidia* and *AMD* GPUs using CUDA-11.5 and ROCm-5.4, respectively.

In [30], the roofline model analysis was used to show that the kernels generated by *GemmForge* resulted in close to the maximum performance on *Nvidia* V100-SXM2 GPU (approximately 94%), but were memory-bound and thus were limited by the global memory bandwidth. It was determined that an operand - i.e., matrix A or B - would reside in L2 cache if it was the same for the entire batched operation. In this case, the arithmetic intensity of a kernel substantially increases. The work also compared the performance of generated kernels against their cuBLAS counterparts, where the former resulted in approximately 2.5 speed–up. It is worth pointing out that the ADER-DG method, applied to the elastic wave propagation problem, results in multiplications of small matrices (see Table 4.1). Thus, in this case, the use of special matrix multiplication hardware units (i.e., *Nvidia*'s Tensor Cores and *AMD*'s matrix cores) may not necessarily improve performance because the amount of data is not enough to completely saturate the hardware units. Moreover, special matrix instructions will add extra latency, particularly on modern architectures, which may negatively affect performance.

GemmForge has been extended to generate CUDA, HIP and SYCL kernels since the publication of work [30]. The original, template-based approach was substituted with the instruction-based one; *GemmForge* defines a set of virtual instructions operating on the matrix data type - e.g., loading a matrix from global to shared memory, storing a matrix from registers to global memory, GEMM, etc. The new approach adds programming flexibility once combined with the *Builder* and *Factory* design patterns. For example, this helped to accommodate two GEMM algorithms in the current version of *GemmForge* (0.0.207) - i.e., *Type-1* and *Type-2*. The second algorithm (*Type-2*) is based on shuffle instructions and aims to minimize shared memory consumption. Each warp of a team loads a part of a matrix B row to registers, and at each iteration of the n-loop, a corresponding thread broadcasts its value to others. Loading tiles of matrix B results in un-coalesced memory access (if $Op(B)$ equals B) due to the adopted column-major matrix layout. However, this GEMM implementation assumes that loads of subsequent rows of matrix B will result in L1 hits - i.e., the corresponding cache lines will stay in L1 cache. The *Factory* method helps to encompass the empirical findings of this study regarding the algorithm selection for a concrete device architecture. For example, *Type-1* is well-suited for *Nvidia* and *AMD* GPUs, while *Type-2* results in better performance on *Intel* Ponte Vecchio GPUs, which was determined during a preliminary study.

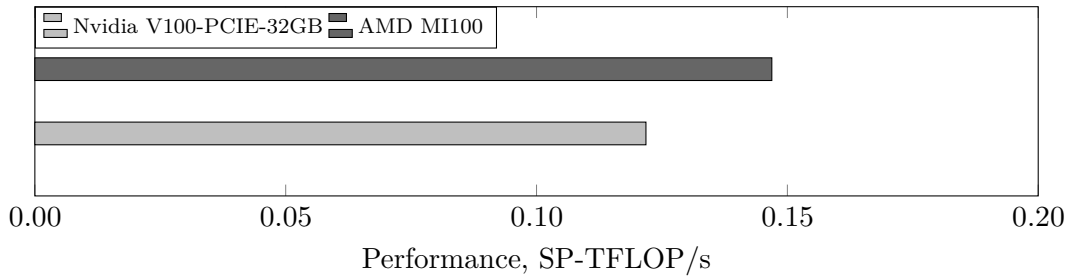


Figure 6.4.: Performance of the generated linear matrix combination kernel on *Nvidia* and *AMD* GPUs using CUDA-11.5 and ROCm-5.4, respectively.

Fig. 6.3 shows the performance of the kernel generated according to Listing 6.1. As can be seen, both GPUs demonstrate similar performance. However, the global memory bandwidth measured on MI100 GPU is approximately 32% higher than the one obtained on V100. Considering that the operation intensity is the same in both experiments, one may expect the kernel to perform better on MI100 - i.e., reaching about 8 SP-TFLOP/s. It is difficult to determine the exact reason for the observed behavior due to the absence of a unified profiling tool for *AMD* and *Nvidia* GPUs. However, I suspect that the kernel's performance on MI100 is mainly limited due to a relatively high L2 cache latency which cannot be entirely overlapped by n interactions of the inner loop. An attempt to software-prefetch one column of matrix A in advance did not help to improve performance on MI100; thus, a better GEMM algorithm for *AMD* GPUs is yet to find.

A linear combinations of matrices in *GemmForge* is given by

$$B_e = \alpha \cdot A_e + B_e \tag{6.2}$$

where e is a batch index; $A_e,\ B_e \in \mathbb{R}^{m \times n}$; and $\alpha,\ \in \mathbb{R}$.

It can be viewed as a stridden variant of the *SAXPY* operation. Therefore, the performance of this kernel type is mainly limited by the global memory bandwidth. *GemmForge* generates the code which performs each batch operation column-wise - i.e., using a team of threads equal to the column size of a matrix. Like the GEMM algorithm, thread blocks can be two-dimensional; thus, a single block can handle multiple batch operations. Each active thread executes 2 arithmetic operations per 3 memory accesses. Therefore, the arithmetic intensities of these kernels are bound to 1/6 and 1/12 regarding single- and double-precision floating-point format, respectively.

Fig. 6.4 depicts the performance of the kernel generated according to Eq. 6.2 using $m = 56$ and $n = 9$. As before, the results were obtained on MI100 and V100 GPUs. The difference in performance strongly correlates with the measured global memory bandwidth of the GPUs - i.e., $\approx$ 1021 GB/s and $\approx$ 770 GB/s on MI100 and V100, respectively. The obtained results are close to the roofline model outline, reaching 86% and 95% of the maximal performance of MI100 and V100 GPUs for the given arithmetic intensity, respectively.

6.3.2. ChainForge

The task decomposition proposed in Section 6.3.1 - i.e., at the level of binary batched operations - entails redundant data round trips between global device memory and compute units. This is caused by storing and loading intermediate results between subsequent data-dependent operations. Fusion of such GPU kernels results in holding intermediate results in low-latency memory - e.g., registers or shared memory. This approach eliminates redundant data movements and, thus, increases the arithmetic intensity of *SeisSol*.

Automatic fusion of GPU kernels is a known optimization technique used for accelerating many scientific (e.g., [36, 118, 37]) and deep learning applications (e.g., [18, 77]), and is always bound to some compiler technologies. Aggregating knowledge from other studies (e.g., [119, 75]), I can formulate three distinct reasons for fusing GPU kernels: (1) to achieve better instruction latency hiding by fusing two data-independent kernels that require different kinds of GPU resources; (2) to eliminate intermediate data round trips by fusing neighboring data-depended kernels; (3) to reduce energy consumption and thus to improve GPU power efficiency. Reason (2) is the most popular because many applications are memory-bound.

In general, an aggressive kernel fusion policy may worsen GPU performance because fused kernels may require too many run-time resources (e.g., registers per thread, shared memory per block, etc.) leading to low occupancy. All works listed above stick to the most generalized approach based on building data dependency graphs from computations, which are used to find candidates for fusion. This typically leads to many possible combinations. The algorithms for finding the best substitution graph are different and can be based on either some rules [18, 77], or empirical searches [36], or exhaustive searches coupled with automatic benchmarking [75] and performance models for pruning search spaces [37], or dynamic programming [119].

In *SeisSol*, the ASTs generated by *YATeTo* can be used as dependency graphs. As mentioned in Section 4.1, *SeisSol*'s macro-kernels mainly consist of sequences of GEMM operations in the context of elastic wave propagation. Therefore, I consider a greedy approach - i.e., fusing the longest sequence of batched GEMM kernels without splitting - because the run-time resources - e.g., shared memory - can be intelligently reused. Moreover, in contrast to the other works, I also exploit the associative property of matrix multiplications to find an optimal ordering of GEMMs within fused kernels. My approach simplifies requirements for code generation. Below, I provide a list of problems that I faced and had to address while working on kernels fusion in *SeisSol*.

Problem 1: One needs to find an efficient matrix-matrix multiplication template that can be used as a high-performance micro-kernel. Here, I can reuse some ideas from *GemmForge* - i.e., implementing a matrix multiplication as a sum of parallel outer products (see Section 6.3.1, the *Type-1* algorithm).

Problem 2: A sufficient number of threads per block and the maximal register array length must be found to generate an efficient code for a fused matrix-multiplication chain.

The data can be estimated during an initial analysis while traversing all GEMM operations within a chain. The largest column size within a chain determines the number of warps required for a thread block, whereas the maximal contraction length defines the register array size needed for a fused kernel. It is worth noting that some intermediate operations may require fewer threads than allocated. In this case, redundant threads need to be disabled by masking.

Problem 3: Multiple operands (matrices) may need to reside in shared memory simultaneously. This can be a result of a specific matrix multiplication order and/or required transpositions of operands within a chain. Therefore, it is necessary to find an algorithm that can estimate and reserve enough shared memory per thread block. Moreover, the reserved memory should be split into blocks to/from where multiple operands can be safely written/read, thus preventing potential race conditions. Additionally, the algorithm should minimize the overall shared memory consumption to avoid scheduling fewer blocks per SM than allowed due to hitting the hardware limits (see Section 5.1). This can be achieved by reusing some memory blocks several times for different operands during the "lifetime" of a fused kernel.

Problem 4: Some computations involve data stored in shared memory and, thus, must be preceded by thread synchronization instructions, which should be automatically inserted into the generated code. A naïve implementation could be quite conservative - i.e., inserting synchronizations before and after each read/write access to each shared memory block. However, an intelligent algorithm should insert the instructions only where they are practically needed.

Problem 5: Chains of matrix multiplications must be automatically identified within the ADER-DG method. In general, this may require some context-free grammar, which will define the structure and rules of a language that will be able to recognize subsequent matrix multiplications within a tensor expression. In *SeisSol*, this can be achieved within the existing code generation workflow - i.e., in *YATeTo*.

Problem 6: In *YATeTo*, the strength reduction step is designed to find an optimal order of matrix multiplications inside a chain for a scalar processor. The algorithm may require some adaptation for the SIMT execution model. Moreover, the algorithm should be aware that the resources of a thread block (i.e., the number of threads, shared memory, and registers) are allocated just before a kernel invocation and cannot be re-configured at run-time.

As a contribution to this part of the work, I developed *ChainForge* - i.e., an open-source *Python* library intended to be used as one of *YATeTo*'s backends - which addresses all problems mentioned above. In the following, I discuss some important implementation details, starting from Problem 3 onward.

Problem 3: Shared memory optimization

The optimal distribution of limited memory resources (e.g., registers) is a well-studied problem in compiler design. It is solved as a variant of the graph coloring problem [3]. The key difference between the register assignment and the problem considered in this study is that, in the former, the number of registers (colors) is strictly given and cannot be increased if necessary. Register spilling occurs when the number of required colors exceeds the number of available hardware registers. In my case, the number of shared memory blocks can be freely adjusted as needed. Using the Liveness analysis [3], the number can be obtained by counting how many blocks must live simultaneously. This will determine the minimum number of blocks (colors) which, in my case, will prevent spilling data from shared to global memory.

The register allocation and Liveness analysis algorithms operate on some low-level Intermediate Representation (IR) of the code, which must be designed for *ChainForge* to solve the problem. In my case, instructions primarily operate on matrices and arrays of registers, which can be represented as variables. Apart from auxiliary instructions, the most relevant ones for solving the allocation problem are 1) *load/store* to/from shared memory from/to global memory, 2) *store* to shared memory from arrays of registers 3) GEMM operations. From the Liveness analysis perspective, 1) and 2) populate a so-called *kill set*, whereas GEMM operands are responsible for *gen set*.

In the following, I explain the details of my implementation using an example of a matrix multiplications chain with some imposed evaluation order (see Eq. 6.3).

$$E_e = \Big(\big(A_e \cdot B_e\big) \cdot \big(C_e \cdot B_e\big) \Big) \cdot D_e \tag{6.3}$$

where e is a batch operation index; $E_e,\ A_e,\ C_e,\ D_e \in \mathbb{R}^{32\times12}$; and $B_e \in \mathbb{R}^{12\times32}$.

According to the *Type-1* GEMM algorithm, matrices A_e and C_e will reside in global memory, while matrices B_e and D_e will be pre-loaded to shared memory. Intermediate results ($T_e^0 = A_e \cdot B_e$, $T_e^1 = C_e \cdot B_e$, and $T_e^2 = T_e^1 \cdot T_e^2$) generated during a kernel execution will also reside in shared memory, which prevents storing and loading them to/from global memory. Fig. 6.3 shows a snippet of the resultant IR code; the graph coloring outcome and the operands' assignment to shared memory blocks are shown in Fig. 6.5. The block size can be determined using Eq. 6.4 since there are no lifetime conflicts between operands within a shared memory block.

$$|\mathcal{B}_i| = \max\big(|M_1|, |M_2|, \ldots, |M_n|\big) \tag{6.4}$$

where $\mathcal{B}_i$ is the i-th block of shared memory; $M_j \mid 1 \leq j \leq n$ are matrices assigned to $\mathcal{B}_i$; and $|\cdot|$ represents the size of a matrix or a block.

Considering the single-precision floating-point format, this approach consumes only 8192 bytes of shared memory per thread block. This is approximately 1.9 times less than the naïve implementation, which stores each matrix in a dedicated shared memory block - i.e., no shared memory reuse. It reduces the overall shared memory consumption per block,

Listing 6.3: Low-level Intermediate Representation (IR) of Eq. 6.3 in *ChainForge*. The IR instructions are shown using the bold dark green font; the register array - the bold dark blue font; shared memory - the bold dark red font; the operands residing in global memory - the bold light blue font; the operands residing in shared memory - the normal black font prefixed with "%" symbol.

```
1  %0 = load_g2s shrmem, B;
2  regs = gemm A, %0;
3  %1 = store_r2s shrmem, regs;
4  clear_regs regs;
5  regs = gemm C, %0;
6  %2 = store_r2s shrmem, regs;
7  clear_regs regs;
8  regs = gemm %1, %2;
9  %3 = store_r2s shrmem, regs;
10 clear_regs regs;
11 %4 = load_g2s shrmem, D;
12 regs = gemm %3, %4;
13 E = store_r2g regs;
```

Figure 6.5.: Graph coloring and the operands' assignment to shared memory blocks for Eq. 6.3.

which may result in scheduling more thread blocks per SM and, thus, may increase the number of ready-to-schedule warps.

Another important aspect, which is worth mentioning, is that matrices move along the memory hierarchy during a kernel execution, starting from global memory and going up to registers. Therefore, each variable needs to be augmented with an attribute describing the current level at which the associated matrix resides at a given time. In *ChainForge*, scoping and nested symbol tables are used to track changes in the locations of all operands between the levels. Because of generating C-like source code, each operand's symbolic name, which is bound to a specific matrix, must be changed at each memory hierarchy level. Therefore, instead of variable names, descriptions of matrices are used as the keys because they uniquely identify operands between multiple scopes.

The input to the code generator is given as a list of GEMM operations. Each operation in the list contains descriptions of its operands. In *ChainForge*, a batched matrix is described by: 1) the number of columns and rows, 2) the formal matrix size, given as a bounding box, and 3) the addressing. Descriptions of intermediate results are deduced by analyzing the corresponding GEMM operations. The initial symbolic names are assigned to all

matrices at the beginning of the code generation process. Matrices explicitly mentioned in the input are labeled as "*in global*"; the deduced ones - i.e., intermediate results - are marked as "*in registers*" because of their initial location within the memory hierarchy.

Problem 4: Threads synchronization

As mentioned, the code generator must prevent threads from simultaneous read/write access to the same memory address. This requires inserting block-level synchronizations into the correct places. A conservative, naïve approach would result in inserting synchronizations before and after each use of shared memory, which could harm the performance of a generated code.

Considering the example given in Fig. 6.3, the naïve approach would insert *SyncThreads*[4] instructions before and after instruction 2. A more intelligent algorithm would recognize that instruction 2 operates on shared memory of block 1, whereas the next GEMM operation uses data only from block 0. Instruction 5 moves the intermediate result ($T_e^1 = C_e \cdot B_e$) from registers to block 0 and thus must follow synchronization. This ensures that all threads in a block have finished reading data from block 0 while executing instruction 4. Therefore, synchronizations surrounding instruction 2, imposed by the naïve approach, become redundant and thus can be removed from the code.

To achieve the desirable result, I implemented a Data Flow Analysis (DFA), which requires three forward passes during the optimization stage of *ChainForge*. The first pass removes all previously inserted instructions of the naïve algorithm. While scanning IR, the second pass appends a list of variables written to shared memory. If a *gemm* instruction is encountered during the scan and one of its operands is in the list, then a *SyncThreads* instruction is inserted before the *gemm*, and the variable list gets emptied. The meaning of the second pass is to guarantee that all necessary synchronizations are performed before executing a *gemm* instruction.

The purpose of the third pass is to ensure that all read operations are completed before the next write operation to the same shared memory block occurs. This is implemented as follows. The pass keeps track of a boolean list while scanning IR. The list size equals the number of shared memory blocks. Here, the True value means that the corresponding block requires synchronization before the next use. The pass starts with initializing all list elements to False. Whenever a *gemm* instruction is encountered, the pass checks whether any of its operands are labeled as "*in shared*". If yes, it assigns True to the corresponding item of the list and continues scanning the intermediate code. If a *SyncThreads* instruction is encountered, which was inserted during the second pass, then the pass simply assigns False to all list elements without any instruction manipulations, as in the previous case. If a write operation to shared memory is encountered, the pass checks whether the destination shared memory block has already been marked with the True value. If yes, a

[4] italic font refers to particular instructions of *ChainForge* IR in this sub-section.

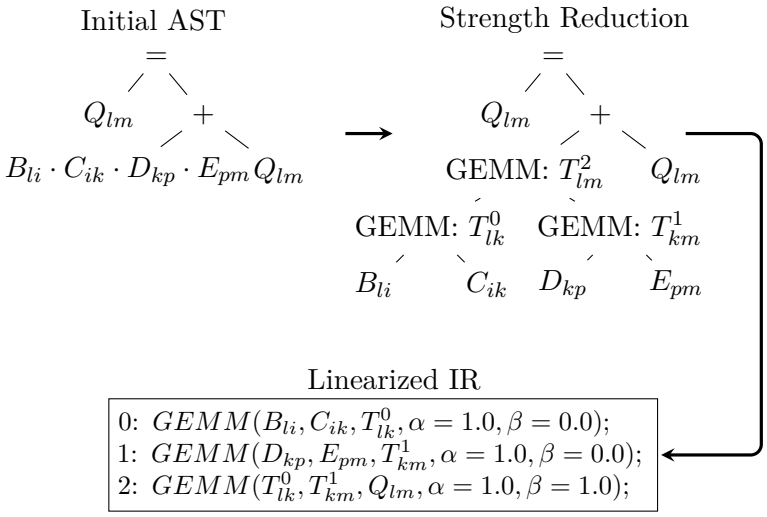


Figure 6.6.: Strength reduction in the *YATeTo* DSL and lowering to the linearized IR.

synchronization instruction gets inserted before the current one and the False value is assigned to all list items.

The application of these passes to the example shown in Eq. 6.3 reduces the number of thread synchronizations from 11 to 5 compared to the naïve approach.

Problem 5: Fused-GEMMs in the *YATeTo* DSL

As the first step, *YATeTo* builds an AST for a given tensor expression. Let's consider Eq. 6.5 as an example, for which its AST is given in Fig. 6.6.

$$Q_{lm} = Q_{lm} + B_{li} \cdot C_{ik} \cdot D_{kp} \cdot E_{pm} \tag{6.5}$$

As discussed in Section 4.2, *YATeTo*'s DSL syntax does not support parenthesized expressions. Thus, the initial AST contains chains of matrix multiplications (as tree nodes) which may not have an optimal evaluation order (see Fig. 6.6). *YATeTo* addresses this problem during the strength reduction step - i.e., finding an optimal sequence of tensor operations which results in minimizing the total number of floating-point operations under a memory constraint [71, 116]. Once the step is finished, the matrices must be grouped according to the found multiplication order and passed to *ChainForge*.

YATeTo maps a given tensor expression to a combination of several binary operations: *GEMM*, *Loop-over-Gemm*, *IndexSum*, *CopyScaleAdd*, etc. In this work, I designed an extra pass for the DSL, which recognizes only GEMM sequences from the linearized form of *YATeTo* IR. The pass utilizes a simple 2-state Finite Automata (FA) shown in Fig. 6.7.

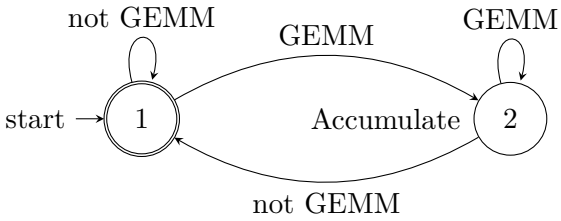


Figure 6.7.: Pattern matching for chains of matrix multiplications in *YATeTo*.

GEMM operations, accumulated at state 2, are removed from the IR whenever the FA returns to the accepting state - i.e., state 1. The removed GEMMs are replaced with a *FusedGEMMs* operation, added to *YATeTo* as a contribution of this work, to hold the accumulated sequence. During *YATeTo*'s code generation phase, every *FusedGEMMs* operation builds GEMM descriptions, puts them into a list, and passes it to *ChainForge*. In the end, *ChainForge* returns the generated code to *YATeTo*.

Problem 6: Optimal Chain Matrix Multiplication Order

Similar to [71], *YATeTo* finds an optimal evaluation order for a chain of tensor operations by decomposing the operations into multiplication ($f_r[\dots] = X[\dots] \times Y[\dots]$) and summation ($f_r[\dots] = \sum_i X[\dots]$) formulae which form a binary tree. The total number of floating-point operations for a tensor chain, which reflects a processor's total work, can be obtained by a post-order traversal of the tree and applying cost functions, as suggested in [71], to each formula. Obviously, a chain may have multiple tree representations. Thus, this optimization problem aims to find the tree with the minimal associated cost. Lam, Sadayappan, and Wenger proved that the problem is NP-complete and developed an efficient search procedure based on exhaustive search. As lengths of tensor chains and tensor dimensions are typically small in *SeisSol* [116], this procedure does not result in a considerable overhead in *YATeTo*.

The algorithm designed by Lam, Sadayappan, and Wenger (which is implemented in *YATeTo*) only considers execution on a scalar processor. In my case, a formula evaluation is supposed to be performed in parallel by several consecutive GPU threads. Therefore, finding an optimal evaluation order may require an additional objective function or/and a slight change of the cost functions for the formulae. A similar approach was made for the MIMD version of the algorithm in [71], where N scalar processors were involved in evaluating a single formula. The authors used an additional objective function to minimize communication overheads. However, data exchange between threads through shared memory is significantly faster in the SIMT model, and thus the MIMD approach is not applicable for this work.

I modified the algorithm proposed in [71] as follows. I changed the cost evaluation to reflect work done by a single GPU thread in the context of a single thread block execution. This is achieved by dividing each intermediate result obtained during a tree traversal by

the dimension size along which the thread-block parallelization is applied. My rationale is based on the fact that a block configuration (i.e., the number of threads, the shared memory size, etc.) is performed just before a kernel invocation and takes into account all operations within a chain. This cannot be re-configured at run-time - i.e., during a kernel execution. Therefore, some intermediate operations within a chain may need fewer threads than allocated. However, two *intermediate operations* result in about the same execution time if they have the same tensor contraction lengths but different sizes of the dimensions along which parallelization is applied. This assumption is plausible because each active GPU thread is given the same amount of work.

The thread-wise cost estimation results in more tree candidates for the final pruning step, during which the candidate that results in the least number of loads to shared memory is selected. The objective of the pruning step is not only to maximize the overall GPU occupancy but also to minimize latencies involved in loading data from global memory to shared one because it may help to reduce delays between computations within a chain.

The cost estimation and the pruning step are merged into *YATeTo*'s workflow. The pruning is implemented by adding penalties to the cost functions equal to the size of operands needed to be loaded from global to shared memory. Penalties are added only once, even if an operand appears more than once in a chain. This helps to mimic the liveness of an operand between multiple operations within a kernel.

6.3.3. Preliminary Performance Analysis

This section starts with a demonstration of the generated code performance using the benchmark shown in Eq. 6.6. The benchmark is derived from *SeisSol*'s implementation of the neighbor surface integral (see Eq. 4.5).

$$Q_e = Q_e + A \cdot B \cdot C \cdot D_e \cdot F_e \tag{6.6}$$

where $Q \in \mathbb{R}^{56\times 9}$; $A \in \mathbb{R}^{56\times 21}$; $B \in \mathbb{R}^{21\times 21}$; $C \in \mathbb{R}^{21\times 56}$; $D \in \mathbb{R}^{56\times 9}$; and $F \in \mathbb{R}^{9\times 9}$. In this example, the matrix sizes correlate to the convergence order 6 (i.e., 56 degrees of freedom) and the number of physical quantities (i.e., 9) involved in the elastic wave propagation problem. The batch size equaled to 10^5 was used for all experiments shown below.

The original cost estimation functions result in the following matrix multiplication order:

$$Q_e = Q_e + A \cdot \Big(\big(B \cdot (C \cdot D_e)\big) \cdot F_e\Big) \tag{6.7}$$

In contrast, applying the modified cost estimation changes the chain evaluation order to:

$$Q_e = Q_e + A \cdot \Big(B \cdot \big(C \cdot (D_e \cdot F_e)\big)\Big) \tag{6.8}$$

Table 6.1.: Cost evaluation statistics.

	Original cost	Cost per GPU thread	Penalty
Eq. (6.7)	52605	2664	774
Eq. (6.8)	57960	1971	81

Table 6.1 shows that the order imposed by Eq. 6.7 results in a lower total cost compared to Eq. 6.8 according to the original cost estimator - i.e., the total number of floating-point operations required for evaluating a tensor expression. However, the total work per GPU thread is lower for the order imposed by Eq. 6.8. Moreover, the order driven by Eq. 6.8 results in fewer explicit loads from global to shared memory, which can be observed from the last column ("Penalty") of Table 6.1.

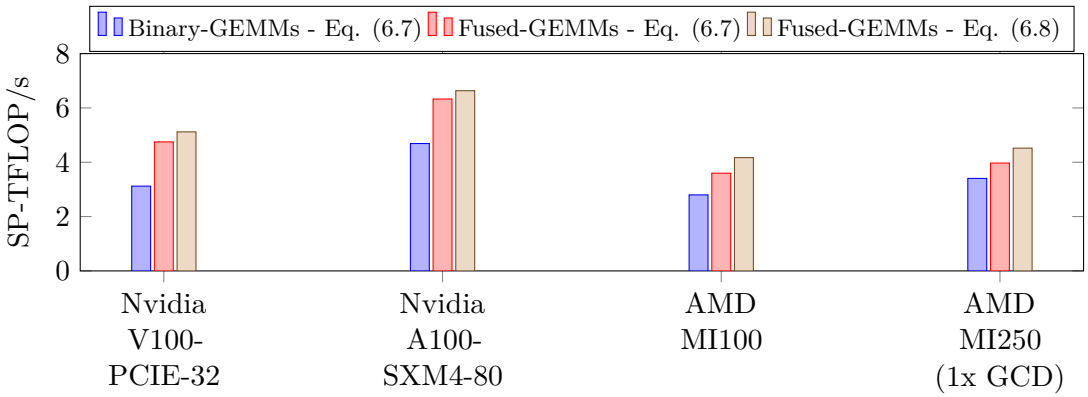


Figure 6.8.: Performance of the benchmarks (Eq. 6.7 and Eq. 6.8) obtained on various *Nvidia* and *AMD* graphics cards using binary and fused GEMM kernels.

Fig. 6.8 shows the results obtained on *Nvidia* and *AMD* GPUs while executing Eq. 6.7 and Eq. 6.8, implemented using the fused and binary batched GEMM operations. The results obtained with *GemmForge* (i.e., binary batched operations) are considered the baseline in this experiment. One can notice that the maximal performance increase was obtained on V100 GPU - i.e., about 52%. The performance obtained on A100 and MI100 GPUs increased by approximately 30% on average, whereas the performance on MI250x increased by only 17%. The ordering imposed by Eq. 6.8 resulted in approximately 6% and 15% performance increase relative to Eq. 6.7 on all tested *Nvidia* and *AMD* GPUs, respectively.

Fig. 6.9 demonstrates the roofline model analysis obtained with the *Nsight Compute* profiler on *Nvidia* V100 GPU. The performance measured by the profiler slightly differs from the one I manually calculated for Fig. 6.8 - i.e., using timers and manually counted floating-point operations. The profiler shows that the generated kernels (Eq. 6.7 and Eq. 6.8) reach close to the maximum achievable performance - i.e., approximately 83% on average. I continue the roofline model analysis in Section 6.3.4, where I demonstrate results obtained on *Nvidia* A100 and *AMD* MI250x GPUs.

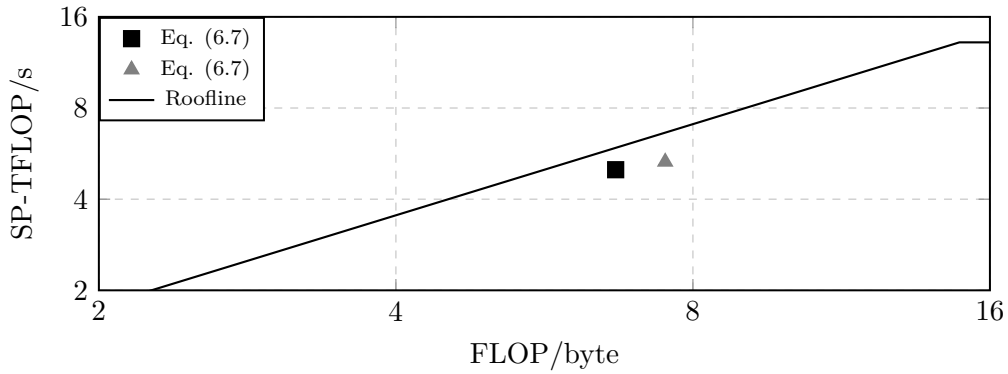


Figure 6.9.: Roofline model analysis obtained on *Nvidia* V100-PCIE-32 using *Nsight Compute*.

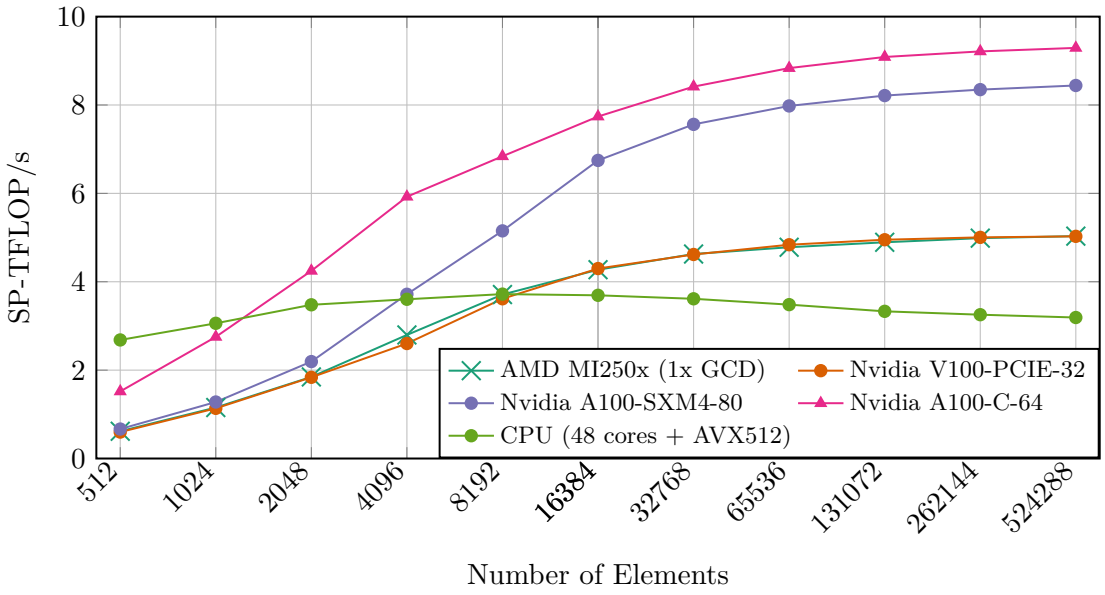


Figure 6.10.: Performance and elapsed time of *SeisSol*-proxy obtained on different single-GPUs using fused GEMM kernels. The CPU performance obtained on a dual-socket *Intel* Skylake Xeon Platinum 8174 is given as the baseline.

Fig. 6.10 shows the performance of *SeisSol*-proxy obtained on different GPUs using convergence order 6 and the single-precision floating-point format. The results obtained on *AMD* MI100 are excluded from the plot because this GPU model provides unified memory support (required for the proxy) through zero-copy memory, which leads to very low performance. From here onward, I refer to *Nvidia* A100-C-64 as the type of A100 GPU installed on the Leonardo supercomputer, which, according to [111], is custom-built for the CINECA supercomputing center and equipped with approximately 20% more SMs in comparison to the standard *Nvidia* A100-SXM4-80 - i.e., used on the Selene supercomputer. As can be noticed, the performance measured on the A100-C-64 GPU does not drop as drastically as the ones obtained with other tested GPUs under low- and medium-sized workloads.

Table 6.2.: Maximum and average speed-ups computed for Fig. 6.10. Note, the x-axis of the plot shown Fig. 6.10 is logarithmic.

	Nvidia A100 C-64	Nvidia A100 SXM4-80	Nvidia V100 PCIE-32	AMD MI250x (1x GCD)
Maximum	2.56	2.25	1.38	1.26
Average	2.48	2.19	1.34	1.20

The CPU performance obtained on *Intel* Skylake Xeon Platinum 8174 is added to the plot as the baseline to reflect the overall speed-up of the GPU version of the proxy application. As mentioned in Section 4.3, the proxy consists of a set of benchmarks containing the main *SeisSol*'s macro-kernels. In this experiment, I used all macro-kernels executed in the same order as in the main application - i.e., *SeisSol*. This reflects the overall performance of *SeisSol*'s wave propagation solver in isolation. This experiment can also be considered as strong scaling of a single LTS time-cluster on a single CPU/GPU.

Unlike CPU performance, which reaches the plateau almost immediately, GPU performance strongly depends on the time-cluster size; it requires having at least 32768 elements in a cluster to reach about 90% of the maximal computational throughput. A further decrease in the time-cluster size leads to a rapid drop in GPU performance most likely because kernels launching overheads get less and less overlapped with computations on a device (see Section 5.3). This behavior was observed on all tested GPUs and can be considered the main takeaway from this experiment.

The maximal and average speed-ups against the baseline are shown in Table 6.2. The values were computed based on the elapsed time of *SeisSol*-proxy, and averaged between 500 repeats. I claim that the values given in Table 6.2 should be treated as the absolute speed-ups because a highly optimized, vectorized, and multithreaded CPU version of *SeisSol*-proxy was used as the baseline. Moreover, a 48-core AVX512 CPU server (i.e., a single node of the SuperMUC-NG supercomputer) was used in the experiment.

All conducted experiments shown above highlight the following contributions of this part of the study. Firstly, I showed that fused GPU kernels considerably improved the overall GPU performance of the ADER-DG method. This establishes the primary direction for the future development of *SeisSol*. Secondly, I showed and discussed the main challenges and aspects related to the automatic fusion of GPU kernels using a single and very specific tensor operation - i.e., GEMM. Extending the compiler-based approach for fusing other chained tensor operations (e.g., tensor product, tensor contraction) or combinations of them is challenging but worth trying to achieve higher GPU performance for other wave propagation models - e.g., viscoelastic one. Thirdly, I extended the algorithm proposed by Lam, Sadayappan, and Wenger in [71] for the SIMT model - i.e., using the thread-wise cost estimation for the multiplication and summation formulae. Lastly, I showed that optimal matrix multiplication ordering is hardware-dependent and must be considered while working on performance portability.

6.3.4. Revisiting the Flux Matrix Decomposition

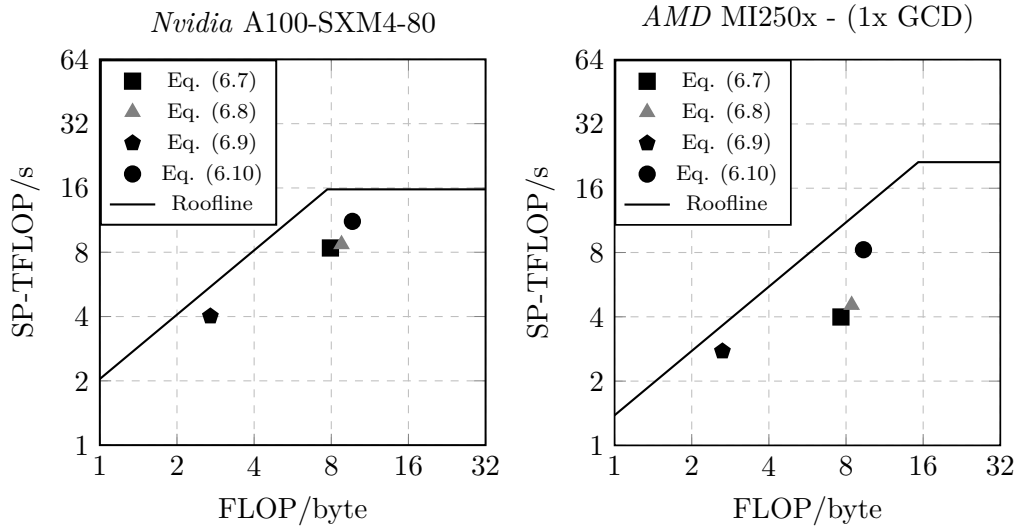


Figure 6.11.: Roofline model analysis obtained on *Nvidia* A100-SXM4-80 and *AMD* MI250x (1x GCD) GPUs using *Nsight Compute* and *Omniperf*, respectively.

Fig. 6.11 shows the roofline model analyses obtained on *Nvidia* A100 and *AMD* MI250x GPUs for the benchmarks defined by Eq. 6.7 and Eq. 6.8. One can observe that the measured performance values significantly deviate from the roofline outline. For example, the performance of Eq. 6.8 on MI250x GPU reached only 37% relative to the maximum. Both *Nsight Compute* and *Omniperfs* identify significant stalls in the load/store pipeline originating between L2 and L1 GPU caches. This probably occurs while reading columns of the constant matrices - i.e., A, B, and C - because they are supposed to reside in L2 cache during the whole execution of the kernels due to the temporary locality. One way to verify this hypothesis is to change Eq. 6.7 as follows

$$Q_e = Q_e + A_e \cdot \Big(B_e \cdot \big(C_e \cdot (D_e \cdot F_e) \big) \Big) \tag{6.9}$$

In this case, matrices A_e, B_e, and C_e are individual for each batch index e in contrast to Eq. 6.7. In this example, there is no data reuse regarding L2 cache. Therefore, I expect the total traffic between global memory and compute units to increase and, thus, the arithmetic intensity of the kernel to drop. This reasoning matches the results obtained during the experiment, shown in Fig. 6.11 (see solid black pentagons). In this case, the generated kernels reached 72% and 76% of the maximum performance of A100 and MI250x GPUs, respectively. This indirectly supports my hypothesis mentioned above. Therefore, I conclude that the performance of the kernels generated according to Eq. 6.7 is primarily limited by the latency of L2 cache.

It is worth mentioning that the benchmark (see Eq. 6.6) is derived from the neighbor integral macro-kernel (see I^{nghb}_{surf} in Eq. 4.5), which is subjected to the flux matrix decomposition (see Eq. 4.4). Uphoff et al. in [117] state that the flux decomposition results in performing fewer floating-point operations at run-time in comparison to using

the original flux matrices in the case of an optimal matrix chain multiplication order. In that work, the authors were focused on optimizing *SeisSol* for *Intel*'s Haswell and Knights Landing CPU microarchitectures and pointed out that the approach led to occupying less memory space and, thus, to fewer data evictions from the top-level CPU caches. Considering differences between CPU and GPU microarchitectures, especially in the design of L2 caches and their sizes, and differences in the task decompositions (see Section 4.3 and Section 6.2), one can assume that the flux matrix decomposition may be redundant for GPUs. This assumption can be tested using the benchmark proposed in Eq. 6.10, where multiplications of matrices A, B, and C are pre-computed in advance and stored in matrix K.

$$Q_e = Q_e + K \cdot (D_e \cdot F_e) \tag{6.10}$$

Looking at Fig. 6.11, one can observe about 1.5x speed-up of the kernels generated according to Eq. 6.10 relative to the ones subjected to Eq. 6.7. The performance of the former reached almost 70% on A100 and 63% on MI250x relative to their maximum based on the estimated arithmetic intensities. In this experiment, the number of coalesced data loads issued by a thread block is reduced from 128 to 56 relative to the implementation of Eq. 6.7.

In this work, I identified two major differences between *Nsight Compute* and *Omniperf* profilers while using them for making the roofline analysis. Firstly, *Nsight Compute* reduces the clocking of SMs during profiling by almost 20%. As a result, the time of a kernel execution becomes longer, which skews the real performance of a kernel. *Omniperf* operates on timing data obtained from a dedicated kernel-run during which the original clocking is used. Secondly, the hardware counters of *Nvidia* GPUs track the arithmetic and logical operations per each CUDA core and each double-precision unit. This approach gives the most precise information about the total number of executed floating-point operations during a test. *AMD* GPUs, on the other hand, only track the arithmetic and logical instructions issued per each SIMD unit - e.g., see [44]. While calculating the total number of executed floating-point operations, *Omniperf* assumes that all threads in a wavefront participate in computations. This approach can overestimate the performance and arithmetic intensity values when thread-mask instructions are used in a kernel - e.g., line 32 in Listing 6.2. This is shown in Fig. 6.12 where the hollow markers depict values obtained from *Omniperf*, whereas the solid ones represent the adjusted results - i.e., using manually counted floating-point operations based on matrix sizes. For example, the difference between the reported and adjusted values reaches approximately 50% in the case of Eq. 6.7 benchmark.

The adjusted values were used in Fig. 6.11 to compensate for the methodological differences between *Nsight Compute* and *Omniprof* profilers and, thus, to demonstrate a fair comparison. Regarding A100-SXM4-80 GPU, the performance values were adjusted using timing data obtained while executing the benchmarks with the original device clocking. Regarding MI250x GPU, the performance and intensity values were obtained using the number of floating-point operations counted manually. In both cases, the roofline model outlines were taken as reported by the tools.

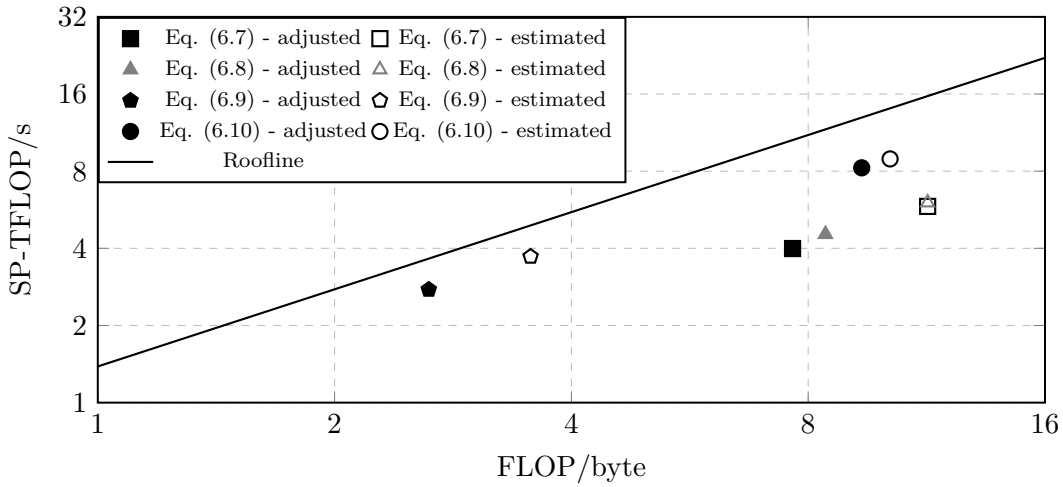


Figure 6.12.: Comparisons of the roofline model obtained with *Omniperf* and its adjusted variant for *AMD* MI250x GPU.

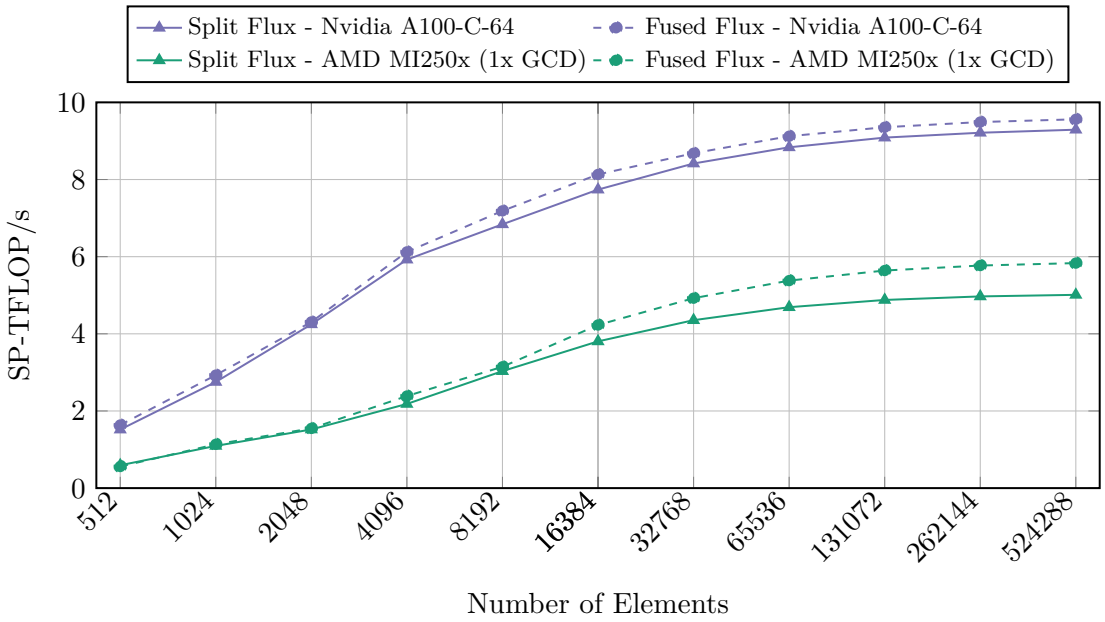


Figure 6.13.: Performance of *SeisSol*-proxy obtained using the split and fused flux matrices.

As shown in Fig. 6.13, on average, the use of the fused flux matrices increases the computational throughput of *SeisSol*-proxy by 3% and 16% on A100 and MI250x GPUs, respectively. It is worth pointing out that I_{surf}^{local} and I_{surf}^{nghb} macro-kernels constitute about 23% and 20% of the total execution time of the computational scheme defined by Eq. 4.1 and Eq. 4.2.

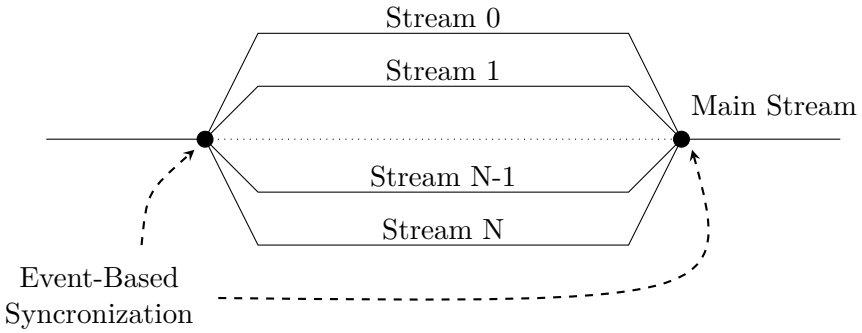


Figure 6.14.: Stream-based fork-join model.

6.4. Concurrent Task Execution

In Section 6.2, a GPU task is defined as an execution of a batched tensor expression. Some GPU tasks are data-independent and thus can be executed concurrently. For example, each face of a tetrahedron is parameterized with h, i, j values which define a flux matrix that needs to be used in the neighbor surface integral macro-kernel - i.e., I_{surf}^{nghb}. Each flux matrix has its unique values and sparsity pattern, and thus, each combination of h, i, j values defines a unique tensor expression. As shown in Algorithm 2, each iteration of the outer most loop (line 4) can result in executing up to 48 data-independent GPU tasks. Some of these tasks can contain just a few elements, which may lead to low utilization of the GPU resources during the execution of such tasks. The same reasoning applies to the local surface integral macro-kernel - i.e., I_{surf}^{local} (see Eq. 4.5).

Modern GPUs support grid-level concurrency - i.e., launching multiple GPU tasks in different streams. GPU programming models refer to a stream as a sequence of asynchronous operations that are executed on a device in the order issued by the host. Each stream is mapped to a single device connection which binds submitted tasks to a specific hardware work queue. Having multiple hardware work queues on a device allows control units to retrieve and schedule several tasks simultaneously if enough computational resources are available. Therefore, executions of independent tasks - i.e., taken from different queues - can be partially or even completely overlapped, leading to better utilization of all hardware components.

GPU programming models like CUDA and HIP allow users to create infinitely many streams. However, their device runtime libraries always bind streams to a concrete number of hardware work queues. This means that some streams may be bound to the same work queue. Therefore, the tasks submitted to such streams will be executed in order, which limits the concurrency specified by a programmer. The programming models also allow users to inject events (time markers) into a flow of tasks. The event-driven model provides fine-grain control on the execution of tasks. For example, scheduling tasks from one stream can be postponed till the completion of an event injected into another stream. Frequent allocations and deallocations of streams and events can introduce noticeable overheads at run-time. Therefore, it is preferable to create all necessary resources only once - i.e., at

Algorithm 3 Stream-based implementation of the Neighbor Surface Integral - i.e., I_{surf}^{nghb}

1: **procedure** ComputeNSI($Device$, $LtsLayer$, $PreComputedData$)
2: $StreamManager \leftarrow Device.getStreamManager()$
3: $OuterTable \leftarrow LtsLayer.getBatchTable()$
4: $M \leftarrow LtsLayer.getClusterSize()$
5: **for** j **from** 1 **to** 4 **do**
6: $StreamManager.fork()$
7: **for** f **from** 1 **to** 48 **do**
8: $condition = Condition(FaceKind :: Regular, FaceKind :: Periodic)$
9: $key \leftarrow Key(KenelName :: NeighborFlux, condition, j, f)$
10: $InnerTable \leftarrow OuterTable[key]$
11: **if not** $InnerTable.empty()$ **then**
12: $stream \leftarrow StreamManager.nextStream()$
13: $h \leftarrow f \mod 3$
14: $i \leftarrow f \mod 12$
15: $\mathcal{F}_{kl}^{+,jih} \leftarrow PreComputedData.getFluxMatrices(i, h, j)$
16: $Q_{lp}^{1:M} \leftarrow InnerTable[InnerKey :: DOFs]$
17: $\mathcal{T}_{tq}^{1:M} \leftarrow InnerTable[InnerKey :: IntegratedDOFs]$
18: $\mathcal{A}_{qp}^{+,1:M} \leftarrow InnerTable[InnerKey :: FluxSolver]$
19: $Q_{lp}^{1:M} = Q_{lp}^{1:M} - \mathcal{F}_{kl}^{+,jih} \cdot \mathcal{T}_{tq}^{1:M} \cdot \mathcal{A}_{qp}^{+,1:M}$ ▷ Launch the task in $stream$
20: **end if**
21: **end for**
22: $StreamManager.join()$
23: **end for**$Device.wait()$
24: **end procedure**

the beginning of a program's execution - and destroy them at the end. Streams and events (as software instances) can be reused multiple times during the life of an application. A programmer is responsible for ensuring that an event has been completed before reusing its software instance.

In this work, the creation, destruction, and management of streams are delegated to a so-called *stream manager*, which is similar to the thread pool concept. The manager allocates a pool of streams and events, and stores them in circular buffers. When a user requests a new stream, the manager increments its internal counter and spins the buffers. When the counter exceeds the number of the allocated streams, the manager resets the counter and moves the iterators of the circular buffers to their heads. The manager also provides a fork-join mechanism, shown in Fig. 6.14, to synchronize encapsulated streams with the application's default stream (also known as the main stream). The fork-join model helps to abstract specific and low-level details of the steam-based synchronization - e.g., events injection and polling - and to create a portability layer that can be used to adapt other GPU programming models. Algorithm 3 shows the stream-based computing applied to the neighbor surface macro-kernel in *SeisSol*. The streams are forked in each iteration of the j-loop (line 5). Internally, the manager injects an event into the default stream and forces other streams to wait for its completion. This guarantees that all tasks

submitted to the main stream are finished before processing tasks submitted to the forked ones. The algorithm fetches the next stream from the circular buffer if the condition in line 10 holds and submits the task shown in line 19 to this stream. The process continues till the end of the f-loop, which follows the join command. At this point, the manager injects events into each forked stream and forces the main stream to wait for their completions. This ensures that all tasks submitted to the forked streams are completed before the next use of the main stream occurs.

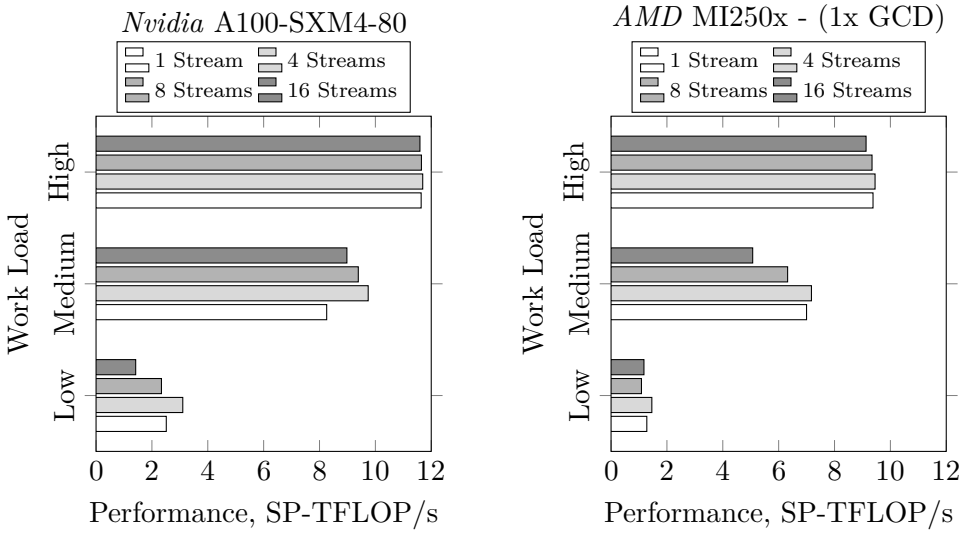
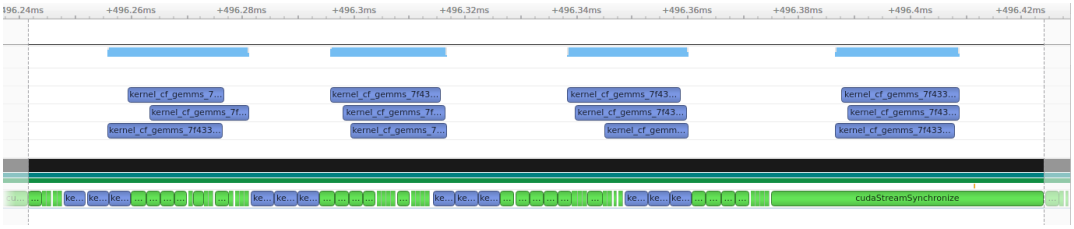


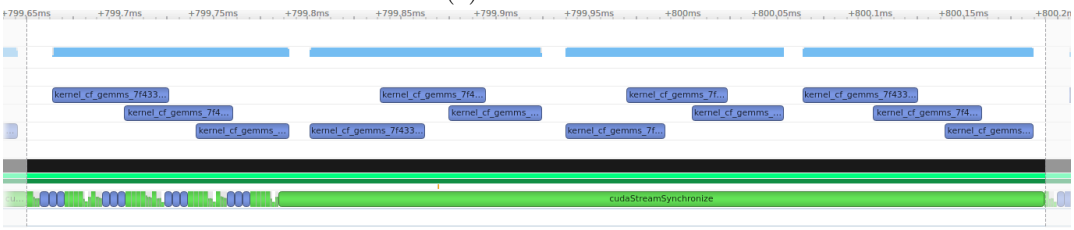
Figure 6.15.: Comparisons of the stream-based implementations of the surface neighbor macro-kernel on different GPUs relative to the number of concurrent streams under different workloads. The workloads - i.e., the number of elements - used in the experiments: "Low" - 1024, "Medium" - 16384, "High" - 262144.

In *SeisSol*, the stream-based execution was applied to all face-parameterized macro-kernels (see Section 4.1), namely: I^{local}_{surf}, I^{nghb}_{surf}, I^{local}_{inter} and I^{nghb}_{inter}. Fig. 6.15 shows how steam-based computing affects the performance of the surface neighbor integral on different GPUs and under different workloads. As expected, the single- and multi-stream implementations are equally efficient under the "High" workload because each offloaded task is large enough to utilize all compute resources completely. Noticeable differences can only be observed under the "Low" and "Medium" workloads, which become relevant for the LTS scheme, especially during strong scaling (see Section 4.4). The use of too many streams - e.g., 16 - imposes considerable synchronization overheads, which may even lead to lower performance relative to the single-stream implementation. According to the obtained results, the optimal number of steams is equal to 4. Similar results were obtained for other face-parameterized macro-kernels on *Nvidia* A100 and *AMD* MI250x GPUs. Therefore, this value - i.e., 4 - was selected as the default circular stream buffer size in *SeisSol*.

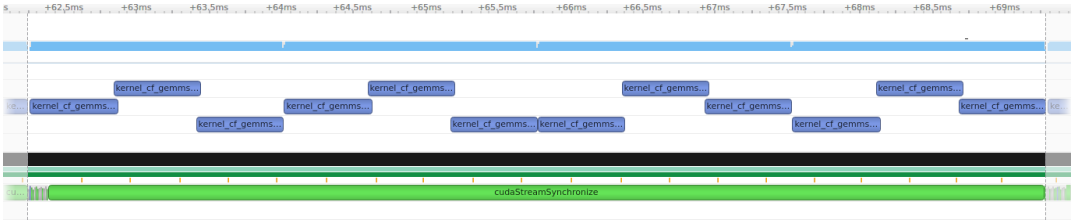
Fig. 6.16 shows tracing results obtained on *Nvidia* A100 GPU. One can observe that the executions of the kernels issued to different streams are almost completely overlapped under the "Low" workload but the overheads, related to the stream synchronizations, dominate. The situation completely changes under the "High" workload - i.e. the kernels'



(a) “Low” workload.



(b) “Medium” workload.



(c) “High” workload.

Figure 6.16.: Tracing of the stream-based CUDA implementations of the surface neighbor macro-kernel under different workloads obtained on *Nvidia* A100-SXM4-80 GPU using the circular stream buffer size equal to 4.

execution is almost fully serialized but the overheads become negligible. In all tests shown in Fig. 6.16, only 3 streams were simultaneously active, even though the stream manager was configured with the circular stream buffer size equal to 4.

6.5. Execution on Distributed Multi-GPU Systems

In this work, the GPU-Aware MPI is used to scale *SeisSol* across multiple GPU-accelerated nodes. The GPU-Awareness adds an extra mechanism to an interface implementation to directly transfer data between GPUs over a network without explicitly involving the host systems. This results in bypassing unnecessary data movement through host memory and, thus, reduces potential overheads associated with GPU-to-GPU data transfers.

The intra-node GPU communication can be implemented in multiple ways: Single Process Multiple GPUs (SPMG), Single Thread Single GPU (STSG), Single Process Single GPU (SPSG), or Multiple Processes Single GPU (MPSG). The SPMG approach may be convenient for simple programs with straightforward sub-partitioning and data exchange. STSG may suit well for executing coarse-grained data-independent tasks on multiple

GPUs, especially when no data exchange between GPUs within a node is required. In both cases, the number of overall MPI processes is equal to the number of the involved GPU nodes, which may lead to lower communication overheads and, thus, to better scaling. The MPSG model may benefit multiphysics simulation software in which solvers communicate via an external coupling library. In this scenario, all solvers may be bound to the same GPUs, and each solver may have no or little knowledge about the others running on the same node. Similar to the stream-based execution model (see Section 6.4), tasks offloaded from different processes to the same GPU run concurrently, which may increase the overall GPU utilization if tasks are small.

In this study, I used the SPSG approach in *SeisSol* due to the irregular and complex domain decomposition involved in the application. This allowed me to reuse many parts of the existing mesh partitioning algorithm and MPI code. In contrast to the CPU version of *SeisSol*, where a single process is used per node or NUMA domain, the SPSG model increases the total number of MPI processes if a node contains more than one GPU. In this case, the average message size becomes smaller, and the overall communication traffic grows. Taking into account the LTS scheme, which splits each sub-partition into multiple time-clusters, the application may become sensitive to network latency.

Hardware topologies of modern HPC nodes become complicated (e.g., see Fig. 6.17). The use of multi-socket CPU systems allows hardware vendors to accommodate more accelerators in a single node and, thus, increase its computational power. In this case, the CPUs are interconnected through a high-speed interconnect fabric - e.g., *Intel* QuickPath Interconnect, *AMD* Infinity Fabric, etc. A node can also be equipped with several Network Interface Controllers (NICs) to increase its network throughput. In such a configuration, GPUs and NICs are usually connected to different sockets. Controlling and managing them from a process residing on a neighboring socket adds extra latency to an application because the data needs to travel through the fabric, which connects sockets. Therefore, correct process binding is very important for latency-sensitive applications. For example, as shown in Fig. 6.17, a process needs to be bound to cores 48-55, belonging to the third NUMA domain, to work efficiently with GPU 0 on a node of the Crusher supercomputer. In this configuration, any message sent from the host initially moves to GPU 0, from where it is transferred to GPU 1 through the GPU-to-GPU Infinity Fabric link and then it moves down to NIC 0 attached to GPU 1 through the PCIe Gen4 ESM interconnect.

An automatic process binding within an HPC application can be difficult to implement because a programmer needs to query and aggregate the locations of all CPUs, GPUs, and NICs allocated for a job and distribute the binding decision to all running processes. Moreover, the solution must be portable across various operating systems, which may have different tools for querying resources. Modern workload managers - e.g., *SLURM*, *Torque*, etc. - are supposed to perform an optimal process binding automatically during job scheduling. However, this approach may still require some interaction with a user via setting additional environment variables. In this work, I stick to the latter approach because 1) it does not require implementing and maintaining complex logic in the application, and 2) many HPC data centers provide detailed descriptions of how to bind resources optimally. Moreover, one can expect that commonly used HPC workload managers will

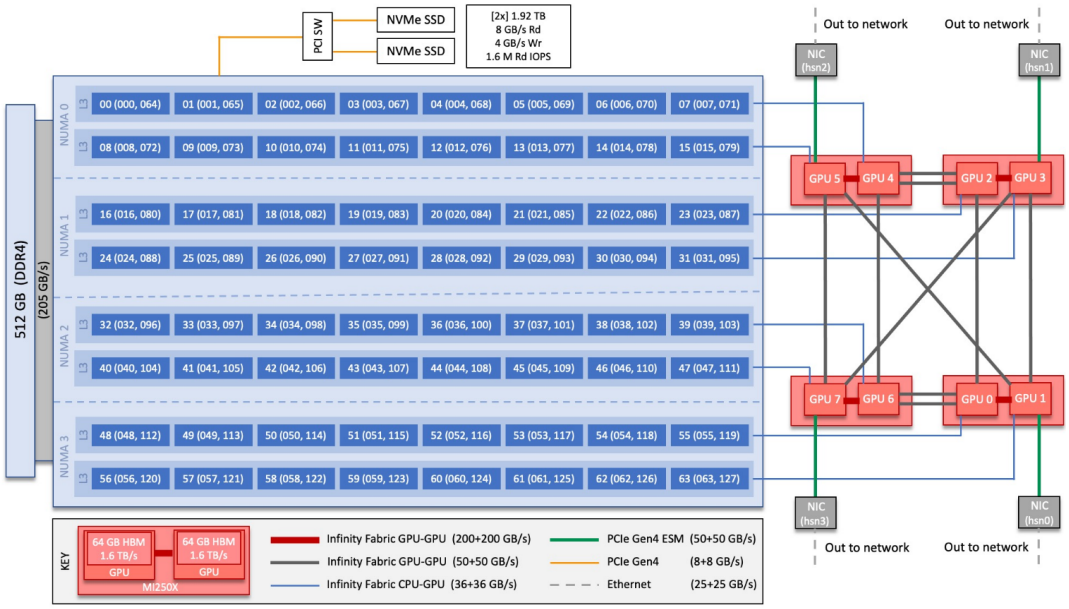


Figure 6.17.: Topology of a single node of the *Crusher* supercomputer. The picture is taken from [85].

be improved in the future regarding the automatic process binding due to a high demand from the HPC community.

In this section, I demonstrate *SeisSol*'s performance on distributed multi-GPU systems using the Layer Over Half-space (LOH.1) test scenario. The scenario aims to simulate an earthquake event using a single kinematic point source located between two adjacent regions with different material properties. Thus, almost all computational resources are concentrated on the wave propagation solver. A detailed description of the scenario can be found in [22]. The original geometry of the scenario is shown in Fig. 6.18, where one can observe a local mesh refinement around the point source. Meshes with approximately 20 million elements were used in this part of the study to demonstrate scaling properties of *SeisSol*. All reported experiments were performed on the LUMI, Leonardo and Selene supercomputers (see A.1).

The key adjustments in the MPI part of *SeisSol* are shown and explained in Section 6.5.1. In Section 6.5.4, I present new versions of the node-weights, which can balance both work and memory between MPI processes during mesh partitioning. These enhancements may be necessary to perform weak scaling studies of *SeisSol* on machines equipped with GPUs with limited onboard memory - e.g., *Nvidia* V100-SXM2-16GB. Section 6.5.3 shows how distributions of elements between LTS time-clusters affect GPU strong scaling performance.

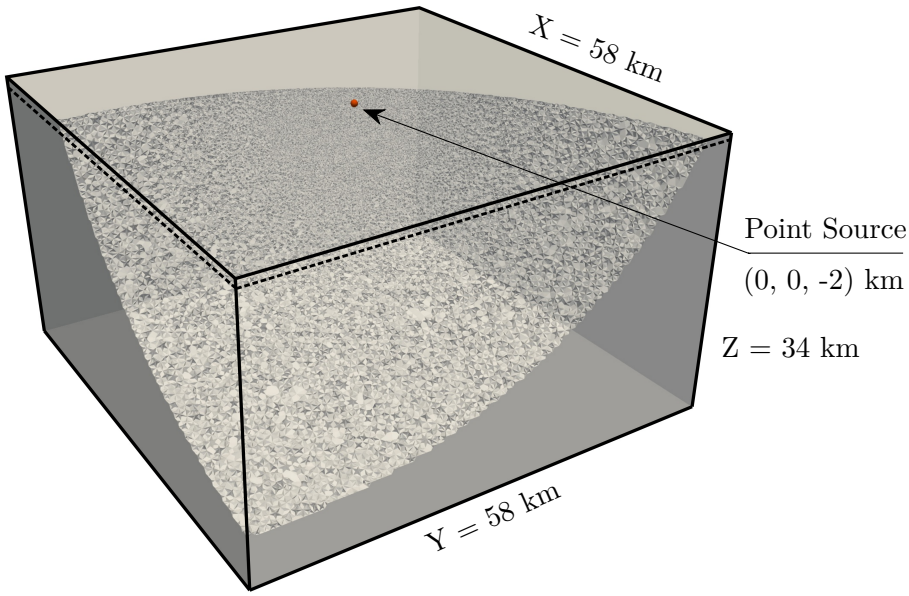


Figure 6.18.: Geometry and a computational mesh of the LOH.1 test scenario.

6.5.1. MPI Buffers Placement

As discussed in Section 4.4, in *SeisSol*, MPI processes exchange DOFs and their derivatives over continuous regions inside *copy* and *ghost* layers of time-clusters using point-to-point non-blocking communication. In Section 6.1, I mentioned that memory for DOFs and their derivatives are allocated using the *unified memory* type.

In [8], Banerjee, Hamidouche, and Panda proposed and implemented the initial and very basic support for unified memory in MPI. Before sending or receiving a message, the authors checked whether the MPI buffers had been allocated with unified memory using *cudaGetPointerAttributes()*. In that case, the authors launched a small CPU kernel performing dummy reads of that memory, which forced to migrate all necessary pages to the host side. After that, a regular host-to-host data exchange was followed. Hamidouche et al. in [45] pointed out that that approach was limited because it did not take any advantage of *Nvidia* GPUDirect Peer-to-Peer and Remote Direct Memory Access (RDMA) technologies, which could provide low-latency and high-bandwidth data exchange between GPUs within and across nodes. In [45], the authors proposed and implemented a more advanced design to support unified memory for intra- and inter-node communication. The key idea was based on pre-allocating auxiliary buffers using regular device memory and copying every unified MPI buffer there before sending or receiving a message. Once combined with the CUDA Inter-Process Communication feature, GPU RDMA, and software pipelining, the new design showed considerable improvements in point-to-point and collective communications for all message sizes.

Manian, Ammar, et al. in [81] and later Manian, Chu, et al. in [82] performed a comparative study of intra- and inter-node point-to-point communication on various *Nvidia* GPU

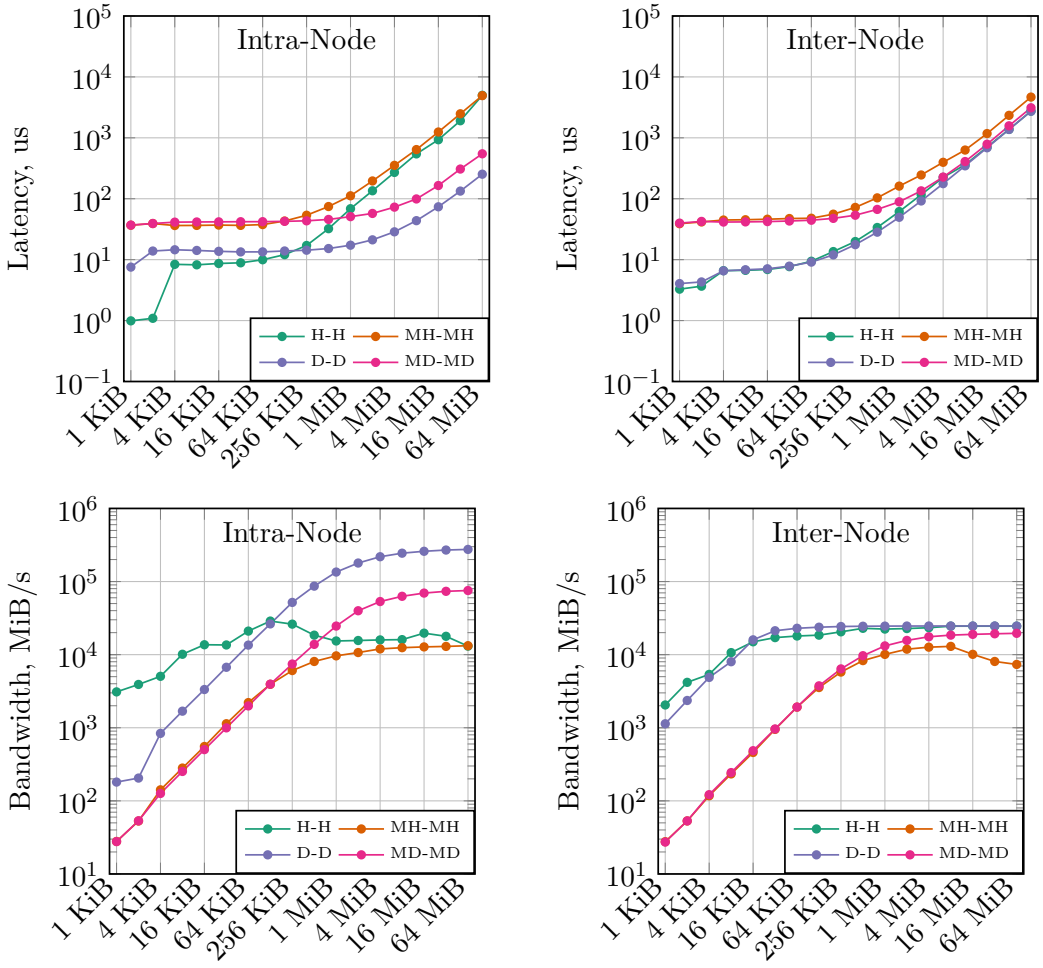


Figure 6.19.: Results of the latency and unidirectional bandwidth tests conducted on the Selene supercomputer.

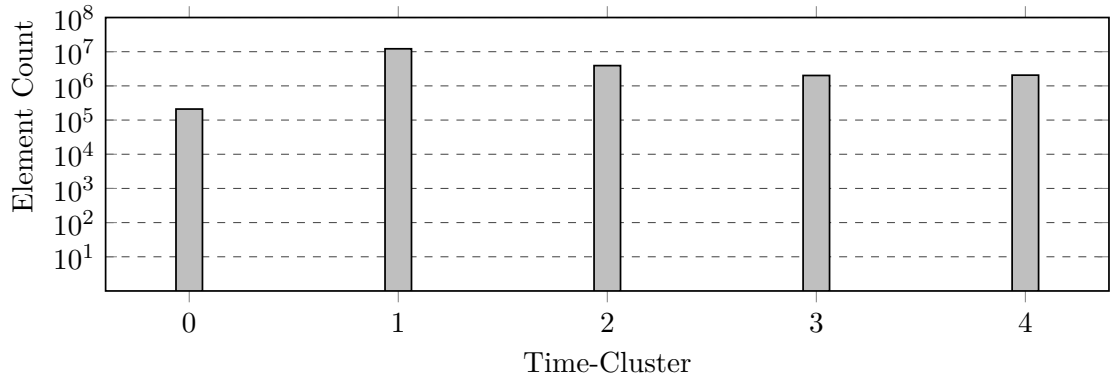
platforms using different types of MPI buffers. The authors extended the OSU Micro-benchmarks [70] by adding an option to allocate the MPI buffers using the unified memory type. In that case, dedicated kernels were launched before sending and after receiving each message to explicitly control page locations of the unified MPI buffers - i.e., either on the host or device. That helped to mimic various scenarios that can happen in a real user application. In those works, a CUDA-Aware MPI library - i.e., MVAPICH2- GDR - was used with the advanced unified memory design proposed in [45]. Similar to [82], the following abbreviations of the MPI buffers regarding their locations are used in this study: H - host; D - device; MH - unified memory when pages reside on the host; and MD - unified memory when pages reside on the device. Furthermore, I refer to a test configuration using a pair separated by the hyphen - e.g., H-H, D-D - where the first element denotes the buffer type of MPI process 0 (sender) and the second one refers to the buffer type of MPI process 1 (receiver).

According to the experiments conducted by Manian, Chu, et al. in [82], exchanging large messages (more than 4 MiB) in the MH-MH and MD-MD configurations does not entail considerable overheads for intra- and inter-node communication. However, sending/receiving small (up to 1 KiB) and medium (between 4 KiB and 1 MiB) messages using the unified MPI buffers results in higher latency and lower bandwidth in comparison to the H-H and D-D configurations. Similar results can be observed in Fig. 6.19, where I depict outcomes of the latency and bandwidth tests conducted on the Selene supercomputer for intra- and inter-node communication. In contrast to [82], in my experiment, the latency of intra-node communication of the MD-MD configuration is about 2.25 times higher compared to the D-D setup, on average. At the same time, the bandwidth of the MD-MD configuration is approximately 3.7 times lower relative to the corresponding baseline. In the experiments shown in Fig. 6.19, the MH-MH and MD-MD configurations were obtained by setting the *UCX_RNDV_FRAG_MEM_TYPE* environment variable of the UCX layer [103] to the *host* and *cuda* values, respectively.

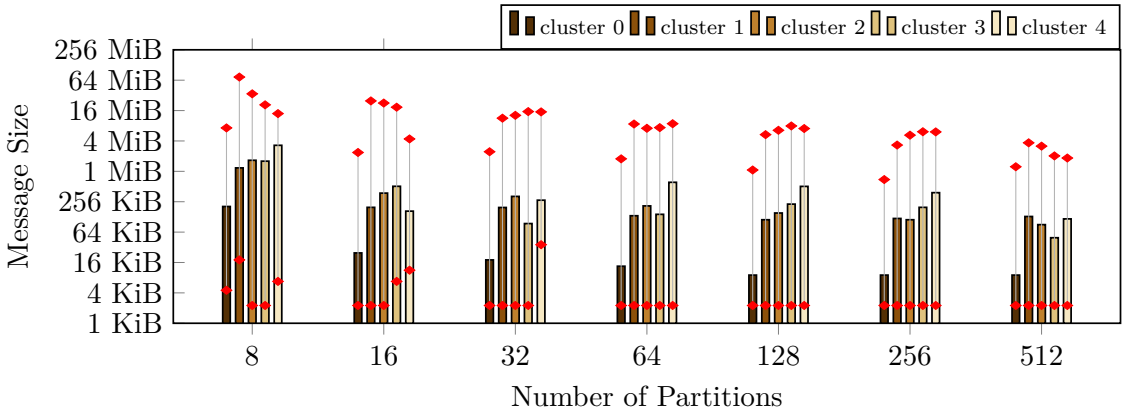
Fig. 6.20 shows the analysis of the average message sizes in *SeisSol* during strong scaling of the LOH.1 scenario with a relatively good distribution of elements between time-clusters. As can be seen, the median message size in *SeisSol* (16 KiB - 1 MiB) falls into the medium message size range, determined above, while partitioning the problem from 8 to 512 sub-domains. Considering that a typical elements distribution is worse than the one given in this experiment, one may expect to observe a poor performance of *SeisSol* under a strong scaling scenario (see Fig. 6.22 as an example - i.e., the solid orange and pink lines).

The abovementioned problem can be alleviated using an algorithm similar to the one proposed in [45]. As shown in Fig. 6.21 - i.e., b) and c), pre-allocated auxiliary buffers on the device or host sides can be used to copy data from *SeisSol*'s MPI buffers before and after each *MPI_ISend* and *MPI_IRecv*, respectively. In [45], the authors could only assume the average message size and, thus, had to fragment messages. The implementation proposed by Hamidouche et al. copied each fragment to the auxiliary buffers and transferred them one by one using software pipelining. Because of the static mesh refinement, the sizes of all communication regions in *SeisSol* can be calculated after the mesh partitioning step. Therefore, I allocate all auxiliary MPI buffers for all time-clusters in advance. This can help to avoid extra latency related to the software pipelining while transferring small and medium messages. Moreover, data copies from the MPI buffers, allocated with unified memory, to auxiliary memory can be done asynchronously and performed as a part of non-blocking communication in *SeisSol*.

In the following, I explain the details of my implementations using the pseudo codes shown in Listing 4 and Listing 5. The receiving process begins with a call to *receiveGhostLayer* method, where the algorithm traverses all communication regions and posts *MPI_IRecv* one by one, directing upcoming messages to the dedicated auxiliary MPI buffers. The method returns a receiving queue, which is initialized to zero at the beginning and gradually filled during the traversal (see lines 6-10). As can be seen, each element of the queue is a compound data structure dedicated to each communication region and contains 1) a pointer to the auxiliary buffer, 2) the associated device stream, 3) the sender process identifier, 4) the communication region identifier and 5) the *MPI_Request* handle returned

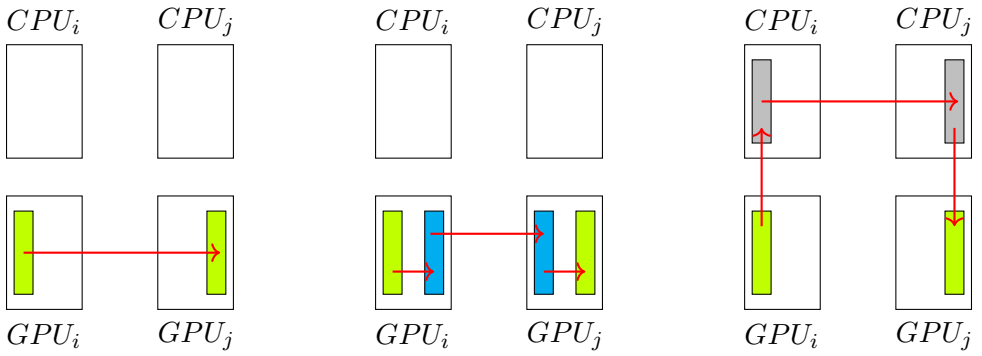


(a) Distribution of 20 million elements used for the LOH.1 test scenario between 5 time-clusters.



(b) Median, maximal, and minimal MPI message sizes of all time-clusters resulted from partitioning the test mesh from 8 to 512 sub-domains. The data were obtained for convergence order 6 using the single-precision floating-point format.

Figure 6.20.: Statistics of the MPI message sizes during strong scaling of the LOH.1 scenario in *SeisSol*.



(a) Direct. (b) Device. (c) Host.

Figure 6.21.: Point-to-point message-passing schemes between GPUs in *SeisSol*. The green color denotes MPI buffers allocated in unified memory; the blue one - regular device memory; the gray color - host memory.

from the corresponding *MPI_IRecv* operation. Each element of a receiving queue can be in one of two possible states: 1) *RequiresMPI_Test* and 2) *RequiresAsyncCopy_Test*. By default, all elements are initialized with the *RequiresMPI_Test* state, indicating that the data transfer for the associated communication region has not been completed yet. While iterating over *interior* LTS layers, *SeisSol* periodically tests receiving queues of each *ghost* layer, thus checking the completeness of the initiated asynchronous communications. The test of a queue is implemented as a for-loop (see lines 24-42), where the state of each region is checked, and the corresponding action is taken. If a region is in the *RequiresMPI_Test* state, the *MPI_Test* function is called with the associated *MPI_Request* handle. If the MPI test succeeds, the algorithm initiates an asynchronous data copy from the auxiliary to the user MPI buffer using the attached device stream (see lines 15-21). Afterward, the test region is immediately switched to the *RequiresAsyncCopy_Test* state (see line 31), and the for-loop iterator shifts. If a region is in the *RequiresAsyncCopy_Test* state, the algorithm checks whether the initiated asynchronous copy operation has been completed using the associated device stream as the handle (see line 36). If the test succeeds, the region gets removed from the queue. The *testReceiveQueue* function succeeds when a passed receiving queue gets empty, indicating that the current process has fully received all pieces of a *ghost* layer from its neighbors. It is worth noting that the proposed implementation can completely hide overheads stemming from the communication and asynchronous data copies on the receiving side if the *interior* LTS layers provide enough work.

The sending process starts with a call to the *sendCopyLayer* method, which immediately calls the *prefetchCopyLayer* function. The function asynchronously copies the user data of all communication regions to the dedicated auxiliary buffers using associated device streams (see lines 4-12). While iterating over the regions, the function creates and fills the compound data structure (see the description above) for each communication region and pushes it to the prefetching queue (see lines 10-11). The queue is initialized to zero at the beginning of the function execution and gets returned at the end, as shown in lines 2 and 13, respectively. Once the *sendCopyLayer* method receives a prefetching queue, it starts repeatedly checking each region, using polling (see lines 19-20), regarding the completeness of the associated copy operation using the attached device stream as the handle (see line 22). If a test succeeds, the method immediately sends the associated auxiliary buffer to the destination using the non-blocking *MPI_Isend* method. The returned *MPI_Request* handle gets assigned to the corresponding field of the compound data structure of the region. After that, the method removes the region from the prefetching queue and immediately adds it to the sending one, initialized to zero at the beginning of the method execution. The process repeats till the entire prefetching queue gets empty, and the sending queue becomes full. When the method finishes, it returns the sending queue to a callee. Afterward, the communication thread of a process can start testing all elements of the queue for their completeness using the *MPI_Test* function. As discussed in Section 4.4, this will force the underlying MPI library to progress the issued non-blocking communications.

The key difference between the sending and receiving sides is that, in the former, the asynchronous data copies cannot be overlapped with some computations. The polling methods, shown in lines 19-20, will force the *sendCopyLayer* method to initiate all non-blocking send operations for an entire *copy* layer, and, thus, the layer must be copied to

Algorithm 4 Receiving Copy-Layer

```
1: procedure receiveGhostLayer(GhostTimeCluster)
2:     ReceiveQueue ←∅
3:     NumRegions ←GhostTimeCluster.getNumRegions()
4:     for i from 1 to NumRegions do
5:         AuxiliaryBuffer ←GhostTimeCluster.getAuxiliaryBuffer(i)
6:         Stream ←GhostTimeCluster.getStream(i)
7:         Source ←GhostTimeCluster.getNeighborRank(i)
8:         Region ←newRegion(AuxiliaryBuffer, Stream, Source, i)
9:         Region.Request ←MPI_Irecv(AuxiliaryBuffer, Source)
10:        ReceiveQueue.append(Region)
11:    end for
12:    return ReceiveQueue
13: end procedure
14:
15: procedure prefetchGhostRegion(Device, Region, GhostTimeCluster)
16:    UserBuffer ←GhostTimeCluster.getUserBuffer(Region.i)
17:    AuxiliaryBuffer ←Region.AuxiliaryBuffer
18:    Stream ←Region.Stream
19:    Device.copyAsync(AuxiliaryBuffer, UserBuffer, Stream)
20: end procedure
21:
22: procedure testReceiveQueue(Device, ReceiveQueue, GhostTimeCluster)
23:    for each Region in ReceiveQueue do
24:        State ←Region.getCurrentState()
25:        switch State do
26:            case RequiresMPI_Test :
27:                TestSuccess ←MPI_Test(Region.Request)
28:                if TestSuccess then
29:                    prefetchGhostRegion(Device, Region, GhostTimeCluster);
30:                    Region.setState(RequiresPrefetchTest)
31:                end if
32:                break
33:            end
34:            case RequiresPrefetchTest :
35:                if Device.isWorkDone(Region.Stream) then
36:                    ReceiveQueue.remove(Region)
37:                end if
38:                break
39:            end
40:        done
41:    done
42:    return ReceiveQueue.empty()
43: end procedure
```

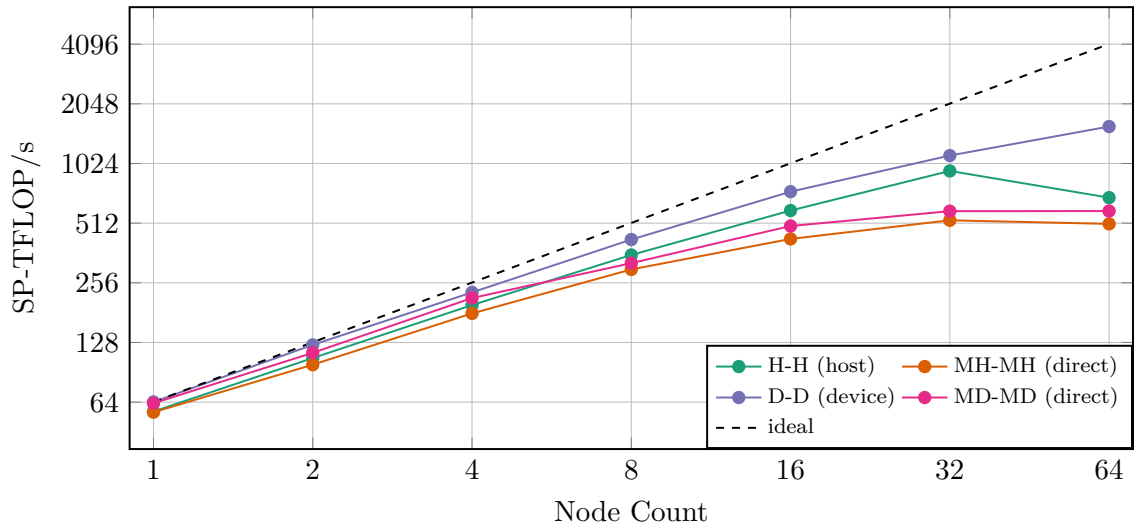


Figure 6.22.: Comparison of the strong scaling performance of the LOH.1 benchmark using different message-passing configurations on the Selene supercomputer using the mesh shown in Fig. 6.20.

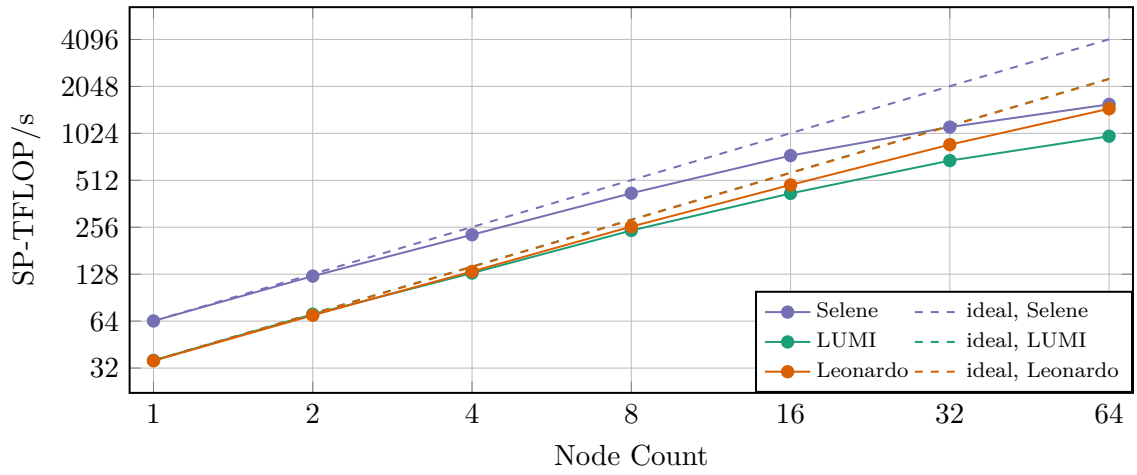
the auxiliary buffers beforehand. However, due to the imposed concurrency, data copies of large messages may be overlapped with instantiations of non-blocking sends of small and medium ones. It is worth mentioning that a GPU-Aware MPI implementation would only be able to start performing the copy to/from managed to non-managed device memory when it begins the sending/receiving message procedure. However, this custom solution allows the copies to happen as soon as possible and gives them longer time to complete.

Fig. 6.22 depicts the strong scaling results obtained on the Selene supercomputer using the mesh shown in Fig. 6.20. The MD-MD and MH-MH configurations were obtained using the original message exchange algorithm - i.e., direct (see Fig. 6.21) - and the MPI buffers, allocated with unified memory, with fragments staging of messages on the host and device, respectively. In both cases, parallel efficiencies are very low - i.e., 14% on average. One can observe that the performance growth stops after 32 nodes. It is worth noting that, on average, staging MPI buffers on the device results in about 13.5% higher performance than staging on the host. A plausible reason for this is the GPUDirect Peer-to-Peer and RDMA technologies used on all Selene nodes. The H-H configuration demonstrates good scaling up to 32 nodes, on which the parallel efficiency reaches almost 51%. After that, the performance suddenly drops by approximately 40%. A detailed profiling of *SeisSol* is required to determine the exact reason for this behavior, which was difficult to perform at the moment of writing because of a need to directly access *Nvidia*'s proprietary supercomputer - i.e., Selene. The best results were obtained with the D-D configuration, which resulted in almost 38% of parallel efficiency on 64 nodes (512 A100 GPUs) relative to the baseline (8 A100 GPUs), reaching approximately 1.58 SP-PFLOP/s.

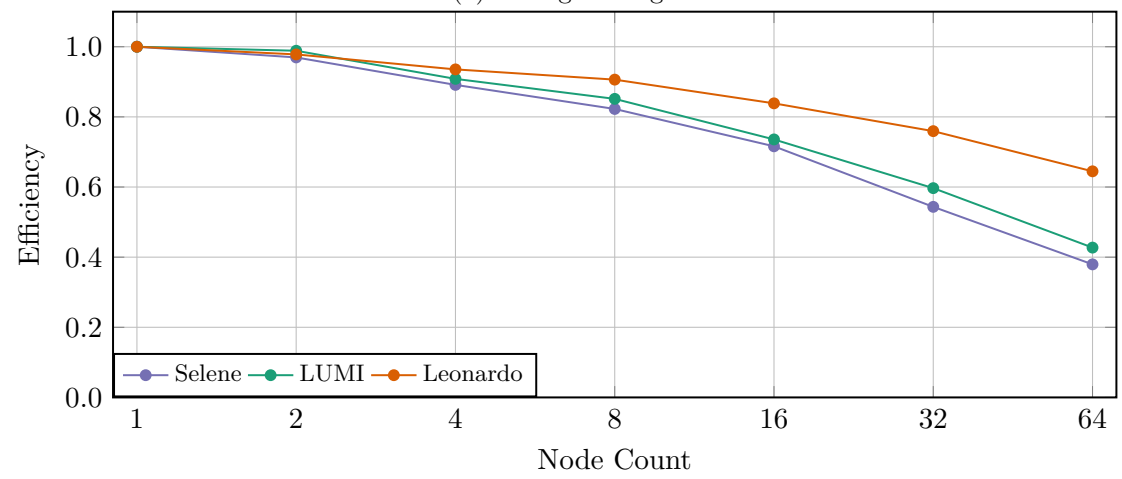
Fig. 6.23 compares the strong scaling performance of *SeisSol* on Selene, Leonardo and LUMI supercomputers using the D-D message-passing configuration. Selene has eight

Algorithm 5 Sending Copy-Layer

```
1: procedure prefetchCopyLayer(Device, CopyTimeCluster)
2:     PrefetchQueue ←∅
3:     NumRegions ←CopyTimeCluster.getNumRegions()
4:     for i from 1 to NumRegions do
5:         AuxiliaryBuffer ←CopyTimeCluster.getAuxiliaryBuffer(i)
6:         UserBuffer ←CopyTimeCluster.getUserBuffer(i)
7:         Stream ←CopyTimeCluster.getStream(i)
8:         Device.copyAsync(UserBuffer, AuxiliaryBuffer, Stream)
9:         Destination ←CopyTimeCluster.getNeighborRank(i)
10:        Region ←newRegion(AuxiliaryBuffer, Stream, Destination, i)
11:        PrefetchQueue.append(Region)
12:    end for
13:    return PrefetchQueue
14: end procedure
15:
16: procedure sendCopyLayer(Device, CopyTimeCluster)
17:    PrefetchQueue ←prefetchCopyLayer(Device, CopyTimeCluster)
18:    SendQueue ←∅
19:    while not PrefetchQueue.empty() do
20:        for each Region in PrefetchQueue do
21:            Stream ←Region.Stream
22:            if Device.isWorkDone(Region.Stream) then
23:                AuxiliaryBuffer ←Region.AuxiliaryBuffer
24:                Destination ←Region.Destination
25:                Region.Request ←MPI_Isend(AuxiliaryBuffer, Destination)
26:                SendQueue.append(Region)
27:                PrefetchQueue.remove(Region)
28:            end if
29:        done
30:    end while
31:    return SendQueue
32: end procedure
33:
34: procedure testSendQueue(SendQueue)
35:    for each Region in SendQueue do
36:        TestSuccess ←MPI_Test(Region.Request)
37:        if TestSuccess then
38:            SendQueue.remove(Region)
39:        end if
40:    done
41:    return SendQueue.empty()
42: end procedure
```



(a) Strong Scaling.



(b) Parallel efficiency.

Figure 6.23.: Comparison of the strong scaling performance of *SeisSol* on the Selene, Leonardo and LUMI supercomputers using the LOH.1 benchmark with the mesh shown in Fig. 6.20 and the D-D message-passing configuration.

Nvidia A100 GPUs per node, whereas each Leonardo and LUMI node is equipped with only four *Nvidia* A100 and *AMD* MI250x cards, respectively. However, each MI250x GPU has two Graphics Complex Dies (GCD), which are considered as two single graphics cards in the SPSG model (see the beginning of Section 6.5); thus, the number of MPI ranks and message sizes between sub-domains remain the same during scaling on Selene and LUMI. Differences in network characteristics between the used supercomputers are shown in Fig. 6.24. It is important to mention that, on LUMI, the latency and bandwidth of intra-node communication were measured between graphics dies located in two different GPUs.

One can observe that the intra-node network characteristics of the Leonardo supercomputer are in between the ones obtained on LUMI and Selene for the large and medium message sizes. At the same time, Leonardo showed the worst results regarding the latency and bandwidth for all tested message sizes in the case of inter-node communication.

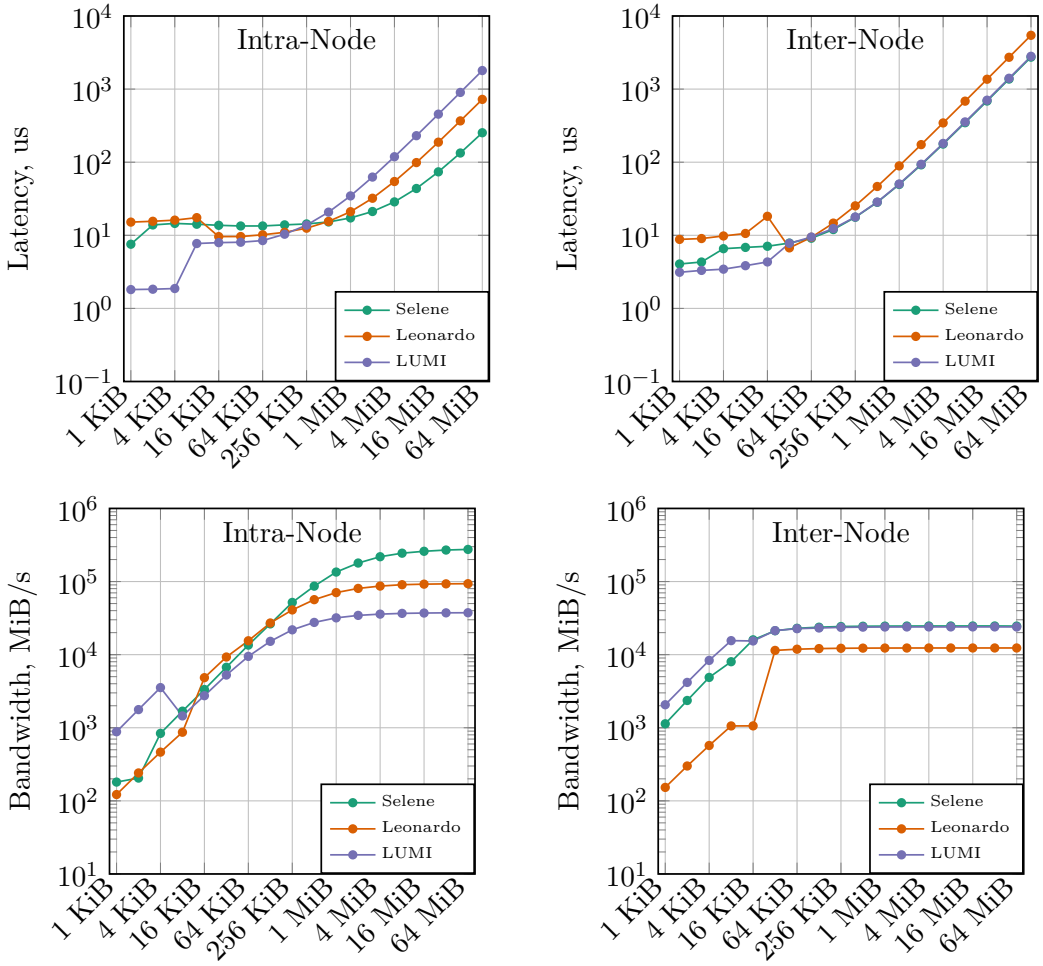


Figure 6.24.: Comparison of the latency and unidirectional bandwidth on the Selene, Leonardo and LUMI supercomputers using D-D message-passing configuration.

Nevertheless, as can be seen from Fig. 6.23, Leonardo demonstrated the best strong scaling results for the LOH.1 test scenario, reaching almost 65% of the parallel efficiency on 64 nodes, while the efficiency obtained on the Selene and LUMI supercomputers was only around 40% at this scale. One can argue that better strong scaling results on Leonardo were achieved due to its setup, which required two times fewer MPI processes and, thus, resulted in fewer messages to exchange. However, despite having worse network characteristics, Leonardo's efficiency on 64 nodes was still almost 5% higher than the one obtained on LUMI with 32 nodes.

The obtained results indicate that, apart from just the networking properties, there exist other factors that limit the strong scaling performance of the ADER-DG method bundled with the LTS scheme. In [96, 98, 97], Rietmann investigated strong scaling properties of the Spectral Element Method applied to the wave propagation problem in an elastic medium. The author observed that the parallel efficiency of his LTS implementation

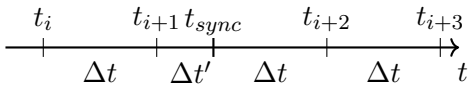


Figure 6.25.: Time integration steps with a synchronization point in between.

drastically dropped during strong scaling on the distributed multi-GPU system used in the experiments (i.e., the Piz Diant supercomputer). Rietmann suggests that it happened due to a small number of elements in the finest mesh refinement level, which could not keep the GPUs adequately busy to mask the overhead of setting up and launching GPU tasks [97]. Rietmann's statement is too general and needs some supporting experiments. In the following two sections, I aim to investigate Rietmann's hypothesis. In the next section, I examine the influence of GPU launching overheads on strong scaling performance. For this purpose, I replaced the traditional kernel launching mechanism with the advanced one - i.e., CUDA Graphs. In Section 6.5.3, I investigate how and to what degree strong scaling efficiency depends on the computational throughput of a device and LTS clustering.

6.5.2. Graph-Based Task Execution

The LTS scheme can significantly shorten some parallel regions, especially under a strong scaling scenario. Offloading such small tasks to GPUs can expose significant kernel launching overheads (see Section 5.3). Concurrent data processing on multiple GPU streams (see Section 6.4) can partially solve this problem, but it is only suitable for processing data-independent tasks. The tasks defined by I_{ader}, I_{vol} and I_{src} macro-kernels (see Section 4.1) are data-dependent and constitute about 65% of the total execution time of the wave propagation solver.

The graph-based execution model aims to reduce overheads associated with launching a sequence of operations on a device. In this model, device kernels are the nodes of a graph, whereas edges encode dependencies between them. A graph can be built explicitly using the corresponding API functions and data structures, or it can be captured by the device driver. The latter is preferable while working with a generated code. Otherwise, all necessary graph-building steps needs to be added to the code generation logic.

While capturing, the driver switches to the mode where it only records kernels with its arguments without executing them. One can use stream-based programming to express complex dependencies between nodes in a graph. Once captured, a graph, which may contain dozens or hundreds of device kernels, is copied to the device, and the associated graph handle is returned to a user. The handle allows the user to launch the entire captured sequence within a single interaction with the device driver. If some kernel argument needs to be changed at run-time, the corresponding graph node must be explicitly rebuilt. If it is not possible, the entire graph must be captured again.

In *SeisSol*, the arguments of most kernels launched inside each time-cluster do not change during a program's execution because of the static mesh refinement and the recording

Algorithm 6 Graph-based implementation of the ADER scheme without source terms and when $t = t_0$ - i.e., I_{ader}.

1: **procedure** ComputeAder($Device$, $LtsLayer$, $PreComputedData$, Δt)
2: $\hat{K}^1_{lt}$, $\hat{K}^2_{lt}$, $\hat{K}^3_{lt} \leftarrow PreComputedData.getStiffnessMatrices()$
3: $A^{1:M}_{qp}$, $B^{1:M}_{qp}$, $C^{1:M}_{qp} \leftarrow PreComputedData.getJacobianMatrices()$
4: $\mathcal{T}^{1:M}_{lp} \leftarrow LtsLayer.getIntegratedDOFs()$
5: $Q^{1:M}_{lp} \leftarrow LtsLayer.getDOFs()$
6: $key \leftarrow GraphKey(KernelType :: ADER, \Delta t)$
7: **if not** $LtsLayer.graphHandleExists(key)$ **then**
8: $Device.beginGraphCapturing()$
9: $LtsLayer.setFirstDerivatives(Q^{1:M}_{lp})$
10: $\mathcal{T}^{1:M}_{lp} \leftarrow \Delta t \cdot Q^{1:M}_{lp}$
11: **for** i **from** 2 **to** $\mathcal{O}$ **do**
12: $\mathcal{D}^{1:M,i-1}_{tq} \leftarrow LtsLayer.getDerivatives(i-1)$
13: $\mathcal{D}^{1:M,i}_{lp} \leftarrow LtsLayer.getDerivatives(i)$
14: $\mathcal{D}^{1:M,i}_{lp} = -\hat{K}^1_{lt}\mathcal{D}^{1:M,i-1}_{tq}A^{1:M}_{qp} - \hat{K}^2_{lt}\mathcal{D}^{1:M,i-1}_{tq}B^{1:M}_{qp} - \hat{K}^3_{lt}\mathcal{D}^{1:M,i-1}_{tq}C^{1:M}_{qp}$
15: $scalar \leftarrow (\Delta t)^i \ / \ i!$
16: $\mathcal{T}^{1:M}_{lp} \leftarrow \mathcal{T}^{1:M}_{lp} + scalar \cdot \mathcal{D}^{1:M,i}_{lp}$
17: **end for**
18: $Device.endGraphCapturing()$
19: $handle \leftarrow Device.getGraphHandle()$
20: $LtsLayer.setGraphHandle(key, handle)$
21: **end if**
22: $handle \leftarrow LtsLayer.getGraphHandle(key)$
23: $Device.launchGraph(handle)$
24: $Device.wait()$
25: **end procedure**

mechanism described in Section 6.2. However, the presence of synchronization points, such as for saving intermediate results to disks, may force a local change in the time step width Δt (see Fig. 6.25), which can be considered as a scaling factor within a tensor expression. Instead of manually updating the corresponding graph nodes, a new graph is recorded and launched. To further reduce the overhead, my implementation stores all graphs that belong to a time-cluster in a hash table, where the key is a pair of a macro-kernel name (encoded with an integer value) and Δt. The concept is illustrated in Algorithm 6. The algorithm checks whether a graph is in the table when entering a parallel region (line 6). If not, it reruns the graph-capturing procedure (lines 8-18), adds the resulting graph to the table (lines 19-20) and immediately launches it (lines 22-24).

Fig. 6.26 depicts the performance of *SeisSol*-proxy obtained on *Nvidia* A100 GPU using the graph-based and stream-based (the baseline) execution models. As expected, the former considerably increases the average GPU performance (by approximately 50%) under low workloads - i.e., within the [512, 8192] range. However, starting from the cluster size equal to 16384, the performance gain becomes indistinguishable relative to

the baseline; the performance increased by only 0.5% within the [16384, 524288] range on average.

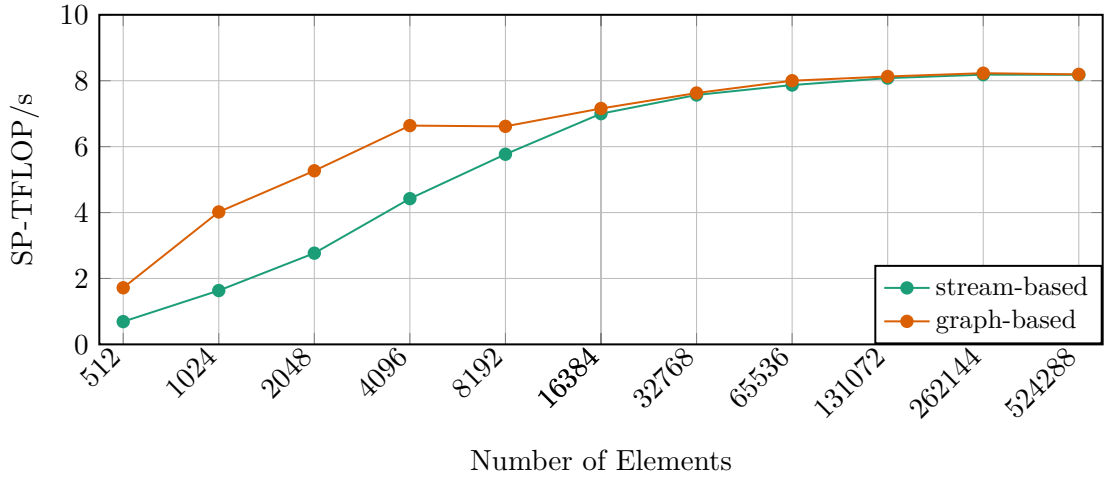


Figure 6.26.: Comparisons of the graph-based and stream-based execution models applied to *SeisSol*-proxy on *Nvidia* A100 GPU.

In general, the graph-based execution model may benefit many applications in the context of strong scaling because, in this case, all parallel regions become smaller at each scaling step. In *SeisSol*, the model would additionally increase hardware performance because of LTS, which entails the presence of small time-clusters (see Fig. 4.4).

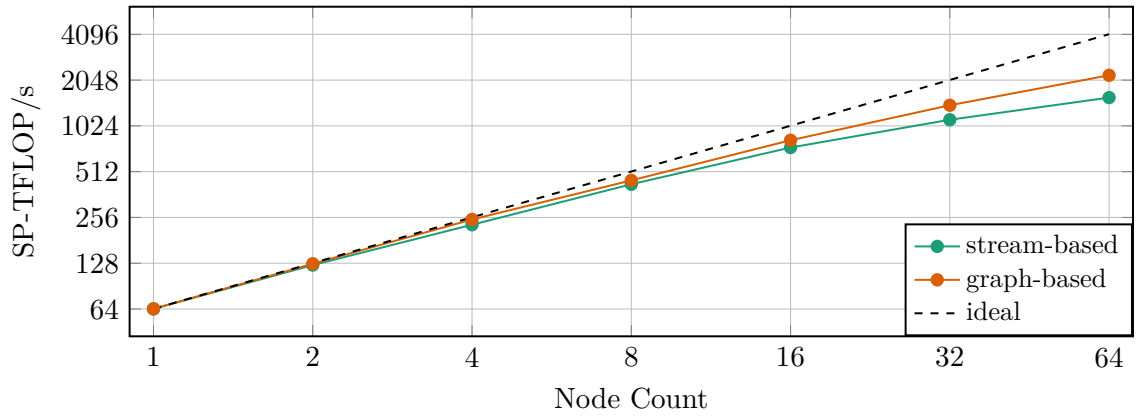


Figure 6.27.: Comparisons of the graph-based and stream-based execution models during strong scaling of the LOH.1 benchmark on the Selene supercomputer. The mesh shown in Fig. 6.20 was used during the experiment.

Fig. 6.27 compares the strong scaling performance of the stream- and graph-based execution models applied to *SeisSol*'s wave propagation solver. The D-D message-passing configuration, discussed in Section 6.5.1, was used in the experiment as the best-performing one on the Selene supercomputer. The experiment was conducted using the same mesh

as in Section 6.5.1 to demonstrate the accumulative improvement of the strong scaling performance of *SeisSol* on distributed multi-GPU systems. The obtained results match the behavior observed in Fig. 6.26. No considerable differences in performance between the two models can be observed during the first phase of the experiment - i.e., scaling from 1 to 8 nodes - because time-clusters in each sub-domain have enough work to hide kernel launching overheads. However, the two lines begin to deviate when the number of nodes reaches 16. The most noticeable improvement can be observed at the last scaling interaction - i.e., on 64 Selene nodes (512 A100 GPUs). The difference in performance between the graph- and stream-based executions reaches approximately 40%. The parallel efficiency of the former achieves about 53% at the end of the experiment.

At the moment of writing, the latest *AMD* ROCm software stack (i.e., version 5.4) provided limited support for the graph-based execution model. Therefore, performing a similar analysis was impossible on *AMD* GPUs.

6.5.3. Influence of LTS clustering on Strong Scaling

A cluster-wise LTS scheme splits mesh elements into sub-sets and updates each with its optimal time integration step width. This approach may drastically reduce the time-to-solution of a simulation by reducing redundant computations. However, as mentioned in Section 3.5, this LTS scheme shrinks parallel regions. Some clusters may fall into a low computational throughput region of a processor, affecting the overall performance. The problem may intensify during strong scaling when more and more clusters fall to the low throughput region due to mesh partitioning. As shown in Fig. 6.10, computational characteristics of CPUs and GPUs significantly differ under *SeisSol*-specific workloads. According to the experiment, the computational throughput of CPUs is almost independent of the problem size compared to GPUs, which experience a prominent performance drop under low workloads. In this section, I investigate the influence of LTS clustering on strong scaling performance.

Eq. 6.11 gives the average performance $\bar{P}$ of a processor for computing N in-order tasks.

$$\bar{P} = \frac{\sum_{i=1}^{N} W_i}{\sum_{i=1}^{N} \tau_i} = \frac{\sum_{i=1}^{N} \tau_i \cdot P_i}{\sum_{i=1}^{N} \tau_i} \tag{6.11}$$

where W_i - work of the i-th task; τ_i - time spent on computing the i-th task; P_i is the computational throughput of a processor while operating on the i-th task;

The specifics of LTS tasking must be considered when applying Eq. 6.11. In the following, I refer to any time instance when the last time-cluster gets updated as a synchronization point because, at this moment, all time-clusters are guaranteed to be in sync (see Fig. 4.2). The total time $\bar{\tau}_l$ spent on updating the l-th time-cluster between two synchronization points can be computed as

$$\bar{\tau}_l = r^{L-l-1} \cdot \tau_l \tag{6.12}$$

where τ_l is the time spent on a single update of the l-th time-cluster; r is the given update ratio for the LTS scheme (see Section 4.4); and L is the total number of time-clusters.

The average performance of the LTS scheme can be obtained by applying Fig. 6.12 to Fig. 6.11, which yields

$$\bar{P} = \frac{\sum_{l=0}^{L-1} \left(r^{L-l-1} \cdot \tau_l \cdot P_l \right)}{\sum_{l=0}^{L-1} \left(r^{L-l-1} \cdot \tau_l \right)} = \sum_{l=0}^{L-1} \omega_l \cdot P_l \tag{6.13}$$

where P_l is the computational throughput of a processor resulting from processing the l-th time-cluster; and ω_l is the performance weight of the l-th cluster, which is given by

$$\omega_l = \frac{r^{L-l-1} \cdot \tau_l}{\sum_{l=0}^{L-1} r^{L-l-1} \cdot \tau_l} \tag{6.14}$$

Fig. 6.28 depicts an evolution of performance weights during strong scaling, computed according to Eq. 6.14. The results were obtained by taking the LTS clustering from Fig. 6.20, and the computational throughput and timing data obtained using *SeisSol*-proxy on *AMD* MI250x GPU (see Fig. 6.10). The data between measured points were linearly interpolated. The partitioning is modeled by dividing each original time-cluster size by a scaling factor, mimicking equal partitioning of the time-clusters between processors.

As can be seen, the performance of the last two clusters - i.e., clusters 3 and 4 - does not noticeably impact the overall performance of the LTS scheme because of their less frequent update rates and, as a result, low weights. The second cluster's weight contributes the most to the performance of LTS. However, once the scale factor exceeds a value of 32, it begins to drop drastically while the weight of the first cluster starts increasing. At this moment, all scaled time-clusters get shifted to the low throughput region. When the scaling factor gets equaled to 512, the performance of the first cluster, which becomes very low (i.e., approximately 0.4 SP-TFLOP/s), constitutes almost 20% of the overall performance because of its significantly increased weight.

A combination of the obtained data (shown in Fig. 6.28) and Eq. 6.13 can be used to estimate the Ideal Parallel Efficiency (IPE) of the LTS scheme during strong scaling for a given clustering and characteristics of a processing unit. In Fig. 6.29, I compare it with the Measured Parallel Efficiency (MPE) obtained during scaling the LOH.1 scenario on MI250x GPUs (on LUMI). The scale factor 8 was chosen as the baseline for IPE to match a single LUMI node which has 8 GPU dies and, thus, requires 8 MPI processes per node. The IPE excludes the presence of a single kinematic point source, load imbalance, communication, differences in partitioning, etc., in contrast to the real scenario; thus, it should be considered the upper bound for the strong scaling LTS performance.

According to Fig. 6.29, even the IPE reaches only 62.5% on 64 nodes (scaling factor 512) under the given conditions. The difference between IPE and MPE is approximately 19.5% at this scaling level, or the same may be interpreted as if *SeisSol*'s parallel efficiency is about 68.3% relative to the upper bound.

I continue the study using a set of wave propagation scenarios with different artificially enforced LTS configurations using the LOH.1 setup as the basis. A single computational

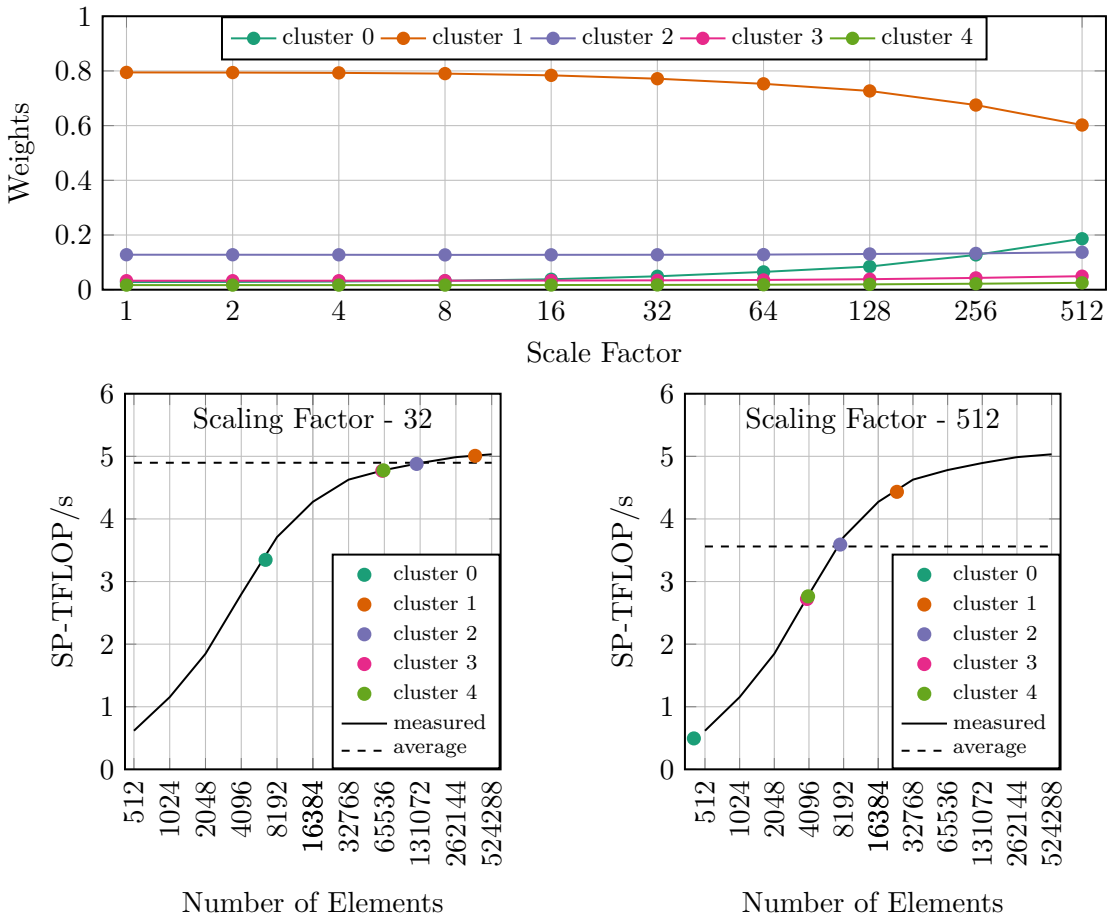


Figure 6.28.: Evolution of performance-weights of LTS time-clusters during strong scaling.

mesh, consisting of 20 million elements, was used for all tests. The mesh was generated in two steps. Firstly, an equidistant cartesian mesh was generated inside a cuboid domain with the longest dimension along the z-axis. Secondly, each cubic element was split into five almost equal tetrahedrons. An example of such a mesh is shown in Fig. 6.30. A particular LTS clustering was enforced through the material parameters - i.e., λ, μ, and ρ (see Section 2.1) - which affected the primary propagation velocities. That allowed me to control the CFL condition inside each mesh element and, thus, helped to build 6 different LTS configurations, depicted in Fig. 6.31. The clusterings are sorted in ascending order relative to the amount of work required to simulate the wave propagation processes for 1 second in *SeisSol* using the convergence order equal to 6.

As can be seen, the LTS Type 1 configuration results in the least work because the largest cluster - i.e., cluster 4 - which constitutes the bulk of the domain, has the lowest time update rate. The first cluster is the smallest one, containing only 12500 mesh elements, and is updated the most frequently. Therefore, this configuration should be the best regarding time-to-solution. However, based on the outcome of the previous discussion, one may expect that it will be the worst regarding strong scaling performance. The LTS Type 6 is completely the opposite. The bulk of the domain is concentrated in the first time-cluster.

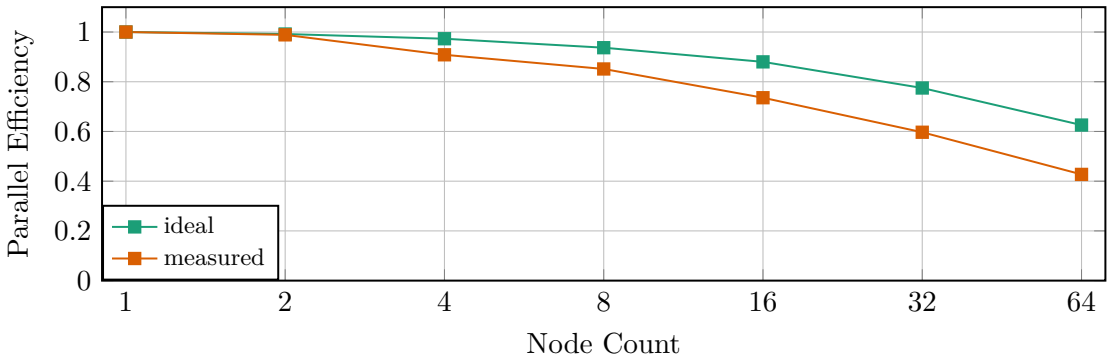


Figure 6.29.: Ideal and measured parallel efficiency during strong scaling of the LOH.1 scenario on *AMD* MI250x GPU.

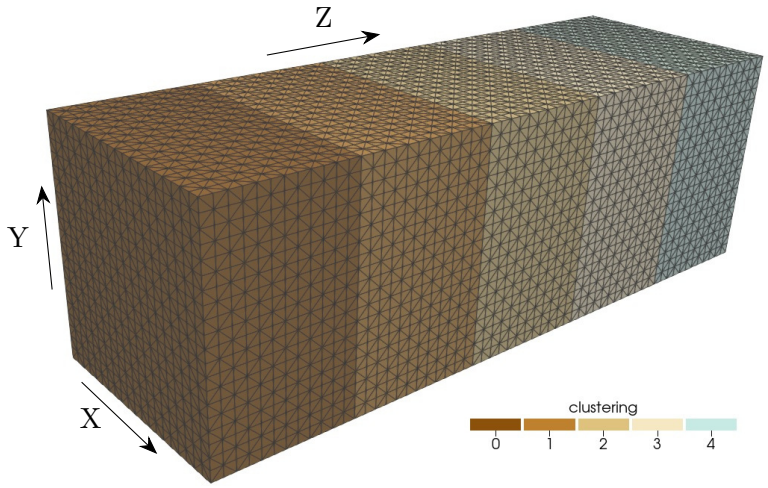
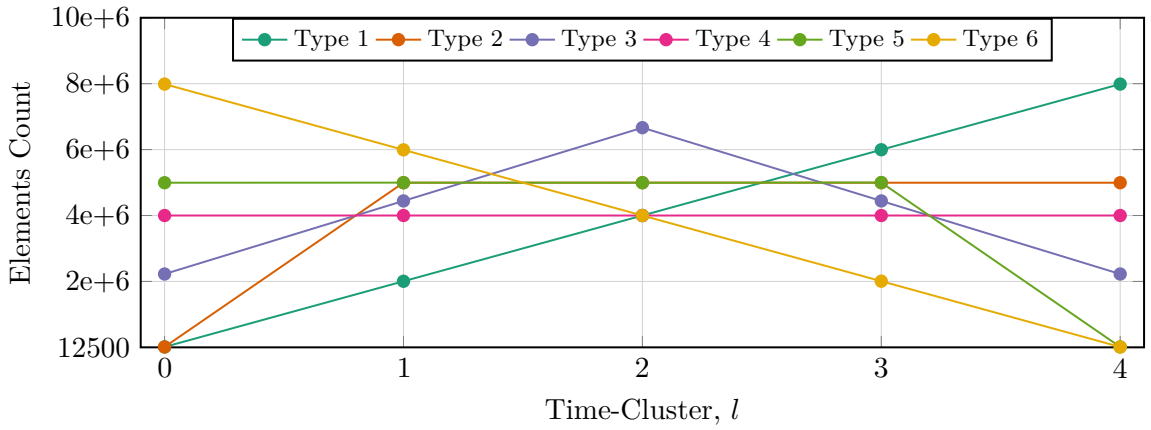
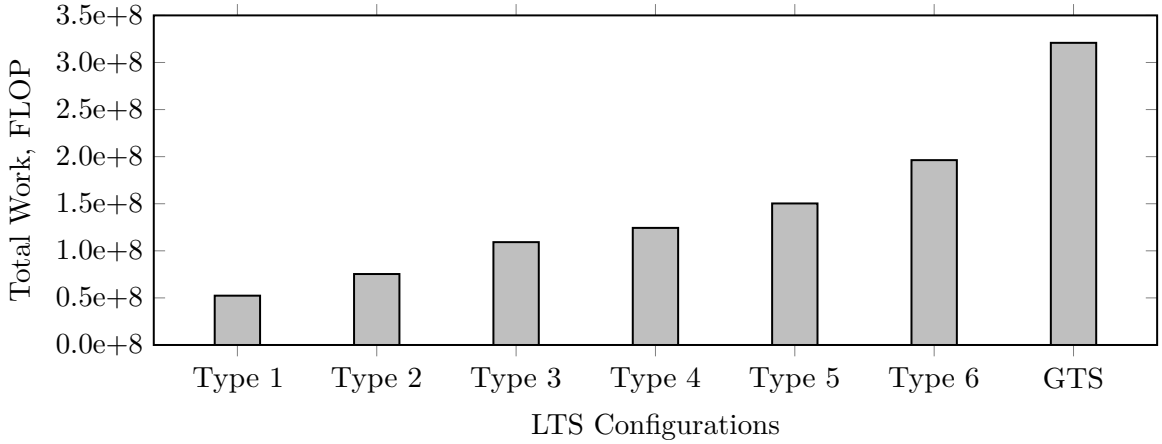


Figure 6.30.: LOH.1 geometry with parametrized LTS clustering.

Sizes of subsequent clusters get gradually reduced while moving toward the right side. Despite being the largest in terms of the required work, one may expect this scenario to scale well because its first cluster will stay longer in the high computational throughput region within the entire scaling range. Moreover, in this case, the low performance of other individual clusters will contribute little to the resulting performance because of their relatively small weights. The LTS Type 2 models a scenario when the first time-cluster contains a tiny subset of elements while the rest are distributed equally among other clusters. Type 4 is opposite to the second LTS configuration and was added just for the symmetry. The LTS Type 3 models a scenario when the most significant part of the domain is concentrated in the middle cluster and then gradually reduces towards both ends. The last proposed scenario - i.e., Type 4 - equally distributes mesh elements among all time-clusters. The results of scaling the GTS scheme applied to the Type 1 scenario are added to the plots for comparison. As can be seen from Fig. 6.30, the GTS scheme results in 6.2 times more work than the one produced by the cluster-wise LTS scheme.



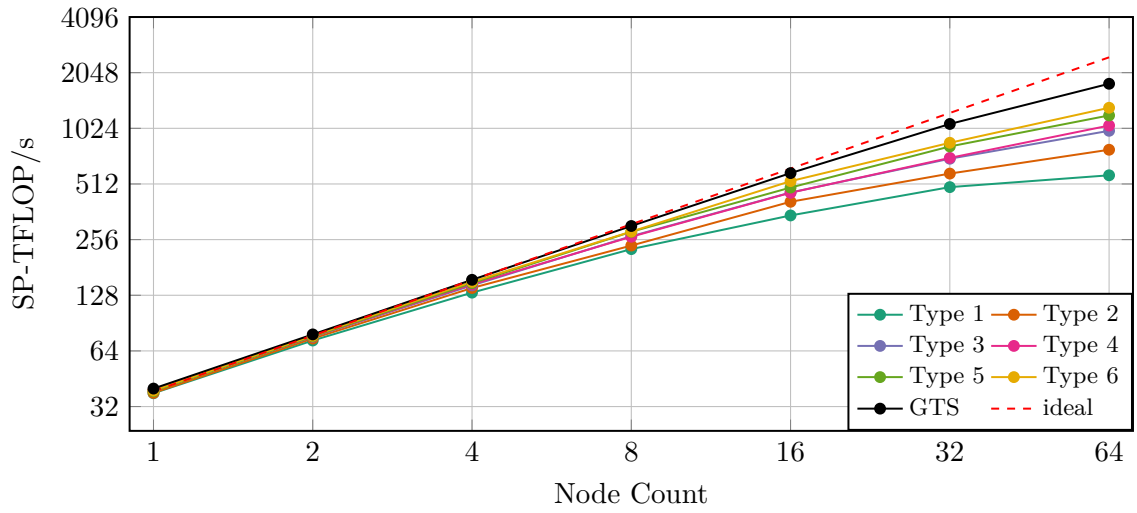
(a) A test set of different elements distributions among LTS time-clusters.



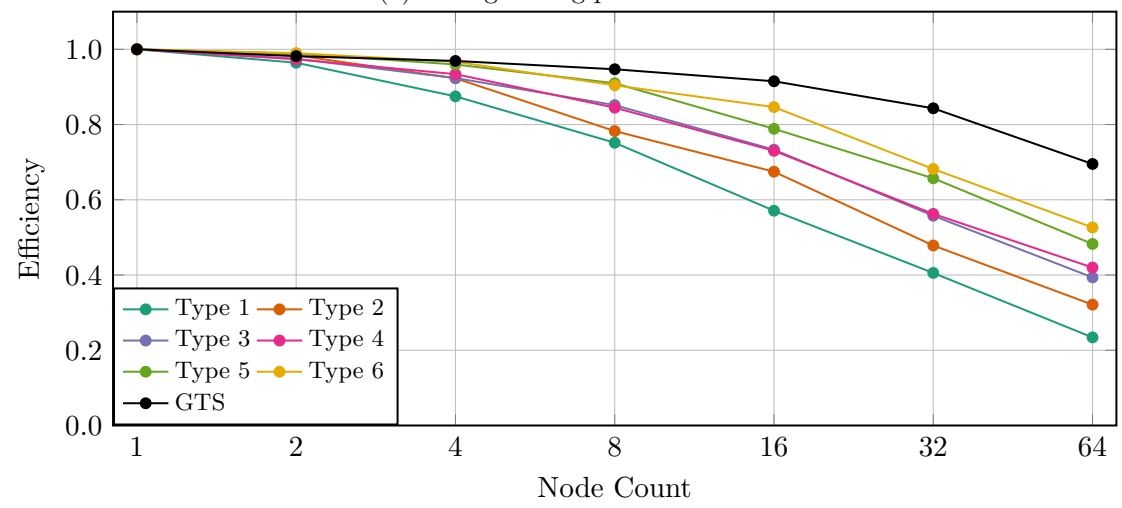
(b) Total works required for performing a 1-second simulation in *SeisSol* for each LTS type.

Figure 6.31.: Tested LTS configurations.

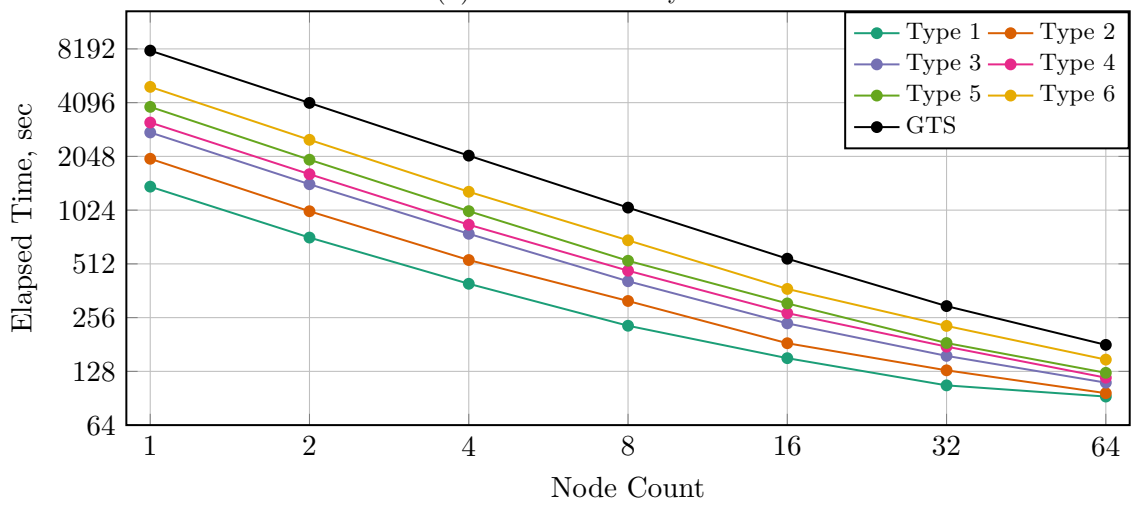
Fig. 6.32 aggregates the results obtained while scaling the abovementioned scenarios on the LUMI supercomputer. As expected, the simulation subjected to the GTS scheme demonstrates the best scalability and the worst run-time performance. On 64 nodes, each MPI process contains approximately 39062 elements in this single cluster configuration, which results in about 92% of the maximal single GPU throughput obtained with *SeisSol*-proxy. The GTS parallel efficiency is close to 70% at this scale, which results in almost 1.8 SP-PFLOP/s of the aggregated GPU performance. As predicted, the best-performing LTS configuration belongs to the Type 6, which reaches close to 1.3 SP-PFLOP/s on 64 LUMI nodes due to relatively good parallel efficiency - i.e., 53%. The LTS Type 1 configuration demonstrated the worst strong scaling performance, reaching only ≈ 0.6 SP-PFLOP/s at the maximum scale. In this case, the parallel efficiency achieves only $\approx$ 23.5%. Nevertheless, this scenario is still 2 times faster compared to the execution of the same setup using GTS. Despite very low parallel efficiency, LTS significantly outperforms the GTS scheme regarding time-to-solution because of its high algorithmic speedup. However, this may not hold for other LTS configurations at a very large scale - e.g., Type 6. However, configurations like Type 6 are rare for many real earthquake scenarios.



(a) Strong scaling performance.



(b) Parallel efficiency.



(c) Time-to-solution.

Figure 6.32.: Strong scaling of different LTS clustering configurations on the LUMI supercomputer.

The reader can observe two important correlations from the obtained results shown in Fig. 6.31 and Fig. 6.32. The first and the most obvious one is that time-to-solution positively correlates to the imposed work. According to the experiments, this statement holds even at a large scale when the parallel efficiency of some LTS configurations drastically drops. The second one relates to the size of the most frequently updated time-cluster, which positively correlates to the resultant strong scaling efficiency.

The outcome of this study suggests two possible approaches to improve *SeisSol*'s strong scaling performance on distributed multi-GPU systems for real production scenarios. The first one is related to the mesh generation process, which is a part of the pre-processing step. For example, skewed tetrahedrons can be avoided if smoother mesh transitioning regions are applied at the borders between refinement regions and the bulk of a computational domain. Skewed elements tend to impose very small time integration step width due to very small inner radiuses, which can negatively affect clustering. Additionally, the user may consider rounding the used CAD geometry at places where two fault planes merge into a single one. Usually, the planes meet at very sharp angles, which may force a mesh generator to produce very small mesh elements in such zones. A slightly modified geometry can still lead to realistic modeling of earthquakes and simultaneously result in better LTS clustering and, thus, better performance. These techniques can also be applied while dealing with complex topological geo-data applied to a free surface. Rounding and smoothing a CAD geometry can remove sudden changes in elevation - e.g., due to steep hills or sharp cliffs - and, thus, improve the resulting mesh quality. A relatively slight increase in the element count and, thus, work can be compensated by better strong scaling GPU performance if the abovementioned changes applied during the meshing process result in better LTS clustering. However, generating a mesh consisting of several hundred million elements is a challenging process that can only be done on distributed-memory systems and typically takes considerable time and effort. Thus, the application of the abovementioned ideas to a real problem can be cumbersome for the end users.

The second approach can take advantage of the specifics of the LTS time-clusters scheduling in *SeisSol*. From Fig. 4.2, the reader can observe that, at some sub-steps, immediate neighboring clusters reach intermediate sync points. In such situations, the corresponding tasks become independent and, thus, may be executed concurrently. Such tasks could be merged into a single one, which would be larger regarding the number of elements. According to Fig. 6.28, the execution of such merged tasks would result in higher computational throughput on GPUs - i.e., due to the increased size. The proposed approach would also reduce the number of individual updates of the first time-cluster by $\frac{r}{r-1}$ times and, thus, its resulting performance weight, which, as shown above, tends to grow during strong scaling. However, an implementation of the proposed idea may entail significant changes in *SeisSol*'s source code. Hence, it is not considered in this work, but it is highly recommended for any follow-up study.

6.5.4. Enchanted Mesh Partitioning in SeisSol

In [30], I could not perform the intended weak scaling analysis of the first GPU implementation of *SeisSol* on the Marconi 100 supercomputer, equipped with 4 *Nvidia* V100-SXM2 GPUs per node. Executions of some parallel processes got aborted because the requested amount of memory exceeded the physical limit - i.e., 16 GB per GPU. The follow-up investigation showed that the problem occurred because of the original mesh partitioning approach, discussed in Section 4.4, which was focused only on LTS-specific work balancing. The approach assumed that the high memory capacity of modern HPC CPU systems could overcome any resulting memory imbalance. In this section, I present two extended versions of the algorithm that can simultaneously balance both work and memory. The enhanced versions helped to complete the weak scaling analysis of *SeisSol* on Marconi 100. Unfortunately, the Marconi 100 system was decommissioned at the moment of writing; therefore, the results presented in this section had to be obtained on a different supercomputer - i.e., LUMI.

Both enhanced versions are based on multi-constraint mesh partitioning applied to vertex weights. The first one, called "Balanced Work and Memory" (BWM), adds a constant value of 1 as the second vertex weight to Eq. 4.9. Together, it can be expressed as follows

$$w_m = \begin{bmatrix} r^{L-l-1} \\ 1 \end{bmatrix} \tag{6.15}$$

where $w_m \in \mathbb{R}^2$.

The allowed load imbalance for the second weight is set to a higher value (i.e., 1.05) than the imbalance for the first one (i.e., 1.01) to favor better work balancing and less constrain the graph partitioner after uncoarsening steps.

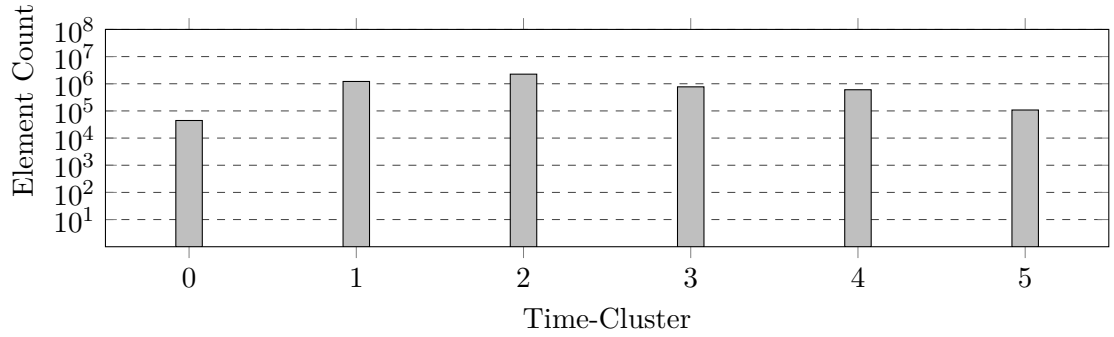
The second version of the algorithm, called "Encoded" (E), sets a binary vector of length L to each graph vertex, where L is the total number of time-clusters. All vector elements are initially set to zero except for the one with index l. This vector element is set to 1 and relates to time-cluster l to which the corresponding mesh element belongs. This can be written as follows

$$w_m^i = \begin{cases} 1, & \text{if } i = l \\ 0, & \text{otherwise} \end{cases} \tag{6.16}$$

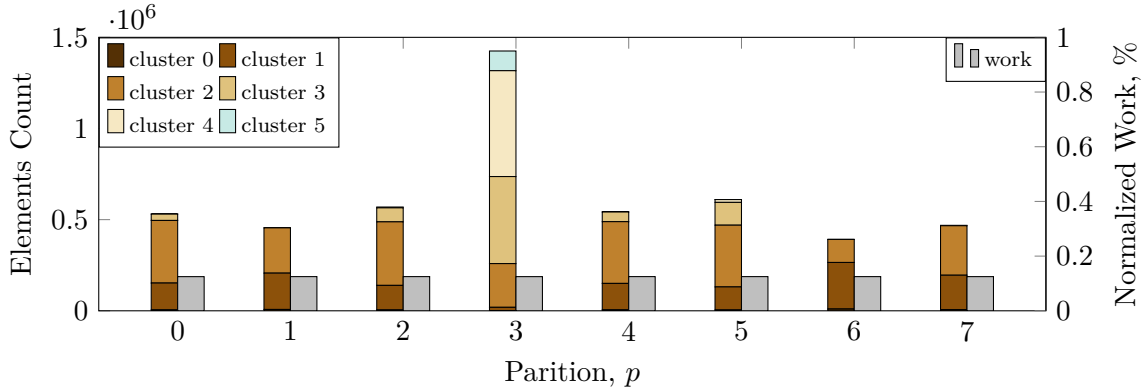
where $w_m \subset \{0,1\}^L$; and i is the binary vector index such that $i \in [0, L)$.

In this case, the allowed load imbalance for each constraint is set to 1.05. This mesh partitioning approach aims to distribute all time-clusters equally between processors. It is worth pointing out that this version does not operate on the definition of work in contrast to Eq. 4.9 and Eq. 6.15.

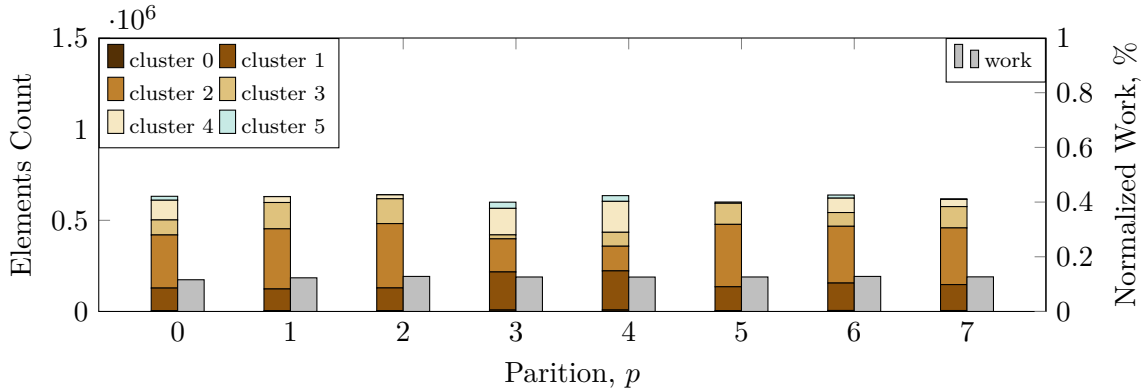
The LOH.1 scenario with a 5 million elements mesh, taken from work [30], is used to demonstrate the difference between the abovementioned partitioning versions. The resulting LTS clustering is shown in Fig. 6.33a.



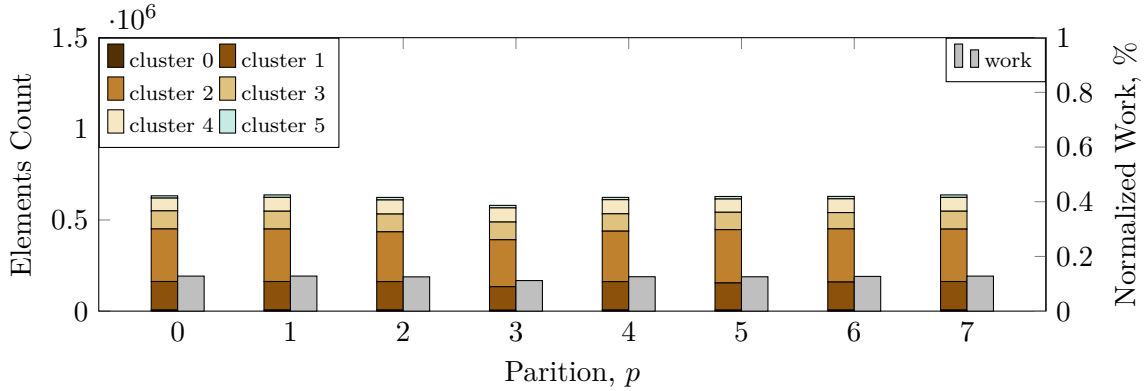
(a) Distribution of 5 million elements used for the LOH.1 test scenario among 6 LTS time-clusters.



(b) Balanced Work.

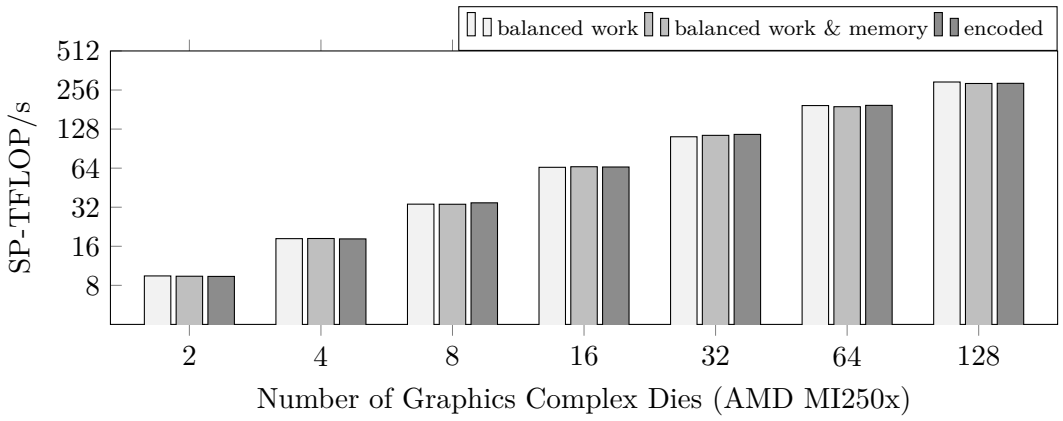


(c) Balanced Work and Memory.

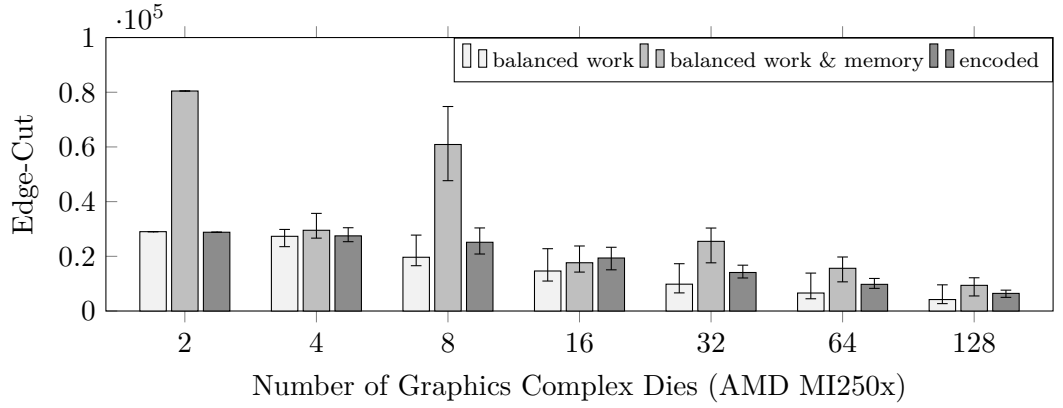


(d) Encoded.

Figure 6.33.: Comparison of different variants of the mesh partitioning algorithm applied to the LOH.1 test scenario.



(a) Strong scaling.



(b) Edge-cut statistics.

Figure 6.34.: Influence of different mesh partitioning versions on *SeisSol*'s strong scaling performance.

As shown in Fig. 6.33b, the original mesh partitioning approach, called "Balanced Work" (BW), results in a very high deviation in distributing mesh elements between processors. In this case, the third processor obtains approximately 2.8 times more elements than the others, leading to a very high memory imbalance - i.e., approximately 2.3. The situation drastically changes once the BWM version of the algorithm gets applied to the same scenario (see Fig. 6.33c). The memory imbalance significantly drops and reaches a value of 1.025. However, in this case, the mesh partitioner had to automatically increase the imposed work imbalance by approximately 1.5% in order to succeed. As shown in Fig. 6.33d, the "Encoded" mesh partitioning version also results in good work and memory balance. As in the previous case, the partitioner automatically raised the imposed work imbalance ($\approx 1.4\%$). The reader can observe that, in the last case, time-clusters get equally distributed among all processors, which is very noticeable compared to the other two cases.

Fig. 6.34 demonstrates how each abovementioned version of the mesh partitioning algorithm affects the parallel execution of *SeisSol* on distributed-memory systems. The most noticeable differences are observed in the median values of the edge-cut, which

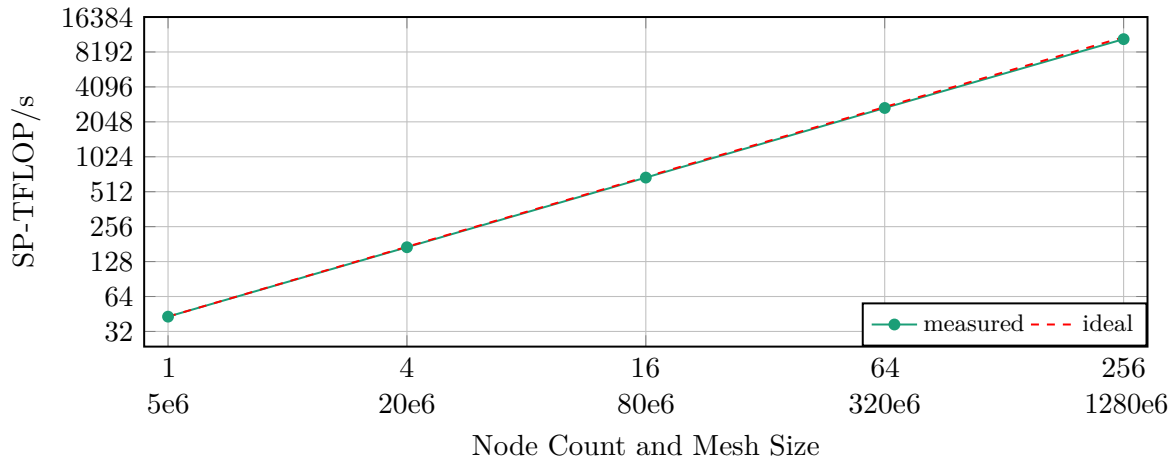
determine how well partitions are connected. Usually, a lower edge-cut value implies less inter-process communication. As shown in Fig. 6.34b, the BW version leads to the lowest edge-cut values within the entire scaling range. Then, the E version follows. In this case, one can observe the smallest deviations of the metric from their corresponding median values, which become very noticeable once the number of GPUs exceeds 32. The BWM version demonstrates the highest edge-cut values in all conducted experiments. It means that, in this case, the partitioner could not achieve good connectivity between sub-domains. Moreover, according to the statistical data shown in Fig. 6.34b, one can determine that the BWM version leads to high communication imbalances due to high deviations of the edge-cut. Nevertheless, according to Fig. 6.34a, *SeisSol*'s strong scaling performance does not change much while switching between different partitioning variants for the used LOH.1 scenario. This could be a result of good communication hiding achieved using asynchronous data exchange between processes while having enough work to process.

Apart from performing weak scaling studies of *SeisSol*, which are mainly used for performance engineering, the outcome of this study can be used in practice to model larger earthquake simulations on a limited number of CPUs or GPUs. It may be especially useful when a user's supercomputing project is restricted to a small subset of computational resources. In this case, a user can switch to one of the enhanced versions of the mesh partitioning algorithm and obtain the desired numerical results without re-meshing a target domain, which may take several iterations and, thus, a lot of time and effort.

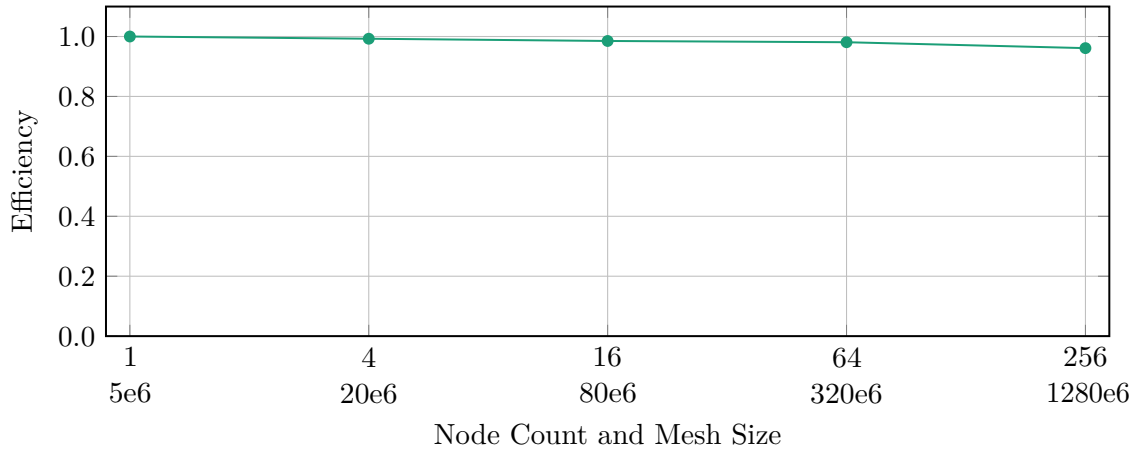
6.5.5. LTS Weak Scaling

This section presents the weak scaling study of *SeisSol* on the LUMI supercomputer, which aims to assess the communication and synchronization overheads as the wave propagation problem scales up. For this purpose, a set of computational meshes were generated using the approach described in Section 6.5.3 - i.e., splitting each element of an initial cartesian mesh into five tetrahedrons. This approach, combined with the "Encoded" mesh partitioning strategy (see Eq. 6.16), helped to preserve almost equal work distribution between GPUs during scaling and, thus, avoid computational or communication imbalance as much as possible. The initial domain of the LOH.1 scenario was set to $25km \times 25km \times 400km$, which was doubled along only the x and y dimensions at each subsequent weak scaling iteration. All generated meshes had the same relative LTS clustering inherited from the experiment shown in Fig. 6.20a. In this experiment, the first scaling iteration involved only intra-node communication because it was performed on 4 *AMD* MI250x GPUs located on a single node. After that, slower inter-node communication between GPUs gradually increased.

The results are depicted in Fig. 6.35 and show an excellent weak scaling performance of *SeisSol* within the entire test range. The parallel efficiency on 256 nodes (2048 MPI processes or 1024 *AMD* MI250x GPUs) achieves about 96%, resulting in approximately 10.5 SP-PFLOP/s. The average workload per GPU was enough to overlap the involved communication well. However, the initial work imbalance, computed based on the elapsed



(a) Weak scaling performance.



(b) Parallel efficiency.

Figure 6.35.: Weak scaling of the LOH.1 test scenario on the LUMI supercomputers.

time of each process and equal to 4%, gradually grows, reaching 6.3% in the end, despite the used meshing technique and partitioning strategy discussed above.

6.6. Source Code Portability

As discussed in Section 6.3.2, higher performance of the generated GEMM kernels is obtained using platforms' native compilers - i.e., *nvcc*, *amdclang*. It is also preferable to use the native SDKs to access some advanced GPU programming features to get better control for managing GPU's resources. This can be achieved by adding some portability layer to *SeisSol* and abstracting some commonly used algorithms.

CUDA and HIP are C-APIs and are syntactically close to each other. For them, the source code portability can be obtained using C-macros. However, SYCL - i.e., a native

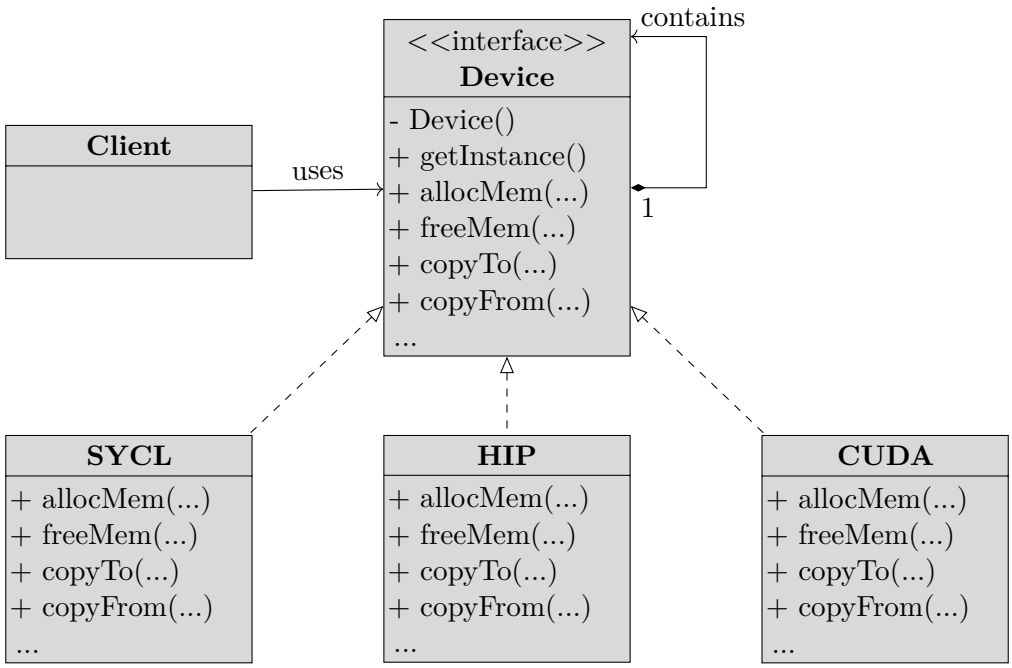


Figure 6.36.: The unified application programming interface in *SeisSol* - i.e., *Device API*.

programming model for *Intel* GPUs - is based on object-oriented programming and has many differences regarding device representation, management, and control relative to CUDA/HIP.

In *SeisSol*, I solved this problem by generalizing CUDA, HIP, and SYCL APIs and providing a unified programming interface - called *Device API*. This approach combines the advantages of the *Adapter* and *Facade* design patterns. As the *Adapter*, *Device* allows the user - i.e., *SeisSol* - to work with incompatible interfaces in a unified manner and, thus, seamlessly switch between GPU programming models if needed. As the *Facade*, the unified interface includes only those components that are practically needed for the client. This design helps to encapsulate and hide low-level details from the user - e.g., device selection, stack memory management (see Section 6.1), stream pool (see Section 6.4), error checking, etc. For APIs like CUDA/HIP, the wrapper adds a state that can be used to collect statistical data. Moreover, *Device* can be extended with other GPU programming models - e.g ., OpenMP, OpenACC - on demand if they provide the same functionality as specified by the interface.

As shown in Fig. 6.36, each GPU programming model is implemented as a sub-class of the abstract *Device* interface. A concrete implementation can selected only at compile-time. In *SeisSol*, the *Device* interface is implemented as the *Singelton*, which provides a single global access point to the unified API. This approach mimics CUDA and HIP C-APIs, which can be called from any place of a source code without any object's context.

The combination of code generation and *Device API* allows *SeisSol*'s users to seamlessly switch between different programming models for the wave propagation solver. For

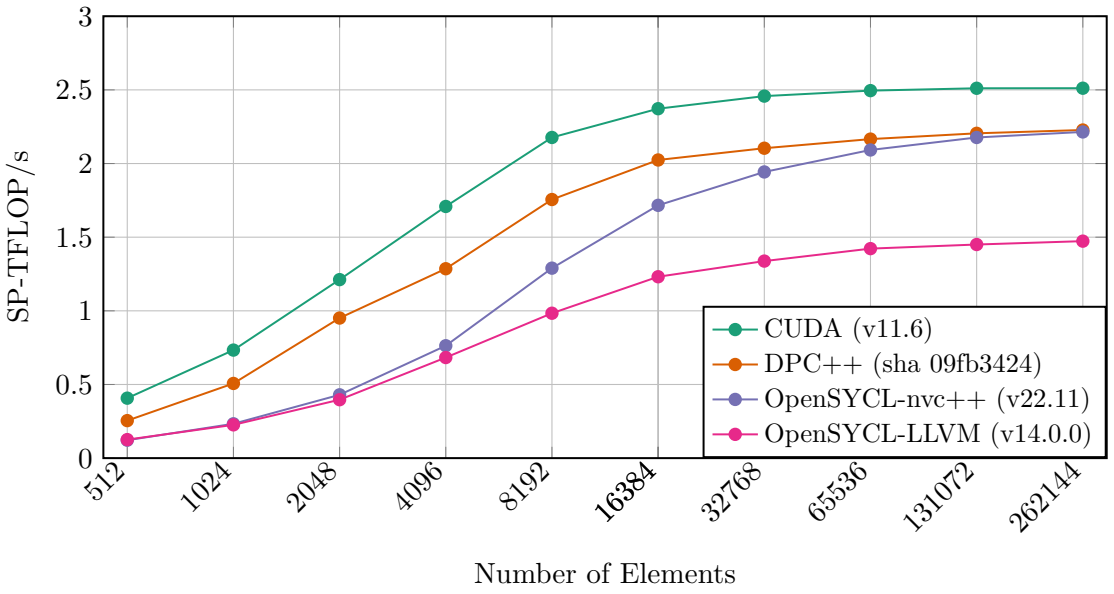


Figure 6.37.: Performance of *SeisSol*-proxy on *Nvidia* RTX 3080 Turbo GPU using different GPU backends.

example, Fig. 6.37 depicts results obtained with *SeisSol*-proxy on a single *Nvidia* RTX3080 Turbo GPU using CUDA and SYCL backends. In this experiment, I demonstrate how different implementations of SYCL perform relative to CUDA. Because the experiment was conducted on *Nvidia* hardware, CUDA implementation of the wave propagation solver is considered the baseline. At the moment of writing, *ChainForge* did not support SYCL; therefore, I used *GemmForge* as *YATeTo*'s GPU backend.

As can be seen, *Intel*'s implementation of SYCL (i.e., DPC++) shows much better results compared to other implementations of the standard; the average performance is only $\approx$ 12.4% lower than the baseline. The used open-source implementation of SYCL (i.e., OpenSYCL[5] [5] version 0.9.4) relies on a compiler backend for generating intermediate GPU code, which is later translated to concrete machine instructions (see Section 5.2). Regarding *Nvidia* GPUs, OpenSYCL can issue the intermediate PTX code using either LLVM or nvc++, a proprietary *Nvidia* C++ compiler. The OpenSYCL-LLVM configuration shows the worth results compared to the others, achieving only $\approx$ 57% of the baseline performance within the entire test range. This may indicate that the resultant PTX code generated by LLVM was less optimized during compilation than the one produced by DPC++. The performance obtained with OpenSYCL-nvc++ is close to the one obtained with DPC++ under high workloads. However, it beings to drop rapidly when the workload gets reduced. The results of both OpenSYCL configurations become indistinguishable within the $[512, 2048]$ range. This outcome suggests that the OpenSYCL implementation results in higher kernel launching overheads compared to DPC++, which may be induced by its complex event-driven mechanism used in the SYCL *Queue* class.

[5] The *OpenSYCL* project was renamed to *AdaptiveCpp*.

The results shown in Fig. 6.37 were obtained using the same versions of *SeisSol*, *Device*, *YATeTo*, and *GemmForge* compiled 4 times for different tested GPU configurations. The experiment additionally shows the source code portability of *SeisSol* to different GPU programming models and, thus, to different platforms. As shown in Fig. 6.10 and Fig. 6.11, the generated GPU code results in high GPU performance on both *Nvidia* and *AMD* platforms. The introduced changes shown in this chapter do not degrade the original high CPU performance of *SeisSol*. Therefore, it also demonstrates the performance portability of *SeisSol* between CPU and GPU computing platforms.

It is worth mentioning the similarities of the resulting software architecture with the key idea of the RAJA project [9, 69] - i.e., the separation of the computational and device management abstractions. Similar to the RAJA library, *YATeTo* only acts as the performance portable layer through which all computations are expressed in *SeisSol*'s wave propagation solver. Meanwhile, the *Device API* abstracts the GPU resource management in *SeisSol* and, thus, acts similarly to the Umpire library [10] in RAJA applications. In general, the *Device API* can be replaced with RAJA and Umpire libraries, which may reduce the cost of the source code maintenance in *SeisSol*. However, it can only be done once SYCL support is added to RAJA, which is still under development at the moment of writing. In this case, RAJA would be used to substitute some commonly used GPU algorithms in *SeisSol* - e.g., parallel reduction - which are currently embedded into the *Device API*.

6.7. Verification and Convergence Study

The analytical solution of plane waves can be found in the following form

$$Q_p(\boldsymbol{x}, t) = a_p e^{i(\omega t - \boldsymbol{k}\cdot\boldsymbol{x})} \tag{6.17}$$

where $a_p \in \mathbb{C}^9$ is vector of amplitudes of the physical quantities - i.e., σ_{11}, σ_{22}, σ_{33}, σ_{12}, σ_{23}, σ_{13}, u_1, u_2, u_3; $\omega \in \mathbb{C}$ is the angular frequency; $\boldsymbol{k} \in \mathbb{R}^3$ is the wave direction; and i is the imaginary unit.

Given the wave direction $\boldsymbol{k}$ and material properties, the vector of amplitudes a_p and angular frequency ω can be found by plugging Eq. 6.17 into Eq. 2.8, which yields the following eigenvalue problem

$$W_{pq} a_q = \omega a_p \tag{6.18}$$

where W_{pq} is a so-called plane-wave operator, equaled to $k_1 A_{pq} + k_2 B_{pq} + k_3 C_{pq}$.

Thus, the plane wave solution at any arbitrary point in time t can be found using Eq. 6.17, taking an eigenpair of the plane-wave operator as the angular frequency and vector of amplitudes. A solution found in this way is a vector of complex numbers which may be difficult to compare with numerical results. However, due to the linearity of the underlying system of PDEs, a new solution can be built using a linear combination of Eq. 6.17 with

its conjugate (see Eq. 6.19).

$$Q_p(\boldsymbol{x},t) = \frac{1}{2}\left(a_p e^{i(\omega t - \boldsymbol{k}\cdot\boldsymbol{x})} + \overline{a}_p e^{-i(\overline{\omega} t - \boldsymbol{k}\cdot\boldsymbol{x})}\right) = Re\left(a_p e^{i(\omega t - \boldsymbol{k}\cdot\boldsymbol{x})}\right) \tag{6.19}$$

The substitution of Eq. 6.19 to Eq. 2.8 results in the same plane-wave operator as in Eq. 6.18 and, thus, requires the same eigenpairs. Each pair satisfies Eq. 2.8 and represents an individual wave. As shown in Eq. 6.20, a complete analytical solution can be obtained as a superposition of individual ones.

$$Q_p(\boldsymbol{x},t) = \sum_{j\in J} Re\left(v_{p_j} e^{i(\omega_j t - \boldsymbol{k}\cdot\boldsymbol{x})}\right) \tag{6.20}$$

where J is a set of eigenpairs of the plane-wave operator W_{pq}; and (ω_j, v_{p_j}) is the j-th eigenvalue pair.

A solution to the plane wave problem implies the infinite domain. However, as shown in [114], it can also be mapped into a unit cube domain using periodic boundary conditions if the wave direction vector is scaled appropriately.

Comparisons of numerical and analytical solutions of the plane wave problem are shown in Fig. 6.38. The experiment was conducted on a single *AMD* MI250x GPU, testing the accuracy of numerical solutions using maximal polynomial degrees $\mathcal{N}$ ranging from 1 to 5 and the single- and double-precision floating-point formats. The following material parameters were used: $\rho = 1$, $\mu = 1$ and $\lambda = 2$. The wave vector $\boldsymbol{k}$ pointed to (π, π, π) direction.

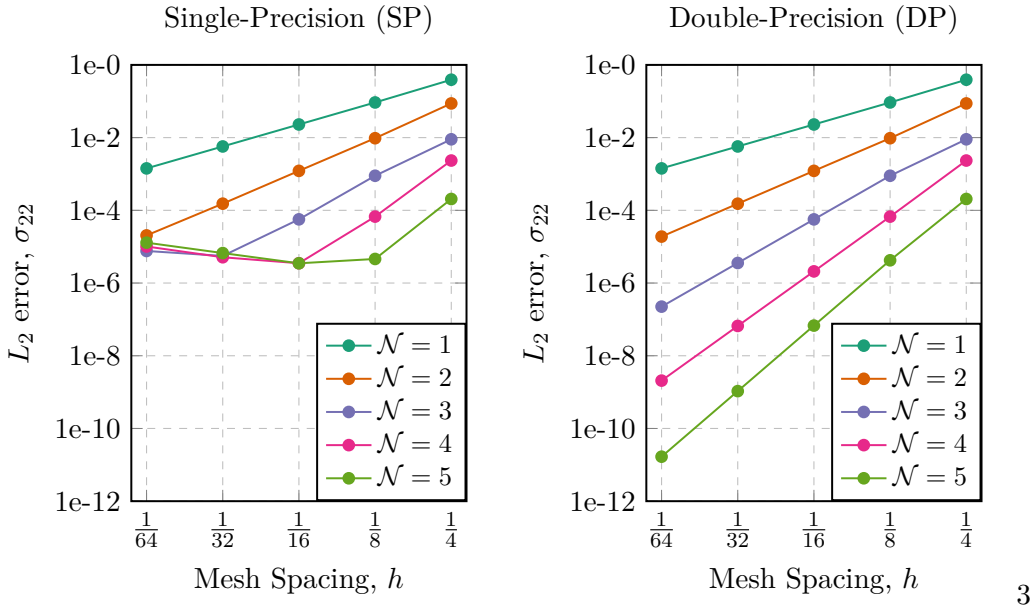


Figure 6.38.: Convergence analysis of the GPU implementation of *SeisSol*'s elastic wave propagation solver.

The convergence of numerical solutions was tested using a set of computational meshes, gradually refined by a factor of 2 along each dimension for each subsequent iteration. The

Table 6.3.: Average empirical convergence orders of the σ_{22} variable obtained while simulating the plane waves with *SeisSol* on *AMD* MI250x GPU using the double-precision floating-point format.

Maximum polynomial degrees, $\mathcal{N}$	1	2	3	4	5
Average empirical convergence order, $\mathcal{O}_e$	2.03	3.04	3.82	5.03	5.89

mesh generation process consisted of two steps. Initially, a cartesian mesh was generated by dividing each domain edge into an appropriate scale factor - i.e., 4, 8, 16, 32, 64. Then, each cubic element was split into five almost equal tetrahedrons. Analytical and numerical solutions were compared after 0.5 seconds of each simulation at $(\mathcal{N}+2)^3$ points in each tetrahedron, defined by the conical product of the one-dimensional Gauss-Jacobi formula (see [106]).

As can be seen, the most precise numerical solution was obtained with the double-precision floating-point format and the imposed converge order 6. The L_2 error of the σ_{22} variable resulted in approximately $1.7e-11$ while using the mesh with the highest resolution.

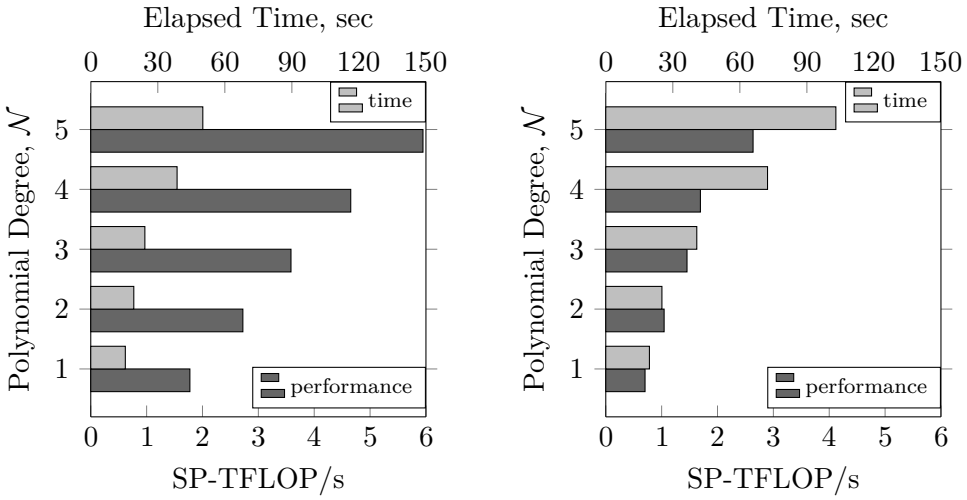
Table 6.3 shows the empirical convergence order derived from Fig. 6.38 using Eq. 6.21 regarding the double-precision floating-point format. As can be seen, the order matches $\mathcal{N}+1$, which is expected according to [53].

$$\tilde{\mathcal{O}}_{L_2} = \log_2 \left(\frac{Error_{L_2}^{h_{i-1}}}{Error_{L_2}^{h_i}} \right) \tag{6.21}$$

where h_i denotes mesh spacing such that higher values of i correspond to finer meshes.

In Fig. 6.38, one can also notice that the error is bound to approximately $5e-06$ when the single-precision floating-point format is used regardless of the imposed mesh spacing and the polynomial degree. This behavior matches the results obtained on x86 CPU platforms by Uphoff and Breuer in [114] and [13], respectively.

Fig. 6.39 demonstrates the performance of *SeisSol*-proxy configured with different polynomial degrees and the floating-point formats. The results were obtained using a fixed number of elements. One can observe that the performance grows almost linearly relative to the maximum polynomial degree because the resulting matrix sizes become larger, and, thus, the generated code better exploits all GPU memory sub-systems. Combining the outcomes shown in Fig. 6.38 and Fig. 6.39, one can conclude that a user may only benefit from the single-precision format and high polynomial degree when using coarse meshes. This configuration may be helpful while working on prototyping an earthquake scenario. Otherwise, it wastes energy because, in this case, higher computational efforts do not increase numerical accuracy.



(a) Single-Precision (SP) (b) Double-Precision (DP)

Figure 6.39.: Performance of *SeisSol*-proxy on *AMD* MI250x GPU regarding different maximal polynomial degrees $\mathcal{N}$ and the floating-point formats. The experiment was conducted with a fixed number of mesh elements equaled to 524288.

6.8. Discussion

The chapter presents various implementation details of the GPU version of *SeisSol*'s wave propagation solver. The discussion contains several major topics: 1) code generation of high-performance GPU kernels, 2) source code and performance portability and 3) the analysis of strong scaling performance of the LTS algorithm on distributed multi-GPU systems.

The outcome of this study shows that the portability of HPC applications (e.g., *SeisSol*) can be conveniently achieved through an intermediate software layer - i.e., code generation. The layer establishes the separation of concerns, which, in general, enhances the maintainability and modifiability of software. Theoretically, software can be re-targeted to a new platform without any (or with little) changes in the high-level source code. If computations are properly abstracted, re-targeting may be performed even by a different team of developers who may have no or little knowledge about the main application domain. This approach has proven itself in all major Artificial Intelligence (AI) frameworks (e.g., PyTorch, TensorFlow) and resulted in many so-called graph compilers (e.g., TVM, XLA). However, at this moment, the approach is not so common in scientific computing because the area is too broad.

This work clearly shows that the original intermediate code representation, established by the *YATeTo* DSL, was insufficient to handle massively parallel systems (e.g., GPUs). The original DSL design aimed to generate highly efficient code for computational sub-tasks, leaving task decomposition to the host application (e.g., *SeisSol*, *tandem*[6]). The CPU-like

[6] https://github.com/TEAR-ERC/tandem

computing model is flexible because it allows users to perform a mixed sub-task execution - i.e., to schedule and execute sub-tasks of different kinds. However, modern heterogeneous programming models are more restrictive since they only consider the SIMD-like program execution. In this model, a computational kernel describes all sub-tasks of a task, which are applied to different pieces of data. This restriction forced me to reconsider the original task decomposition in *SeisSol* and introduce batched computations in *YATeTo* to accommodate heterogeneous computing.

The technique discussed in Section 6.3.2 extends the DSL further. As shown in this study, the average GPU performance of the ADER-DG method can be improved by more than 35% by fusing subsequent batched GEMM kernels. The idea of kernel-fusion is not novel. Many graph compilers have successfully exploited this optimization method for accelerating AI models for GPUs. The pattern-matching, similar to the one shown in Section 6.3.2, is widely used for finding candidates for fusion. In contrast to the AI graph compilers, which usually fuse a single explicit or implicit[7] GEMM operation with subsequent pointwise computations, the method proposed in this work tackles a bit more complicated problem - i.e., a fusion of subsequent batched GEMM kernels. The outcome of this study is limited to only GEMM operations; however, the work can be further extended if needed. The ideas implemented in *ChainForge* can serve as a basis for follow-up research and development.

Unified memory is convenient for programming heterogeneous systems. It allows a programmer to focus solely on developing highly efficient device kernels without spending time on writing, maintaining, and optimizing the code for host-to-device data transfers. However, the results presented in this chapter show that it is better to use regular device memory for MPI buffers for latency critical HPC algorithms (e.g., LTS). In this case, the MPI runtime uses more efficient communication protocols that do not perform host-to-device data transfers in the background. Such protocols result in minimal latency and maximal bandwidth for point-to-point communication between accelerators (see Fig. 6.19).

Results presented in Section 6.5.2 and Section 6.5.3 suggest that computational throughput characteristics can significantly impact the strong scaling performance of the LTS scheme on distributed-memory systems. The parallel efficiency of the algorithm is very sensitive to 1) LTS clustering (see Fig. 6.32) and 2) the performance of a computational unit under low and medium workloads (see Fig. 6.10). Section 6.5.3 demonstrates a simple mathematical model that can only estimate the upper bound of the strong scaling performance for a specific device and clustering. Because the computational throughput is non-linear regarding the problem size, it is difficult to find a simple rule of thumb that users can utilize to estimate the number of GPUs for a particular mesh. The conclusion of Section 6.5.3 suggests that the strong scaling performance of the LTS scheme can be algorithmically improved by merging LTS tasks of neighboring clusters into a single one at common synchronization sub-intervals (see Fig. 4.2). As mentioned, it will reduce the overall number of small-sized high-frequency tasks, allowing the LTS algorithm to better exploit the GPU hardware capabilities.

[7] a variant of direct convolution through the implicit GEMM operation

7. Implementation of Dynamic Rupture

As discussed in Section 2.2, rock sliding along fractures in Earth's crust generates seismic waves that radiate in the media. The ability to simulate this process accurately is essential for achieving realistic simulations of earthquakes. The sliding process involves friction, which can be described by multiple friction laws.

The simulation of the dynamic rupture process is incorporated into the ADER-DG method through a specially designed boundary condition. As shown in Fig. 3.6.3, it involves projecting the volumetric solution inside each affected tetrahedron onto a set of Gaussian points for which an inverse Riemann problem must to be solved at multiple time sub-intervals. The relation between the friction coefficient and slip rate magnitude, defined by a friction law, becomes necessary to close the resulting systems of equations. Some models (e.g., Linear Slip Weakening) become cost-effective but require a careful choice of parameters to achieve accurate results. Conversely, the others, which usually describe the underlying physical process more precisely (e.g., Aging Law), results in significant computational efforts. Finally, partial solutions at each sub-interval and each Gaussian point must be aggregated into corresponding flux contributions and projected back to the modal basis.

As can be seen from Eq. 2.22 and Eq. 2.23, the relations described by friction laws involve non-linear functions. The *YATeTo* DSL cannot express such computations because it is mainly designed to handle linear combinations of tensors and sequences of tensor products. Therefore, the dynamic rupture code implementation relies on regular programming techniques instead of code generation.

The previous attempts to adapt heterogeneous computing in *SeisSol* were summarized in [50, 49], where the *Intel* Xeon Phi coprocessors were primarily used as accelerators. In both works, the authors used only a single quadrature point in time, saying that it did not negatively influence accuracy and convergence behavior. Therefore, the authors decided to leave the dynamic rupture computations on the host, arguing that running such code sections on the higher-clocked CPU cores was advantageous because they could better handle 1) non-linear operations (e.g., square root, division, etc.) and 2) low computational loads resulting from a small number of dynamic rupture elements. Moreover, in [49], the authors admit that they could only support the dynamic rupture physics for the GTS scheme.

This work extends the original approach in several directions. Firstly, as required by Eq. 3.44, all $\mathcal{O}$ quadrature points in time are considered, which leads to more accurate numerical results. This, however, increases the computational loads of the dynamic

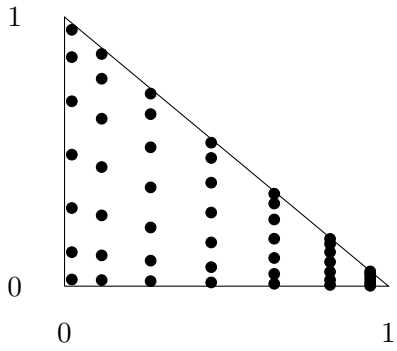
rupture tasks. Thus, the tasks are fully offloaded to accelerators. As a consequence, it also eliminates the necessity of exchanging data between the host and device, which, as mentioned in [50], can be difficult to fully overlap with other computations. Lastly, in this work, the LTS scheme is used for the wave propagation and dynamic rupture solvers, which adds additional acceleration to numerical simulations.

This chapter is structured as follows. Section 7.1 introduces the task decomposition of the dynamic rupture solver. Then, in Section 7.2, I review various programming models for heterogeneous computing and select more suitable ones to provide a portable implementation of the dynamic rupture code in *SeisSol*. I test and compare different implementations of the selected models on the same hardware. The results of the comparisons determine the selection of a specific programming model, which is used for the rest of this work. In this section, I also compare the run time of the Linear Slip Weakening and Aging friction laws using the same computational mesh and test environment. Section 7.3 contains strong scaling results of the TPV-5 test scenario obtained on the Selene, Leonardo, and LUMI supercomputers. In Section 7.4, I demonstrate verification of numerical results obtained with the GPU implementation of the dynamic rupture code to ensure its correctness. Finally, Section 7.5 summarizes the main outcomes of this part of the study and concludes the chapter.

Due to the large configuration space of *SeisSol*, this chapter demonstrates the results obtained mainly with the single-precision floating-point format and convergence order equal to 6. This is done for convenience for comparing different experiments shown in this work. Results shown in Fig. 6.39 can be used to compute the appropriate coefficients to extrapolate the shown performance data to different floating-point data types and convergence orders.

7.1. Parallelization

Theoretical and numerical aspects of the dynamic rupture process are explained in Section 2.2 and 3.6.3, respectively. The computations start with projecting the solutions from two neighboring tetrahedrons, which share a dynamic rupture face, onto a set of 2D Gaussian points, shown in Fig. 7.1. The number of Gaussian points is chosen to have the same strength as the strength of the 3D polynomial modal basis used in the wave propagation solver. The projections are defined by Eq. 3.45 and Eq. 3.45, and involve the corresponding Vandermonde matrices - i.e., $\Phi_m(\boldsymbol{\xi}^f(\boldsymbol{\chi}_i))$ and $\Phi_m(\boldsymbol{\xi}^g(\tilde{\boldsymbol{\chi}}^h(\boldsymbol{\chi}_i)))$. In *SeisSol*, the projections are considered to be a part of the wave propagation solver and, thus, are implemented with the *YATeTo* DSL. The parameterization of the Vandermonde matrices with f, g, and h values defines a set of unique tensor expressions, which, similar to the local and neighboring macro-kernels (I^{local}_{surf} and I^{nghb}_{surf}, respectively), can be executed concurrently and, thus, are implemented using the stream-based execution model (cf. Fig. 6.14).



Strength	Number Gaussian Points
4	25
5	36
6	49
7	64

Figure 7.1.: Left: Gaussian points within the canonical triangle required for the Stroud quadrature rule of strength 6. Right: Dependencies of the strengths of the Stroud rule on the number of Gaussian points.

After projecting the volumetric solution onto the fault, the inverse Riemann problem, discussed in Section 3.6.3, must be numerically solved for each Gaussian point at each time sub-interval (see Fig. 3.7). As in many scientific algorithms, computations between time steps impose data dependencies and thus are not concurrent. For example, the values of the state variables (see Eq. 2.21) determined at each Gaussian point get updated between time sub-intervals. At the same time, computations between Gaussian points within each time integration sub-step are mostly data-independent and can be done in an embarrassingly parallel manner. However, there are a few exceptions regarding the SIMT model, where threads' cooperation is required - e.g., data resampling within a rupture face followed by point-wise operations. In such rare cases, a programming implementation of the model (e.g., CUDA, HIP, or SYCL) may need to use shared memory and block-wise thread synchronizations. After updating the stress and velocity components according to Eq. 3.56 and Eq. 3.64, solution vectors $\hat{Q}^b_{il}$ and $\hat{Q}^c_{il}$ can finally be assembled at all Gaussian points. After that, the adjusted flux contributions can be computed for both sides of the fault according to Eq. 3.44.

In *SeisSol*, the CPU parallelization of the dynamic rupture code combines both multi-threading and vectorization. A sub-task is defined as a computation of flux contributions from both sides stemming from a single dynamic rupture face; thus, a sub-task includes updates of all Gaussian points enclosed by a rupture face (see Eq. 7.1). The CPU threads are forked at the beginning of the dynamic rupture code and keep operating until the end (Eq. 3.44), making sub-tasks coarse-grained. Sub-tasks are distributed between threads using the *OpenMP* static scheduling policy. Higher sub-task performance can be achieved by applying vectorization, for which the number of Gaussian points must be padded to a multiple of the vector register length. In *SeisSol*, the *OpenMP* SIMD directives are used to enable auto-vectorization, resulting in performance portability among various CPU platforms.

On GPUs, many point-local computations can be performed by individual GPU threads, which mostly operate on their local data. However, sometimes, GPU threads need to be grouped into blocks to operate on individual rupture faces due to the abovementioned

exceptions - i.e., when a thread's cooperation is required. In such cases, the block size can be equal to the number of Gaussian points, which is determined by the quadrature rule. Thus, the GPU task decomposition can be directly inherited from the CPU version of the code, in contrast to the discussed implementation of the wave propagation solver (see Section 6.2). It is worth noting that, despite operating on small blocks regarding their sizes, the GPU scheduler can still generate enough active warps by assigning several blocks to each SM, which should lead to good GPU occupancy.

In contrast to the CPU version of the code, which opens the parallel region only once, the GPU variant consists of a series of kernels, implementing one or several consecutive dynamic rupture equations. This approach adds flexibility to the software design because the main host thread changes the control flow and selects appropriate kernels based on the chosen friction law, using the curiously recurring template pattern, so-called static polymorphism. The asynchronous kernel execution adapted by many GPU programming models (see Section 5.1) can help to overlap the associated kernel launching overheads with computations if the workload is enough. However, due to the used LTS scheme in the dynamic rupture code, added as a contribution of work [114] to *SeisSol*, the proposed GPU implementation can experience problems similar to the one discussed at the beginning of Section 6.5.2.

7.2. Portability

In contrast to *SeisSol*'s implementation of the wave propagation solver, where most of the computations are generalized by the *YATeTo* DSL, acting as a performance portable layer, the dynamic rupture code in *SeisSol* is written in a conventional manner - i.e., without code generation. Therefore, designing a portable solution, at least for the GPU implementation, is necessary to reduce maintenance costs. Apart from the source code portability, it is desirable to achieve a performance portable solution that can cooperate with the one designed for the wave propagation solver, which, as discussed in Chapter 6, is based on vendor-native GPU programming models - e.g., CUDA, HIP, SYCL.

As shown in [54, 64], the performance of some OpenACC device kernels can be close to the one implemented with CUDA. However, the authors admit that it often requires some manual code tuning. The OpenACC standard has limited compiler support. For example, the LLVM implementation of the C/C++ compilers does not natively support OpenACC. The support is provided by the *Clacc* extension [25] of the LLVM project, which acts as a transpiler, converting OpenACC code to OpenMP in place. Another example is the compiler suite from *Cray*, which only supports OpenACC for their Fortran compiler. Moreover, at the moment of writing, no compiler implementors claimed to support OpenACC for the upcoming *Intel* GPUs. Therefore, OpenACC was excluded from the list of possible candidates.

The OpenMP standard excels in the HPC market; thus, it is widely adapted by many compiler vendors. As shown in [92], in some cases, device kernels implemented with

OpenMP can outperform their OpenACC counterparts due to better use of team-local (i.e., shared) memory and registers. In theory, the code written with OpenMP should be portable to a wide range of accelerators without considerable code changes. In practice, some compiler implementations contain only a subset of the standard. Thus, a programmer may need to use C-macros and write different OpenMP statements for different compilers or even their versions. The same holds for OpenACC. This fact seriously violates the requirement to source code portability for both models. As an advantage, both OpenACC and OpenMP have native compatibility with CUDA and HIP regarding the memory models and the device selection mechanisms, making them convenient for mixed GPU programming.

SYCL is a single-source programming model acting as a high-level C++ abstraction layer to target various heterogeneous hardware accelerators. The known SYCL implementations involve the compilation process. For example, *Intel*'s *DPC++* compiler, a fork of the LLVM project, must be used to compile the entire application; offloading to a particular device is enabled using appropriate compiler flags. Another example is the *OpenSYCL* project, formerly known as *hipSYCL*, which must be linked against an installed LLVM library. During processing, *OpenSYCL* utilizes the LLVM library to build LLVM intermediate representations of device kernels and then delegates the rest of the compilation process to an appropriate compiler backend - e.g., NVPTX, AMDGPU. The Unified Shared Memory (USM) model, adapted in the latest SYCL standard, allows a program to use raw C/C++ pointers inside device kernels. The standard delegates the responsibility to programmers to ensure that memory is accessible on the device through provided pointers. Therefore, it is theoretically possible to bundle SYCL with CUDA/HIP if both runtimes have their contexts attached to the same device. However, in contrast to OpenACC and OpenMP, the device selection mechanism in SYCL is generally incompatible with CUDA and HIP. Therefore, it may be difficult to ensure that both runtimes operate with the same device. Moreover, the latest standard version allows the implementors to interact directly with the native runtimes to manage the state or memory of a device; before, the standard required interaction only through the OpenCL runtime. Therefore, a SYCL implementation may internally change the global state of the CUDA/HIP context used by a programmer, which may lead to unexpected behavior of an application during run time. Regarding the upcoming Intel GPUs, SYCL is a good candidate for *SeisSol* because, in this case, the same GPU programming model will be used for both solvers - i.e., the wave propagation and dynamic rupture.

Kokkos and RAJA may also be worth considering. However, the SYCL model already covers many functionalities which Kokkos and RAJA can provide; therefore, these models were not considered in this work.

Summarizing the above discussion, only the OpenMP and SYCL programming models were chosen for implementing a performance-potable solution for the dynamic rupture solver in *SeisSol* in this work. Regarding OpenMP, *nvc++*, *clang*, and *gcc* compilers were used in this study. I selected *DPC++* and *OpenSYCL* to assess the potential performance of the SYCL model. The tests were conducted using the TPV-5 test scenario taken from the SCEC benchmark collection [47].

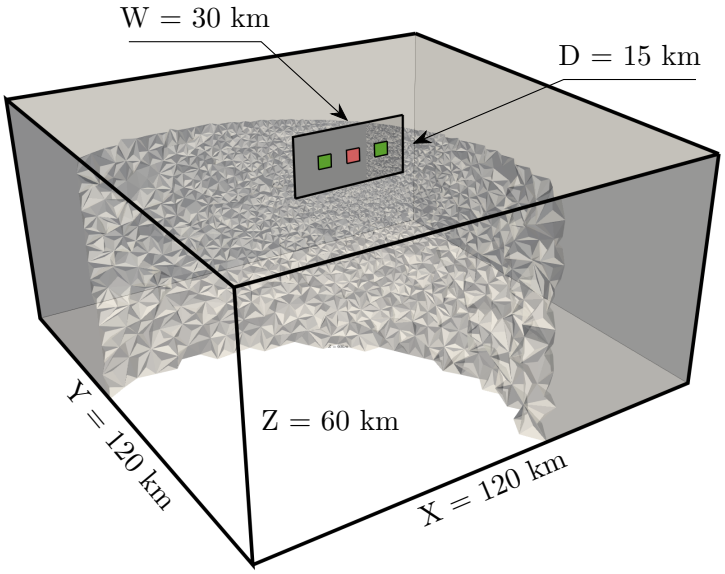


Figure 7.2.: Geometry and mesh refinement of the TPV-5 test scenario.

As shown in Fig. 7.2, the scenario has a vertical strike-slip fault inside a homogeneous cuboid domain. The earthquake rupture is artificially nucleated at the center of the fault - i.e., in a small square zone shown in red color. The rupture spontaneously begins from the nucleation region and moves toward the rest of the fault surface. As the process involves, the rupture encounters two square patches (shown in green), along which the initial stress conditions are set to different values compared to other parts of the fault surface. In this scenario, the nucleation is imposed by setting the initial shear stress higher than the initial static yield stress inside the nucleation patch. The failure affects all parts of the fault plane, including the nucleation patch.

The top of the cuboid domain represents a free surface, whereas the rest act as absorbing boundaries. The material properties are homogeneous in this scenario. The material density equals 2760 kg/m^3, while the first and second Lamé parameters are about 32.04 GPa. The details of the TPV-5 scenario regarding the initial conditions are given in [93].

In the TPV-5 scenario, the friction is subjected to the Linear Slip-Weakening law (see Eq. 2.23) with homogeneous parameters along the fault. The static and dynamic friction coefficients are equal to 0.677 and 0.525, respectively. The critical distance is set to 0.4, whereas, the cohesion coefficient equals zero.

The experiment was conducted using a one million elements mesh, containing almost 50 thousand rupture elements along the fault, on a single *Nvidia* A100 GPU. The baseline solution was obtained on a single *AMD* EPYC 7763 CPU using all 64 cores and making use of the AVX256 instruction set via 1) the LIBSXMM library (used as the main *YATeTo* backend for the wave propagation solver) and 2) auto-vectorization (used for the dynamic

Table 7.1.: Comparison of the OpenMP and SYCL parallel programming models on a single AMD EPYC 7763 64-core CPU and *Nvidia* A100-PCIE-40GB GPU. The performance results were obtained using one million elements mesh of the TPV-5 test scenario and averaged among the first 1000 time steps.

Hardware	64 cores	1 core 1 GPU	1 core 1 GPU	1 core 1 GPU	1 core 1 GPU	1 core 1 GPU	1 core 1 GPU
Model	OpenMP v4.0	DPC++ v2023WW13	OpenSYCL v9.4	OpenSYCL v9.4	OpenMP v5.0	OpenMP v4.5	OpenMP v4.5
Compiler	gcc v11.2.0	clang v17.0	nvhpc v22.11	clang v14.0.6	nvhpc v22.11	clang v14.0.6	gcc v11.2.0
Kernels time, sec	197.009	88.209	88.245	88.515	91.331	107.171	1048.261
Elapsed time, sec	197.052	88.373	88.412	88.682	146.431	115.154	1056.119
Speed-up	1.000	2.229	2.228	2.222	1.346	1.711	0.187
Performance, SP-TFLOP/s	2.023	6.110	6.108	6.089	4.559	4.869	0.514

rupture code). The performance data were averaged among the first 1000 time steps. The results of the experiment and comparisons are shown in Table 7.1.

As can be seen from Table 7.1, the SYCL implementation of *SeisSol*'s dynamic rupture solver resulted in the best performance for the given scenario. The difference in run time between all tested SYCL implementations - i.e., DPC++ and OpenSYCL - is less than 0.5%. The average speedup against the baseline achieved a value of 2.2. On the other hand, results obtained with the OpenMP implementations were inconsistent regarding performance. In all three cases, one can observe substantial differences between the time spent on executing GPU kernels and the elapsed time. It indicates that the tested OpenMP implementations entail significant overheads caused by redundant synchronizations between invocations of GPU kernels, even though the "*nowait*" clauses were explicitly specified for all OpenMP target regions.

The best results with the *Nvidia* C++ compiler - i.e., *nvhpc* - were obtained while using "*target teams loop*" and "*loop bind*" OpenMP clauses which are parts of the fifth version of the standard. The clauses add a descriptive behavior to OpenMP, which is known to be a prescriptive model [26]. The "*loop*" directives only prescribe that a next for-loop statement must be parallelized and leave it up to the compiler to decide how it should be done. As can be seen from Table 7.1, this approach resulted in the shortest time spent on executing GPU kernels among all tested OpenMP implementations. The tested versions of *clang* and *gcc* compilers do not support the OpenMP "*loop*" construct; thus, the constructs were replaced with "*target teams distribute*" and "*parallel for*" directives for which I had to explicitly specify the number of teams, the team size and scheduling for each encountered *for-loop* - i.e., "*schedule(static,1)*". Even though the kernels generated by the *clang* compiler performed slightly worse than the ones generated by *Nvidia* C++, the *clang* version of the code outperformed the *Nvidia* C++ implementation by approximately 21% with respect to the elapsed time because of significantly smaller overheads. The *gcc* version of the OpenMP code resulted in the worst performance among all tested implementations of the dynamic rupture code and was almost 5 times slower than the baseline implementation.

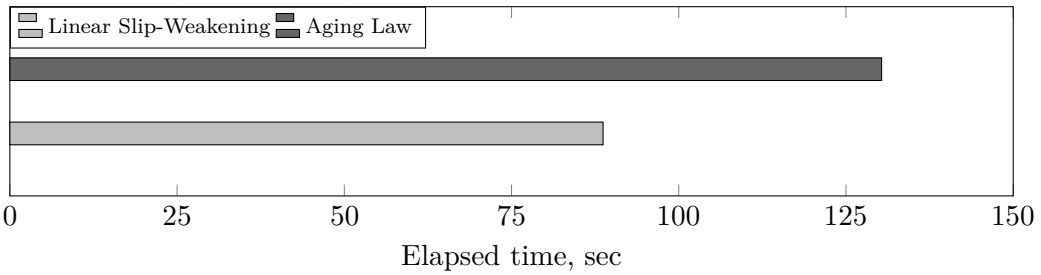


Figure 7.3.: Comparisons of the Linear Slip-Weakening and Aging friction laws regarding performance on a single *Nvidia* A100-PCIE-40GB GPU using the same one million elements mesh for both cases.

The results obtained during the experiment determined the choice of SYCL for two reasons. Firstly, all SYCL implementations of the dynamic rupture solver outperformed their OpenMP counterparts. Secondly, all tested SYCL implementations had a little deviation in run time, which highlighted the portability of the standard.

It is worth pointing out that the performance values reported in Table 7.1 must be treated with care. The floating-point counters used in *SeisSol* do not count operations occurring after the projections of volumetric DOFs onto rupture faces (see Eq. 3.45 and Eq. 3.46). This happens because the counting is based on the code generation, which, as mentioned above, is not a part of the dynamic rupture code implementation. Because the inserted timers are kept running, and some floating-point operations are not considered, the obtained performance values (measured in TFLOP/s) are lower than they must be. This is relevant for both the CPU and GPU implementations of *SeisSol*.

Fig. 7.3 shows a difference in run–time between the SYCL implementations of the Linear Slip-Weakening (see Eq. 2.23) and Aging friction (see Eq. 2.22) laws. The Aging law belongs to a class of "State and Rate" models, which, as discussed in Section 3.6.3, define a non-linear system of equations that need to be solved using the Newton-Raphson algorithm. The required number of interactions until convergence of the algorithm strongly depends on the local fault state at each Gaussian point. Therefore, some sub-tasks are more computationally expensive than the others. This highlights the main difference between the "State and Rate" and the Linear Slip-Weakening models, where, in the latter, all sub-tasks are equal and computationally cheap because the slip rate magnitude at each Gaussian point can be directly obtained from Eq. 3.63.

The experiment was conducted using the same computational mesh, the same hardware, and the same implementation of the SYCL standard. The initial conditions for simulating the rupture process subjected to the Aging law were taken from the TPV-101 test scenario [94] and adjusted to the fault geometry used in the test (see Fig. 7.2). For a relative comparison with Table 7.1, the simulations were performed only for the first 1000 time steps. The results show that the Aging law implementation of friction is almost 32% more computationally expensive than the Linear Slip-Weakening one. This number should be treated as an estimate because the rupture process did not fully evolve during the first

1000 steps and, thus, many sub-tasks, in the case of the Aging law, require only a few Newton-Raphson iterations to converge.

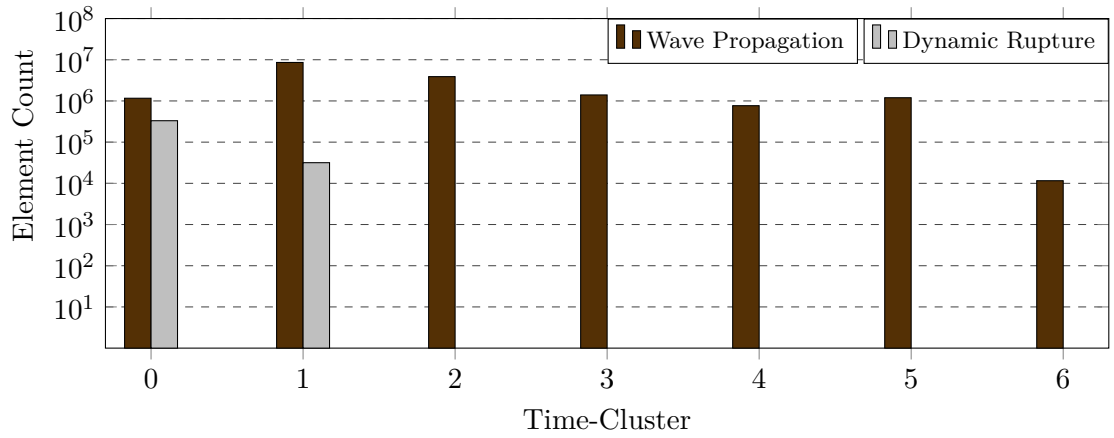
7.3. Strong Scaling

This section aims to demonstrate the strong scaling properties of *SeisSol* when the fracturing in Earth's crust occurs due to slipping rocks along a fault. For consistency with the manuscript, the scaling experiment was conducted using the TPV-5 test scenario on Selene, LUMI, and Leonardo supercomputers. The network characteristics of each machine are shown in Fig. 6.24.

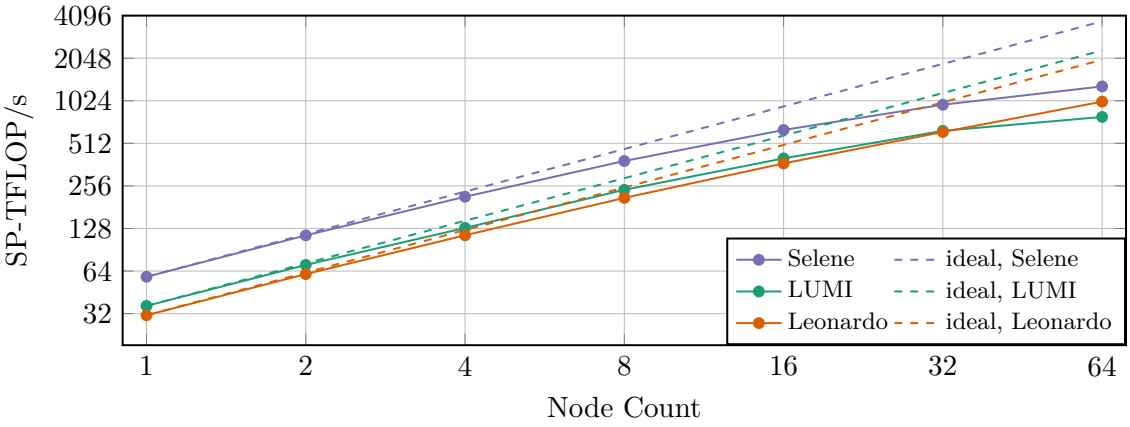
The computational mesh used in the experiment contains 17 million tetrahedrons and 360 thousand rupture elements. It was generated in two steps. Firstly, the first half of the computational domain was built using the Netgen [109] mesh generator, taking advantage of the geometrical symmetry of the scenario along the vertical fault. Netgen resulted in a significantly better mesh quality than the results produced by Gmsh [42], which is traditionally used as a part of *SeisSol*'s workflow. Secondly, the generated half was copied and mirrored along the fault. The resulting LTS clustering is shown in Fig. 7.4a. As can be seen, the wave propagation domain is distributed more or less equally between the first five clusters. The area along the fault consists of elements having approximately the same sizes, which can be deduced from the very tight distribution of dynamic rupture elements between two time-clusters.

As mentioned in Section 6.3.3, the custom-built *Nvidia* A100-C-64 GPUs perform better under low- and medium-sized *SeisSol*'s workloads than the *Nvidia* A100-SXM4-80 and *AMD* MI250x GPUs (see Fig. 6.10). As discussed in Section 6.5.3, the LTS clustering and the computational throughput characteristic of a computational device determine strong scaling properties of the cluster-wise LTS schemes. Because the clustering is fixed in this experiment, one should expect better scaling of the TPV-5 scenario on the Leonardo supercomputer and obtain results similar to the ones shown in Fig. 6.23.

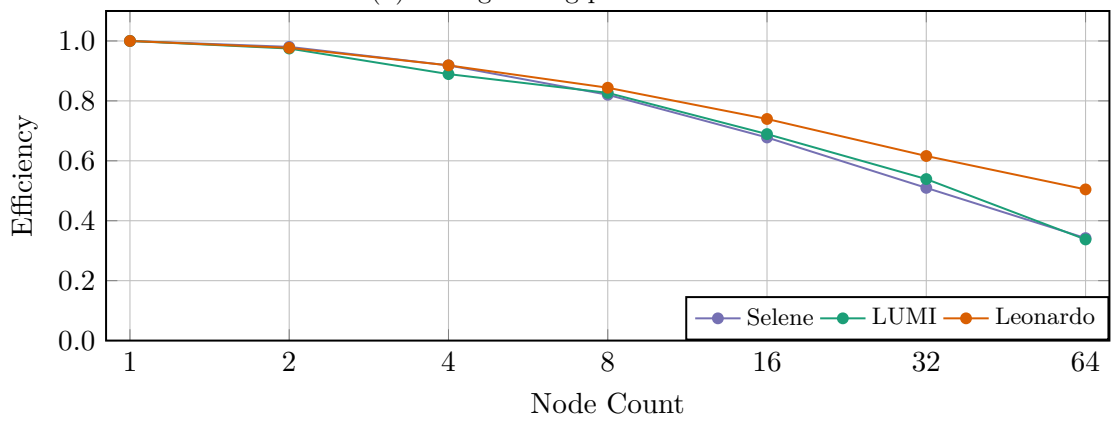
The experiment results are shown in Fig. 7.4 and match the abovementioned hypothesis. As expected, *SeisSol* showed the best parallel efficiency on the Leonardo supercomputer despite having the worst inter-node network characteristics, according to Fig. 6.24. The graph shows that, in the case of Leonardo, the efficiency stayed slightly above 50% while running the TPV-5 scenario on 64 nodes. In two other cases, the efficiency dropped below 35% at the same scale. One can observe that *SeisSol* on the Leonardo supercomputer managed to outperform the same setup on LUMI by approximately 28% at a 64-node scale due to higher parallel efficiency, even though the initial single-node performance obtained on Leonardo was about 5% lower than on LUMI. By extrapolating data onto a 128-node scale, one can imagine that *SeisSol* would keep scaling up on Leonardo and match Selene's configuration for this particular scenario. However, in this case, the difference in the single-node performance between these two machines is even more significant - i.e., approximately 1.9 times.



(a) Resulting LTS clustering for the used mesh of the TPV-5 scenario.



(b) Strong scaling performance.



(c) Parallel efficiency.

Figure 7.4.: Strong scaling of the TPV-5 scenario using 17 million elements mesh on the Selene, Leonardo and LUMI supercomputers.

7.4. Verification

The following summarizes the verification procedure of the proposed dynamic rupture code implementation. In contrast to Section 6.7, there is no an analytical solution that can be used to compare numerical results. However, the CPU version of *SeisSol* is regularly verified against a suite of benchmarks [46] and has been validated against various real events [117, 113, 124], etc. Therefore, the CPU implementation, configured with the double-precision floating-point format, is used as a reference to compare values of physical quantities obtained with the GPU implementation. The verification is conducted for the TPV-5 test scenario after the first 12 seconds of the modeled earthquake event using a computational mesh with approximately one million tetrahedrons in the wave propagation domain and 55 thousand elements along the fault.

In *SeisSol*, physical quantities are computed in absolute units - e.g., m/s, MPa, etc. Therefore, it is more convenient to use the reference error based on the L_2 norm to compare numerical results (see Eq. 7.1).

$$Error^{q}_{L_2} = \frac{||q_{gpu} - q_{cpu}||_2}{||q_{cpu}||_2} \tag{7.1}$$

where q_{gpu} is the reference solution of quantity q obtained with the GPU implementation; q_{cpu} is the numerical solution of physical quantity q obtained with the reference implementation - i.e., CPU.

Table 7.2.: Relative error of the GPU implementation of the dynamic rupture solver obtained for the TPV-5 test scenario. The CPU version of the code configured with the double-precision floating-point format was used as the reference.

Quantity	Single Precision	Double Precision
Slip, $\|\Delta \boldsymbol{d}\|$	2.208e-09	6.856e-17
Slip Rate, $\|\Delta \boldsymbol{u}\|$	5.558e-07	1.569e-10
Rupture Velocity, V_r	3.370e-05	2.344e-06

Quantity	Single Precision	Double Precision
Friction Coefficient, μ	1.016e-11	2.303e-18
Normal Stress, $\|\sigma_n\|$	4.783e-10	2.505e-28
Shear Traction, $\|\tau\|$	2.899e-08	1.964e-14

According to the obtained data, shown in Table 7.2, the numerical results of the CPU and GPU implementations are almost identical. As expected, the relative errors obtained with the single-precision floating-point format are slightly higher but do not exceed $\approx$ 0.01%. This demonstrates the correctness of numerical results and, thus, indirectly proves the correctness of the GPU implementation.

7.5. Discussion

This chapter mainly explains task decomposition and source code portability of the GPU version of *SeisSol*'s dynamic rupture solver. Additionally, the chapter presents the strong

scaling behavior of *SeisSol* on distributed multi-GPU systems using the TPV-5 test scenario. The scenario involves both wave propagation and dynamic rupture solvers, and thus, all contributions presented in this and the previous chapter.

The coupling between the wave propagation and dynamic rupture solver involves a linear transformation that projects a 3D modal basis onto a 2D nodal one. In this work, the coupling is implemented using the *YATeTo* DSL because it is convenient, portable, and efficient. Thus, the task decomposition of this part of the code is similar to the one discussed in Chapter 6.

The rupture solver itself mainly involves pointwise operations, which are data-independent. The task decomposition is trivial in this case. A single block of threads is used to update all Gaussian points encompassed by a single rupture element. Thus, the total number of blocks equals the number of rupture elements in a time-cluster.

As discussed throughout the chapter, non-linear functions are broadly involved in the computations performed by the solver. At the moment of writing, the *YATeTo* DSL does not support such operations. Therefore, I had to find an alternative solution to provide the source code portability and high-performance computing. In this work, multiple implementations of OpenMP and SYCL standards were considered. The experiments show that all tested SYCL implementations result in about the same run time of the rupture solver, which is entirely opposite to OpenMP. Moreover, the designed SYCL implementation of the solver is approximately 1.65 times faster than the fastest OpenMP variant.

The results presented in Fig. 7.4 demonstrate that the strong scaling behavior of the TPV-5 scenario is similar to the one obtained with the LOH.1 test-case. The scaling lines obtained on the LUMI, Selene, and Leonardo supercomputers have the same trend as the one shown in Fig. 6.23. One can notice similarities in the structure of the first LTS time-clusters between these two cases (see Fig. 6.20a and 7.4a), which, as discussed in Section 6.5.3, is the key component determining strong scaling behavior on distributed multi-GPU systems.

8. Implementation of Off-fault Plasticity

As discussed in Section 2.1, the underlying system of PDEs is built on top of the linear stress-strain model. Stress states of some elements can exceed the yielding criterion at the end of a time integration step - i.e., after adding flux contributions from neighboring elements (see Eq. 4.2). The criterion determines the transition of a material from elastic to plastic state. Such situations lead to unrealistic modeling of the wave propagation process.

In Section 2.4, it was shown that plasticity can be simulated by the return-mapping algorithm, which treats stress states obtained with purely elastic material properties as predictors. The correction step determines the amount of plastic deformations, which need to be taken into account to bring the stress states of the affected elements back to the yield surface.

The return-mapping algorithm is often used in *SeisSol* to account for the non-linear behavior of a material. For example, all production scenarios, which will be discussed in Chapter 9, are based on the off-fault plasticity model. The algorithm is not computationally intensive. However, leaving these computations on CPUs would negatively impact the overall performance of *SeisSol* because of frequent data transfers between the host and device. A GPU implementation eliminates this problem and, thus, increases the computation throughput of earthquake simulations on heterogeneous computing systems. This chapter describes only the key problems faced during the development of the GPU version of the off-fault plasticity macro-kernel.

The performance data in this chapter are shown for the single-precision floating-point format and convergence order 6, which is done mainly for illustration purposes.

8.1. Parallelization and Portability

The GPU implementation of the off-plasticity kernels is based on the idea proposed by Wollherr, Gabriel, and Uphoff in [125] - i.e., simulating the yielding process entirely on a nodal basis. Firstly, the solution defined on a modal basis (see Section 3.3) is projected onto a set of three-dimensional nodal points $\boldsymbol{\xi_i}$ (see [125]), equal to the number of basis functions of the modal basis.

As discussed in Section 2.1, a solution inside each element is composed of 6 stress and 3 velocity components. Taking into account the established memory layout in *SeisSol*

(see Section 4.1), a numerical solution can be viewed as a concatenation of the stress $\mathcal{S} \in \mathcal{R}^{\mathcal{B}\times 6}$ and velocity $\mathcal{U} \in \mathcal{R}^{\mathcal{B}\times 6}$ matrices.

$$Q_{lp}^{m} = \left(\begin{array}{c|c} \mathcal{S} & \mathcal{U} \end{array} \right) \tag{8.1}$$

As shown in Section 2.4, modeling of the yielding process operates only on the stress components. Thus, the change of basis can be implemented as follows

$$\hat{\mathcal{S}}_{ip}^{m} = \Psi_{l}(\boldsymbol{\xi_i})\mathcal{S}_{lp}^{m} = \hat{\mathcal{V}}_{li}\mathcal{S}_{lp}^{m} \tag{8.2}$$

where $\mathcal{S}_{lp}^{m}$ is the stress components of the m-the element on a modal basis; $\hat{\mathcal{S}}_{ip}^{m}$ is the stress components of the m-the element on a nodal basis; $\hat{\mathcal{V}}_{li}$ is the Vandermonde matrix.

The outcome of Eq. 8.2 determines values of the yield function $F(\boldsymbol{\sigma})$ at all nodal points inside each tetrahedron. The values obtained using Eq. 2.28 help to determine points with internal stress states outside the yield surface - i.e., $F(\boldsymbol{\sigma}) > 0$. For these points, the required plastic strain increments can be found using Eq. 2.33. The increments are used to correct the internal stress states of the affected points according to Eq. 2.32 and, thus, return them back to the yield surface - i.e, $F(\boldsymbol{\sigma}) \leq 0$. The corrected stress states are brought back to the modal basis by applying the inverse Vandermonde matrix, which, together with the Vandermonde matrix, is pre-computed in advance.

$$\mathcal{S}_{lp}^{m} = \hat{\mathcal{V}}_{li}^{-1}\hat{\mathcal{S}}_{ip}^{m} \tag{8.3}$$

Eq. 8.2 and Eq. 8.3 define simple tensor expressions; therefore, their corresponding GPU kernels were implemented using the *YATeTo* DSL. However, as can be seen from Eq. 2.28, 2.34, and 2.33, checking the yield criterion and the correction step involve non-linear and point-wise functions, which are not supported by *YATeTo* and, thus, had to be implemented conventionally - i.e., without code generation.

In *SeisSol*, a sub-task of the return mapping algorithm is defined as the adjustment of the stress state of a single element in a given LTS time-cluster. In the GPU implementation, I assign a team of threads equal to the number of nodal points to process each sub-task. This can be also viewed as using a single GPU thread per nodal point. The sub-tasks are unequal because the correction step and the transformation back to the modal basis must be applied only to those elements that satisfy $F(\boldsymbol{\sigma}) > 0$ condition, which is known only during run time.

A naïve implementation would require data rearrangements involving complex parallel reduction algorithms to process the elements outside the yield surface. The implementation proposed in this work avoids it by writing the results of the yield condition checks into a vector of booleans. The vector is passed to subsequent kernels, allowing sub-tasks to execute the GPU code conditionally. This scenario forced me to extend the *YATeTo* DSL and the GPU kernel generators to support conditional batch execution. An example can be seen in lines 8, 9 and 10 of Listing 6.2, where each team of threads decides whether to execute a kernel or skip it depending on the array values associated with each team.

At first glance, it is desirable to use the SYCL programming model to implement device kernels for Eq. 2.28, 2.34, 2.33, etc. This would result in a portable solution that would be

similar to the dynamic rupture solver discussed in Section 7.2. As mentioned above, it is also preferable to utilize the device kernels generated by the *YATeTo* DSL. Moreover, some generic algorithms from the *Device API* could also be used to initialize temporary memory buffers required for intermediate results and manipulate device memory through pointers. However, each change in the programming model - i.e., from SYCL to CUDA/HIP and vice versa - would require synchronizing the entire device to preserve the correct execution order. Otherwise, works submitted to a SYCL device queue, and the default *Device API* stream would run concurrently, resulting in race conditions and, thus, leading to data corruptions. Such behavior results from the SYCL device queue construction, which implicitly allocates and associates a GPU stream with a queue. The stream is used internally by a standard implementation to run submitted tasks; therefore, it is not exposed to the user. Thus, the only way to impose correct execution order between tasks submitted to a CUDA/HIP stream and a SYCL queue is through global device synchronization.

In this work, I used an alternative solution and implemented device kernels for Eq. 2.28, 2.34, 2.33, etc., separately for each supported GPU programming model in *SeisSol* - i.e., CUDA, HIP, and SYCL. This approach helped to avoid excessive device synchronizations caused by the need to change programming models too frequently. All tasks are submitted to the same GPU stream (or queue) and, thus, executed in order. The downside of this approach is that a mistake found in one implementation must to be manually corrected in two others, increasing the cost of source code maintenance.

8.2. Verification and Comparison

The GPU implementation of the return mapping algorithm is verified using the TPV-13 test scenario taken from the SCEC benchmark collection [47]. The wave propagation domain is a cuboid with homogeneous material properties - i.e., $\rho = 2700\, kg/m^3$, $\mu \approx \lambda \approx 29.40\, GPa$. As shown in Fig. 8.1, the depth is equal to 42 km, whereas, the length and width are almost 72 km long. Similar to the TPV-5 scenario, the top of the domain represents a free surface; the rest act as absorbing boundaries.

The fault surface is 30 km wide, 15 km deep, and inclined at 60° downward from the horizontal line. The scenario includes 3×3 km nucleation patch residing on the fault plane. The patch is located at $(0, 12)$ km in the fault coordinates. In contrast to the TPV-5, in this scenario, the nucleation is achieved by decreasing the static friction coefficient, which makes the initial shear stress exceed the fault strength. The details of the TPV-13 scenario regarding the initial conditions and fault parameters are given in [95].

Similar to the TPV-5, the TPV-13 test scenario operates on the Linear Slip-Weakening friction law (see Eq. 2.23)) and homogeneous parameters along the fault. However, in this case, the static and dynamic friction coefficients are equal to 0.70 and 0.10, respectively. The critical distance is set to 0.5. The cohesion coefficient is negative, with the absolute value equal to 200000.

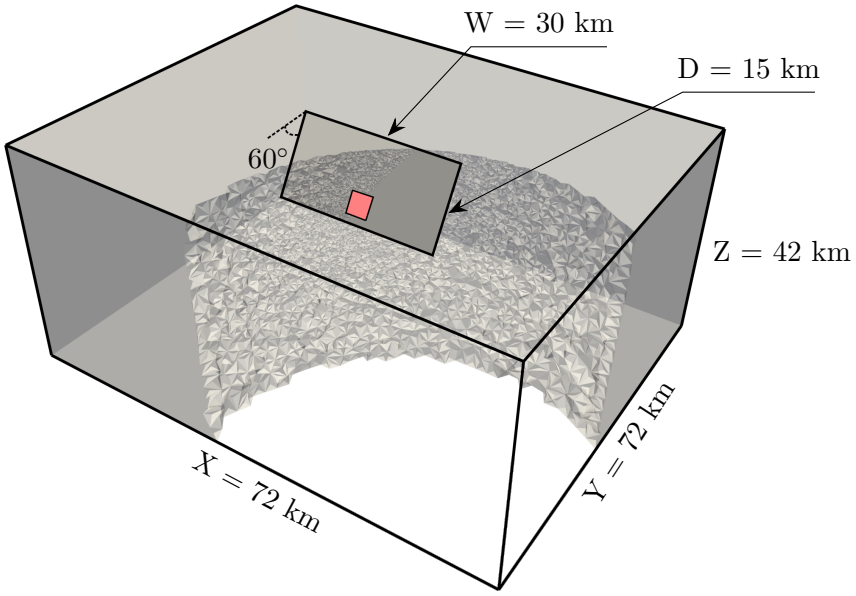


Figure 8.1.: Geometry and mesh refinement of the TPV-13 test scenario.

Table 8.1.: Relative error of the GPU implementation of the dynamic rupture solver obtained for the TPV-13 test scenario. The CPU version of the code configured with the double-precision floating-point format was used as the reference.

Quantity	Single Precision	Double Precision
Slip, $\lvert\Delta \boldsymbol{d}\rvert$	1.361e-08	6.717e-16
Slip Rate, $\lvert\Delta \boldsymbol{u}\rvert$	0.003e-00	1.429e-12
Rupture Velocity, V_r	1.717e-05	1.049e-07

Quantity	Single Precision	Double Precision
Friction Coefficient, μ	6.999e-05	9.398e-13
Normal Stress, $\lvert\sigma_n\rvert$	6.087e-05	1.720e-12
Shear Traction, $\lvert\tau\rvert$	7.154e-05	5.607e-13

Table 8.1 shows the relative errors of numerical results obtained with the GPU implementation. The presented data were obtained following the methodology described in Section 7.4. In contrast to the TPV-5 benchmark, the largest error obtained with the single-precision floating-point format is just slightly above 0.3%. This proves that the proposed GPU implementation of the off-fault plasticity model matches the CPU one, which, as mentioned in Section 7.4, has been validated against other seismic benchmarks and applications.

Fig. 8.2 depicts the evolution of the velocity magnitude captured by a point receiver near the fault - i.e., 3 *km* away from the center of the fault plane along the direction toward the free surface. One can clearly observe that the results obtained with the off-fault plasticity model are smoother. The sudden jumps in velocity at 3.69 and 4.37 seconds after the beginning of the simulation are noticeably dumped. After the second jump, the velocity starts gradually vanishing to zero. However, it stays at a slightly higher level in comparison to the velocity profile obtained with the pure elastic model. The observed

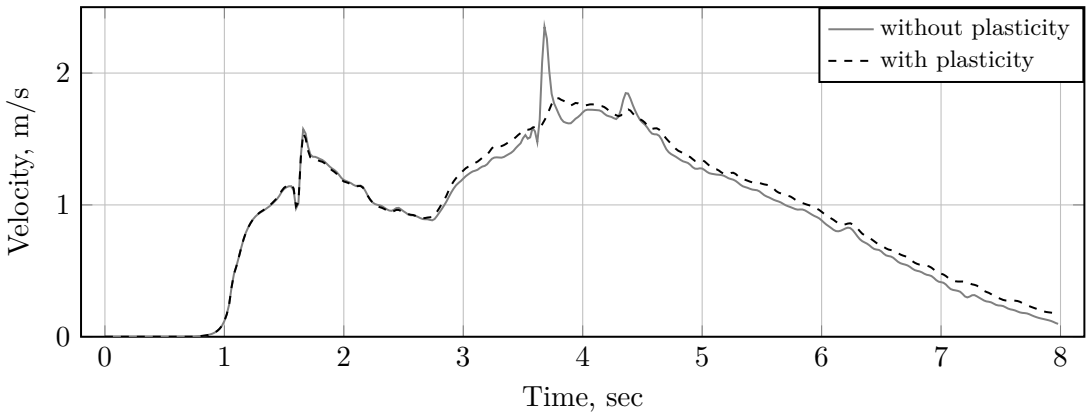


Figure 8.2.: Comparison of the velocity magnitudes with and without the off-fault plasticity model at a receiver located 3 *km* away from the center of the fault plane along the normal direction toward the free surface (see Fig. 8.1).

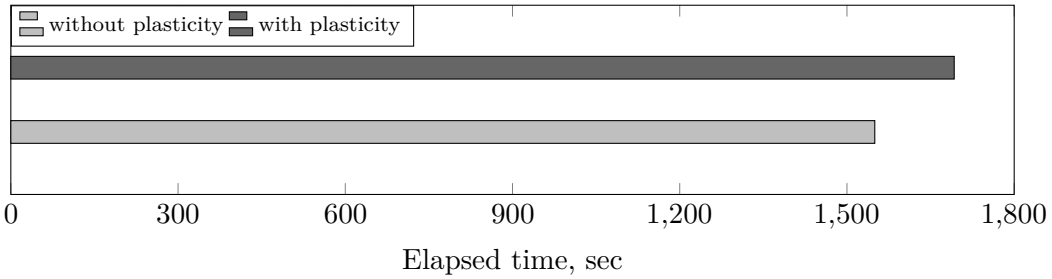


Figure 8.3.: Comparisons of the elapsed time of the TPV-13 test scenario with and without the off-fault plasticity model on a single *Nvidia* A100-PCIE-40GB GPU.

behavior is analogous to the results presented in [125], focused on simulating the wave propagation process in the viscoplastic media.

While collecting data for Fig. 8.2, the run time of two simulations was also tracked. The results are shown in Fig. 8.3, where one can see that the off-fault plasticity macro-kernel increases the time-to-solution by approximately 9.2%. Meanwhile, the performance measured on a single Nvidia A100-PCIE-40GB GPU dropped only by about 4.5% - i.e., from 6.08 to 5.81 SP-TFLOP/s. The overheads measured in this experiment match the ones reported in [125], which was performed using the CPU version of *SeisSol*.

8.3. Discussion

The chapter reveals the implementation details of the return-mapping method in the DG formulation adapted for heterogeneous computing. The task decomposition proposed in this part of the study is similar to the one discussed in Chapter 7 - i.e., a single block of threads is assigned to process all Gaussian points inside a mesh element. However,

in contrast to the dynamic rupture algorithm, the return-mapping method operates on a 3D nodal basis, which naturally requires more Gaussian points to preserve the same polynomial strength within a mesh element. Thus, the method operates on larger thread blocks.

As mentioned at the beginning of the section, the method adds more realistic behavior to the wave propagation media. Fig. 8.3 clearly shows that the method is not computationally expensive; the overall workload increases only by 4.5%. Taking into account the advantages of the method - i.e., more accurate results of numerical earthquake simulations - the increased computational cost is marginal.

At this point, all three major components of *SeisSol* are well-adapted for heterogeneous computing. All parallel tasks inside the main computational loop are performed on a device; thus, no extra host-to-device data transfers are required (except for writing intermediate results to disks). Thus, the host system only submits computational tasks to a device and tries to make it as busy as possible during the entire earthquake simulation process. This scenario is desirable for any heterogeneous program. However, it may be challenging to accomplish it in practice for certain scientific applications. In this study, I managed to achieve it and, thus, maximize the overall computational throughput of *SeisSol*.

9. Numerical Simulations and Supercomputing

This chapter demonstrates three complete production earthquake simulations using the GPU version of *SeisSol*, which contains all contributions presented in this work. All simulations involve elastic wave propagation, dynamic rupture, off-fault plasticity models, as well as heterogeneous material properties. As discussed in Section 8.3, all computations are adapted for heterogeneous computing. Thus, the bulk of computations are performed by accelerators without host-to-device data transfers inside the main computational loop, except for the data output operations.

The first scenario is the 2023 Kahramanmaraş earthquake (shown in Section 9.1), which models friction using the Linear Slip-Weakening law. The second one is the 2019 Ridgecrest earthquake (shown in Section 9.2), which incorporates the Fast Velocity Weakening friction law - i.e., a member of the "State and Rate" family. The last scenario is the 2018 Sulawesi earthquake, which, as in the previous case, involves the Fast Velocity Weakening friction law and, most importantly, a fully 3D coupled tsunami model. The computations required for tsunami modeling were also adapted for heterogeneous computing and considered yet another contribution of this work. The numerical results of the 2018 Sulawesi earthquake and tsunami are presented in Section 9.3.

The 3D geometries with incorporated topography data, initial and boundary conditions, material properties, etc., of all discussed earthquake models are taken from the scientific artifacts of works [112, 68, 57, 108]. For consistency with the entire manuscript, the convergence order is chosen to equal 6 for all simulations. The numerical results and performance data were obtained on the LUMI and Leonarod supercomputers. Due to the project budget constraints on both supercomputers, the termination criteria of all simulations - i.e., the abortion time - were reduced.

9.1. 2023 Kahramanmaraş Earthquake

The first M_w 7.8 earthquake struck southern and central Turkey as well as northern and western parts of Syria on the 6th of February, 2023. As shown in Fig. 9.1, it happened on the Nurdağı-Pazarcık Fault (NPF), which branches from about 700 km long left-lateral strike-slip East Anatolian Fault (EAF). After approximately 9 hours, another M_w 7.6 earthquake occurred about 90 km away toward the northwest from the epicenter of the first one, on the Çardak fault. Seismic waves generated by the earthquakes propagated

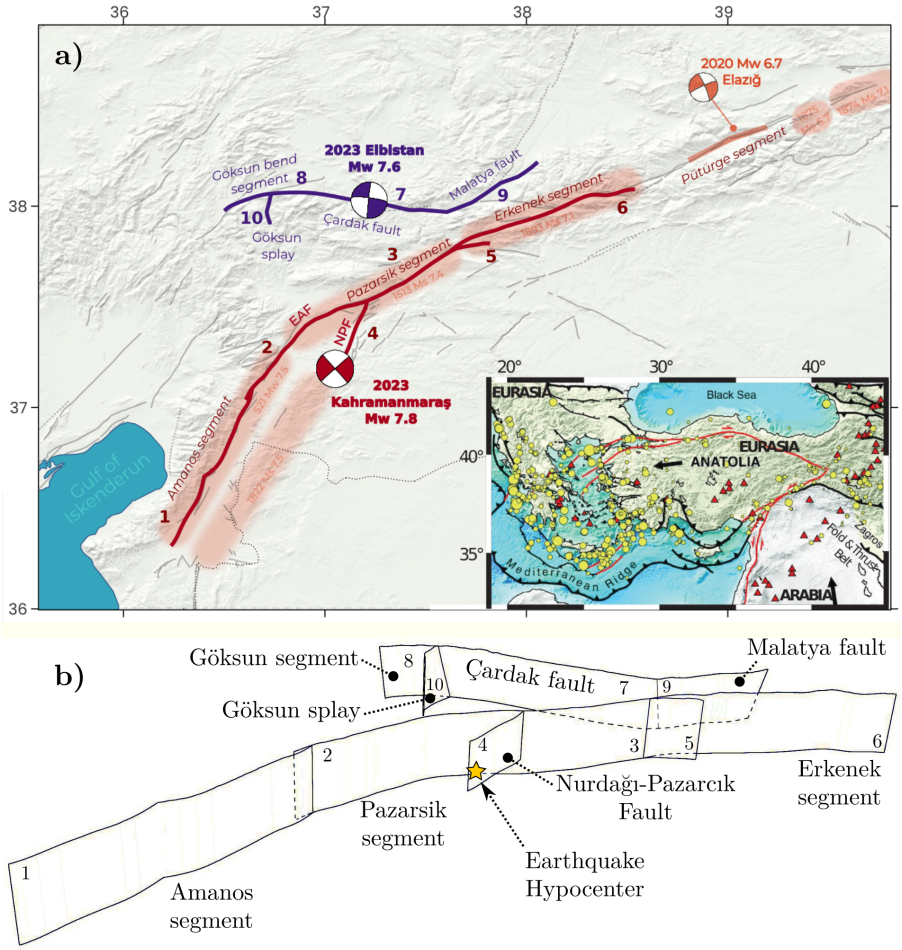


Figure 9.1.: Regional tectonic map around the Kahramanmaras region and the 3D model of the fault system. The map was taken from [57].

up to the surface and intensified shaking in the area surrounded by the fault systems. According to [20], the rupture produced by the mainshock extended over approximately 300 km, resulting in surface displacements of up to 5 m. The second quake caused a shorter rupture (i.e., about 100 km long) but led to larger land displacements of up to 7–8 m.

The consequences of the earthquake doublet were devastating. The official death toll in Turkey and Syria exceeded 57 thousand people because the affected areas were densely populated. Numerous smaller aftershocks of varying magnitudes continued to shake the region, further damaging the affected buildings and, thus, worsening the rescue operations. The first M_w 7.8 event is considered the most powerful earthquake recorded in Turkey since 1939.

As shown in Fig. 9.1, the fault geometry comprises ten curved, intersecting segments dipping between between 90 degrees along EAF and 70 degrees along the Çardak fault. The average depth of the fault system is approximately 20 km.

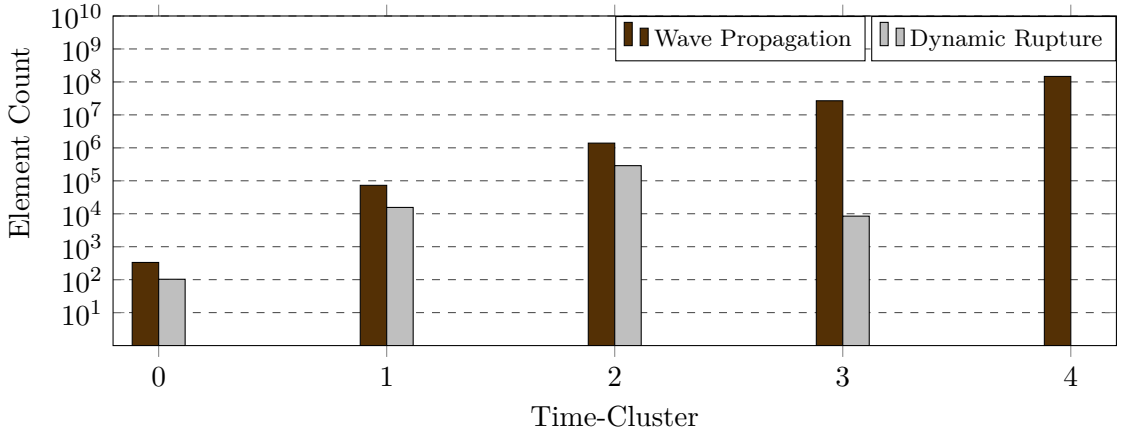


Figure 9.2.: LTS clustering of the Turkey computational mesh, consisting of approximately 175 million tetrahedrons and about 300 thousand rupture elements.

The earthquake sequence resulted in unexpected ruptures across the fault segments. The numerical simulation presented in this section is based on the initial conditions, fault geometry, relative fault strength, prestress, and material properties taken from the artifacts of work [57]. According to [57], the simulation parameters were retrieved from 1) geodesy and seismicity, 2) regional seismo-tectonics, 3) static slip inversion, and 4) earthquake kinematics. The scenario incorporates the Linear Slip-Weakening friction law (see Eq. 2.23) with constant static and dynamic friction coefficients on all faults. The critical slip distance on segments 7, 8 and 9 (see Fig. 9.1) is set to a higher value than for the other parts to impose larger fracture energy for the main faults hosting the second event.

Numerical results presented in this section were obtained with a well-refined computational mesh containing approximately 175 million tetrahedrons and around 300 thousand rupture elements - i.e., about 5.6 times larger compared to the one used in [57]. The LTS clustering is shown in Fig. 9.2. As can be observed, most of the mesh elements belong to the last time-cluster. That is approximately 147 million tetrahedrons or, in other words, 84% of all mesh elements. The last two clusters together comprise almost 99% of the entire mesh. The LTS distribution of dynamic rupture and wave propagation elements between the first three time-clusters indicates that the average element size sharply grows from the fault surfaces to the bulk of the computational domain. Such meshing is understandable from the user's perspective since it is supposed to significantly reduce time-to-solution. However, according to the experiments shown in Section 6.5.3, this approach substantially limits the strong scaling capabilities of *SeisSol* on distributed multi-GPU systems. As an example, parallel efficiency of the LOH.1 LTS Type 1 scenario (see Fig. 6.32), which contains 70% of all elements in the last two time-clusters, reaches only about 40% on 64 nodes of the LUMI and Selene supercomputers.

The simulation was performed on the LUMI supercomputer, using 512 *AMD* MI250x GPUs and 1024 MPI processes. The numerical results were obtained using the single-precision floating-point because the LSW friction law involves simple arithmetical operations (i.e., subtraction, division, min) and, thus, is numerically stable. In this work, only the first 48

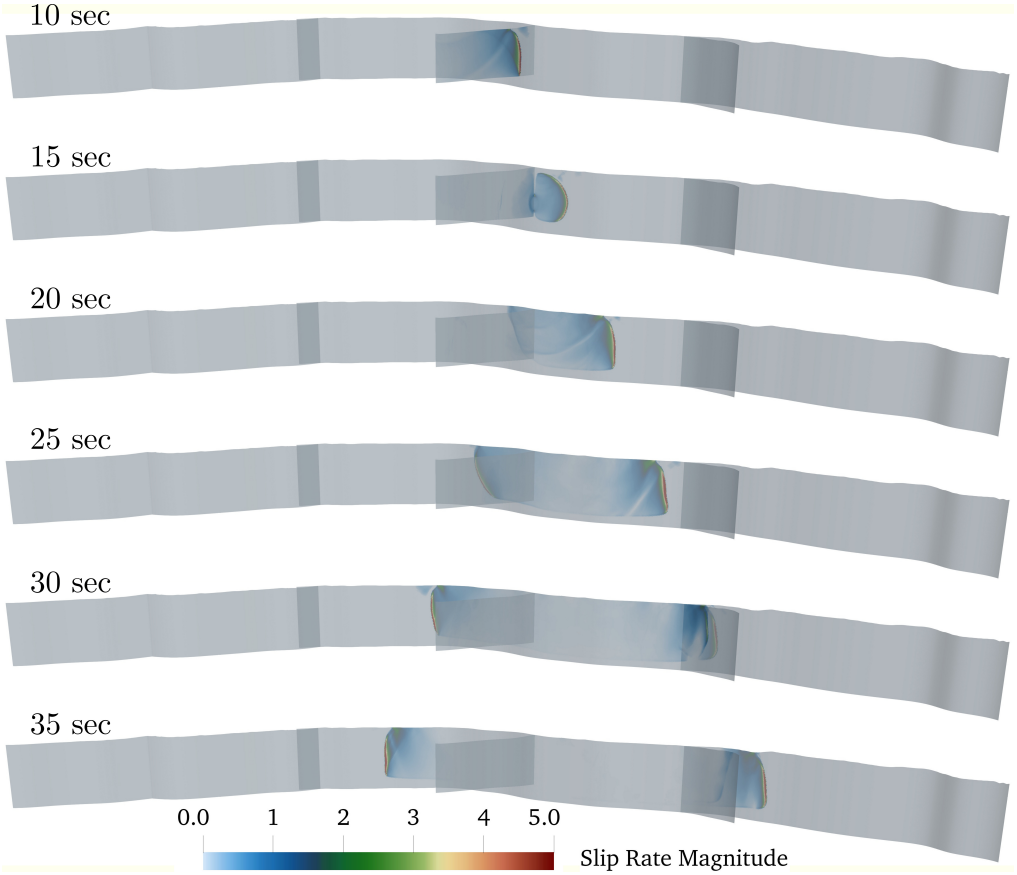


Figure 9.3.: Snapshots of the absolute slip rate along the East Anatolian Fault obtained during a numerical simulation of the 2023 Kahramanmaraş earthquake.

seconds of the 2023 Kahramanmaraş earthquake were simulated, which was enough to capture the rupture branching at the NPF-EAF junction. The simulation took 1 hour and 13 minutes and resulted in 1.52 SP-PFLOP/s. The parallel efficiency, estimated using Fig. 7.4, slightly exceeded 35%.

Simulation results are shown in Fig. 9.3, which demonstrates the rupture propagation along the East Anatolian Fault. In 12 seconds after the beginning of the simulation, the rupture reaches the NPF-EAF junction, which is almost 30 km away from the earthquake hypocenter. Then, the rapture branches and starts moving to the opposite sides along the Pazarcik segment - i.e., toward 1) the south of Turkey and north-west of Syria, and 2) the center of Turkey. In 15 seconds after the branching, the rupture reaches the left tip of the Erkenek fault, which is 46 km away from the junction. As can be seen, the rupture front propagates almost 1.7 times faster along the northwest direction than toward the southwest. According to the simulation results, the average displacement on the free surface is about 3-4 m along the EAF fault. The propagation patterns and rupture behavior closely match the results reported in [57], which can be found in the supplementary materials attached to that publication. Unfortunately, the provided initial conditions and material parameters did not result in triggering the second M_w 7.7 earthquake event on the Çardak fault.

9.2. 2019 Ridgecrest Earthquake Sequence

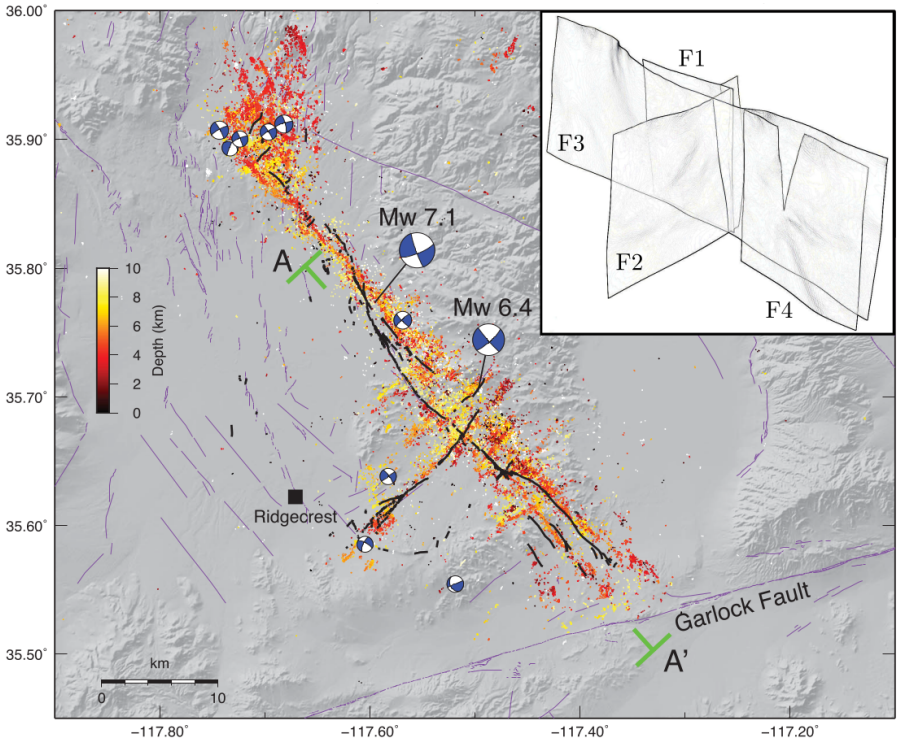


Figure 9.4.: Seismic activity around the Ridgecrest region and the 3D model of the fault system. The map was taken from [99].

The Ridgecrest scenario models a sequence of earthquakes that occurred in California on the 4th and 5th of July, 2019. The sequence included M_w 6.4 Searles Valley foreshock followed by M_w 7.1 Ridgecrest mainshock. The mainshock has been the biggest earthquake event captured in southern California since the 1999 Hector Mine earthquake [128]. The total economic loss was estimated at approximately 4-5 billion dollars according to [19]. Fortunately, the earthquakes caused no serious injuries and fatalities because the most severe ground motion occurred far away from densely populated regions. Most of the damage occurred at the China Lake Naval Air Weapons Station, where approximately 230 buildings were heavily affected.

The 3D fault system, shown in Fig. 9.4, was constructed using the radar images collected from orbiting satellites, relocated seismicity, and selected focal mechanisms [108]. The system contains four quasi-orthogonal intersecting fault segments, referred to as F1, F2, F3, and F4. Segments F1, F3, and F4 are right-lateral faults trending from northwest to southeast, whereas F2 is left-lateral, trending from northeast to southwest. The largest segment (i.e., F3) stretches for approximately 45 km. Its dip angle changes between 80 degrees to the southwest and 70 degrees to the northeast along the fault. In the southeast, the F3 fault branches into two subparallel strands separated approximately 7 km apart. Each strand keeps trending toward the southeast for about 12 km. The F2 segment is

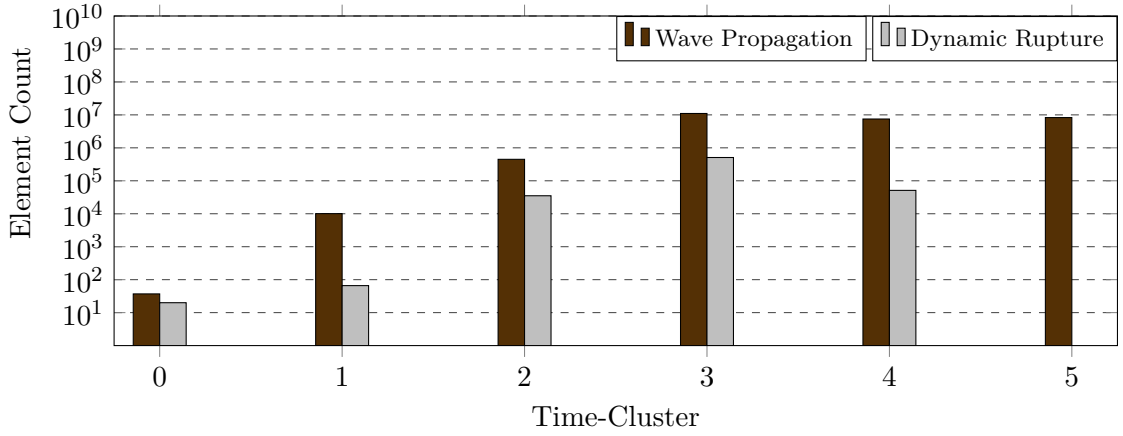


Figure 9.5.: LTS clustering of the Ridgecrest computational mesh, consisting of approximately 27 million tetrahedrons and about 600 thousand rupture elements.

almost orthogonal to F3 and stretches about 20 *km* long. The average depth of the system is approximately 15 *km*.

A deep rupture across the F1 segment activates the conjugate left-lateral fault F2, which is critically prestressed. After a while, the rupture starts propagating across both faults, with a higher speed along F1. In a few seconds after that, the rupture process spontaneously stops along the F1 segment before reaching the surface. However, the process keeps evolving on the F2 fault, moving toward the southwest and up. In the end, the rupture eventually reaches the surface and breaks it. As mentioned by Taufiqurrahman et al. in [108], the rupture along the F2 segment can be described as a narrow slip pulse, which re-accumulates a significant amount of shear stresses. The re-accumulated stresses contribute to subsequent reactivations during the mainshock. Detailed descriptions of the earthquake dynamics of the Ridgecrest sequence can be found in [99, 55, 128, 108].

In this study, computational mesh, initial, and boundary conditions are taken from the updated artifacts[1] of work [108], where the authors aimed to find a link between the shocks of the Ridgecrest earthquake sequence using the CPU version of *SeisSol*. The computational mesh comprises approximately 27 million tetrahedrons and about 600 thousand rupture elements. The LTS clustering of the mesh is depicted in Fig. 9.5, showing that approximately 98% of all elements are in the last three time-clusters. This clustering is slightly better compared to the one resulting from the 2023 Kahramanmaras earthquake simulation discussed in Section 9.1. However, the same reasoning suggests that the Ridgecrest scenario with this particular mesh will not scale well on muli-GPU systems.

[1] https://zenodo.org/record/7352554

The scenario is based on the Fast Velocity Weakening (FVW) friction law, which belongs to the "State and Rate" family, similar to the Aging friction law, and given by

$$\begin{cases} \mu_f = a \sinh^{-1}\left[\frac{U}{2U_0}\exp\left(\frac{\psi}{a}\right)\right] \\ \frac{d\psi}{dt} = -\frac{U}{L}\left(\psi - a\ln\left[\frac{2U_0}{U}\sinh\left(\frac{\mu_{ss}(U)}{a}\right)\right]\right) \end{cases} \tag{9.1}$$

where μ_{ss} coefficient is given by

$$\mu_{ss}(U) = \mu_\omega + \frac{f_0 - (b-a)\ln\left(\frac{U}{U_0}\right) - \mu_\omega}{\left(1 + \left[\frac{U}{U_\omega}\right]^8\right)^{1/8}} \tag{9.2}$$

where f_0 is the reference friction coefficient; μ_ω is the weakening friction coefficient; U_ω is the weakening slip velocity. The rest of the coefficients are the same as for Eq. 2.22.

As can be seen from Eq. 2.22, computations stemming from the FVW friction law involve non-linear functions such as exp, sinh, ln. The use of the single-precision floating-point format can result in noticeable truncation errors. Thus, the double-precision format is used for the Ridgecrest earthquake simulation to prevent fast-growing accumulated numerical errors, which may eventually lead to the divergence of a numerical solution. The simulation was conducted on the Leonardo supercomputer. The first 40 seconds of the earthquake simulation took 6 hours and 1 minute while using 128 *Nvidia* A100-C-64 GPUs - i.e., 32 GPU nodes. The accumulated performance reached approximately 0.22 DP-PFLOP/s.

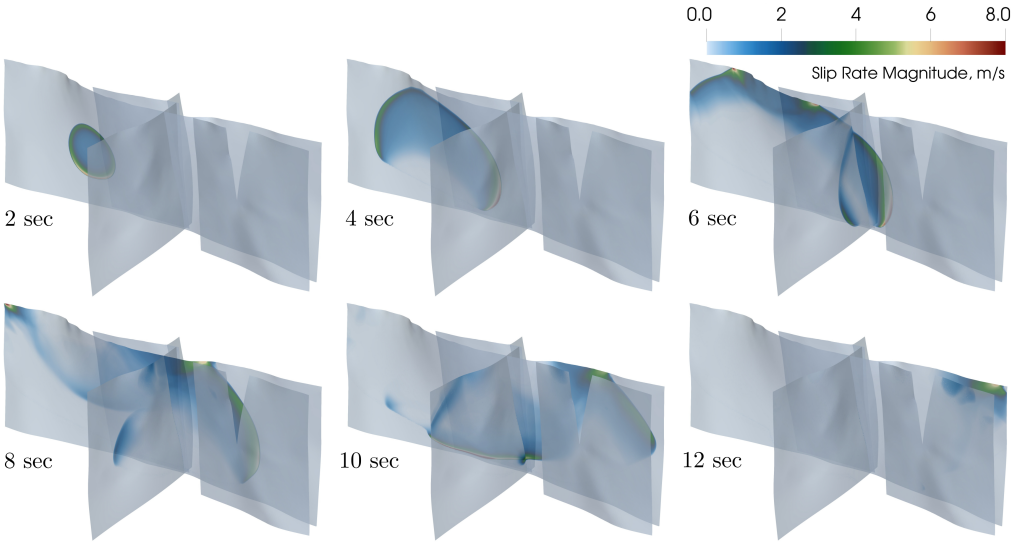


Figure 9.6.: Snapshots of the absolute slip rate along the Ridgecrest fault system.

Simulation results are shown in Fig. 9.6, which demonstrates the rupture propagation along the Ridgecrest fault system. In [108], Taufiqurrahman et al. modified *SeisSol*'s

source code to impose the second stress nucleation region around the hypocenter of the mainshock at a specific point in time. That approach allowed the authors to model triggering the mainshock by the foreshock. The modified version from [108] dates 21 October 2021 and, thus, does not include the most recent and essential changes in *SeisSol*'s source code, including the GPU implementation of the FVW friction law. Because the most recent version of *SeisSol*, at the moment of writing, was used for that study, I only managed to reproduce wave and rupture propagation caused by the mainshock.

The first snapshot clearly shows the location of the earthquake hypocenter. After 4 seconds of the simulation, one can observe that the rupture mainly propagates toward the northwest, southeast, and up. After one more second, the rupture front reaches the free surface in the northwest and passes over the junction of the F2-F3 segments, located approximately 12 km away toward the southeast from the hypocenter. In the top right-most snapshot, one can also observe a rupture front along the F2 segment propagating toward the southwest and trending upwards, as shown in subsequent snapshots. After 8 seconds, the rupture front starts reaching the free surface in the southeastern part of the F3 segment. At the same time, one can notice a reflected rupture wave in the northwestern part of the F3 segment moving downwards. The reflected wave hits the bottom of the segment after 10 seconds of the simulation. According to the simulation results, the average displacement on the free surface reached approximately 4 m along the fault.

9.3. 2018 Palu, Sulawesi Earthquake and Tsunami

On September 28, 2018, a M_w 7.5 strike-slip earthquake struck Donggala Regency - i.e., a region in the Central Sulawesi Province, Indonesia. The event occurred due to a tectonic movement of the left lateral Palu-Koro fault within the Molucca Sea microplate [48]. The earthquake ruptured a 180 km long section of the fault at a very high speed, exceeding the shear wave velocity - i.e., a super-shear earthquake. The event triggered a local but devastating tsunami that hit Palu Bay, surrounded by the provincial capital, Palu City, and its neighborhood. According to [90], the maximum flow depth and height reached 8 m and 10 m, respectively. As shown in Fig. 9.7, the earthquake epicenter was approximately 70 km north of the capital. The hypo-central depth was estimated around 20 km [48].

Destructive flows of mud and soil destroyed large residential areas adjoined to the coastline. According to the National Disaster Management Authority, the death toll caused by the earthquake and tsunami exceeded 3 thousand people; 4 thousand were injured. More than 70 thousand houses and nearly 3 thousand schools were damaged [107].

The 3D fault model, taken from work [112], was reconstructed using focal mechanisms and geodetic data and includes three major intersecting segments of the Palu-Koro fault (see Fig. 9.7) - i.e., Northern, Palu, and Saluki. The Northern segment stretches about 86 km long from north to south, dipping 65 degrees toward the east. The Palu segment is approximately 79 km long, stretching from northwest to southeast, and has the same dip

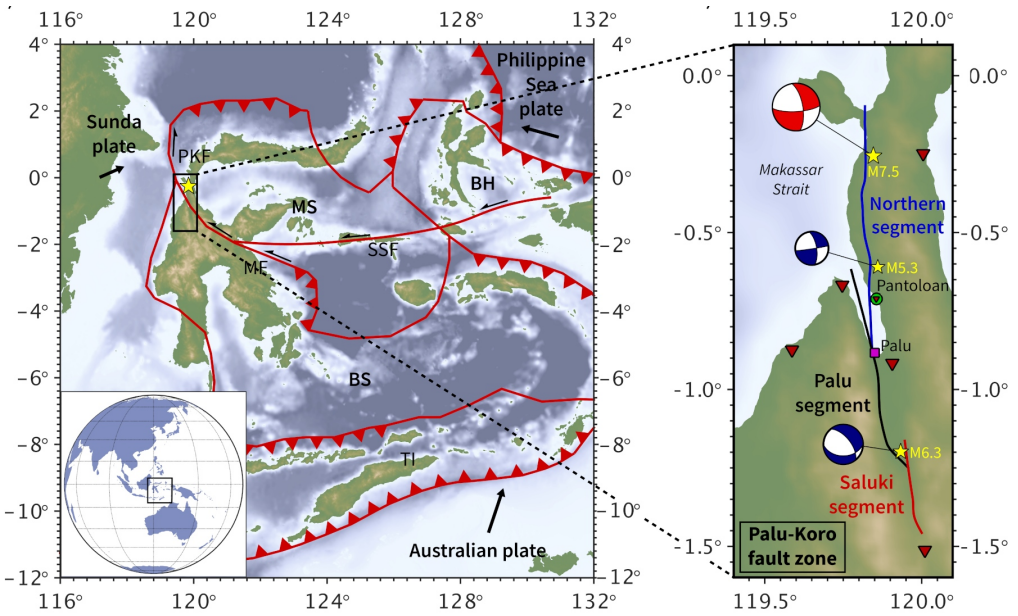


Figure 9.7.: Tectonic setting of the 2018 Palu, Sulawesi earthquake and its epicenter. The zoomed region displays the area of interest, focusing on the Northern, Palu, and Saluki segments. The following abbreviations are used: BH – Bird's Head plate; BS – Banda Sea plate; MF – Matano fault zone; PKF – Palu-Koro fault zone; MS – Molucca Sea plate; SSF – Sula-Sorong fault zone; TI – Timor plate. The image is taken from [112].

direction as the Northern segment. The Saluki segment is vertical, trending from north to south.

The numerical simulation presented in this section is based on the initial and boundary conditions, fault geometry, and material properties taken from the artifacts[2] of works [112] and [68]. Ulrich in [112] describes that a highly overstressed circular patch with a radius of 1.5 *km* is placed at the hypocenter to initiate fault failure for this earthquake-tsunami model. The scenario incorporates the Fast Velocity Weakening friction law (see Eq. 9.1). Apart from the rupture dynamics and seismic wave propagation, the model also includes tsunami propagation. The fundamental concepts of tsunami modeling and the corresponding contributions are briefly explained in the following.

Following work [78], the ocean dynamics in *SeisSol* is modeled as wave propagation in the ocean. The model prescribes a free surface upper boundary condition in the presence of a gravitational field. The governing equations describe water motions as small perturbations about a rest state in hydrostatic equilibrium. The equations are based on the conservation of mass and momentum of fluid, which are linearized about the hydrostatic state. It is

[2] https://zenodo.org/record/5159333

given by

$$\begin{cases} \frac{\partial p'}{\partial t} + K\frac{\partial u_i'}{\partial x_i} = \rho g u_3' \\ \rho\frac{\partial u_i'}{\partial t} + \frac{\partial p}{\partial x_i} = -\delta_{i2}\frac{\rho g}{K}p' \end{cases} \tag{9.3}$$

where K is the constant bulk modulus of the fluid; g is the gravitational acceleration equal to 9.81 m/s^2; ρ is the fluid density; $p'(\boldsymbol{x}, t)$ and $u_i'(\boldsymbol{x}, t)$ are perturbations of pressure and velocities, respectively.

System 9.3 is referred to as the acoustic wave equation, expressed in velocity-pressure formulations. It can be written in the form of Eq. 2.8. According to [78], the right-hand side can be equated to zero because of its small magnitude, which can be neglected. The system operates with four unknowns, namely: p', u_1', u_2', u_3'. The space-dependent Jacobian matrices for the system can be written as

$$A_{qp}^{acc} = \begin{pmatrix} 0 & K & 0 & 0 \\ \frac{1}{\rho} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}, B_{qp}^{acc} = \begin{pmatrix} 0 & 0 & K & 0 \\ 0 & 0 & 0 & 0 \\ \frac{1}{\rho} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}, C_{qp}^{acc} = \begin{pmatrix} 0 & 0 & 0 & K \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ \frac{1}{\rho} & 0 & 0 & 0 \end{pmatrix} \tag{9.4}$$

It can be analytically shown, that the eigenvalues of the Jacobian matrices are equal to $(-c, 0, c)$, where c is given by $\sqrt{K/\rho}$. Thus, System 9.3 is also hyperbolic. Therefore, it can be numerically solved using the ADER-DG method as shown in Chapter 3. In [68], Krenz et al. embedded the unknowns of System 9.3 into the same vector of quantities as for the elastic wave propagation problem using Eq. 9.5.

$$\begin{aligned} \sigma_{11} = \sigma_{22} = \sigma_{33} = -p \\ \sigma_{12} = \sigma_{23} = \sigma_{13} = 0 \end{aligned} \tag{9.5}$$

That approach made both systems of PDEs structurally look the same, with the only difference in the choice of flux matrices and boundary conditions. Despite introducing additional computational overheads, it allowed the authors to preserve the same data layouts for both implementations and, thus, reuse the same computational kernels. In this work, I stick to the same approach.

The seafloor represents the interface between the elastic and acoustic domains, which requires coupling. In *SeisSol*, it is established through the structure of the Riemann problem at the interface boundaries. The derivation is similar to Eq. 3.6 - 3.7 but takes into account a different set of the right-going waves. Eventually, the flux between the boundaries takes the form of Eq. 3.11 (see details in [68]). This means that the coupling is seamless from the algorithmic point of view and, thus, does not demand any additional CPU or GPU code. The implementation only requires initializing element local flux matrices appropriately.

The free ocean surface needs to be constrained through boundary conditions to close System 9.3. In this model, the atmospheric pressure p_a is imposed on the free surface, which is mathematically translated into

$$p(x_1, x_2, \eta(x_1, x_2, t), t) = p_a \tag{9.6}$$

where $\eta(x_1, x_2, t)$ is the displacement of water relative to the sea level along x_3 direction.

The Taylor expansion allows us to linearize Eq. 9.6. This assumes small variations of η at the sea level, which is a valid assumption for modeling tsunami propagation far from the coastal area. Writing the result in terms of pressure perturbations leads to a more convenient form of the boundary condition, namely

$$p'(x_1, x_2, 0, t) = \rho g \eta(x_1, x_2, t) \tag{9.7}$$

Apart from Eq. 9.7, System 9.3 requires additional constrains. It is usually given by a so-called kinematic boundary condition, which establishes the relation between the motion at the surface and its shape (see [100] for details). The condition prescribes the continuity of the ocean surface, which means that no wave breaking can occur. At $z = \eta(x_1, x_2, t)$, it can be written as

$$u_3 = \frac{\partial \eta(x_1, x_2, t)}{\partial t} + u_1 \frac{\partial \eta(x_1, x_2, t)}{\partial x_1} + u_2 \frac{\partial \eta(x_1, x_2, t)}{\partial x_2} \tag{9.8}$$

Linearization of Eq. 9.8 regarding velocity perturbation at the sea level (i.e., $z = 0$) yields

$$\frac{\partial \eta(x_1, x_2, t)}{\partial t} = u_3'(x_1, x_2, 0, t) \tag{9.9}$$

As discussed in Section 3.6, the DG method requires boundary conditions to be set through numerical fluxes by solving a proper inverse Riemann problem. In the case of the free ocean surface, artificial ghost cells can be used to impose Eq. 9.7 and Eq. 9.8 on the surface boundaries. As shown in [68], the state in each aligned ghost cell (denoted using "+" superscript) must follow Eq. 9.10 to obtain correct numerical fluxes.

$$\begin{aligned} p^{+,\prime} &= 2\rho g \eta - p^{-,\prime} \\ u_n^{+,\prime} &= u_n^{-,\prime} \end{aligned} \tag{9.10}$$

where "$-$" denotes a cell inside the acoustic domain aligned to the free surface; subscript n denotes the normal direction of the boundary between the cells.

The displacement η can be obtained by solving Eq. 9.11, which stems from the inverse Riemann problem.

$$\frac{\partial \eta}{\partial t} = u_n^{-,\prime} - \frac{1}{Z}\left(\rho g \eta - p^{-,\prime}\right) \tag{9.11}$$

where $Z = c\rho$ is impedance.

In *SeisSol*, the ADER scheme is used to evaluate the displacement from Eq. 9.11. The solution is based on the Taylor expansion.

$$\eta(t) = \sum_{i=0}^{\mathcal{O}-1} \frac{(t - t_0)^i}{i!} \frac{\partial^i \eta(t_0)}{\partial t^i} \tag{9.12}$$

Similar to Eq. 3.26, the derivatives required to compute Eq. 9.12 can be found by differentiating Eq. 9.11 recursively.

$$\frac{\partial^i \eta}{\partial t^i} = \frac{\partial^{i-1} u_n^{-,\prime}}{\partial t^{i-1}} - \frac{1}{Z}\left(\rho g \frac{\partial^{i-1} \eta}{\partial t^{i-1}} - \frac{\partial^{i-1} p^{-,\prime}}{\partial t^{i-1}}\right) \tag{9.13}$$

The first time derivative (i.e., $\partial^0\eta/\partial t^0$) is equal to the displacement η at the beginning of a time step and, thus, given. The derivatives for pressure and velocities perturbations can be retrieved from the results of *ader* macro-kernel (see Eq. 4.7) and, thus, also given.

The implementation of the gravitational free surface boundary consists of five steps. Firstly, the integrated DOFs of the tetrahedrons with at least one face aligned to the free ocean surface are stored in a temporary memory buffer. This is implemented as a part of evaluating the "*ader*" macro-kernel (see Eq. 4.7). It is worth pointing out that all necessary temporary buffers are allocated beforehand - i.e., during *SeisSol*'s initialization phase. Thus, allocations and de-allocations of them do not involve any additional overheads. Secondly, the saved time derivatives of pressure and velocity perturbations are projected from the 3D modal onto the 2D nodal basis in the face normal directions. The step is similar to mapping volumetric solutions to the 2D dynamic rupture Gaussian points (see Eq. 3.45 and Eq. 3.46). Thirdly, displacement derivatives, computed using Eq. 9.13, are summed together according to Eq. 9.12. The integrated displacements are saved into another temporary memory buffer for later data processing. The fourth step happens after evaluating all local flux contributions - i.e., after I^{local}_{surf} macro-kernel. During this step, the states in the adjacent ghost cells are computed according to Eq. 9.10 using the saved integrated displacement values. Lastly, the computed ghost states get projected from the 2D nodal back onto the 3D modal basis, multiplied with $\mathcal{A}^{+,m}_{qp}$ flux matrices, and added to the corresponding intermediate solutions - i.e., to the left-hand side of Eq. 4.1.

Steps 1, 2, and 5 involve only matrix multiplications, and thus, their GPU implementations are based on the kernels generated by the *YATeTo* DSL. Steps 3 and 4 require manipulating individual columns of the projected integrated DOFs matrices and also involve computations of the reciprocals of the impedance values. At the moment of writing, these operations are not supported by the DSL and, thus, implemented without code generation.

Computations involved in steps 3 and 4 are data-independent and operate only on point-local data. Therefore, the task decomposition for them is trivial and similar to the implementation of the dynamic rupture kernels. A single GPU thread is assigned to process a nodal point; the block size equals the number of the 2D nodal points. Following the reasoning discussed in Section 8.1, GPU kernels for steps 3 and 4 are implemented separately for each supported GPU backend - i.e., CUDA, HIP, and SYCL - due to their small sizes and tight integration with the surrounding generated code.

The computational mesh used for the Palu earthquake-tsunami simulation comprises approximately 89 million tetrahedrons and about 100 thousand rupture elements. The LTS clustering of the mesh is shown in Fig. 9.8. As can be seen, the tetrahedrons are distributed between 8 LTS time-clusters. Approximately 99.5% of them belong to the last 4 clusters. The first cluster is the smallest one and contains only 12 mesh elements. As in two previous cases, the clustering indicates that the simulation will not scale well on muli-GPU systems with the given space discretization.

It is worth noting that the dynamic rupture elements are distributed only between time-clusters 2, 3, and 4. The pattern may indicate that the mesh contains very skewed

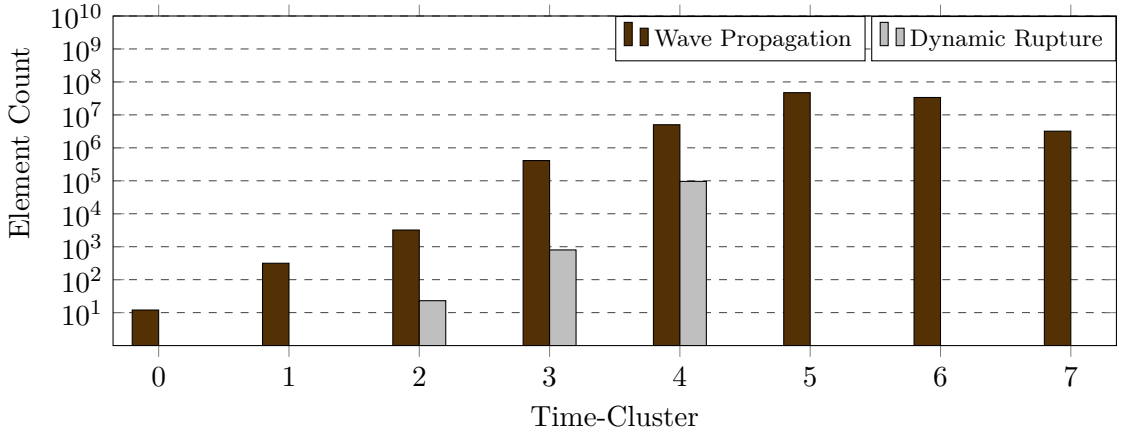


Figure 9.8.: LTS clustering of the Palu computational mesh, consisting of approximately 89 million tetrahedrons and about 100 thousand rupture elements.

tetrahedrons far from the fault segments. Usually, it results from applying a box mesh refinement with a very small transitioning region regarding the average element size from inside the box outward (see Fig. A.3). A mesh generator may fail to fulfill a good mesh quality within such a tiny volume. As a result, the generator can make a decision to skew some elements to avoid a program failure. I suggests that, in such cases, a refinement box can be slightly reduced in size while increasing the transitioning region. This can give the generator a better opportunity to optimize a mesh. And it may not necessarily increase the overall element count. The smallest time step width imposed by a given CFL condition can be increased by eliminating skewed elements, leading to fewer time integration steps. Moreover, it can reduce the number of time-clusters, making the first ones larger and, thus, increasing GPU performance (see Fig. 6.10).

The correctness of the aforementioned acoustic coupling and boundary conditions was checked by comparing numerical and analytical solutions of the Earthquake-Tsunami benchmark scenario proposed by Krenz et al. in [68]. The analysis proved the correctness of the GPU implementation and also revealed that the numerical solution quickly diverges when the single-precision floating-point format is used. Thus, the double-precision format is necessary for any earthquake-tsunami scenario in *SeisSol*. In this study, the 2018 Palu, Sulawesi earthquake and tsunami simulation was conducted on the Leonardo supercomputer using 256 *Nvidia* A100-C-64 GPUs on 64 GPU nodes. The first 92 seconds of the simulation took 13 hours and 9 minutes and resulted in almost 0.4 DP-PFLOP/s.

The fault system geometry and simulation results are shown in Fig. 9.9. Fig. 9.9b depicts the shear and rupture fronts traveling along the Palu segment from north to south. As can be seen from the snapshot, the rupture front covered more than 80 *km* in 17 seconds. The collected numerical data revealed that the front crossed Palu Bay in approximately 4 seconds. The resulting fast slip pulse generates radiating shear waves, forming a V-like shape on the free surface, also known as the Mach cone. Nevertheless, it is worth pointing out that perturbations of the Earth caused by seismic waves alone could not lead to such a devastating tsunami event similar to the one in question.

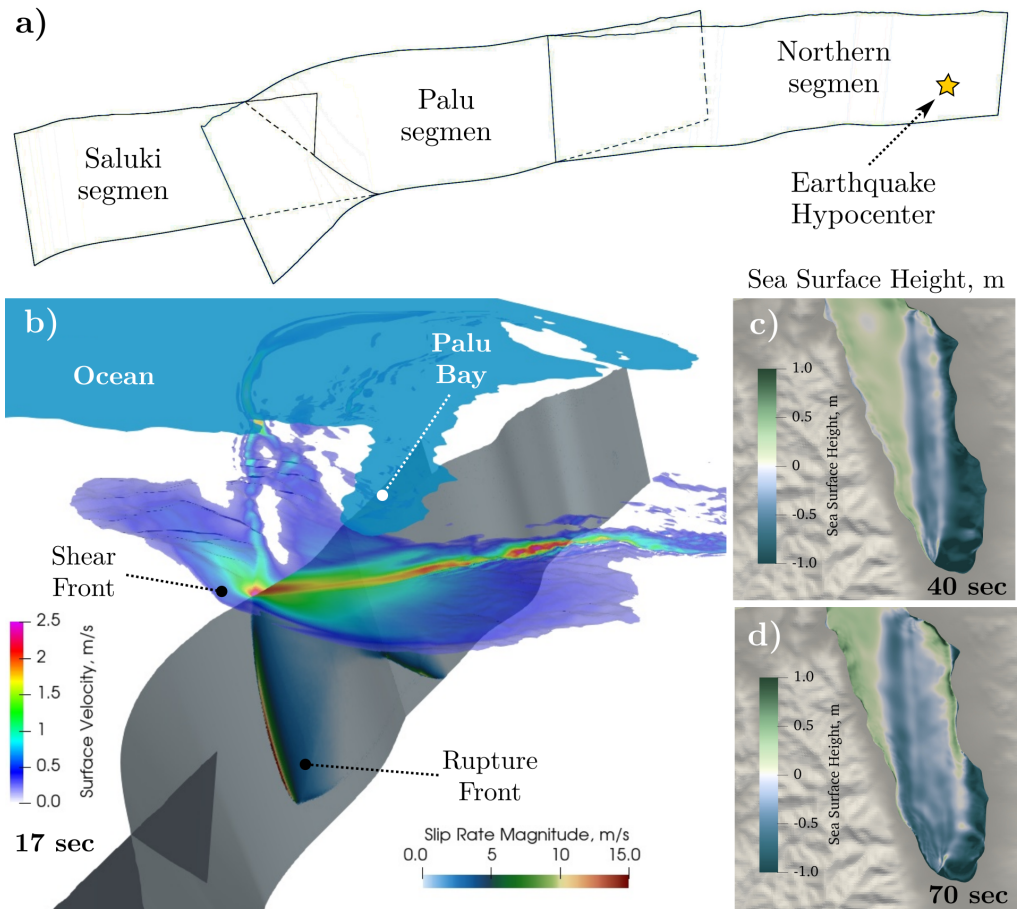


Figure 9.9.: Numerical results of the 2018 Palu, Sulawesi earthquake-tsunami scenario obtained on the Leonardo supercomputer. (a) the 3D model of the fault system. (b) the rupture and shear Mach fronts at 17 seconds. (c) and (d) snapshots of vertical water displacement of the ocean surface at 40 and 70 seconds.

The earthquake resulted in rapid vertical displacements of the sea bottom of the bay. Fig. 9.10 shows displacements of two neighboring points at the bottom located approximately at the center of the bay. The points belong to two adjacent elements with a shared edge, laying on the fault surface of the Northern segment. Fig. 9.10 illustrates the local process at the sea bottom around the test points. Nevertheless, it depicts a common pattern observed from numerical results. As can be seen from the plot, the western part of the bay was lifted up to approximately 0.65 m in about 4 seconds. Meanwhile, the eastern part went about 0.95 m down in around 8 seconds. After some initial perturbations, the displacement difference between the two parts stabilized and became equal to approximately 1.4 m.

The abrupt change in the water level creates a difference in the potential energy of the water between the two parts of the bay. The difference is substantial due to the enormous mass of water in the western part of the bay, which is connected to the ocean. The potential difference transforms into the kinetic water energy, establishing the flow from

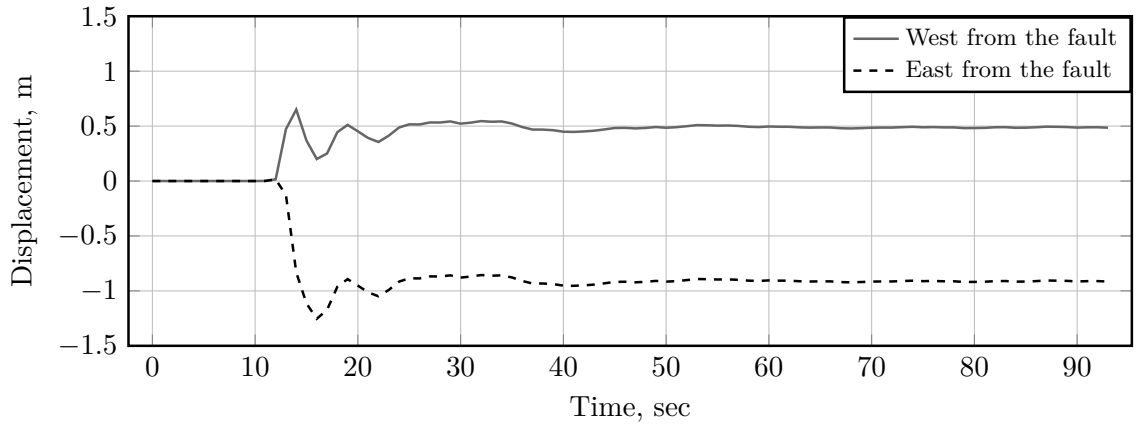


Figure 9.10.: Vertical displacements of the sea bottom at two arbitrarily chosen points around the center of Palu Bay. The points belong to the centers of adjacent elements aligned to the rupture surface of the Northern segment.

left to right. As can be seen from Fig. 9.9c, the falling water columns hit the free surface of the eastern part of the bay, forcing the water under it to get displaced downwards and then toward the east. This process generates tsunami waves, which rapidly travel toward the shore. According to Fig. 9.9d, the wavefront reaches the northeastern shore of Palu Bay approximately 70 seconds after the beginning of the earthquake, resulting in a heavy strike. At the same time, one can observe small amplitude tsunami waves reflected from the southern shore of the bay. All in all, the process results in a complex wavefield pattern, which can be studied further.

Table 9.1.: *SeisSol*'s performance data collected on 50 CPU/GPU supercomputing nodes during simulations of the 2018 Palu, Sulawesi earthquake-tsunami scenario. The performance data obtained on the SuperMUC-NG and Mahti supercomputers are taken from work [68].

Supercomputer name	SuperMUC-NG	Mahti	Leonardo
Resources per node	2x Intel Xeon 8174 CPUs, 48 cores, AVX512	2x AMD Rome 7H12 CPUs, 128 cores, AVX2	4x Nvidia A100-C-64 GPUs
Performance, DP-TFLOP/s	67.95	116.10	358.65

Numerical results presented in Fig. 9.9c and Fig. 9.9d are identical to the ones shown in [68], which were obtained on several CPU-based supercomputers. This fact additionally proves the correctness of the GPU implementation of *SeisSol*. Further, the publication reports *SeisSol*'s performance measured on the SuperMUC-NG and Mathi supercomputers over a wide range of CPU nodes. The authors provided detailed information about their 50-node runs on both machines using the same scenario and the same 89 million elements mesh as the ones used for this study. To compare the CPU- and GPU-based executions, an additional 50-node run was performed on the Leonardo supercomputer in this study.

The results are summarized in Table 9.1. As can be seen, the computational throughput of *SeisSol* is almost 5.3 times higher on the Leonardo nodes compared to the SuperMUC-NG ones and approximately 3.1 times higher than on the Mahti nodes. However, as shown in [68], *SeisSol* scales well on the CPU-based supercomputers - i.e., reaching approximately 72% of parallel efficiency on 1600 SuperMUC-NG nodes and about 73% on 700 Mahti nodes. As mentioned above, it is not possible to achieve similar strong scaling results on GPU-accelerated machines, at the moment of writing, due to the very complex LTS clustering resulting from the provided computational mesh.

The last experiment of this work is a short 1024-GPU run conducted on 256 Leonardo nodes using 518 million elements mesh taken from the artifacts of work [68]. The refined mesh contains almost 6 times more tetrahedrons in the elastic and acoustic wave propagation domains and approximately 2.5 times more dynamic rupture elements. The resultant LTS clustering is shown in Fig. A.4. Similar to the previous case, the tetrahedrons are distributed between 8 LTS time-clusters, whereas the number of the dynamic rupture clusters increased by a factor of 2 - i.e., from 3 to 6. The aggregated performance of the run reached slightly above 1.81 DP-PFLOP/s. According to the result reported in [68], the authors achieved 0.99 DP-PFLOP/s on 768 SuperMUC-NG nodes using the same computational mesh and setup.

It is worth mentioning that, in contrast to work [68], the results shown in this chapter were obtained using a so-called wiggle factor. The factor slightly reduces the global minimal time step width, leading to a more compact LTS clustering, which is beneficial for GPU-like execution (see Section 6.5.3). Typically, the change forces some elements around the time-cluster boundaries to move from clusters with smaller time step widths to their direct neighbors with larger ones. As a result, some clusters, usually the last ones, can be merged entirely. For example, the use of an optimal wiggle factor reduced the number of time-clusters in the elastic and acoustic wave propagation domains from 13 to 8 in this work. Therefore, the reader should be a bit careful when comparing the CPU and GPU performance values reported in this section.

9.4. Discussion

This chapter presents simulations of three complete production earthquake scenarios performed on two of the most powerful supercomputers in Europe at the moment of writing - i.e., LUMI and Leonardo. Both supercomputers are distributed multi-GPU systems. The largest simulation presented in this study involved the computational mesh consisting of 518 million tetrahedrons and was conducted on 1024 Nvidia A-C-64 GPUs, resulting in 1.81 DP-PFLOP/s. In this work, the used GPU counts were constrained because of the limits in 1) the strong scaling performance and 2) supercomputing project budgets. The former was mainly caused by the resulting distributions of mesh elements between LTS time-clusters, which are illustrated in Fig. 9.2, Fig. 9.5, and Fig. 9.8.

The 2023 Kahramanmaraş and the 2019 Ridgecrest scenarios model earthquake sequences using two different approaches. In the former case, the aftershock must be triggered by the seismic waves caused by the mainshock. A careful choice of the initial and boundary conditions is required to achieve such an effect. In the latter case, the mainshock occurs after the foreshock. The triggering, in this case, is imposed through the second stress nucleation region around the hypocenter of the mainshock at a specific point in time and requires specific source code modifications. In this work, I managed to reproduce only the mainshocks of both earthquake sequences due to the reasons described in Section 9.1 and Section 9.2. Nevertheless, the obtained numerical results match the observations reported in [57, 108].

The demonstrated 2018 Sulawesi earthquake-tsunami scenario is the most advanced because it involves simultaneous modeling of four phenomena: 1) wave propagation, 2) dynamic rupture, 3) off-fault plasticity, and 4) ocean dynamics. As discussed in Section 9.3, *SeisSol*'s ocean dynamics and wave propagation implementations share many common components. Coupling between them is done through a specially designed boundary condition developed by Krenz et al. in [68]. As the contribution of this work, I adapted the gravitational free surface boundary condition for heterogeneous computing environments to achieve better GPU performance. The reader can find the details in Section 9.3. The numerical results obtained during the earthquake-tsunami simulation are identical to the ones presented in [68]. The latter were obtained on several CPU-based supercomputers. This fact additionally proves the correctness of all GPU-adapted components of *SeisSol*. The comparison between the CPU and GPU performance of the scenario is shown in Table 9.1.

10. Conclusions

This study aimed to investigate whether a highly tuned CPU-based HPC application designed for simulating seismic wave propagation and earthquake dynamics - i.e., *SeisSol* - can be extended to efficiently utilize distributed multi-GPU systems. The results show that all necessary computations can be fully adapted for heterogeneous computing and optimally mapped to GPU-like computational units. The number of GPU service tasks inside the main computational loop can be significantly minimized. The combination of these two factors results in a high-performance GPU implementation of *SeisSol*, capable of simulating real earthquake production scenarios. The results also show that a suitable choice of abstractions and design patterns allows the CPU and GPU versions of *SeisSol* to coexist in a single codebase. In the following, I describe the key finding of this study.

Because all computational tasks generated by elastic wave propagation, dynamic rupture, and off-fault plasticity solvers run on GPUs, no additional host-to-device data transfers are needed inside the main computational loop except for writing intermediate results to disks. Additionally, the GPU version of *SeisSol* utilizes a pool of device memory, which implements the LIFO policy - i.e., stack. The pool is used for fast allocations and deallocations of temporary memory required for storing intermediate computational results. Simple pointer arithmetic is much faster than communication with GPU runtime libraries, which may consist of several layers and interact with the operating system kernel space. Therefore, the memory pool design reduces the number of GPU service tasks and, thus, increases the overall performance of the application.

GPU programming models used in this work (i.e., CUDA, ROCm, and SYCL) provide different APIs for handling device service tasks - e.g., selection and initialization of devices, memory allocation and deallocation, data transfers, etc. In this work, I demonstrate that this problem can be solved by unifying programming interfaces using the Adapter and Facade design patterns. The additional software layer redirects user requests to concrete API calls and provides access only to a required subset of APIs, thus reducing the overall software maintenance cost. The interface is also used to provide custom GPU service tasks by combining several primitive ones and to abstract some commonly used GPU algorithms - e.g., parallel reduction. Of course, an additional level of indirection entails overheads. However, GPU programming models are primarily asynchronous. Therefore, such inefficiencies on the host can be well hidden by computations on a device.

In this study, I demonstrate that the original software design, which is based on mixed sub-task execution, can be adapted for heterogeneous computing by splitting each CPU task into multiple ones containing sub-tasks of the same kind. Due to the static mesh refinement used in *SeisSol*, this pre-processing step needs to be done only once - i.e.,

during the initialization phase. During this step, the data required for processing each split task are recorded into batches and stored in a specially designed multi-level hash table. The necessary batches for each GPU task can be efficiently retrieved from the table using the keys, which encode the execution control flow path of a task as an integer value. During a task execution, the GPU hardware maps each sub-task to a free streaming multiprocessor. Thus, all multiprocessors concurrently perform the same computations on different pieces of data. This leads to a better utilization of GPU resources. This approach significantly simplifies the code generation design because the control flow selection logic, which sometimes is not trivial, stays on the host.

Sometimes, a task split can generate many small-sized data-independent fragments. Offloading such fragmented tasks to a GPU may degrade its performance; the workload of a task may not be enough to completely utilize all available multiprocessors. In this study, I address this problem by concurrently executing data-independent tasks on multiple GPU streams (or queues in the case of SYCL). This approach gives a GPU scheduler enough sub-tasks to fill all streaming processors with enough work. I show that the number of service tasks in the main computational loop can be minimized by allocating and managing a pool of streams. At some point, streams must be synchronized to ensure that all submitted GPU tasks are completed. A synchronization of streams involves overhead proportional to the pool size. I show that the optimal stream count equals 4 for *Nvidia* A100 and *AMD* MI250x GPUs. This pool configuration increases the GPU performance by approximately 19% compared to the single-stream design.

In *SeisSol*, computational micro-kernels, executed by CPU sub-tasks, are generated with the *YATeTo* DSL. The DSL maps each tensor expression to an abstract syntax tree in which each intermediate node represents a binary tensor operation. After the semantic analysis, *YATeTo* generates vectorized CPU code for each operation represented by a node during a post-order tree traversal. As one of the first steps of this work, I extended the DSL to generate GPU code for batched binary operations - i.e., batched GEMMs.

On CPUs, the intermediate results of computations are small and fit into the top-level caches. Thus, vectorization and abandoned data caching are the key components determining the high CPU performance of *SeisSol*. GPU caches are not persistent between subsequent invocations of GPU kernels. Thus, the initial GPU implementation of *SeisSol*, implemented using batched binary operations, has inherently lower arithmetic intensity than its original CPU counterpart; the GPU application redundantly moves intermediate results between the device memory and compute units. In this study, I demonstrate that the average GPU performance of the ADER-DG method can be considerably improved by fusing subsequent batched GEMM GPU kernels during code generation. A suitable intermediate code representation of batched GEMM operations makes it easier to implement various code transformations to minimize shared memory consumption and block-level thread synchronizations inside a fused kernel. The results show that fusion increases GPU performance by more than 35% compared to executing binary batched GEMM operations in sequence. Overall, the final GPU implementation of *SeisSol* results in 2-2.5x speed-up relative to the original CPU version when comparing performance on a single HPC GPU and a single 48-core AVX512 CPU server - i.e., a single node of the SuperMUC-NG supercomputer. Currently, the kernel fusion is limited to only

batched GEMM operations. More advanced wave propagation models - e.g., viscoelastic - involve high-order tensor operations. Therefore, the presented code-generation approach needs to be better generalized and extended. However, fusing a long sequence of high-order operations in a single GPU kernel is not trivial and requires a follow-up study.

Adapting a complex software application like *SeisSol* for heterogeneous computing environments requires an incremental approach; parallel regions are developed, offloaded, and tested one by one. During this process, a developer spends significant time developing the code for copying input and output data to and from a device. The process is dynamic and error-prone because, once two adjacent parallel regions are ported to GPUs, redundant host-to-device copies must be removed. During the development, it may even be required to restore the original host implementation of a parallel region to perform debugging. The use of unified (managed) memory significantly simplifies the process. This memory type is pageable: the device driver manages data migration between host and device memory when a page fault occurs. Automatic data migration is less efficient than explicit data copying because it involves extra algorithmic overhead and cannot fully utilize the bandwidth of the bus connecting the host and a device. However, once all parallel regions are offloaded, most of the data resides on the device during the whole execution of a program. Thus, no data migration occurs between the host and a device, except for writing intermediate results to a disk.

Many modern implementations of the MPI standard are adapted to pass messages to/from MPI buffers allocated with unified memory. However, I demonstrate that, while being convenient, this approach significantly reduces the application's performance during strong scaling. In this work, I extended the original non-blocking message-passing algorithm of *SeisSol* to copy data from unified to device memory asynchronously before exchanging data between GPUs over a network. This design involves a more efficient communication protocol, which results in lower latency and higher bandwidth through a communication channel. The results show that the improved algorithm increases strong scaling performance by almost 2.6 times.

A cluster-wise LTS algorithm splits elements of a computational mesh into sub-sets - i.e., time-clusters - and updates each with its optimal time integration step width. This approach drastically reduces the time-to-solution of a simulation by reducing redundant computations. However, the algorithm results in shrinking parallel regions. Some clusters may fall into a low computational throughput region of a device, affecting the overall performance of the algorithm. The problem intensifies during strong scaling when more and more clusters fall to the low throughput region due to mesh partitioning. In these cases, the workloads of GPU tasks may not be enough to hide kernel launching overheads, leading to a significant performance loss.

In this study, I investigate the effect of graph-based execution on the strong scaling performance of the LTS algorithm. Computational graphs are composed of GPU tasks and stored directly on a device with all required kernel arguments. A single graph launch is required to schedule executions of all GPU tasks enclosed by a graph in the order specified by the graph edges. This work demonstrates how one can build computational graphs for the ADER-DG method. The results show that the graph-based model increases the

average GPU performance by approximately 50% for small LTS clusters and the strong scaling performance of the entire LTS algorithm by almost 40%. Despite a considerable performance boost, the parallel strong scaling efficiency of the LTS algorithm reaches only 53% on distributed multi-GPU systems. This means the algorithm has other performance limiters that require further investigation.

As a part of this study, I investigate why the strong scaling parallel efficiency of the LTS algorithm drops faster on distributed multi-GPU systems than on distributed-memory CPU machines. By using the GTS scheme on a single GPU and CPU and varying the problem size, I show that the GPU throughput rapidly drops starting at a particular problem size, whereas CPU performance stays almost flat within the entire test range. This experiment concludes that there is a dependency between the computational throughput characteristics of a device and LTS clustering. I demonstrate this by conducting multiple strong scaling experiments on the LUMI supercomputer using the same computational mesh with various LTS clustering configurations, enforced by manipulating the material parameters of the domain. The results show that the performance varies within a wide range and strongly depends on clustering - i.e., from 0.57 to 1.32 SP-PFLOP/s on 256 *AMD* MI250x GPUs. The worst results were obtained when the sizes of the time-clusters linearly grew from the cluster with the highest update rate to the cluster with the lowest one. The best results were obtained in the opposite case. The former type of clustering commonly occurs in simulations of real production earthquake scenarios.

SeisSol's implementation of the LTS scheme constrains update rates of adjacent clusters to integer numbers. It results in having common synchronization points between them. At these points, adjacent clusters have no data dependencies and, thus, can be updated concurrently. In theory, neighboring time-clusters can be temporarily merged and updated as a single one. This approach should reduce the overall number of small-sized GPU tasks. Moreover, the execution of merged tasks will be shifted to a higher computational throughput region. The batched data required for executing the merged tasks can be prepared in advance - i.e., during the initialization phase. The proposed solution should theoretically improve the strong scaling performance of the LTS algorithm on distributed multi-GPU systems, which I suggest trying in a follow-up research.

Computations resulting from the dynamic rupture solver involve many non-linear operations. These cannot be expressed with *YATeTo* because the DSL is designed for generating source code for tensor operations. In this work, I compare several implementations of the OpenMP and SYCL standards for offloading the GPU tasks generated by the dynamic rupture solver. The results show that SYCL results in better portability compared to OpenMP. Moreover, the designed SYCL implementation of the rupture solver is approximately 1.65 times faster than the fastest OpenMP variant.

The final results of this thesis include simulations of three complex earthquake scenarios obtained with the fully adapted GPU version of *SeisSol* on the LUMI and Leonardo supercomputers. The scenarios include the 2023 Kahramanmaraş earthquake, the 2019 Ridgecrest earthquake, and the 2018 Palu earthquake-tsunami event. The last one is the most advanced because it involves simultaneous modeling of four phenomena: 1) wave propagation, 2) dynamic rupture, 3) off-fault plasticity, and 4) ocean dynamics. This

scenario required the adaptation of the gravitational free surface boundary condition from work [68] to GPUs for better performance. The obtained numerical results are identical to the ones reported in [68]. The largest Palu simulation, presented in this work, involved a computational mesh consisting of 518 million tetrahedrons and was conducted on 256 nodes of the Leonardo supercomputer (1024 Nvidia A-C-64 GPUs), resulting in 1.81 DP-PFLOP/s. For comparison, the same setup resulted in 0.99 DP-PFLOP/s on 768 SuperMUC-NG nodes, according to the results reported in [68].

The GPU version of *SeisSol* is mature and ready for performing complex earthquake simulations on top-tier supercomputers. Currently, it is challenging to sustain high performance on distributed multi-GPU systems during strong scaling. However, an in-depth study of an extreme-scale earthquake event may involve multiple highly resolved 3D simulations, which can run in parallel on different sets of GPU nodes. Therefore, researchers can efficiently utilize large portions of GPU-accelerated supercomputers despite limited scaling capabilities. The outcome of this study is not limited to *SeisSol*. The results, findings, and ideas shared in this thesis may help others to adapt and optimize their scientific applications for heterogeneous computing environments.

A. Appendices

A.1. Supercomputers

A.1.1. LUMI

LUMI belongs to the Hewlett Packard Enterprise Cray EX family of supercomputers. The LUMI-G partition consists of 2560 GPU nodes. Architecturally, it is similar to the Crusher supercomputer regarding the single-node design (see Fig. 6.17). Each node is powered by four *AMD* MI250X GPUs and controlled by a single 64-core *AMD* Trento CPU. Individual GPUs within a node communicate via the GPU-to-GPU *AMD* Infinity Fabric interconnect, delivering 100 GB/s bidirectional bandwidth. Each GPU gets attached to an individual 50 Gb/s network card via PCIe Gen4 ESM bus. The 200 Gb/s Cray Slingshot-11 network connects all LUMI-G nodes using a Dragonfly topology.

The MI250X GPU is based on the *AMD* CDNA2 architecture (see [28]) and consists of two Graphics Compute Dies. Each compute die is exposed to the Operating System (OS) as a single accelerator with 110 Compute Units and 64 GB of HBM2e memory. Thus, applications based on the Single Process Single GPU model must use 2 MPI processes to utilize two graphics dies simultaneously and, thus, the entire MI250X GPU.

OS	SUSE Linux Enterprise Server 15 SP4
MPI	CRAY MPICH v8.1.25
GPU driver	AMD v5.16.9.22.20
GPU Stack	ROCm v5.4
Host Compiler	amdclang v15.0.0
Device Compiler	amdclang v15.0.0

Table A.1.: The used software stack on the LUMI-G partition.

A.1.2. Leonardo

Leonardo is a member of the Atos BullSequana XH family of supercomputers. The Booster partition of Leonardo encompasses 3456 GPU nodes connected with the 200Gb/s *Nvidia* Mellanox HDR InfiniBand network, featuring a Dragonfly+ topology. Each GPU node consists of a single 32-core Intel Xeon 8358 CPU, four custom *Nvidia* Ampere 64GB-HBM2

GPUs (referred to as A100-C-64 in this work), and two dual-port Mellanox HDR100 ConnectX-6 InfiniBand network cards, providing 400 Gb/s bidirectional bandwidth for inter-node communication in total. Individual GPUs within a node are connected using the NVLink 3.0 interconnect technology, which provides 200 GB/s bidirectional bandwidth per GPU pair.

OS	Red Hat Enterprise Linux 8
MPI	OpenMPI v4.1.5
GPU driver	Nvidia v525.105.17
GPU Stack	NVIDIA HPC SDK v12.1
Host Compiler	clang v15.0.0
Device Compiler	nvcc v12.1

Table A.2.: The used software stack on the Leonardo Booster partition.

The details of the single-node and network hardware designs of the Leonardo supercomputer are explained in [111].

A.1.3. Selene

Selene is a proprietary *Nvidia*'s supercomputer, which is based on the DGX SuperPOD technology. The supercomputer consists of four SuperPODs, each containing 140 DGX A100 nodes. Each node is powered by eight *Nvidia* A100-SXM4-80, controlled by two 64-core AMD Rome 7742 CPUs. The NVLink 3.0 technology is used to provide equal bandwidth and latency for intra-node GPU-to-GPU communication. Eight single-port Mellanox HDR ConnectX-6 InfiniBand network cards, connected to four PCIe switches, are installed in each node. Each card provides 200 Gb/s bidirectional bandwidth. Additionally, each node is equipped with two dual-port Mellanox HDR ConnectX-6 InfiniBand cards, reserved for communicating with the storage network. The nodes of the Selene supercomputer are connected together using Full Flat-tree network topology.

OS	Ubuntu 20.04
MPI	OpenMPI 4.1.5, UCX 1.15.0
GPU driver	Nvidia v515.65.01
GPU Stack	Nvidia HPC SDK 23.5
Host Compiler	gcc 13.1
Device Compiler	nvcc v11.8

Table A.3.: The used software stack on the Selene supercomputer.

A.2. Supplementary Materials

A.2.1. Influence of LTS clustering on Strong Scaling

Leonardo: Analysis without making use of CUDA-Graphs

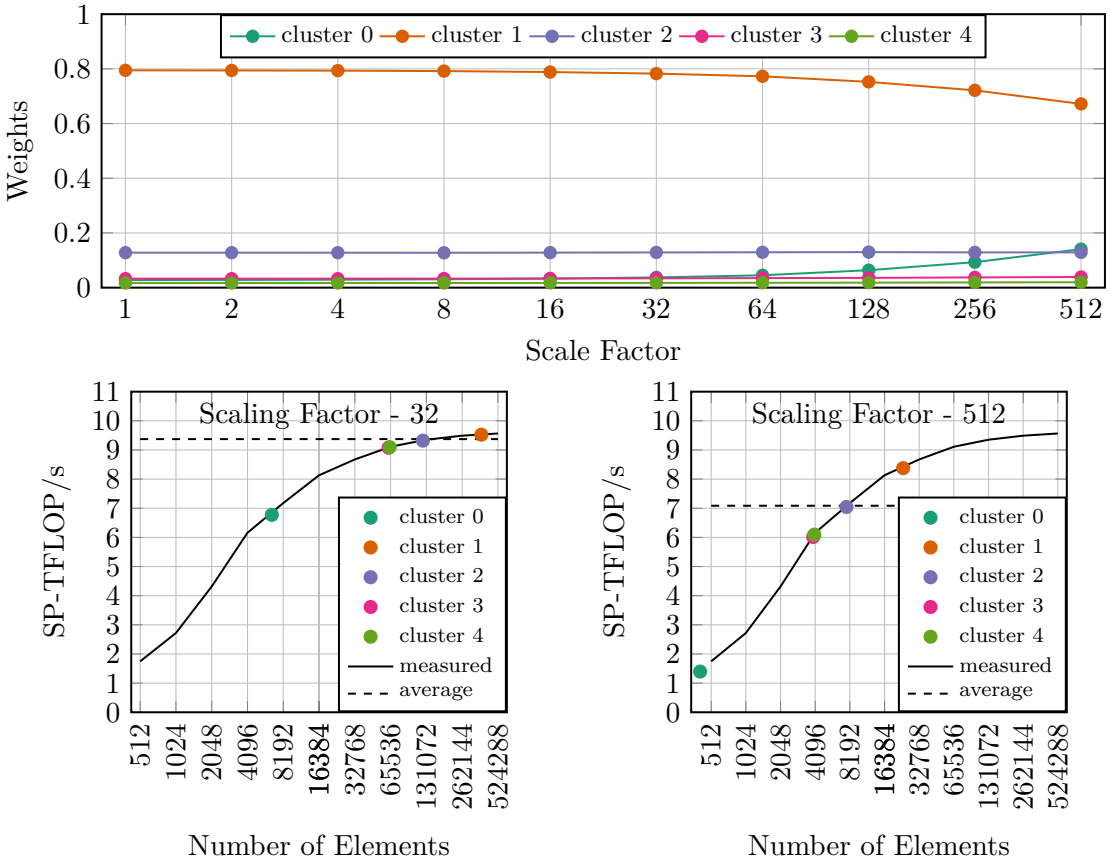


Figure A.1.: Evolution of performance-weights of LTS time-clusters during strong scaling.

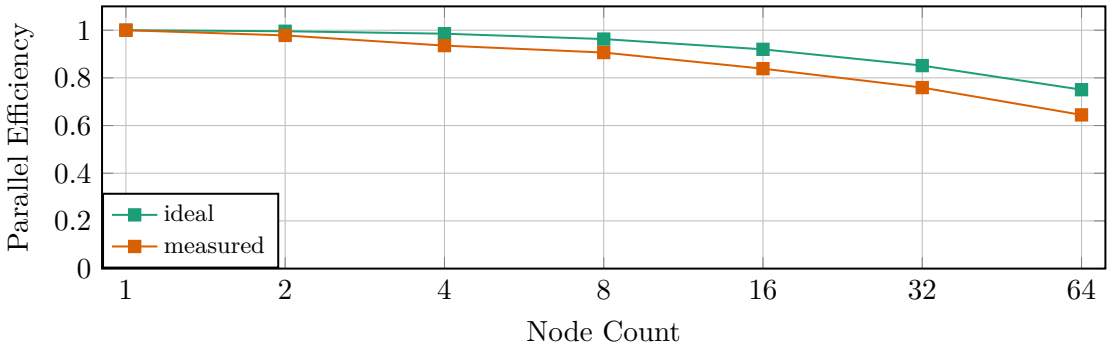


Figure A.2.: Ideal and measured parallel efficiency during strong scaling of the LOH.1 scenario on *Nvidia* A100-C-64 GPU.

A.2.2. 2018 Palu, Sulawesi Earthquake and Tsunami

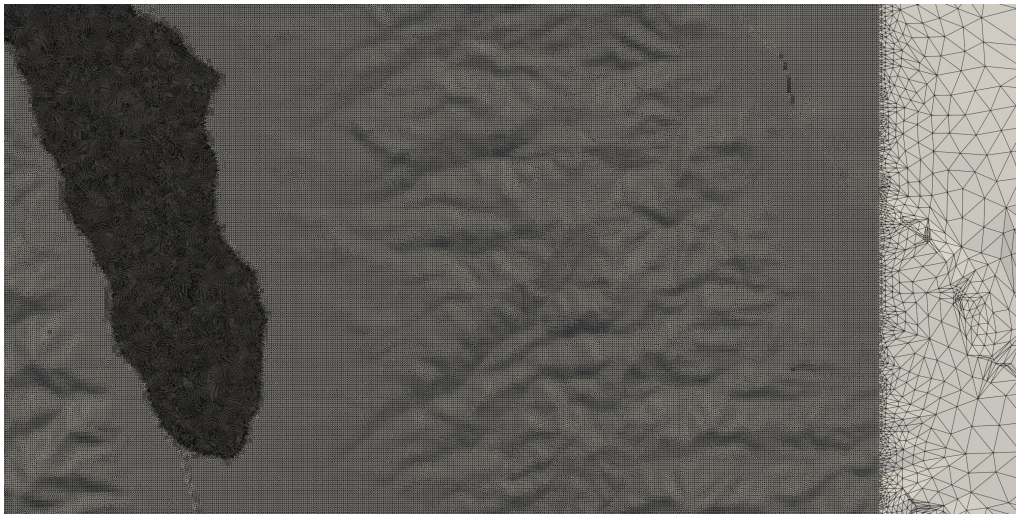


Figure A.3.: A snippet of the 89 million elements mesh used for the numerical simulation of the 2018 Palu earthquake-tsunami scenario.

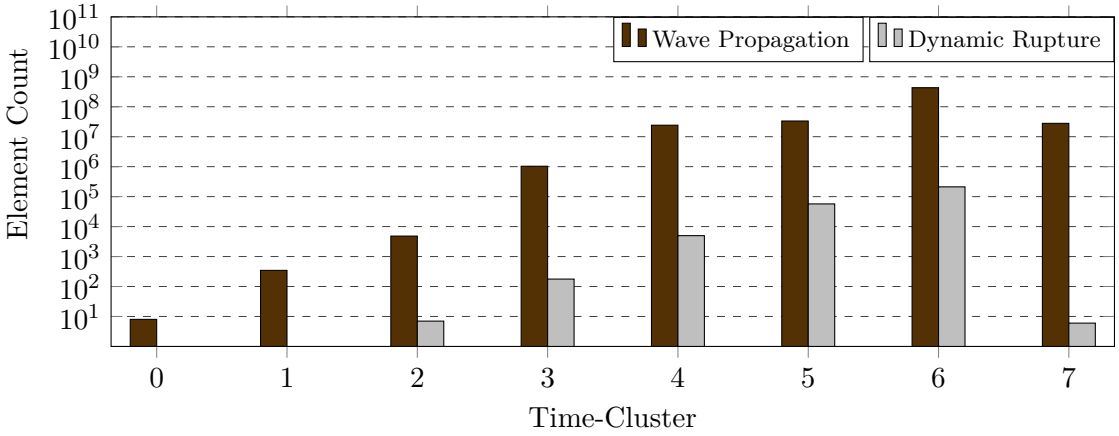


Figure A.4.: LTS clustering of the Palu computational mesh, consisting of approximately 518 million tetrahedrons and about 275 thousand rupture elements.

Bibliography

[1] Tor M Aamodt, Wilson Wai Lun Fung, and Timothy G Rogers. "General-purpose graphics processor architectures". In: *Synthesis Lectures on Computer Architecture* 13.2 (2018), pp. 1–140.

[2] Daniel S Abdi, Lucas C Wilcox, Timothy C Warburton, and Francis X Giraldo. "A GPU-accelerated continuous and discontinuous Galerkin non-hydrostatic atmospheric model". In: *The International Journal of High Performance Computing Applications* 33.1 (2019), pp. 81–109.

[3] Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. *Compilers, principles, techniques, and tools.(Rep. with corrections.)* 2020.

[4] Leandro R Alejano and Antonio Bobet. "Drucker–prager criterion". In: *The ISRM Suggested Methods for Rock Characterization, Testing and Monitoring: 2007-2014*. Springer, 2014, pp. 247–252.

[5] Aksel Alpay and Vincent Heuveline. "One Pass to Bind Them: The First Single-Pass SYCL Compiler with Unified Code Representation Across Backends". In: *IWOCL & SYCLcon 2023* (2023), p. 3585351.

[6] DJ Andrews. "Rupture dynamics with energy loss outside the slip zone". In: *Journal of Geophysical Research: Solid Earth* 110.B1 (2005).

[7] Zhen Guo Ban, Yan Shi, Qi Yang, Peng Wang, Shi Chen Zhu, and Long Li. "GPU-accelerated hybrid discontinuous Galerkin time domain algorithm with universal matrices and local time stepping method". In: *IEEE Transactions on Antennas and Propagation* 68.6 (2020), pp. 4738–4752.

[8] Dip Sankar Banerjee, Khaled Hamidouche, and Dhabaleswar K Panda. "Designing high performance communication runtime for GPU managed memory: early experiences". In: *Proceedings of the 9th Annual Workshop on General Purpose Processing using Graphics Processing Unit*. 2016, pp. 82–91.

[9] David A Beckingsale, Jason Burmark, Rich Hornung, Holger Jones, William Killian, Adam J Kunen, Olga Pearce, Peter Robinson, Brian S Ryujin, and Thomas RW Scogland. "RAJA: Portable performance for large-scale scientific applications". In: *2019 ieee/acm international workshop on performance, portability and productivity in hpc (p3hpc)*. IEEE. 2019, pp. 71–81.

[10] David A Beckingsale, Marty J McFadden, Johann PS Dahm, Ramesh Pankajakshan, and Richard D Hornung. "Umpire: Application-focused management and coordination of complex hierarchical memory". In: *IBM Journal of Research and Development* 64.3/4 (2019), pp. 00–1.

[11] Jean-Pierre Berenger. "A perfectly matched layer for the absorption of electromagnetic waves". In: *Journal of computational physics* 114.2 (1994), pp. 185–200.

[12] Alexander Breuer, Alexander Heinecke, and Michael Bader. "Petascale local time stepping for the ADER-DG finite element method". In: *2016 IEEE international parallel and distributed processing symposium (IPDPS)*. IEEE. 2016, pp. 854–863.

[13] Alexander Nikolas Breuer. "High Performance Earthquake Simulations". PhD thesis. Technische Universität München, 2015.

[14] Otto T Bruhns. "History of plasticity". In: *Encyclopedia of Continuum Mechanics* (2020), pp. 1129–1190.

[15] Miha Cernetic, Volker Springel, Thomas Guillet, and Rüdiger Pakmor. "High-order Discontinuous Galerkin hydrodynamics with sub-cell shock capturing on GPUs". In: *Monthly Notices of the Royal Astronomical Society* 522.1 (2023), pp. 982–1008.

[16] Jesse Chan and Tim Warburton. "GPU-accelerated Bernstein–Bézier discontinuous Galerkin methods for wave problems". In: *SIAM Journal on Scientific Computing* 39.2 (2017), A628–A654.

[17] Jieyang Chen, Nan Xiong, Xin Liang, Dingwen Tao, Sihuan Li, Kaiming Ouyang, Kai Zhao, Nathan DeBardeleben, Qiang Guan, and Zizhong Chen. "TSM2: optimizing tall-and-skinny matrix-matrix multiplication on GPUs". In: *Proceedings of the ACM International Conference on Supercomputing*. 2019, pp. 106–116.

[18] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. "{TVM}: An automated {End-to-End} optimizing compiler for deep learning". In: 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18). 2018, pp. 578–594.

[19] CALIFORNIA SEISMIC SAFETY COMMISSION. *Findings and Recommendations from the Ridgecrest Earthquake Sequence of July 2019*. 2019. url: https://ssc.ca.gov/wp-content/uploads/sites/9/2020/08/19-03_ridgecrest.pdf.

[20] Luca Dal Zilio and Jean-Paul Ampuero. "Earthquake doublet in Turkey and Syria". In: *Communications Earth & Environment* 4.1 (2023), p. 71.

[21] James Daniell and Armand Vervaeck. "Damaging earthquakes database 2011 the year in review". In: *CEDIM Earthquake Loss Estimation Series, Research Report No* 1 (2012).

[22] Steven M Day, Jacobo Bielak, Doug Dreger, R Graves, S Larsen, KB Olsen, and A Pitarka. "Tests of 3D elastodynamic codes: Final report for Lifelines Project 1A02". In: *Pacific Earthquake Engineering Research Center* (2003).

[23] Steven M Day, Luis A Dalguer, Nadia Lapusta, and Yi Liu. "Comparison of finite difference and boundary integral solutions to three-dimensional spontaneous rupture". In: *Journal of Geophysical Research: Solid Earth* 110.B12 (2005).

[24] Josep de la Puente, J-P Ampuero, and Martin Käser. "Dynamic rupture modeling on unstructured meshes using a discontinuous Galerkin method". In: *Journal of Geophysical Research: Solid Earth* 114.B10 (2009).

[25] Joel E Denny, Seyong Lee, and Jeffrey S Vetter. "Clacc: Translating openacc to openmp in clang". In: *2018 IEEE/ACM 5th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC)*. IEEE. 2018, pp. 18–29.

[26] Bronis R de Supinski, Thomas RW Scogland, Alejandro Duran, Michael Klemm, Sergi Mateo Bellido, Stephen L Olivier, Christian Terboven, and Timothy G Mattson. "The ongoing evolution of openmp". In: *Proceedings of the IEEE* 106.11 (2018), pp. 2004–2019.

[27] Advanced Micro Devices. *AMD CDNA Architecture*. 2020. url: https://www.amd.com/content/dam/amd/en/documents/instinct-business-docs/white-papers/amd-cdna-white-paper.pdf.

[28] Advanced Micro Devices. *AMD CDNA2 Architecture*. 2021. url: https://www.amd.com/system/files/documents/amd-cdna2-white-paper.pdf.

[29] Advanced Micro Devices. *"AMD Instinct MI200" Instruction Set Architecture: Reference Guide*. Feb. 2022. url: https://www.amd.com/content/dam/amd/en/documents/instinct-tech-docs/instruction-set-architectures/instinct-mi200-cdna2-instruction-set-architecture.pdf.

[30] Ravil Dorozhinskii and Michael Bader. "Seissol on distributed multi-gpu systems: Cuda code generation for the modal discontinuous galerkin method". In: *The International Conference on High Performance Computing in Asia-Pacific Region*. 2021, pp. 69–82.

[31] Benchun Duan and Steven M Day. "Inelastic strain distribution and seismic radiation from rupture of a fault kink". In: *Journal of Geophysical Research: Solid Earth* 113.B12 (2008).

[32] Michael Dumbser and Martin 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 (2006), pp. 319–336.

[33] Eric M Dunham, David Belanger, Lin Cong, and Jeremy E Kozdon. "Earthquake ruptures with strongly rate-weakening friction and off-fault plasticity, Part 1: Planar faults". In: *Bulletin of the Seismological Society of America* 101.5 (2011), pp. 2296–2307.

[34] Fionn Dunne and Nik Petrinic. *Introduction to computational plasticity*. OUP Oxford, 2005.

[35] Dominik Ernst, Georg Hager, Jonas Thies, and Gerhard Wellein. "Performance Engineering for a Tall & Skinny Matrix Multiplication Kernel on GPUs". In: *arXiv preprint arXiv:1905.03136* (2019).

[36] Jirí Filipovic, Jan Fousek, Bedrich Lakomỳ, and Matú Madzin. "Automatically optimized GPU acceleration of element subroutines in finite element method". In: 2012 Symposium on Application Accelerators in High Performance Computing. IEEE. 2012, pp. 141–144.

[37] Jiří Filipovič, Matúš Madzin, Jan Fousek, and Luděk Matyska. "Optimizing CUDA code by kernel fusion: application on BLAS". In: *The Journal of Supercomputing* 71.10 (2015), pp. 3934–3957.

[38] HSA Foundation. *HSA Platform System Architecture Specification Version 1.0*. Jan. 2015. url: http://hsafoundation.com/wp-content/uploads/2021/02/HSA-SysArch-1.01.pdf.

[39] Martin Fuhry, Andrew Giuliani, and Lilia Krivodonova. "Discontinuous Galerkin methods on graphics processing units for nonlinear hyperbolic conservation laws". In: *International Journal for Numerical Methods in Fluids* 76.12 (2014), pp. 982–1003.

[40] A-A Gabriel, J-P Ampuero, LA Dalguer, and Paul Martin Mai. "Source properties of dynamic rupture pulses with off-fault plasticity". In: *Journal of Geophysical Research: Solid Earth* 118.8 (2013), pp. 4117–4126.

[41] Rajesh Gandham, David Medina, and Timothy Warburton. "GPU accelerated discontinuous Galerkin methods for shallow water equations". In: *Communications in Computational Physics* 18.1 (2015), pp. 37–64.

[42] Christophe Geuzaine and Jean-François Remacle. "Gmsh: A 3-D finite element mesh generator with built-in pre-and post-processing facilities". In: *International journal for numerical methods in engineering* 79.11 (2009), pp. 1309–1331.

[43] Nico Gödel, Steffen Schomann, Tim Warburton, and Markus Clemens. "GPU accelerated Adams–Bashforth multirate discontinuous Galerkin FEM simulation of high-frequency electromagnetic fields". In: *IEEE Transactions on magnetics* 46.8 (2010), pp. 2735–2738.

[44] Nick Hagerty, Veronica Melesse Vergara, and Arnold Tharrington. *Studying performance portability of lammps across diverse gpu-based platforms*. Tech. rep. Oak Ridge National Lab.(ORNL), Oak Ridge, TN (United States), 2022.

[45] Khaled Hamidouche, Ammar Ahmad Awan, Akshay Venkatesh, and Dhabaleswar K Panda. "CUDA M3: Designing efficient CUDA managed memory-aware MPI by exploiting GDR and IPC". In: *2016 IEEE 23rd International Conference on High Performance Computing (HiPC)*. IEEE. 2016, pp. 52–61.

[46] Ruth A Harris, Michael Barall, Brad Aagaard, Shuo Ma, Daniel Roten, Kim Olsen, Benchun Duan, Dunyu Liu, Bin Luo, Kangchen Bai, et al. "A suite of exercises for verifying dynamic earthquake rupture codes". In: *Seismological Research Letters* 89.3 (2018), pp. 1146–1162.

[47] Ruth A Harris, Michael Barall, R Archuleta, E Dunham, B Aagaard, Jean Paul Ampuero, Harsha Bhat, V Cruz-Atienza, L Dalguer, Phillip Dawson, et al. "The SCEC/USGS dynamic earthquake rupture code verification exercise". In: *Seismological Research Letters* 80.1 (2009), pp. 119–126.

[48] Hemanta Hazarika, Divyesh Rohit, Siavash Manafi Khajeh Pasha, Tsubasa Maeda, Irsyam Masyhur, Ardy Arsyad, and Sukiman Nurdin. "Large distance flow-slide at Jono-Oge due to the 2018 Sulawesi Earthquake, Indonesia". In: *Soils and Foundations* 61.1 (2021), pp. 239–255.

[49] Alexander Heinecke, Alexander Breuer, Michael Bader, and Pradeep Dubey. "High order seismic simulations on the Intel Xeon Phi processor (Knights Landing)". In: *International Conference on High Performance Computing*. Springer. 2016, pp. 343–362.

[50] Alexander Heinecke, Alexander Breuer, Sebastian Rettenberger, Michael Bader, Alice-Agnes Gabriel, Christian Pelties, Arndt Bode, William Barth, Xiang-Ke Liao, Karthikeyan Vaidyanathan, et al. "Petascale high order dynamic rupture earthquake simulations on heterogeneous supercomputers". In: *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE. 2014, pp. 3–14.

[51] Alexander Heinecke, Greg Henry, Maxwell Hutchinson, and Hans Pabst. "LIBXSMM: accelerating small matrix multiplications by runtime code generation". In: *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE. 2016, pp. 981–991.

[52] Verena Hermann. "ADER-DG-Analysis, further Development and Applications". PhD thesis. lmu, 2011.

[53] Jan S Hesthaven and Tim Warburton. *Nodal discontinuous Galerkin methods: algorithms, analysis, and applications*. Springer Science & Business Media, 2007.

[54] Tetsuya Hoshino, Naoya Maruyama, Satoshi Matsuoka, and Ryoji Takaki. "CUDA vs OpenACC: Performance case studies with kernel benchmarks and a memory-bound CFD application". In: *2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*. IEEE. 2013, pp. 136–143.

[55] Susan E Hough, Eric Thompson, Grace A Parker, Robert W Graves, Kenneth W Hudnut, Jason Patton, Timothy Dawson, Tyler Ladinsky, Michael Oskin, Krittanon Sirorattanakul, et al. "Near-field ground motions from the July 2019 Ridgecrest, California, earthquake sequence". In: *Seismological Research Letters* 91.3 (2020), pp. 1542–1555.

[56] Yoshiaki Ida. "Cohesive force across the tip of a longitudinal-shear crack and Griffith's specific surface energy". In: *Journal of Geophysical Research* 77.20 (1972), pp. 3796–3805.

[57] Zhe Jia, Zeyu Jin, Mathilde Marchandon, Thomas Ulrich, Alice-Agnes Gabriel, Wenyuan Fan, Peter Shearer, Xiaoyu Zou, John Rekoske, Fatih Bulut, et al. "The complex dynamics of the 2023 Kahramanmaraş, Turkey, M w 7.8-7.7 earthquake doublet". In: *Science* 381.6661 (2023), pp. 985–990.

[58] Ali Karakus, Noel Chalmers, K Świrydowicz, and Tim Warburton. "A GPU accelerated discontinuous Galerkin incompressible flow solver". In: *Journal of Computational Physics* 390 (2019), pp. 380–404.

[59] George Karypis and Vipin Kumar. "A Coarse-Grain Parallel Formulation of Multilevel k-way Graph Partitioning Algorithm." In: *PPSC*. 1997.

[60] George Karypis, Kirk Schloegel, and Vipin Kumar. "Parmetis: Parallel graph partitioning and sparse matrix ordering library". In: (1997).

[61] Martin Käser and Michael 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.

[62] Martin Käser, P Martin Mai, and Michael Dumbser. "Accurate calculation of fault-rupture models using the high-order discontinuous Galerkin method on tetrahedral meshes". In: *Bulletin of the Seismological Society of America* 97.5 (2007), pp. 1570–1586.

[63] Ronan Keryell, Ruyman Reyes, and Lee Howes. "Khronos SYCL for OpenCL: a tutorial". In: *Proceedings of the 3rd International Workshop on OpenCL*. 2015, pp. 1–1.

[64] Mikhail Khalilov and Alexey Timoveev. "Performance analysis of CUDA, OpenACC and OpenMP programming models on TESLA V100 GPU". In: *Journal of Physics: Conference Series*. Vol. 1740. 1. IOP Publishing. 2021, p. 012056.

[65] Andrew C Kirby and Dimitri J Mavriplis. "Gpu-accelerated discontinuous galerkin methods: 30x speedup on 345 billion unknowns". In: *2020 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE. 2020, pp. 1–7.

[66] Andreas Klöckner, Tim Warburton, Jeff Bridge, and Jan S Hesthaven. "Nodal discontinuous Galerkin methods on graphics processors". In: *Journal of Computational Physics* 228.21 (2009), pp. 7863–7882.

[67] Marcin Knap and Paweł Czarnul. "Performance evaluation of unified memory with prefetching and oversubscription for selected parallel cuda applications on nvidia pascal and volta gpus". In: *The Journal of Supercomputing* 75.11 (2019), pp. 7625–7645.

[68] Lukas Krenz, Carsten Uphoff, Thomas Ulrich, Alice-Agnes Gabriel, Lauren S Abrahams, Eric M Dunham, and Michael Bader. "3D acoustic-elastic coupling with gravity: the dynamics of the 2018 Palu, Sulawesi earthquake and tsunami". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2021, pp. 1–14.

[69] Lawrence Livermore National Laboratory. *RAJA Performance Portability Layer*. June 2023. url: https://github.com/LLNL/RAJA.

[70] Network-Based Computing Laboratory. *OSU Micro-Benchmarks*. June 2023. url: https://mvapich.cse.ohio-state.edu/benchmarks.

[71] Chi-Chung Lam, P Sadayappan, and Rephael Wenger. "Optimal reordering and mapping of a class of nested-loops for parallel execution". In: *International Workshop on Languages and Compilers for Parallel Computing*. Springer. 1996, pp. 315–329.

[72] Chi-Chung Lam, P Sadayappan, and Rephael Wenger. "Optimal reordering and mapping of a class of nested-loops for parallel execution". In: *Languages and Compilers for Parallel Computing: 9th International Workshop, LCPC'96 San Jose, California, USA, August 8–10, 1996 Proceedings 9*. Springer. 1997, pp. 315–329.

[73] Dong Uk Lee, Kyung Whan Kim, Kwan Weon Kim, Kang Seol Lee, Sang Jin Byeon, Jae Hwan Kim, Jin Hee Cho, Jaejin Lee, and Jun Hyun Chun. "A 1.2 V 8 Gb 8-channel 128 GB/s high-bandwidth memory (HBM) stacked DRAM with effective I/O test circuits". In: *IEEE Journal of Solid-State Circuits* 50.1 (2014), pp. 191–203.

[74] Randall J LeVeque et al. *Finite volume methods for hyperbolic problems*. Vol. 31. Cambridge university press, 2002.

[75] Ao Li, Bojian Zheng, Gennady Pekhimenko, and Fan Long. "Automatic horizontal fusion for GPU kernels". In: 2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO). IEEE. 2022, pp. 14–27.

[76] Tianyi Li and Allan M Rubin. "A microscopic model of rate and state friction evolution". In: *Journal of Geophysical Research: Solid Earth* 122.8 (2017), pp. 6431–6453.

[77] Guoping Long, Jun Yang, Kai Zhu, and Wei Lin. "Fusionstitching: Deep fusion and code generation for tensorflow computations on gpus". In: *arXiv preprint arXiv:1811.05213* (2018).

[78] Gabriel C Lotto and Eric M Dunham. "High-order finite difference modeling of tsunami generation in a compressible ocean from offshore earthquakes". In: *Computational Geosciences* 19.2 (2015), pp. 327–340.

[79] Jacob Lubliner. *Plasticity theory*. Courier Corporation, 2008.

[80] Jacob Lubliner and Panayiotis Papadopoulos. *Introduction to solid mechanics*. Springer, 2016.

[81] Karthik Vadambacheri Manian, AA Ammar, Amit Ruhela, C-H Chu, Hari Subramoni, and Dhabaleswar K Panda. "Characterizing cuda unified memory (um)-aware mpi designs on modern gpu architectures". In: *Proceedings of the 12th Workshop on General Purpose Processing Using GPUs*. 2019, pp. 43–52.

[82] Karthik Vadambacheri Manian, Ching-Hsiang Chu, Ammar Ahmad Awan, Kawthar Shafie Khorassani, Hari Subramoni, and DK Panda. "OMB-UM: Design, implementation, and evaluation of CUDA unified memory aware MPI benchmarks". In: *2019 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*. IEEE. 2019, pp. 82–92.

[83] Stefano Markidis, Steven Wei Der Chien, Erwin Laure, Ivy Bo Peng, and Jeffrey S Vetter. "Nvidia tensor core programmability, performance & precision". In: *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE. 2018, pp. 522–531.

[84] Chris Marone. "The effect of loading rate on static friction and the rate of fault healing during the earthquake cycle". In: *Nature* 391.6662 (1998), pp. 69–72.

[85] Veronica Melesse Vergara, Reuben Budiardja, Matt Davis, Matthew Ezell, Jesse Hanley, Christopher Zimmer, Michael Brim, Wael Elwasif, and Dan Dietz. *Approaching the Final Frontier: Lessons Learned from the Deployment of HPE/Cray EX Spock and Crusher supercomputers*. Tech. rep. Oak Ridge National Lab.(ORNL), Oak Ridge, TN (United States), 2022.

[86] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard Version 4.0*. June 2021. url: https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf.

[87] Axel Modave, Andreas Atle, Jesse Chan, and Tim Warburton. "A GPU-accelerated nodal discontinuous Galerkin method with high-order absorbing boundary conditions and corner/edge compatibility". In: *International Journal for Numerical Methods in Engineering* 112.11 (2017), pp. 1659–1686.

[88] Axel Modave, Amik St-Cyr, and Tim Warburton. "GPU performance analysis of a nodal discontinuous Galerkin method for acoustic and elastic models". In: *Computers & Geosciences* 91 (2016), pp. 64–76.

[89] Dawei Mu, Po Chen, and Liqiang Wang. "Accelerating the discontinuous Galerkin method for seismic wave propagation simulations using the graphic processing unit (GPU)—single-GPU implementation". In: *Computers & Geosciences* 51 (2013), pp. 282–292.

[90] Abdul Muhari, Fumihiko Imamura, Taro Arikawa, Aradea R Hakim, and Bagus Afriyanto. "Solving the puzzle of the September 2018 Palu, Indonesia, tsunami mystery: clues from the tsunami waveform and the initial field survey data". In: *Journal of Disaster Research* 13.Scientific Communication (2018), sc20181108.

[91] NVIDIA. *NVIDIA Tesla P100: Whitepaper*. 2019. url: https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf.

[92] Swaroop Pophale, Swen Boehm, and Verónica G Vergara Larrea. "Comparing high performance computing accelerator programming models". In: *High Performance Computing: ISC High Performance 2019 International Workshops, Frankfurt, Germany, June 16-20, 2019, Revised Selected Papers 34*. Springer. 2019, pp. 155–168.

[93] The SCEC/USGS Spontaneous Rupture Code Verification Project. *Documentation for The Problem, Version 5*. June 2023. url: https://strike.scec.org/cvws/tpv5docs.html.

[94] The SCEC/USGS Spontaneous Rupture Code Verification Project. *Documentation for The Problem, Versions 101 and 102*. June 2023. url: https://strike.scec.org/cvws/tpv101_102docs.html.

[95] The SCEC/USGS Spontaneous Rupture Code Verification Project. *Documentation for The Problem, Versions 12 and 13*. June 2023. url: https://strike.scec.org/cvws/tpv12_13docs.html.

[96] Max Rietmann. "Local time stepping on high performance computing architectures". In: (2015).

[97] Max Rietmann, Marcus Grote, Daniel Peter, and Olaf Schenk. "Newmark local time stepping on high-performance computing architectures". In: *Journal of Computational Physics* 334 (2017), pp. 308–326.

[98] Max Rietmann, Daniel Peter, Olaf Schenk, Bora Uçar, and Marcus Grote. "Load-balanced local time stepping for large-scale wave propagation". In: *2015 IEEE International Parallel and Distributed Processing Symposium*. IEEE. 2015, pp. 925–935.

[99] Zachary E Ross, Benjamín Idini, Zhe Jia, Oliver L Stephenson, Minyan Zhong, Xin Wang, Zhongwen Zhan, Mark Simons, Eric J Fielding, Sang-Ho Yun, et al. "Hierarchical interlocked orthogonal faulting in the 2019 Ridgecrest earthquake sequence". In: *Science* 366.6463 (2019), pp. 346–351.

[100] Tatsuhiko Saito. *Tsunami generation and propagation*. Springer, 2019.

[101] Christopher H Scholz. "Earthquakes and friction laws". In: *Nature* 391.6662 (1998), pp. 37–42.

[102] Jonas Schreier. "Optimization of small matrix multiplication kernels on Arm". In: (2021).

[103] Pavel Shamis, Manjunath Gorentla Venkata, M Graham Lopez, Matthew B Baker, Oscar Hernandez, Yossi Itigin, Mike Dubman, Gilad Shainer, Richard L Graham, Liran Liss, et al. "UCX: an open source framework for HPC network APIs and beyond". In: *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*. IEEE. 2015, pp. 40–43.

[104] Martin Siebenborn, Volker Schulz, and Stephan Schmidt. "A curved-element unstructured discontinuous Galerkin method on GPUs for the Euler equations". In: *Computing and Visualization in Science* 15 (2012), pp. 61–73.

[105] Paul Springer and Paolo Bientinesi. "Design of a high-performance GEMM-like tensor–tensor multiplication". In: *ACM Transactions on Mathematical Software (TOMS)* 44.3 (2018), pp. 1–29.

[106] Arthur H Stroud and Don Secrest. "Gaussian quadrature formulas". In: *(No Title)* (1966).

[107] Jafril Tanjung, Yasushi Sanada, Fajar Nugroho, Syafri Wardi, et al. "Seismic analysis of damaged buildings based on postearthquake investigation of the 2018 Palu Earthquake". In: *GEOMATE Journal* 18.70 (2020), pp. 116–122.

[108] Taufiq Taufiqurrahman, Alice-Agnes Gabriel, Duo Li, Thomas Ulrich, Bo Li, Sara Carena, Alessandro Verdecchia, and František Gallovič. "Dynamics, interactions and delays of the 2019 Ridgecrest rupture sequence". In: *Nature* (2023), pp. 1–8.

[109] Netgen Team. *Netgen*. Version 5.3.1. 2023. url: https://ngsolve.org.

[110] Eleuterio F Toro. *Riemann solvers and numerical methods for fluid dynamics: a practical introduction*. Springer Science & Business Media, 2013.

[111] Matteo Turisini, Giorgio Amati, and Mirko Cestari. "LEONARDO: A Pan-European Pre-Exascale Supercomputer for HPC and AI Applications". In: *arXiv preprint arXiv:2307.16885* (2023).

[112] Thomas Ulrich. "On the rupture processes of large earthquakes using three-dimensional data-integrated dynamic rupture simulations". PhD thesis. lmu, 2020.

[113] Thomas Ulrich, Alice-Agnes Gabriel, Jean-Paul Ampuero, and Wenbin Xu. "Dynamic viability of the 2016 Mw 7.8 Kaikōura earthquake cascade on weak crustal faults". In: *Nature communications* 10.1 (2019), p. 1213.

[114] Carsten Uphoff. "Flexible model extension and optimisation for earthquake simulations at extreme scales". PhD thesis. Technische Universität München, 2020.

[115] Carsten Uphoff and Michael Bader. "Generating high performance matrix kernels for earthquake simulations with viscoelastic attenuation". In: *2016 International Conference on High Performance Computing & Simulation (HPCS)*. IEEE. 2016, pp. 908–916.

[116] Carsten Uphoff and Michael Bader. "Yet another tensor toolbox for discontinuous Galerkin methods and other applications". In: *ACM Transactions on Mathematical Software (TOMS)* 46.4 (2020), pp. 1–40.

[117] Carsten Uphoff, Sebastian Rettenberger, Michael Bader, Elizabeth H Madden, Thomas Ulrich, Stephanie Wollherr, and Alice-Agnes Gabriel. "Extreme scale multi-physics simulations of the tsunamigenic 2004 sumatra megathrust earthquake". In: *Proceedings of the international conference for high performance computing, networking, storage and analysis*. 2017, pp. 1–16.

[118] Mohamed Wahib and Naoya Maruyama. "Scalable kernel fusion for memory-bound GPU applications". In: SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. IEEE. 2014, pp. 191–202.

[119] Guibin Wang, YiSong Lin, and Wei Yi. "Kernel fusion: An effective method for better power efficiency on multithreaded GPU". In: 2010 IEEE/ACM Int'l Conference on Green Computing and Communications & Int'l Conference on Cyber, Physical and Social Computing. IEEE. 2010, pp. 344–350.

[120] Zoltán Wéber. "Estimating source time function and moment tensor from moment tensor rate functions by constrained L 1 norm minimization". In: *Geophysical Journal International* 178.2 (2009), pp. 889–900.

[121] W Hwu Wen-mei. *Heterogeneous System Architecture: A new compute platform infrastructure*. Morgan Kaufmann, 2015.

[122] Niklas Wintermeyer, Andrew R Winters, Gregor J Gassner, and Timothy Warburton. "An entropy stable discontinuous Galerkin method for the shallow water equations on curvilinear meshes with wet/dry fronts accelerated by GPUs". In: *Journal of Computational Physics* 375 (2018), pp. 447–480.

[123] Sebastian Wolf, Martin Galis, Carsten Uphoff, Alice-Agnes Gabriel, Peter Moczo, David Gregor, and Michael Bader. "An efficient ADER-DG local time stepping scheme for 3D HPC simulation of seismic waves in poroelastic media". In: *Journal of Computational Physics* 455 (2022), p. 110886.

[124] Stephanie Wollherr, Alice-Agnes Gabriel, and P Martin Mai. "Landers 1992 "reloaded": Integrative dynamic earthquake rupture modeling". In: *Journal of Geophysical Research: Solid Earth* 124.7 (2019), pp. 6666–6702.

[125] Stephanie Wollherr, Alice-Agnes Gabriel, and Carsten Uphoff. "Off-fault plasticity in three-dimensional dynamic rupture simulations using a modal Discontinuous Galerkin method on unstructured meshes: implementation, verification and application". In: *Geophysical Journal International* 214.3 (2018), pp. 1556–1584.

[126] Xinhui Wu, Ethan J Kubatko, and Jesse Chan. "High-order entropy stable discontinuous Galerkin methods for the shallow water equations: curved triangular meshes and GPU acceleration". In: *Computers & Mathematics with Applications* 82 (2021), pp. 179–199.

[127] Yidong Xia, Lixiang Luo, and Hong Luo. "OpenACC-based GPU acceleration of a 3-D unstructured discontinuous galerkin method". In: *52nd Aerospace Sciences Meeting*. 2014, p. 1129.

[128] Han Yue, Jianbao Sun, Min Wang, Zhengkang Shen, Mingjia Li, Lian Xue, Weifan Lu, Yijian Zhou, Chunmei Ren, and Thorne Lay. "The 2019 Ridgecrest, California earthquake sequence: Evolution of seismic and aseismic slip on an orthogonal fault system". In: *Earth and Planetary Science Letters* 570 (2021), p. 117066.

List of Figures

List of Tables

Code Listings

List of Algorithms