Fakultät für Informatik
Technische Universität München

Technische Universität München

Fakultät für Informatik

Lehrstuhl für Wissenschaftliches Rechnen

Template-based Code Generation for a Customizable High-Performance Hyperbolic PDE Engine

Jean-Matthieu Gallard

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

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:

Prof. Dr. Julien Gagneur

Prüfende der Dissertation:

1. Prof. Dr. Michael Georg Bader
2. Prof. Dr. Tobias Weinzierl

Die Dissertation wurde am 22.07.2021 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 26.09.2021 angenommen.

Abstract

Systems of hyperbolic partial differential equations (PDE) are used to model physical phenomena important to numerous scientific fields, from neutron star mergers in astrophysics to earthquakes and tsunamis in seismology. ExaHyPE ("An Exascale Hyperbolic PDE Engine") is a customizable high-performance engine, that can be used by an interdisciplinary research team to simulate these phenomena. It employs a discontinuous Galerkin (DG) method combined with explicit one-step arbitrary high-order derivative (ADER) time-stepping on adaptive Cartesian meshes. The ADER-DG numerical scheme is broken down into customizable cell-local single-thread compute kernels, which form the performance-critical components of the engine. These kernels can be optimized toward a target hardware, and also tailored toward a given application, giving the engine its adaptability. To customize its kernel, ExaHyPE relies on code generation.

My main contribution to ExaHyPE is the development of the Kernel Generator. To guide its design, I defined three roles ExaHyPE's users could take, and distinguished implementing the kernels' algorithms from performing the architecture-aware optimizations. Taking inspiration from web application development practices, the Kernel Generator follows a Model-View-Controller architectural pattern and uses the Jinja2 template engine. Jinja2's template language abstracts the code to be generated, isolating low-level optimization macros, from architecture-oblivious algorithmic templates using them. This separation decouples and streamlines the workflows of ExaHyPE's identified user roles, which is illustrated by the development of the ExaSeis application from the point of view of each role. Furthermore, I optimized the generated kernels toward modern CPU architectures, focusing on Skylake used by the SuperMUC-NG supercomputer. I employed aggressive vectorization and reformulated tensor contractions as Loop-over-GEMM using the specialized BLAS library LIBXSMM as backend. The performance-critical kernels were further optimized in incremental steps, each introducing a new kernel variant: first re-engineering the numerical schemes to increase cache-awareness and reduce memory stalls, then using hybrid data layouts to increase vectorization opportunities. Using the most advanced variants, ExaHyPE's applications can be fully vectorized, with a Navier-Stokes solver achieving 31.7% of peak performance on a single node of SuperMUC-NG.

Acknowledgements

First of all, I would like to thank my supervisor Michael Bader, for his continuous support and guidance throughout my dissertation project. Without him and Nikola Tchipev, I would not have started this PhD project.

My gratitude goes to all members of the ExaHyPE consortium who made this research possible, in particular to Anne Reinarz, Leonhard Rannabauer, and Philipp Samfaß, for their support within the TUM team, as well as to Dominic Charrier and Sven Köppel, for our fruitful coding weeks pushing the project forward. I am very grateful to ExaHyPE's PIs Michael Dumbser and Tobias Weinzierl, for welcoming me in Trento and Durham, and for showing me other aspects of the engine and its underlying mathematical model. I would also like to thank Carsten Uphoff, his valuable insights into code optimization gave me new perspectives on ExaHyPE and how to further optimize it.

I am deeply thankfull to all my colleagues at the chair of SCCS in Garching, for the positive and relaxing atmosphere at the chair, and to Hans-Joachim Bungartz for fostering it.

Finally, I want to thank my family and my friends, for their invaluable support during this journey.

ExaHyPE has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement no. 671698 (`www.exahype.eu`). I also acknowledge support by the Leibniz Supecomputing Centre (`www.lrz.de`), which provided the computing resources on SuperMUC-NG (grant nos. pr48ma & pr83no).

Contents

Part I

Introduction to an ADER-DG solver

1 Introduction

1.1 Motivations for a customizable exascale hyperbolic PDE engine

In our modern understanding of physics, the evolution of many types of systems can be determined by systems of Partial Differential Equations (PDE). In particular, physical phenomena propagating in a wave-like form are described by hyperbolic PDE systems. Some of these phenomena, such as earthquakes and tsunamis, can have a direct impact on the life of millions of people. Whereas others, for instance the gravitational waves produced during the merger of a binary neutron star system, provide insight into the fundamental laws of physics and the history of our universe. As the hyperbolic PDE systems describing these phenomena often do not have a usable analytical solution, the only way to compute the evolution of these systems is to do it step by step with a numerical solver. Grand challenge simulations, such as neutron star mergers, are problems of this kind of particular scientific interest and difficulty.

Developing a numerical solver for a grand challenge simulation requires not only expertise in the phenomenon's specific domain area, but also in numerical schemes to develop sophisticated high-order numerical methods. The simulation itself uses a large amount of computational power to be performed with a high enough resolution, required to produce scientifically useful results. For this reason, the solver also needs to be designed for modern supercomputing platforms, which requires expertise in high performance computing and code optimization to fully exploit the performance potential of modern hardware architectures. Therefore, the simulation of a grand challenge from the ground up requires the combined work of an interdisciplinary research team over multiple months or even years.

The consortium behind the ExaHyPE project ("An Exascale Hyperbolic PDE Engine") aims to prevent such long development processes by providing a software engine for modelling and simulating a wide range of hyperbolic PDE systems. To solve systems of this kind, ExaHyPE employs a state of the art discontinuous Galerkin (DG) method combined with explicit one-step arbitrary high-order derivative (ADER) time-stepping on adaptive Cartesian meshes. As a high-performance engine, domain experts can use ExaHyPE's generic canonical PDE system to model various kind of problems, including grand challenges, and thus simulate them on modern supercomputing platforms.

Furthermore, ExaHyPE's ADER-DG numerical scheme is broken down into customizable cell-local single-thread compute kernels, which are executed in parallel and form the performance-critical components of the engine. Using code generation, the kernels are au-

tomatically optimized toward a target hardware, and tailored toward a given application. The code generation itself is customizable, so that the produced kernels used by the engine can be modified, should new requirements arise. Experts in numerical methods can implement more advanced numerical schemes as new kernel variants, and experts in code optimization can add support for new hardware architectures to the kernels' fine tuning. The architecture of the code generation utilities aims at streamlining its expansions, and also clearly separates the algorithms' abstractions from the low-level architecture specific optimizations so that experts in one field can work independently of the other.

This thesis focuses on the code generation in ExaHyPE. First, I present how code generation structures the overall design of ExaHyPE and how this and the architecture of the code generation utilities themselves streamlines the workflow of an interdisciplinary research team implementing a grand challenge simulation. Then, I introduce my optimization of the generated kernels toward Intel's Skylake CPU architecture, with both low-level code optimization and algorithm modifications such as the introduction of new formulations of some kernels and the use of hybrid data layouts.

1.2 Thesis structure

This thesis is structured in four parts.

Part I provides introductory information. Chapter 2, outlines the mathematical background of ExaHyPE and introduces the ADER-DG algorithm and its kernels.

Part II focuses on the design of ExaHyPE and the code generation utilities. Chapter 3 introduces three user roles and with them justifies the use of code generation and presents the modular design of ExaHyPE. Chapter 4 discusses how I redesigned the code generation utilities using a Model-View-Controller architectural pattern and the Jinja2 template engine, to fulfill the previously identified user roles' requirements. Chapter 5 presents a real use case of application implementation, with the development of the ExaSeis application from the point of each of the user roles.

Part III discusses the optimization of ExaHyPE's kernels. Chapter 6 introduces the code optimization background necessary for the later chapters of this part. Chapter 7 presents my overall optimization of kernels, using modern compiler capabilities and a Loop-over-GEMM formulation to optimize the tensor contractions with a BLAS library. Chapter 8 focuses on my optimization of the linear SpaceTimePredictor kernel in successive steps, each introducing a new kernel variant and tackling the bottlenecks I identified when benchmarking the previous variant. Chapter 9 outlines further proofs of concepts to gain insight on potential future work.

Part IV concludes this thesis with a summary of my scientific contribution.

2 An ADER-DG Solver for hyperbolic PDE systems

This chapter introduces the numerical concept behind ExaHyPE. In Section 2.1, we discuss hyperbolic PDE systems and introduce the canonical PDE system solved by ExaHyPE. It can be used to formulate the PDE systems of various physical phenomena, that can therefore be simulated with ExaHyPE. The ADER-DG numerical scheme is then briefly outlined in Section 2.2, and we see how its steps can be isolated into independent kernels. The kernels are the critical components to customize and optimize ExaHyPE, their generation and optimization is the focus of this thesis.

As the numerical scheme itself is not the focus of this work, only the key concepts are outlined. A more detailled description of the ADER-DG scheme as well as its mathematical justifications can be found in [1, 2].

2.1 Hyperbolic PDE systems and related applications

2.1.1 ExaHyPE's Canonical PDE System

Hyperbolic PDE systems can be used to model a wide range of phenomena involving waves. ExaHyPE's solves systems that can be expressed in the form of the following *canonical PDE system* (2.1):

$$\underbrace{\mathbf{P}}_{\text{material matrix}} \frac{\partial}{\partial t}\mathbf{Q} + \nabla \cdot \underbrace{\mathbf{F}(\mathbf{Q})}_{\text{flux}} + \underbrace{\sum_{i=1}^{d} \mathbf{B}_i(\mathbf{Q}) \frac{\partial \mathbf{Q}}{\partial x_i}}_{\text{ncp}} = \underbrace{\mathbf{S}(\mathbf{Q})}_{\text{source}} + \underbrace{\sum_{i=1}^{n_{\text{ps}}} \delta_i}_{\text{point sources}} \quad , \tag{2.1}$$

where on a computational domain $\Omega \subset \mathbb{R}^d$ with $\mathbf{Q} : \Omega \subset \mathbb{R}^d \mapsto \mathbb{R}^\nu$ being the state vector of the ν conserved variables:

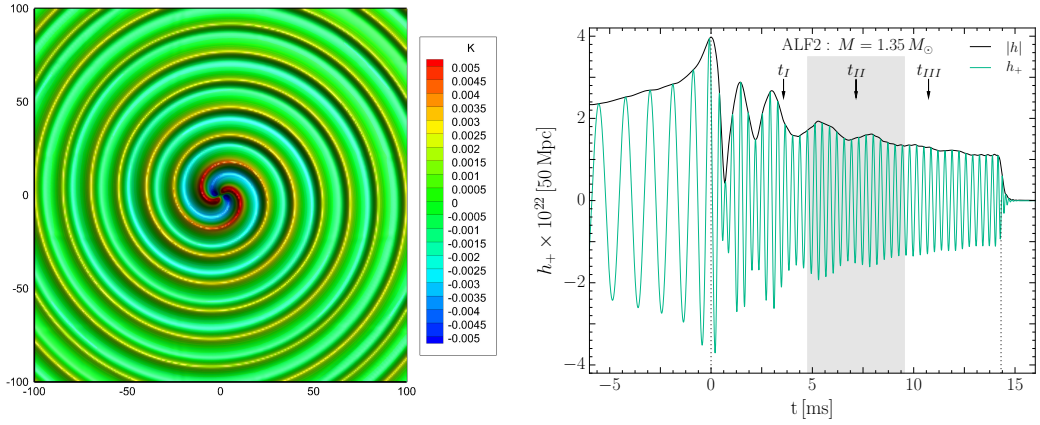
- $\mathbf{P}$ is the *material matrix* term,
- $\mathbf{F}(\mathbf{Q})$ is the *flux* term,
- $\mathbf{B}_i(\mathbf{Q})$ represents a non-conservative part and thus $\sum_{i=1}^{d} \mathbf{B}_i(\mathbf{Q}) \frac{\partial \mathbf{Q}}{\partial x_i}$ is the *non-conservative product* (ncp) term,
- $\mathbf{S}(\mathbf{Q})$ is the *source* term,
- δ_i are the given n_{ps} moment-tensor *point sources*.

This canonical PDE system is highly customizable as each term can be disabled if desired. For example, if no material matrix is required to model a given phenomenon, then it can be set to the identity matrix in the application. Likewise setting $\mathbf{S}(\mathbf{Q}) = 0$ removes this particular term from the system. ExaHyPE's numerical scheme automatically adapts itself to its application's PDE system's formulation to avoid wasting computations on unused terms.

2.1.2 Grand challenge simulations in astrophysics and seismology

In its initial project proposal, ExaHyPE aimed to be able to perform two grand challenge simulations. These applications PDE system formulations are taken from ExaHyPE's release paper [3].

Binary neutron stars merger with GRMHD



(a) Extrinsic curvature

(b) Gravitational waves at 50 Mpc

Figure 2.1: **(a)**: Contour colors of the trace of the extrinsic curvature generated by two rotating Gaussian density distributions, figure taken from [4]. **(b)**: Gravitational waves for the *ALF2-M135* binary at a distance of 50 Mpc, figure taken from [5].

The first grand challenge simulation originates from astrophysics with the simulation of gravitational waves emitted by the merger of a binary neutron stars, see e.g [6, 7, 8]. Figure 2.1a illustrates the typical wavefield generated by a rotating binary, here idealized Gaussian density distributions, and Figure 2.1b shows the gravitational waves predicted by a simulation of a neutron stars merger using the *ALF2-M135* model, performed by Hanauske et al. [9]. Here the merger lasted close to 15ms with the first doted line at $t = 0$ marking the start of the merger and the second one 14ms later the formation of the resulting black hole's event horizon. Interest in this topic grew over the last decade, with on the one hand the increasing computational capacity making such simulations possible, and on the other hand the gravitationnal wave detectors LIGO and Virgo coming online. The two detectors measured the gravitational waves emitted by a neutron star merger

for the first time in 2017 [10], hence providing real world measurements to compare the models to.

In astrophysics, a neutron star internal structure can be modelled as an electrically ideally conducting fluid with comparable hydrodynamic and electromagnetic forces. To do so the equations of classical magnetohydrodynamics (MHD) are used to model the corresponding fluid dynamics. Furthermore, due to the strong gravitational fields of such astrophysical objects, the background space-time is included in the model in the form of a non-conservative product using the standard $3+1$ split of General Relativity (GR) to decompose the four-dimensional space-time manifold into 3D hyper-surfaces parameterised by a time coordinate t [11].

Following the form of ExaHyPE's canonical PDE system (2.1), and using Einstein summation convention over repeated indexes, the resulting GRMHD model can be written as:

$$\frac{\partial}{\partial t}\underbrace{\begin{pmatrix}\sqrt{\gamma}D\\ \sqrt{\gamma}S_j\\ \sqrt{\gamma}\tau\\ \sqrt{\gamma}B^j\\ \phi\\ \alpha_j\\ \beta\\ \gamma_m\end{pmatrix}}_{=\mathbf{Q}} + \nabla\cdot\underbrace{\begin{pmatrix}\alpha v^i D-\beta^i D\\ \alpha T^i_j-\beta^i S_j\\ \alpha(S^i-v^i D)-\beta^i\tau\\ (\alpha v^i-\beta^i)B^j-(\alpha v^j-\beta^j)B^i\\ 0\\ 0\\ 0\\ 0\end{pmatrix}}_{=\mathbf{F}(\mathbf{Q})} + \underbrace{\begin{pmatrix}0\\ \sqrt{\gamma}(\tau\partial_j\alpha-\frac{1}{2}T^{ik}\partial_j\gamma_{ik}-T^j_i\partial_j\beta^i)\\ \sqrt{\gamma}(S^j\partial_j\alpha-\frac{1}{2}T^{ik}\beta^j\partial_j\gamma_{ik}-T^j_i\partial_j\beta^j)\\ -\beta^j\partial_i(\sqrt{\gamma}B^i)+\alpha\sqrt{\gamma}\gamma^{ji}\partial_i\phi\\ \sqrt{\gamma}\alpha c_h^2\partial_j(\sqrt{\gamma}B^i)-\beta^j\partial^j\phi\\ 0\\ 0\\ 0\end{pmatrix}}_{=\sum_{i=1}^3\mathbf{B}_i(\mathbf{Q})\frac{\partial\mathbf{Q}}{\partial x_i}} = 0,$$

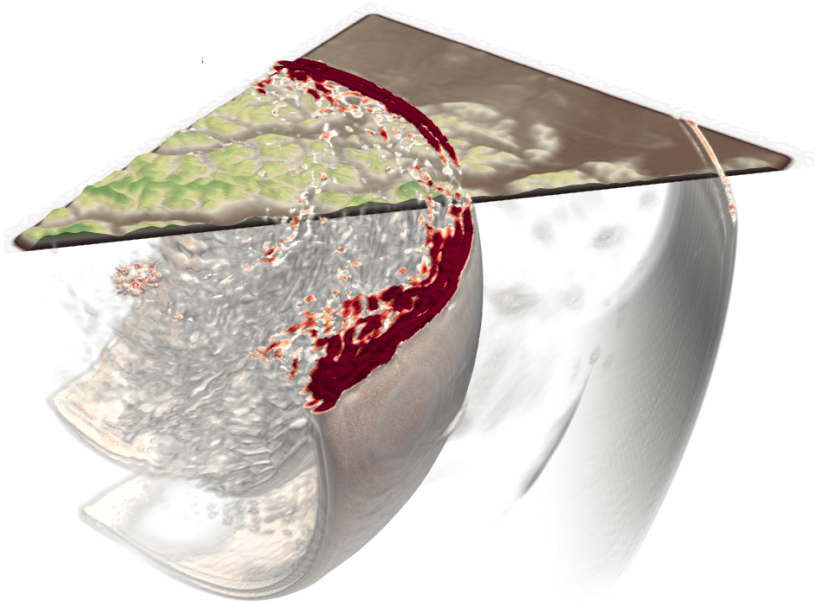
where $i,j=1,2,3$ and $m=1...6$.

The details of this formulation can be found in [11, 12, 13].

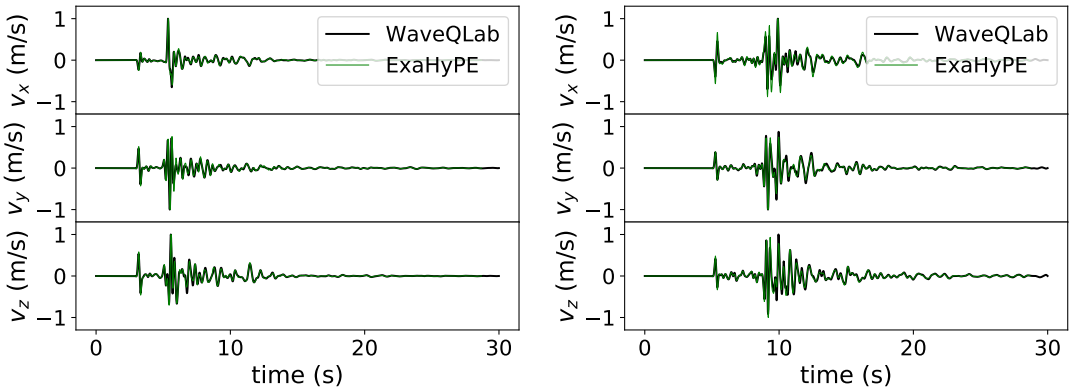
Waves propagation in elastic solids

Coming from seismology, the second grand challenge simulation concerns long-range seismic risk assessment. Earthquakes can be simulated by Hooke's law and the conservation of momentum to model the velocities, stress and strain of a heterogeneous medium [16]. The implementation of a related ExaHyPE application, ExaSeis, is presented and discussed in Chapter 5. Figure 2.2 shows some results obtained with ExaSeis, here the simulation of an earthquake at mount Zugspitze [15].

While ExaHyPE uses adaptive Cartesian meshes, they can be extended to allow the modelling of complex topography. A first approach, used by the ExaSeis application, maps ExaHyPE's adaptive Cartesian mesh to a complex topography via high-order curvilinear transformations [17, 18]. A second one, used in another application, represents the topography as a smooth field using a diffuse interface method [19].



(a) Seismic wavefield



(b) Seismograms

Figure 2.2: **(a)**: 3D snapshots of the absolute velocity of the propagating seismic wavefield for the Zugspitze model at $t = 15$, simulated with ExaSeis. **(b)**: resulting seismograms (green) compared with a reference implementation using the finite difference code WaveQLab [14]. Figures adapted from [15]

Ignoring mesh transformations, the propagation of waves in elastic solids is modeled by:

$$\frac{\partial}{\partial t}\underbrace{\begin{pmatrix}\sigma\\ \rho v\end{pmatrix}}_{=\mathbf{Q}} + \nabla\cdot\underbrace{\begin{pmatrix}0\\ \sigma\end{pmatrix}}_{=\mathbf{F}(\mathbf{Q})} + \underbrace{\begin{pmatrix}E(\lambda,\mu) & 0\\ 0 & 0\end{pmatrix}\cdot\nabla\begin{pmatrix}v\\ \sigma\end{pmatrix}}_{=\sum_{i=1}^{d}\mathbf{B}_i(\mathbf{Q})\,\frac{\partial\mathbf{Q}}{\partial x_i}} = \underbrace{\delta}_{=\sum_{i=0}^{1}\delta_i},$$

where the vector v denotes the velocity, ρ the mass density, and σ the stress tensor, which can be written in terms of its six independent components. The moment-tensor point source δ is used to model the initial impulsion at the epicenter of the earthquake.

2.1.3 Other applications

Other physical phenomena described by hyperbolic PDE systems can also be simulated using ExaHyPE. These formulations are taken from ExaHyPE's release paper [3].

Shallow Water Simulations

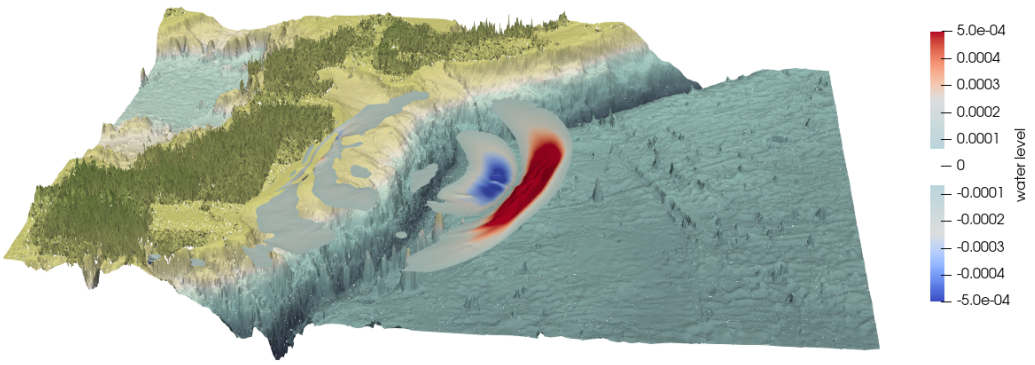


Figure 2.3: Simulation of the 2011 Tohoku tsunami using shallow water equations, figure adapted from [20]

The shallow water equations describe fluid flow in cases whose horizontal length scales are considerably greater than the vertical length scale. They are commonly used in atmospheric and oceanic modeling, for example to model tsunamis caused by underwater earthquakes, such as illustrated in Figure 2.3 with a simulation of the 2011 Tohoku tsunami performed, using ExaHyPE, by Rannabauer et al. [20]. Unlike the previous grand challenge simulations, shallow water simulations only use two spatial dimensions and a small amount of variables, with a much simpler PDE system. Thus, they are also useful for teaching purposes.

They can be written as:

$$\underbrace{\frac{\partial}{\partial t}\begin{pmatrix} h \\ hu \\ hv \\ b \end{pmatrix}}_{=\mathbf{Q}} + \nabla \cdot \underbrace{\begin{pmatrix} hu & hv \\ hu^2 & huv \\ huv & hv^2 \\ 0 & 0 \end{pmatrix}}_{=\mathbf{F}(\mathbf{Q})} + \underbrace{\begin{pmatrix} 0 \\ hg\,\partial_x(b+h) \\ hg\,\partial_y(b+h) \\ 0 \end{pmatrix}}_{=\sum_{i=1}^{2} \mathbf{B}_i(\mathbf{Q})\,\frac{\partial \mathbf{Q}}{\partial x_i}} = 0,$$

where h denotes the height of the water column, (u, v) the horizontal flow velocity, b the bathymetry and g the gravity constant.

Compressible Navier-Stokes models

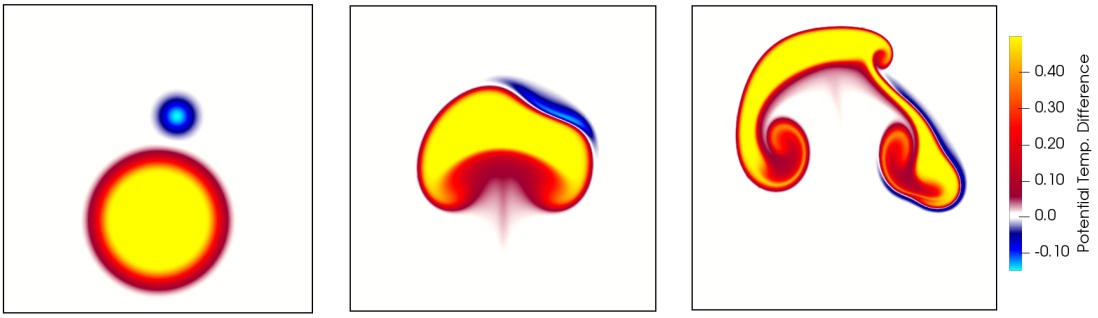


Figure 2.4: Collision of two air bubbles at different temperatures in ExaHyPE, using the compressible Navier-Stokes equations. Figures adapted from [21].

With some minor modifications to the canonical PDE system (2.1), ExaHyPE is extensible to some non-hyperbolic equations, such as the parabolic compressible Navier-Stokes equations [22]. Figure 2.4 shows the collision of two air bubbles at different temperatures, simulated using these equations in ExaHyPE by Krenz et al. [21].

Navier-Stokes models, used to simulate the dynamics of viscous fluids, are given by:

$$\underbrace{\frac{\partial}{\partial t}\begin{pmatrix} \rho \\ \rho v \\ \rho E \end{pmatrix}}_{=\mathbf{Q}} + \nabla \cdot \underbrace{\begin{pmatrix} \rho v \\ v \otimes \rho v + Ip + \sigma(\mathbf{Q}, \nabla\mathbf{Q}) \\ v \cdot (I\rho E + Ip + \sigma(\mathbf{Q}, \nabla\mathbf{Q})) - \kappa\nabla(T) \end{pmatrix}}_{=\mathbf{F}(\mathbf{Q},\nabla\mathbf{Q})} = \underbrace{\begin{pmatrix} 0 \\ -gk\rho \\ 0 \end{pmatrix}}_{=\mathbf{S}(\mathbf{Q})},$$

where viscous effects in the flux term are modelled by the stress tensor $\sigma(\mathbf{Q}, \nabla\mathbf{Q})$. To accomodate this application's requirement, the flux from (2.1) was expanded to allow $\nabla\mathbf{Q}$ as input in a previous work presented in [23].

Further details on the implementation of these equations and the related applications can be found in [21, 24].

2.2 ADER-DG solver

To numerically solve any PDE system taking the canonical form (2.1), ExaHyPE employs a high-order discontinuous Galerkin (DG) approach. First introduced by Reed et al. [25] for the neutron transport equation in nuclear physics, DG schemes were subsequently extended to general hyperbolic PDE systems in a serie of papers by Cockburn et al. [26, 27, 28, 29, 30].

Within the DG framework, the *arbitrary high-order derivative* (ADER) DG approach, first introduced by Toro and Titarev [31], allows for higher-order accuracy in time and space. The ADER-DG scheme avoids the problem of increasing number of stages for increasing polynomial degree exhibited by classical Runge-Kutta-DG schemes, leading to better performance and time-to-solution [32]. ExaHyPE uses the ADER-DG formulation proposed by Dumbser et al. [2].

2.2.1 Numerical method outline

We here only briefly outline the numerical method, detailled in [2], to introduce the corresponding algorithm. Furthermore for the sake of simplicity, we restrict ourselves to a simplified form of the canonical PDE system using only a flux term:

$$\frac{\partial}{\partial t}\mathbf{Q} + \nabla \cdot \mathbf{F}(\mathbf{Q}) = 0, \text{ on } \Omega \subset \mathbb{R}^d, d \in 2, 3. \tag{2.2}$$

To solve this PDE system using the ADER-DG scheme, the computational domain Ω is discretized on a space-time Cartesian grid Γ of cubic cells K, and the state vector $\mathbf{Q}$ is replaced by q_h, a piecewise high-order polynomial of degree N. Within each mesh element (cell), the polynomial basis of q_h is constructed as tensor products of Lagrange polynomials over Gauss-Legendre nodes.

The ADER-DG methods works in three steps using a weak formulation of its PDE system. To obtain the weak formulation, we multiply (2.2) with a space-time test function θ_h from same space of piecewise polynomials as q_h, and we integrate it over a space-time slice $K \times [T, T + \Delta T]$ made of one cell and a time interval:

$$\int_K \int_T^{T+\Delta T} \theta_h \frac{\partial q_h}{\partial t} \, \mathrm{d}x\mathrm{d}t + \int_K \int_T^{T+\Delta T} \theta_h \, \nabla \cdot F(q_h) \, \mathrm{d}x\mathrm{d}t = 0 \tag{2.3}$$

Prediction

For each cell K, we can implicitly solve the weak formulation (2.3) to produce a cell-local *space-time prediction* $q_h^*|_K$. This can be done by performing an integration by parts on the first term of (2.3), which results in a fix point problem [1]. To solve the resulting fix point problem, ExaHyPE relies by default on a robust method using Picard iterations as introduced by Dumbser et al. [33]. If the PDE system is linear, then ExaHyPE can instead

use a more efficient Cauchy-Kowalewsky procedure, as initialy proposed by Gassner et al. [34].

The prediction is made while ignoring neighboring cells, and therefore, yields jumps along the cell faces in q_h^* and $F(q_h^*)$.

Riemann solver

At each face-adjacent cell boundary, we solve the Riemann problem resulting from the previous step to obtain a numerical flux $G(q_h^*, F(q_h^*))$ for each of the two cells.

By default ExaHyPE employs a Rusanov flux. However, this can be changed by the user to take into account the application's specificities.

Correction

In the *correction* phase, we traverse the mesh again and in each cell we solve

$$\int_K \theta_h \left(q_h(T+\Delta T) - q_h(T)\right) \mathrm{d}x = \underbrace{-\int_K \int_T^{T+\Delta T} \nabla\theta_h \cdot F(q_h^*)\,\mathrm{d}x\mathrm{d}t}_{\text{Volume Integral}} + \underbrace{\int_{\partial K} \int_T^{T+\Delta T} \theta_h\, G(q_h^*, F(q_h^*))\,\mathrm{d}s\mathrm{d}t}_{\text{Surface Integral}} \tag{2.4}$$

The equation (2.4) is derived from (2.3) by partially integrating both terms [33].

The two terms on the right-hand side of (2.4) can be computed separately. Thus, the correction step is split into a *volume integral* and a *surface integral* substep, with only the latter requiring the numerical normal flux from the previous step.

Stable time step computation

Adjustments to the time step increment ΔT may be needed due to nonlinear effects or mesh adaptations. They would need to be performed before the first step at the start of the ADER-DG loop to determine a valid ΔT.

An upper bound on the time step increment is expressed by the Courant–Friedrichs–Lewy (CFL) condition [35]:

$$\Delta T \leq \frac{\mathrm{CFL}_N}{d\,(2N+1)} \frac{dx}{|\lambda_{\max}|}, \tag{2.5}$$

where dx is mesh size, $|\lambda_{\max}|$ the maximum signal velocity, and $\mathrm{CFL}_N < 1$ is a stability factor that depends on the polynomial order [19, 33].

2.2.2 ADER-DG algorithm

The steps of the ADER-DG scheme can be slightly reordered to increase data locality, resulting in the simplified Algorithm 1.

Algorithm 1 Simplified algorithm implementing a ADER-DG scheme with reordered steps to increase data locality

$T \leftarrow 0$
for cell $K \in \Gamma$ **do**
  $q_h(0)|_K \leftarrow$ `adjustSolution`$(K, 0)$ // Initialization
end for
while $T < T_{\text{final}}$ **do**
  $\Delta T \leftarrow$ `timestep`$(q_h(T))$ // Time increment
  for cell $K \in \Gamma$ **do**
    $(q_h^*|_K, F(q_h^*)|_K) \leftarrow$ `predictor`$(q_h(T)|_K, \Delta T)$ // Prediction
    $(q_h^*|_{\partial K}, F(q_h^*)|_{\partial K}) \leftarrow$ `extrapolate`$(q_h^*|_K, F(q_h^*)|_K)$ // Extrapolation
    $q_h(T + \Delta T)|_K \leftarrow q_h(T)|_K$
    $q_h(T + \Delta T)|_K$`+=volumeIntegral`$(F(q_h^*|_K, \Delta T))$ // Volume Integral
  end for
  for face-connected cells $K_1, K_2 \in \Gamma$ **do**
    $(G(q_h^*, F(q_h^*))_{K_1}, G(q_h^*, F(q_h^*)))_{K_2} \leftarrow$ `RiemannSolve`(
      $q_h^*|_{\partial K_1}|_{\partial K_2}, q_h^*|_{\partial K_2}|_{\partial K_1}, F(q_h^*)|_{\partial K_1}|_{\partial K_2}, F(q_h^*)|_{\partial K_2}|_{\partial K_1})$ // Riemann Solver
    $q_h(T + \Delta T)|_{K_1}$`+=faceIntegral`$(G(q_h^*, F(q_h^*))_{K_1})$ // Surface Integral
    $q_h(T + \Delta T)|_{K_2}$`+=faceIntegral`$(G(q_h^*, F(q_h^*))_{K_2})$ // Surface Integral
  end for
  $T \leftarrow T + \Delta T$
end while

We start by initializing the representation q_h of the state tensor Q in each cell. The goal of the simulation is to evolve the tensors $q_h(\cdot)|_K$ in each cell K.

Thus, as long as the simulation time is below the desired final time, the main loop proceeds in steps.

1. First we compute the optimal stable time step increment using the CFL conditions.
2. Then in each cell and without communication between cells:
   a) We compute local space-time prediction $q_h^*|_K$ and the associated predicted flux $F(q_h^*)|_K$, for example using Picard iterations.
   b) These predictions are projected to the cell boundary to prepare for the Riemann solver step. The projected state $(q_h^*|_{\partial K})$ and flux $(F(q_h^*)|_{\partial K})$ predictions are one dimension smaller and need to be stored for later.
   c) We perform the volume integral part of the correction step immediatly as it only requires the cell-local state and flux predictions as input, these predictions are not required afterward and can be discarded after this.

3. Following the prediction step, for each pair of face-connected cells, we solve the local Riemann problem using the prepared projections of the predictions. Here communication may be required to obtain both cells projected predictions.
4. The resulting numerical fluxes are then applied as the surface integral part of the correction step on their corresponding cell.

2.2.3 Expansion of the ADER-DG scheme

The simplified Algorithm 1 can be expanded in multiple ways. These expansions are not directly relevant to this thesis and thus are here only briefly outlined.

Adaptive mesh refinement

Adaptive Mesh Refinement (AMR) enables the use of cells of different sizes, which can be exploited by applications to only have a fine mesh on "interesting" parts of the domain, greatly reducing the number of cells required to obtain scientifically relevant results. AMR can be trivially added to the ADER-DG algorithm by including restriction and prolongation operators to perform a new step to refine or coarsen cells in accordance with application-specific criteria and to adapt the projected solutions around the Riemann solver step when cells interact with neighbors of a different resolutions [2].

A posteriori limiter

The ADER-DG scheme can suffer from numerical instabilities introduced by discontinuities in the DG solution. For this reason, Algorithm 1 can be extended with an *a posteriori limiter* as described by Dumbser et al. [1] and Zanotti et al. [2]. In short, at the end of the time step before updating the cell, we check if the new cell solution verifies both numerical criteria, using a relaxed discrete maximum principle, and physical ones specificied by the application, e.g. the water column height in a shallow water application has to be positive in each cell. If a cell does not verify all criteria, then it is marked as troubled. It and its surrounding are projected to a finer regular mesh in their state before the time step. Then the time step is recomputed using a more robust ADER-WENO finite volume solver *limiter* and the resulting solution for the troubled cell is projected back to the DG mesh. The implementation of the a posteriori limiter in ExaHyPE's ADER-DG scheme is detailled in [36].

As a beneficial side effect, the a posteriori limiter also increases the resilience of the ADER-DG scheme to soft errors by detecting and correcting faults which would otherwise lead to a fatal failure [37].

Fused steps

Without change to them, the steps of Algorithm 1 can be reordered to fuse some loops, therefore reducing the amount of mesh traversals and communication required. This

"fused" variant of the ADER-DG algorithm was implemented in ExaHyPE by Dominic Charrier and is described in [38, 36].

2.2.4 Kernels

Algorithm 1 relies on cell or cell-boundary local computations such as the `predictor`. In ExaHyPE, they are isolated into *kernels*. As each kernel only processes a single cell or face-boundary of the mesh, and this independently of the rest of it, they are implemented as single-threaded functions. The scheme's parallelization is then achieved by distributing sub-domains of the mesh and executing the kernels in parallel.

From Algorithm 1 we can identify the step-related main kernels:

- **SpaceTimePredictor (STP)**: This kernel generates the local prediction $q_h(T)|_K$ and also performs the extrapolation by projecting it and its associated flux to the cell boundary to obtain $q_h^*|_{\partial K}$ and $F(q_h^*)|_{\partial K}$.
- **VolumeIntegral**: It performs the volume integral substep of the correction step as described in (2.4) using the prediction from the STP kernel. For optimization purpose, it is often integrated to the STP kernel as its last substep.
- **RiemannSolver**: It solves the Riemann problem at each face-boundary to produce the numerical flux $G(q_h^*, F(q_h^*))_{K_i}$ required for the surface integral correction. By default it implements a Rusanov flux.
- **FaceIntegral**: It performs the surface integral substep of the correction step as described in (2.4) for one of a cell's faces, using the associated numerical flux from the RiemannSolver.
- **StableTimeStepSize**: This kernel computes the optimal time step increment $(\Delta T)|_K$ for a cell using the CFL condition. The time step increment used for the current time step is then the minimum over all the cells.

We also have the following supporting kernels:

- **SolutionAdjustment**: It initializes the cells $(q_h(0)|_K)$ and can be used to override the value of a cell during the simulation.
- **SolutionUpdate**: The prediction and correction steps only compute an update to the local solution. At the end of the time step, this kernel applies the update to the current solution $q_h(T)|_K$ to replace it with $q_h(T + \Delta T)|_K$.
- **BoundaryConditions**: Cells at the domain boundary are missing some face neighbors for the RiemannSolver kernel. This kernel applies the application's defined boundary condition to obtain the missing data.

To support the a posteriori limiter, some additional kernels are also available to detect troubled cells and perform a time step using a finite volume scheme, which itself is also decomposed into kernels. Furthermore, projection kernels are used to scale up or down a

cell or cell's face for the adaptive mesh refinement scheme and for the limiter's projection back and forth between the DG space and the FV mesh.

The kernels are the critical components of the ADER-DG scheme in multiple regards.

First, they are the components who call the application's PDE terms. Thus by using kernels, ExaHyPE isolates the overall ADER-DG scheme from the application's implementation of the PDE system, with the kernels at the interface between the two.

Second, in a simulation, almost all floating point operations are performed either directly inside the kernels or in the application's functions called by the kernels. The high-order approach of ADER-DG, compared to other schemes such as Finite Volume, results in numerous tensor contraction operations in the kernels, increasing their arithmetic intensity and ensuring they are compute-bound and not memory-bound on modern CPU architectures. Therefore their optimization is key to ExaHyPE's performance. In particular, the SpaceTimePredictor kernel is the prime target for optimizations as it consumes up to 90% of an application's runtime, as was reported in ExaHyPE's second project report [5].

Finally, the kernels implement the fine details of the ADER-DG scheme. By having them isolated as a black box, we can swap them to change the numerical scheme implemented. For example, as mentioned when outlining the prediction step, we use Picard iterations for nonlinear applications and a Cauchy-Kowalewsky procedure for linear ones. As the prediction step is performed by the SpaceTimePredictor kernel, linear and nonlinear implementations of this kernel are available. Likewise the RiemannSolver employing a Rusanov flux can be replaced by another one if necessary.

For these reasons, the kernels are the focus of this work, from their generation using a customizable *Code Generation* utility, to their optimization toward modern CPU-based architectures using vectorization, hybrid data layouts, and support for highly optimized BLAS libraries.

Part II

Design of ExaHyPE and its Kernel Generator

3 Architecture of ExaHyPE

This chapter presents the overall architecture of ExaHyPE. As this is an overview of the whole engine, multiple concepts introduced here are detailled in later parts of this thesis.

At the beginning of the project, a separation between two types of ExaHyPE's users was made to guide the design process of the engine. This separation was later further refined into three user roles, each with separate areas of expertise and responsibilities. From the description and analysis of these user roles, we defined four requirements for ExaHyPE's design as the project matured from its prototype stage. To fulfill these requirements, ExaHyPE is designed using a modular architecture with its components optimized and bound by generated code.

The user role description and analysis are performed in Section 3.1, ending with the introduction of the design requirements. Section 3.2 justifies the choice of code generation combined with a modular design to obtain the required customizability and separation of concerns. Then, looking at the resulting workflow to develop an application, ExaHyPE's components are identified. Finally Section 3.3 presents each component and outlines the design choices made for it and the paradigm it follows.

3.1 User roles

3.1.1 Domain experts and engine developers

After looking at the usual development process of a grand challenge application by a medium-sized interdisciplinary team, the ExaHyPE development team defined early on a clear separation between the domain experts on one side, and the experts in numerical schemes and code optimization on the other.

The *domain experts* are the typical users of ExaHyPE, with an expertise in fields not related to computer science. They desire to use ExaHyPE to easily simulate the mathematical model they developed to describe a given physical phenomenon of their field. For example such domain experts could be the astrophysicists that have formulated the GRMHD model shown in Section 2.1.2.

The latters are the ones that can adapt the engine, both to the needs of the application developed by the domain experts, and to further optimize it to fully exploit the potential of the hardware on which the simulation is performed. These *engine developers* are typically not experts in the applications field, but instead have a background in High

Performance Computing (HPC) related fields. The members of the ExaHyPE consortium developing the engine itself, such as myself, belong to this group. However, we also expect some members of an external interdisciplinary team to take this role, should the application developed by their team require a new numerical scheme or be run on a new hardware architecture.

This split between domain experts and engine developers is also seen in other PDE framework where the users can customize the engine itself, such as Firedrake [39, 40].

3.1.2 Formalization of user roles

While the initial split between domain experts and engine developers isolated the domain experts from more HPC oriented experts, we quickly noticed that the latter group needed to be further refined. As described in Section 2.2.4, the critical engine components are isolated in the kernels that need to be tailored toward a given application and optimized for the target architecture. Thus, the engine developers are expected to perform these two kind of optimizations of the kernels. However, they require two separate areas of expertise: On the one hand adapting the algorithm of the kernels to modify the overall numerical scheme, and on the other hand performing a low-level optimization of the implementation. Therefore, I created two new roles from the engine developer one: the *algorithm expert* and the *optimization expert*.

Some leftover responsibilities were left in the engine developer description, such as implementing the overarching ADER-DG algorithm or the parallelization scheme. However, these responsibilities are expected to be taken by members of the ExaHyPE consortium while developing the engine, and not require further user input once the project is finalized, as the ADER-DG overall scheme itself should not require further modifications. Consequently, no user role was defined for this. I also formalized in the same way the role of the domain expert as an *application expert*.

Each member of a team working with ExaHyPE can be described as an expert taking one or many of these roles, each focusing on a different area of expertise. The proper formalization and analysis of each role guided our design choice to improved ExaHyPE's user experience and facilitate, as much as possible, their work, as an engine that cannot be efficiently used would be mostly useless to scientific community.

Application experts

The application experts are the domain experts writing an ExaHyPE application. They have the mathematical and domain specific knowledge required to formulate a hyperbolic PDE system describing the physical world phenomenon to be simulated, using the form given by the canonical system (2.1). Furthermore, once the application's PDE system is defined, they also define the problem-specific initial and boundary conditions as well as criteria for mesh refinement, if desired, and admissibility of solutions, in the case of expected numerical instabilities. All these have to be implemented in the form of *user functions* in the application.

Application experts are not expected to be well-versed in computer science and the advanced skillset required to optimize a low-level programming language like C++. Therefore, they require a straightforward user API that allows them to simply describe their application's requirements toward the engine and have it automatically tune its kernels and prepare a skeleton code where they can implement the user functions.

Algorithm experts

Despite having multiple numerical schemes already built-in, an application may offer specific numerical optimization opportunities, or in a worse case scenario it can require, e.g. for stability reasons, a specific numerical scheme not already implemented in the engine. The algorithm experts are responsible for modifying existing kernels' algorithms or implementing new variants of them to respond to these opportunities or requirements. They have an expertise in numerical solvers and good progamming skills to implement them when optimization and parallelization concerns can be ignored.

Algorithm experts do not need to be domain experts in the field of the application, or have the low-level optimization skills required to fully exploit the target architecture. Thus they desire an abstract way to write the required algorithm without consideration for the application or architecture specific optimizations, and then have the engine add these automatically.

Optimization experts

Similarly to the numerical solvers, low-level architecture-targeted optimizations are already built-in for some hardware architectures, in an abstracted form to be used by the algorithm expert. The optimization experts implement new abstracted optimizations to improve the engine's performance on new target hardware architectures. They are experts in code optimization with a background in computer science and the expertise in the optimization techniques for the given target architecture.

Optimization experts are not expected to need to know every numerical scheme consideration behind the algorithms implemented by the kernels to be able to optimize them. As optimization techniques may change radically depending on the target architecture, the optimization experts desire tools to be able to provide multiple variants of an optimization and then have the engine automatically select, configure and apply the most appropriate one.

3.1.3 Derived engine's design requirements

From the descriptions of the three user roles, we see three separate fields of expertise and tasks to perform. ExaHyPE's design needs to separate each role from one another to allow each one to work independently of the others, as we cannot expect every user to be a master of all trades capable of undertaking all three roles. With this, as well as

taking ExaHyPE's nature as an engine into account, we defined requirements to guide ExaHyPE's design.

1. *Application and engine separation* – There is a clear separation between the application implementing the description of the phenomenon to be simulated and the engine itself implementing and optimizing the required solver to perform the simulation.

This first design requirement expresses the fact that ExaHyPE is first and foremost an engine to be used by domain experts, later formalized under the role of application experts. They need to be able to ignore the complexity of implementing an exascale-ready numerical solver, and therefore they require an abstraction layer between the user API they interact with and the engine complexity.

2. *Engine customizability* – Through its customizable kernels, the engine can automatically adapt itself, both to match its application's requirements and to fully exploit the capabilities of the hardware running the application.

A static "generic" engine would not be able to achieve the desired performance and adapts its user API to the application experts' requirements. As described earlier, the kernels are the critical components of the engine both performance-wise and to modify the engine's numerical scheme. Thus, this second design requirement follows from the first one by mandating that the kernels themselves can change between applications or targeted hardware, and that the engine automatically exploits this capability.

3. *Engine expandability* – The customizability of the kernels itself can be expanded by users.

As expressed by the formalization of the user roles of algorithm and optimization experts, should the engine lack some desired features or not be optimized toward a given architecture, then these users need to be able to add the missing features or optimization. The new features or optimization are then part of the engine and can be reused by another application.

4. *Algorithm and optimization separation* – The kernels' algorithms and low-level optimizations can be modified independently of one another.

Following the previous design goal, this one expresses that the algorithm and optimization experts are separated user roles with separate areas of expertise. Thus, streamlining their workflow requires their work areas to be separated from one another, similarly to how the separation between application and engine in the first design goal improved the workflow of the application experts.

The first two requirements were made by ExaHyPE's development team at the beginning of the project and guided our design of the overall architecture of the engine, that is discussed in the rest of this chapter. As the project matured, I introduced the last two design requirements during the formalization of the user roles of algorithm and optimization experts. These last two requirements focus on the customizability of the kernel themselves. Hence, they only affected the design of a specific component of ExaHyPE, the Kernel Generator, and their fulfillment will be the focus of the following chapter.

3.2 Overall architecture and design of ExaHyPE

In this section, we discuss the high-level design and architectural choices made in order for ExaHyPE to fulfill the first two requirements given by the analysis of the user roles.

3.2.1 Customizability with code generation

The second requirement focuses on the customizability of the engine through its kernels, as ExaHyPE needs to be able to efficiently support multiple kinds of applications, various numerical schemes and multiple target architectures, each with their specific optimizations. In other engines, frameworks, and libraries, two main approaches to automatically customize a code can be distinguished: templating and code generation.

Templating

In the templating approach, the compiler itself performs the desired customization at compile time. To do so, language specific features guiding the compiler behavior, e.g. templates and macros in C++, are heavily used by the projects following this approach. An example of this is the BLAS library Eigen [41], that relies on a templatized API to have the compiler produce and compile a code finely tuned to the desired use cases.

```
1 Eigen::Map<Eigen::Matrix<double,12,6>, Eigen::Aligned, Eigen::
    ↪ OuterStride<12> > A(A_array);
2 Eigen::Map<Eigen::Matrix<double,6,6>,  Eigen::Aligned, Eigen::
    ↪ OuterStride<8>  > B(B_array);
3 Eigen::Map<Eigen::Matrix<double,12,6>, Eigen::Aligned, Eigen::
    ↪ OuterStride<12> > C(C_array);
4 C.noalias() += A*B;
```

Listing 3.1: Matrix multiplication $C+ = A \cdot B$ using Eigen.

The code excerpt in Listing 3.1 illustrates how such customization is performed in the use case of a matrix multiplication $C+ = A \cdot B$: To perform the matrix multiplication involving C++ standard arrays with Eigen, three `Eigen::Map` objects are used to map the arrays to an Eigen's internal representation. Each `Eigen::Map` uses templates parameters, some being themselves templates. In this example the `Eigen::Matrix` parameters

specify the scalar type used, `double`, and the matrix size, `A` is a 12×6 matrix. Using C++ templates, the compiler produces the correct version of the template for this code and optimizes it accordingly, also with architecture-aware optimizations if a target architecture is specified at compilation. The same can be done on each matrix multiplication in the codebase so that the compiler produces all the required code to perform each of them efficiently.

Another library using this templating approach is the Peano framework [42, 43] used to parallelize ExaHyPE.

Code generation

Code generation relies on an external tool, the code generator, to produce a customized source code using multiple configuration parameters as input. For example, the LIBXSMM BLAS library [44] provides a code generator that takes as inputs a target architecture and a matrix multiplication's parameters, such as the size of the involved matrices and operation coefficients, to produce a highly tuned function performing the very specific matrix multiplication operation requested.

```
1  void gemm(const double* A, const double* B, double* C) {
2    __asm__ __volatile__(
3      \\[...] assembly code
4      "addq $96, %%rdi\n\t"
5      "vfmadd231pd %%ymm5, %%ymm0, %%ymm9\n\t"
6      "vfmadd231pd %%ymm5, %%ymm1, %%ymm12\n\t"
7      "vfmadd231pd %%ymm5, %%ymm2, %%ymm15\n\t"
8      \\[...] assembly code
9     );
10 }
```

Listing 3.2: Matrix multiplication $C+ = A \cdot B$ using LIBXSMM.

The code excerpt in Listing 3.2 performs the same matrix multiplication as the previous example with Eigen. To do so the specification of the desired matrix multiplication is given to LIBXSMM that then produces a source file. This function is directly written in assembly code and performs $C+ = A \cdot B$ in an efficient architecture-aware way, as long as its inputs satisfy the properties used when generating it. If other kinds of matrix multiplication are required, multiple functions can be generated and then used for each desired specification.

Another example is the SableCC framework [45], that can produce a object-oriented Java code for parsing a language defined by a user-made grammar.

Comparison

Both approaches are able to produce a highly tuned software with the level of customizability desired by ExaHyPE, as demonstrated by the frameworks and libraries using one or the other. However, when analyzing the advantages and drawbacks of each one, code generation was chosen for ExaHyPE overall design, with templating being used to a lesser degree in some specific areas, as will be described in Section 3.3.2.

Indeed, the templating approach suffers from many relevant drawbacks in regard to our design requirements. First, the language specific features used are often advanced ones that application experts without advanced programming skills cannot be expected to master, as they would be the ones responsible for properly setting up templated calls similar to the ones seen in Listing 3.1. Second, as the actual "source code" is generated on the fly by the compiler at compile time, it cannot be easily read by users for improvement, benchmarking, or debugging purposes. Furthermore, the templatized source code itself is often hard to parse when C++ templates and macros are heavily used. Finally, while minor compared to the other drawbacks, as the work performed by the compiler is greatly increased, this approach can lead to very long compilation times, especially on bigger projects such as ExaHyPE.

On the other hand when using code generation, glue code can be generated to configure and bind various engine components together, thus hiding this process from the users, in particular application experts, and allowing a clear separation between the various components of the engine without needing a common API for variants as the glue code can be adapted at the same time to match the chosen variant's specific API. In particular, the application can be separated from the engine itself and automatically bound to it with glue code. Glue code can also be used to bind third party generated code or libraries to the engine. Furthermore with code generation, since the tuned source code is generated in proper files, it can easily be read, analyzed, and modified before compilation to benchmark it or try out variations of it. Finally, it also can be used to generate a skeleton code to be worked on. This is again particularly helpful to the application experts, which are then provided a clear work area to focus on, with generated descriptions and placeholder functions to work on.

The most significant cost of using code generation is the need for a separate code generator utility. The code generator itself becomes a project in the project, leading to new design challenges, requirements regarding third party libraries, and more concepts and potentially programming languages to work with. However, as will be discussed in detail in Chapter 4, I turned this drawback around as ExaHyPE can now be split between a static core engine requiring no further adaptations, and the code that needs to be customized being delegated to the code generator, the two parts being later bound together by glue code. Thus, the work area of the algorithm and optimization experts is moved from the engine to the code generation utility, that itself can be designed to fulfill the last two design requirements.

While other PDE engine, such as DUNE [46, 47] or the YATeTo [48] toolbox from SeisSol, also use code generation to produce the optimized kernels of their numerical schemes, their code generation utilities themselves are not expected to be expanded by their users.

3.2.2 Development of an ExaHyPE application

ExaHyPE being a C++ engine, at a high level the development process of an ExaHyPE application can at first be divided into the usual three steps of developing a C++ application: writing the source code, compiling it, and running the resulting executable. With code generation involved, two extra steps are added: setting up the code generation and generating code. In ExaHyPE these two extra steps are performed before writing any source code, which results in a five steps development process.

(1) Setting up the code generation

The requirements and properties of the desired application are described in a *specification file* by application experts.

(2) Using the code generation

Using the specification file as input, the following code is produced:

- **Kernels**: The components of the engine implementing the numerical scheme's steps in highly optimized C++ functions, as described in Section 2.2.4.
- **Skeleton code**: Used to write the application code in the next step.
- **Makefile**: Already configured, to simplify the compilation step.
- **Glue code**: To bind the desired schemes from ExaHyPE core with the generated kernels and the application code.

The produced kernels are the result of the work of both algorithm experts, for the numerical scheme employed, and optimization experts, for the low-level architecture specific tuning.

(3) Writing the application

Application experts complete the application's skeleton code. Only the required user functions are present in the skeleton code, for example an application using only the flux term of the canonical PDE (2.1) does not ask for the implementation of a source function.

(4) Compilation

The code is compiled into an executable using a Makefile generated during the second step. This local Makefile configures and includes a global engine Makefile. As compiler specific

optimizations dependent on the target architecture are set up here, the optimization expert can either modify the global engine Makefile or the generation of its configuration in the local Makefile to further optimize the produced executable.

(5) Running the application

The executable produced in the previous step is run with the specification file as input to provide runtime parameters. The analysis of the simulation plots and results requires the domain expertise of the application experts.

3.2.3 Modular architecture

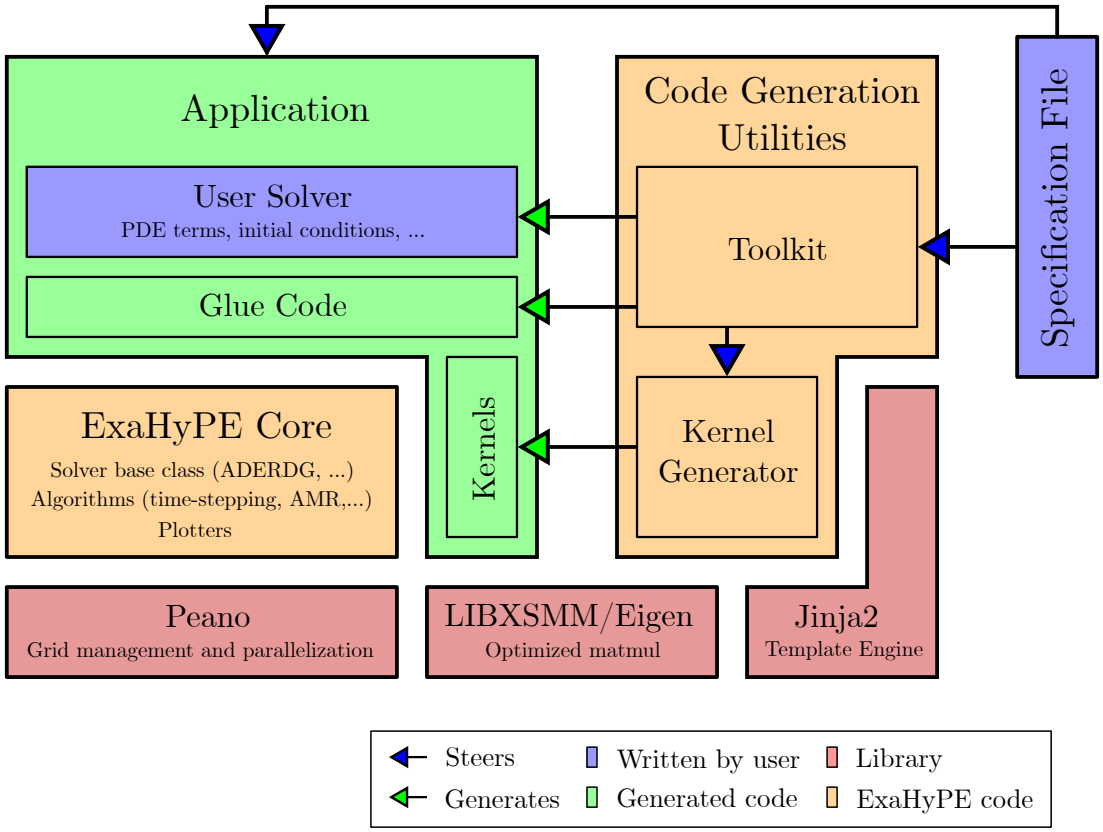


Figure 3.1: Overall architecture of ExaHyPE. The user, here an application expert, only writes the specification file and the user solver (in blue). The code generation utilities uses the specification file to generate the optimized kernels, as well as the glue code binding everything together.

From the previoulsy defined development process of an ExaHyPE application, we can identify distinct components of the overall engine's architecture with an application, as

illustrated in Figure 3.1. Each component is here only briefly introduced to provide an overview of its role in the architecture and its relation to the application development process. They will be discussed individually in more detail in the next section.

The first one is the *specification file* written in the first step by the application expert. It is where an application is described, both for the code generation utilities and for the final executable. Therefore, it contains static settings, e.g. the desired solver scheme and the list of terms from the canonical PDE (2.1) required for this application, but also modifiable settings for the execution of the simulation itself, e.g. the domain size and end time of the simulation.

The second component is the code generator used in the second development step, separated into two utilities. The first one is the *Toolkit* that is responsible for generating the glue code, the application's skeleton code, and the local Makefile. The second one is the *Kernel Generator*, that generates the appropriate optimized kernels for the described application and the desired target architecture. The Toolkit acts as the frontend of the code generation and is given the specification file as input. It calls automatically the Kernel Generator, however the latter can also be used independently through its own API or command line interface. Being one of the main contributions of this thesis, the design and development of the Kernel Generator will be detailed in Chapter 4.

The third component is the application. It is made of a directory containing generated files, in particular a *user solver* class generated as a skeleton code to be completed by the application experts in the third development step. For more complex applications, additional source code and third party libraries may be added or linked here by the users. It also contains the kernels, which are the functions implementing the steps of the overarching numerical scheme. The kernels, the engine, and the user solver are all bound together by generated glue code in the application.

The last proper ExaHyPE component is the *ExaHyPE core* itself. It contains the high-level descriptions of the overarching ADER-DG numerical scheme with its extensions, such as the a posteriori limiter described in Section 2.2.2. It also deals with the shared-memory and distributed-memory parallelization through Peano, as well as the plotting and logging.

Finally the whole engine relies on multiple third party open-source libraries, in red in Figure 3.1. Notable ones are:

- **Peano** [42, 43]: A framework for dynamically adaptive Cartesian meshes, provides the parallelization logic used by the core.
- **LIBXSMM** [44]: A BLAS library capable of generating matrix multiplication functions highly optimized toward intel architectures.
- **Jinja2** [49]: A web template engine in Python3 used by the Toolkit and Kernel Generator to render the abstraction of the code to be generated.

This architecture is highly modular, enforcing a strong separation of concerns between the components and enabling the engine adaptibility. Code generation is critical to this

architecture, not only for customizing the kernels, but also for providing the required glue code to bind everything together. ExaHyPE's architecture fulfills its first two design requirements identified in Section 3.1.3:

1. The application experts work only on the specification file and the user solver, where they can ignore the engine complexity.
2. The Kernel Generator produces optimized kernels and is worked on by the two other user roles.

3.2.4 Installing ExaHyPE

ExaHyPE's architecture relies on several third party libraries. In accordance with our overall goal of improving the user experience, we made the installation process of ExaHyPE and its dependencies as simple as possible.

ExaHyPE is an open source project. It uses Git for version control and its repository is hosted by the GitLab web service from the Leibniz Supercomputing Centre at `https://gitlab.lrz.de/exahype/ExaHyPE-Engine`. Git was chosen due to it being the current de facto industry standard distributed version control system.

All third party libraries used by ExaHyPE are open source and available as Git repositories. To simplify their integration, they are included in the ExaHyPE project as Git submodules. A shell script is provided in ExaHyPE to download and prepare the dependencies, as some require additional setup, e.g. LIBXSMM C++ source code has to be compiled to obtain the `gemm_generator` used by the Kernel Generator.

Installing ExaHyPE has been simplified to two easy steps done as command line:

1. Cloning the ExaHyPE repository using Git:

```
git clone https://gitlab.lrz.de/exahype/ExaHyPE-Engine.git
```

2. Using the submodule script to download and set up its dependencies:

```
./ExaHyPE-Engine/Submodules/updateSubmodules.sh
```

Once installed, users can immediatly start using ExaHyPE to develop their application or try out provided examples.

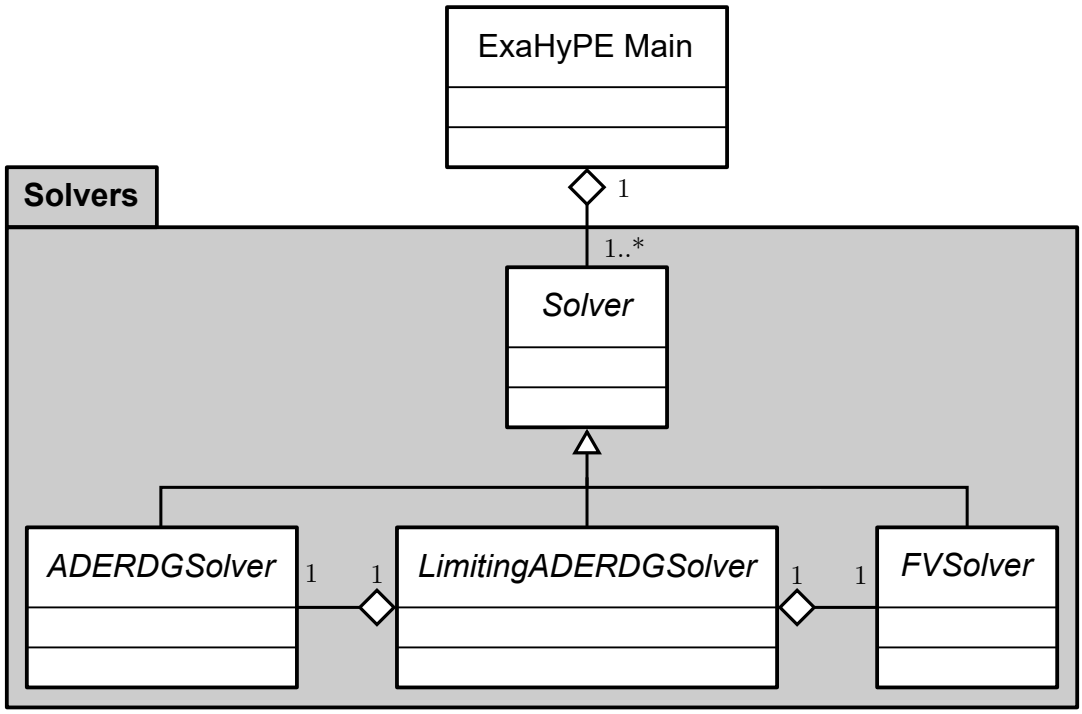
3.3 Programming languages and paradigms in ExaHyPE

The highly modular nature of its architecture allows ExaHyPE to use multiple programming languages and paradigms, each time chosing the ones best suited to the component considered.

3.3.1 Solvers: object-oriented programming

ExaHyPE targets CPU based architecture. Hence, we chose C++ as base programming language for the project. As the ExaHyPE core is not a performance critical component, the standard C++11 is used without compiler specific or outside standard optimization. It follows the *Object Oriented Programming* (OOP) paradigm to run the application in the engine using polymorphism, seeing it as an instance of a base abstract `Solver` class.

We here focus on this class and its inheritances, from the core to its realization in the application. A detailled description of the architecture and implementation of the ExaHyPE core and its other constituents can be found in [36].



UML Diagram 3.2: UML diagram of the solver class hierarchy in ExaHyPE core.

UML diagram 3.2 models how the ExaHyPE core uses polymorphism with an abstract base class `Solver` to handle various numerical schemes, implemented as child classes. Three such classes exist in ExaHyPE:

- `ADERDGSolver` implementing the ADER-DG scheme described in Section 2.2.
- `FiniteVolumesSolver` implementing an ADER-WENO Finite Volumes scheme as a robust backup solver for the a posteriori limiter [1].

- `LimitingADERDGSolver` combining the previous two solvers with the aposteriori limiting scheme logic to obtain a more robust ADER-DG scheme [2, 36], as described in Section 2.2.3.

It is at this level, still in the ExaHyPE core, that the parallelization required logic is performed with delegation to Peano, as well as the global algorithm of the chosen numerical scheme. These classes are lacking two critical components, abstracted by pure virtual functions needing to be implemented. First the implementation of the specific steps of their scheme, that are isolated in the kernels as described in Section 2.2.4 so that they can be highly customized and optimized by the Kernel Generator. Then the actual implementation of the PDE system and scenario-specific functions, e.g. the initial conditions of the simulation that need to be implemented in the application. An application can define multiple solvers, in particular when using the a posteriori limiter, however, from here on only single solver ADER-DG applications are considered.

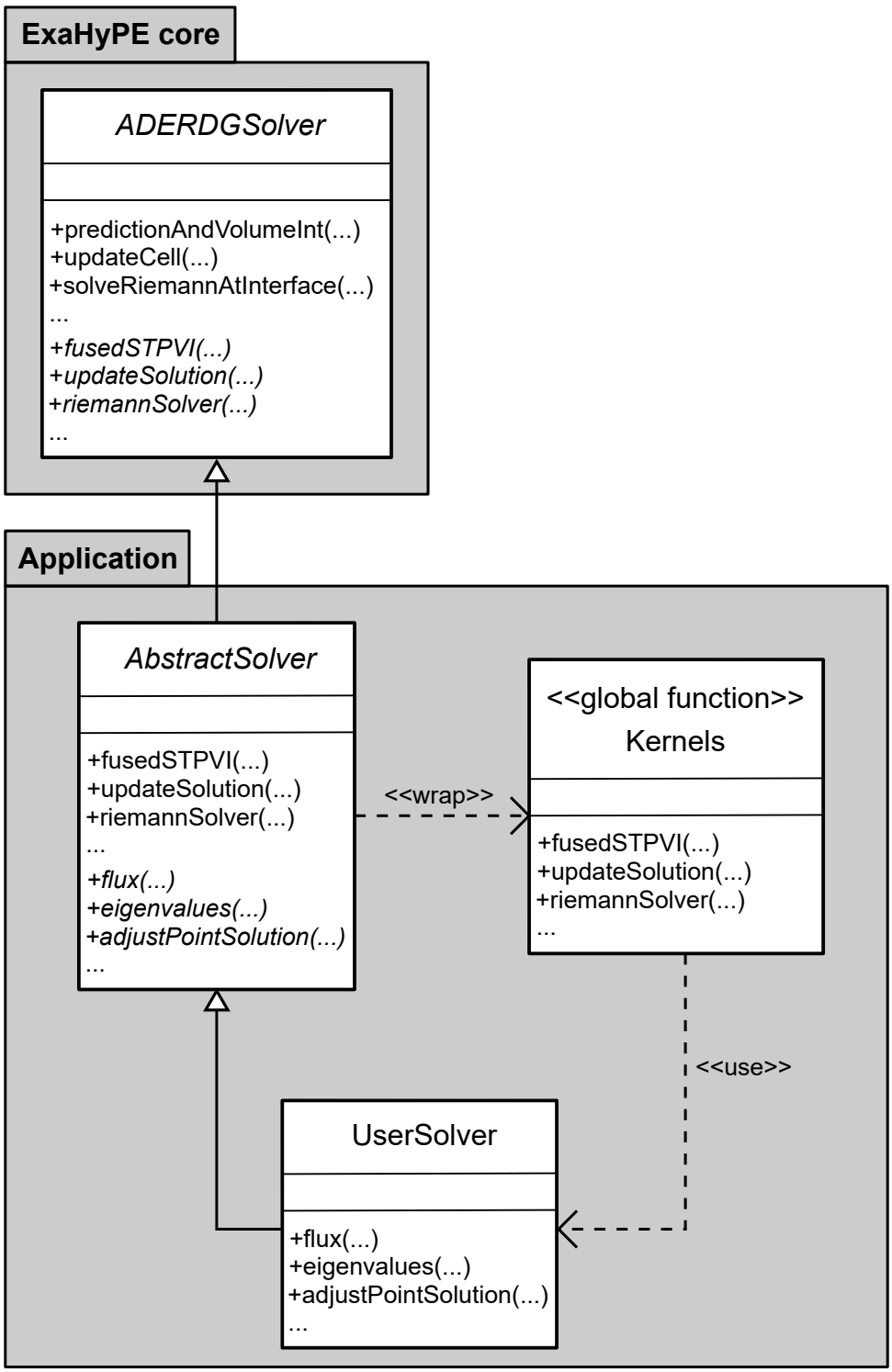
UML diagram 3.3 shows how the application is bound to the engine core using a further level of inheritance with the `AbstractSolver`. This abstract class inherits from the application chosen solver class, here an `ADERDGSolver`, and is generated as glue code by the Toolkit. It is there that the kernels, implemented as free functions, are bound to the solver scheme by wrapping them in overloaded member functions. In accordance with the application's specification file, it also defines the user functions to be implemented by the application's realization as virtual functions.

Finally the `AbstractSolver` is inherited by the `UserSolver`, which is the concrete class defining the application. A skeleton code of the class implementation and its header are generated by the Toolkit for the application experts to complete with the proper *user functions* specific to the PDE system solved and the scenario simulated. As some application experts are more familiar with Fortran than C++, it is possible to implement the member functions of the `UserSolver` as wrapper around a Fortran code[1]. ExaHyPE's compilation process is able to seamlessly combine the two languages with its supported compilers.

Design evaluation

This design choice enforces a clear separation of concerns between the inheritance layers, while ensuring that from the point of view of ExaHyPE's main loop an application is the concrete realisation of a solver. The choice of OOP also facilitates the introduction of glue code as a simple inheritance layer between the ExaHyPE core and the application. Finally OOP is a well-known paradigm, here used in its simplest form, which helps guide the work of application experts when implementing the application related functions.

[1] However, if one uses Fortran, it is important to note that despite it not being visible at this level, the function will be called using shared-memory parallelism, and thus, in accordance with the Fortran standard, need to be properly marked as such to avoid memory corruption at runtime due to the compiler using static memory addresses. Declaring the functions as RECURSIVE is enough to prevent this.



UML Diagram 3.3: UML diagram of the binding between ExaHyPE core, kernels and application using OOP, with the `AbstractSolver` class acting as glue code.

On the other hand, ExaHyPE core's abstract solver classes are very complex classes, with over 3000 lines of code, performing multiple tasks. Furthermore, they are deeply connected to the Peano framework to handle the parallelization of the application. This is akin to the "God Class", also known as "The Blob", anti-pattern [50]. Hence, while being finished from a purely functional perpective, these classes should be re-engineered and split into multiple subclasses working together, each performing one task. However, due to limited development time and the lesser relevance of this part of the codebase with regard to performance, this was not prioritized as of today. It is important to note here, that no user is expected to directly work on these classes, so this concern is not relevant from a purely user perspective.

3.3.2 Kernels: highly-tuned free functions

The kernels are the performance-critical components of an application, isolating each of the steps of the overarching numerical scheme. Two types of kernels exist, the generic kernels and the optimized ones.

Both are C++ free functions that are automatically bound to the engine by the Toolkit generated `AbstractSolver` class wrapping them. They call the `UserSolver`'s user functions, e.g. the flux function, to bind them to the numerical scheme. Kernels are declared together in a single header file properly setting up their namespace.

The optimized kernels are produced by the Kernel Generator, steered using the provided specification file. The generated code is located in a subdirectory of the application. Multiple levels of optimization are possible through options available in the specification file, each level increasing the optimization requirements on the application side with a more advanced user API presented to the application experts.

On the other hand, the generic kernels are C++ free functions that are already implemented with the engine, and thus cannot be fully customized. They are implemented in a C++ standard compliant way and use the simplest data layout to be able to work on any architecture with any compiler at the cost of lacking performance. When developing an application, the generic kernels are usually used to test the validity of the implemented `UserSolver` before moving on to the more performant optimized kernels.

Despite the restrictions put on their implementation, the generic kernels still benefit from some customization through the templating method described in Section 3.2.1. Listing 3.3 shows a simplified code excerpt illustrating the kind of templating performed on the generic kernels using C++ templates. A template parameter `SolverType` is used to give these functions access to the user solver class. Compile-time constants of the application, such as the number of variables used declared with the C++11 `constexpr` specifier, can be accessed through it. Furthermore, using simple template metaprogramming, branching can be omitted during compilation. For example, in Listing 3.3 a boolean template parameter `useFlux` is given and used for branching inside the function. As this parameter is resolved to a known constant boolean value at compile time, the compiler can recognize that the branch is either always dead, i.e. `if(false)`, and remove it, or always executed,

```
1 template <bool useFlux, typename SolverType>
2 void kernel(SolverType& solver, [...]) {
3   constexpr int numberOfVariables = SolverType::numberOfVariables;
4   //[...]
5   if(useFlux){
6     //performs flux operations [...]
7   }
8   //[...]
9 }
```

Listing 3.3: Simplified code illustrating the use of template metaprogramming to give the generic kernels some customization capabilities.

and remove the branching and associated unnecessary test. Thus, with template metagramming, the generics kernels can be customized to solve only the actual PDE system required by disabling the parts corresponding to unused terms of the full canonical PDE system (2.1).

These templatized kernel functions are bound to the solver in the `AbstractSolver` as described earlier. When doing so, the template parameters are hard-coded in the glue code wrapping the kernel, which for this example could be:

```
1 kernel<true, MySolver>(*static_cast<MySolver*>(this), [...]);
```

3.3.3 Specification file: JSON

As described in Section 3.2.2, the first step to develop a new ExaHyPE application is to describe it in a specification file. This file contains the informations needed by the Toolkit and Kernel Generator to produce the required files and set up an adapted skeleton `UserSolver` class. Furthermore, it is used at runtime to obtain some parameters such as the length of the simulation or the domain's dimensions. The specification file being used to store data of multiple type, by default, is written in JavaScript Object Notation (JSON). This standardized format[2] is easily readable and widely-used, making it likely that users are already familiar with it. It also provides the required flexibility to store multiple kinds of data.

Listing 3.4 shows a valid specification file describing a linear elastic wave application. In it we can see that string, integers and floating point numbers are supported, as well as arrays and objects storing key-value pairs. The specification file itself is a JSON object constituted of multiple blocks, the most important ones being the `solver` block describing a list of solvers and the `computational domain` block specifying the domain relevant runtime parameters. In Listing 3.4 we can see that here only one solver is used, it is an ADER-DG solver working at polynomial order 5 on a PDE system of 9 variables

[2] ISO/IEC 21778:2017 [51], complemented by RFC 8259 [52].

```
{
  "project_name": "Elastic",
  "paths": {
    "peano_kernel_path": "./Peano",
    "exahype_path"     : "./ExaHyPE",
    "output_directory" : "./999_todelete"
  },
  "architecture": "hsw",
  "computational_domain": {
    "dimension": 3,
    "end_time": 2.3,
    "offset": [   0.0,  0.0,  0.0 ],
    "width":  [  30.0, 30.0, 30.0 ]
  },
  "solvers": [{
    "type": "ADER-DG",
    "name": "ElasticWaveSolver",
    "order": 5,
    "variables": 9,
    "maximum_mesh_size": 3.0,
    "maximum_mesh_depth": 1,
    "time_stepping": "globalfixed",
    "aderdg_kernel": {
    "nonlinear": false,
    "terms": ["flux", "ncp"],
    "space_time_predictor": {"split_ck":true},
    "implementation": "optimised"
    },
    "plotters": [{
      "type": "probe::ascii",
      "name": "ProbeWriter1",
      "time": 0.0,
      "repeat": 0.005,
      "output": "./seismogram_1km",
      "variables": 9,
      "select": {"x": 16.0, "y": 15.0, "z": 15.0}
    }]
  }]
}
```

Listing 3.4: Minimal example of a specification file written in JSON for an elastic wave application with one plotter.

using optimized kernels. Its PDE system uses the flux and non-conservative product terms of the canonical PDE system (2.1) and a `split_ck` advanced optimized scheme is chosen for the SpaceTimePredictor kernel. Other relevant specification file blocks not present in Listing 3.4 are the shared and distributed memory configurations.

Using JSON to format the specification file helps with parsing it, as robust JSON parsers exists for any major programming language, either natively like in Python or through

open source library such as "JSON for Modern C++" [53] used in ExaHyPE. While JSON is the recommended format, ExaHyPE also supports its own legacy custom language used in an earlier version that is automatically translated to JSON. This is only recommended for backward compatibility as newer features are not supported in the legacy language. Moreover, ExaHyPE can be expanded to support other formats, such as XML or YAML, by translating them to JSON in the Toolkit.

3.3.4 Toolkit and Kernel Generator: MVC Python3 utilities

The Toolkit and Kernel Generator are used during the code generation step. Thus, they are not subject to performance requirements beyond not exceeding more than a couple of seconds to do their work. Furthermore, as projects in the project, their designs and architectures can be optimized independently of the rest of ExaHyPE to better fulfill their goals. Their design and the code generation process itself are described in detail in Chapter 4 and only summarized here.

Both are written in Python3 and follow a Model-View-Controller architectural pattern. Python3 was chosen for its ease of use as well as its widespread usage in the HPC community, making it likely that the users assuming the roles of algorithm or optimization experts are already familiar with it and that Python3 is already installed on the target HPC cluster. Furthermore, to generate code, they rely on the Jinja2 web template engine [49]. The code to be generated is abstracted by Jinja2 templates in the form of C++ blended with templating language tokens. This allows the code generation to highly customize the generated code while keeping its abstraction close to the expected result, and helps streamline the expansion of the code generation process, fulfilling ExaHyPE's third design requirement from Section 3.1.3. Furthermore, this also enabled us to satisfy ExaHyPE's fourth design requirement by separating the algorithms from the optimizations.

Only a basic Python3 installation is required as all dependencies are provided as git submodules of ExaHyPE. While these could be directly installed from package in Python3 using `pip3`, we experienced that it is not always allowed to do so on clusters and having to setup a local installation of Python3 can be cumbersome. Providing them as submodules simplifies greatly the installation of ExaHyPE as described in Section 3.2.4.

A bash script `toolkit.sh` encapsulating the call to Toolkit is provided with it. It automatically checks that the correct version of Python is available as well as the required submodules. To call the Toolkit, a user simply needs to run the script with the specification file as parameter:

```
./Toolkit/toolkit.sh mySpecFile.exahype
```

3.3.5 Compilation: Makefile

To compile the application, ExaHyPE comes with a main Makefile. This Makefile is configured by environment variables and through them automatically detects the files to

be compiled. Since an application can also use Fortran code, the Makefile also calls a Fortran compiler if it detects any Fortran source file, the produced object files being linked with the ones from the C++ source code at the end of the compilation process.

To automatically configure the Makefile, the Toolkit generates a local Makefile in the application's directory. This local Makefile sets up the environment variables using the specification file's provided parameters and then calls the main Makefile. For example, the `architecture` parameter found in Listing 3.4 specifies the target architecture and is hard-coded in the local Makefile, so that the main one can use architecture specific compiler options, e.g. to specify the desired SIMD instruction set to use. Likewise the relative paths to the application and the various engine components required are also hard-coded in the local Makefile. Finally it also sets up some preprocessor variables used in C++ macros in the core's code.

While being ready to use without further configuration by simply using the command `make` in the application directory, the user can export environment variables to change the default compilation behavior. Commonly used options include the compilation in debug mode instead of release mode with `export MODE=DEBUG` or specifying another compiler to use. Additional preprocessor variables also can be set using the corresponding environment variables as both Makefile only append to them.

The choice of using a Makefile was made at the beginning of the project, as one from a previous project could be repurposed. As a drawback of this choice, the main Makefile has becomed increasingly complex as new compilation options were added. However, like the solver classes from the core, no user is expected to work directly on it, with some very rare exceptions from optimization experts, hence negating most of this drawback. On the other hand, using a main Makefile configured by a local Makefile provides a lot of customization opportunity and facilitates the integration of the code generation logic to the compilation process through the local Makefile. It also streamlines the compilation process to the extreme from the user's point of view, as compilation often requires no further thought and only a single `make` command.

3.3.6 Parallelization: Peano framework

The Cartesian meshes used by ExaHyPE's ADER-DG scheme are provided by the third party Peano framework (version 3) [42, 43]. Peano generates the meshes using a cache-efficient tree structure capable of adaptive mesh refinements. Figure 3.4 shows an example of the mesh traversal performed by Peano on a 2D adaptive mesh, following the Peano space-filling curve [54].

Peano is deeply integrated into the ExaHyPE core. When performing the mesh traversal, it fully takes care of the shared-memory and distributed-memory parallelization for the engine. Domain decomposition for distributed-memory parallel simulations is realised by forking off or merging subtrees of the mesh. Message passing in Peano is enabled by MPI. The shared-memory parallelization is performed by parallel-for loops with Intel

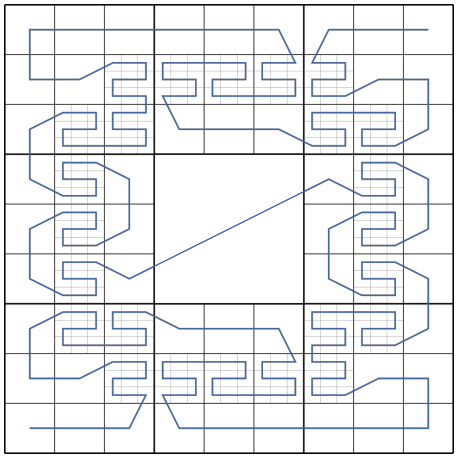


Figure 3.4: Peano space-filling curve running through a 2D adaptive mesh. Figure adapted from [3].

Threading Building Blocks (TBB) in regular substructures in the domain's tree [55]. Large applications are expected to use both MPI and TBB together [56].

4 Design of the Kernel Generator and Toolkit

In this chapter, the focus is on the architecture of the Kernel Generator and Toolkit and how I re-engineered the initial utilities to their current form.

In Section 4.1, using the role analysis from the previous chapter and the overall design requirements introduced in Section 3.1.3, in particular the last two, I introduced three design goals for the Kernel Generator. By making a parallel between these requirements and web application development, I chose to redesign the first version of the Kernel Generator using the *Model-View-Controller* architectural pattern supported by the Jinja2 *Template Engine* [49]. The re-engineering of the Kernel Generator is presented in Section 4.2. After this re-engineering proved to us the benefits of this architecture, together with two colleagues, we fully rewrote the Toolkit to follow the same design, as is presented in Section 4.3. Finally in Section 4.4, I used advanced features of Jinja2 to split the code abstraction in *algorithmic templates* and *optimization macros*, thus successfully separating the work area of the application and optimization experts, as required by the last engine's design requirement to enable an efficient team work while minimizing the required communication and overlapping knowledges.

4.1 MVC frameworks for code generation

4.1.1 Motivation for a new design

In the early stages of the ExaHyPE project, the Kernel Generator was a collection of Python3 scripts, each generating a different kernel directly using Python's `write` to produce the resulting file. A single frontend script was used to call the Kernel Generator through the command line interface, it then called the respective generation scripts for each file to be generated. This version of the Kernel Generator fulfilled its primary goal:

> 1. *Customizability* – The Kernel Generator produces customized kernels that are tailored toward its application and optimized toward a given hardware architecture.

Nevertheless, this was not done optimally. To illustrate the issues, the code excerpt in Listing 4.1 shows how the initialization of the trivial initial guess from the nonlinear SpaceTimePredictor kernel was produced in its script. We here see that the readability of this code is poor, as C++ is mixed with Python logic. Furthermore, the excerpt itself

```
1 l_sourceFile.write('  for(int ijk=0;ijk<'+str(l_nSpaceDof)+';ijk++)
    ↪ {\n')
2 # replicate luh(:,i,j,k) nDOFt times
3 for i in range(0, self.m_config['nDof']):
4     l_sourceFile.write('    std::memcpy(&lqh[ijk*'+str(l_blockWidth)+
      ↪ '+'+str(i*l_colWidth)+'], '\
5         '&luh[ijk*'+str(self.m_config['nVar'])+'], '+\
6         str(self.m_config['nVar'])+'*sizeof(double));\n')
7 # close for loop
8 l_sourceFile.write('  }\n')
```

Listing 4.1: Code excerpt from the first Kernel Generator using raw Python to generate code.

needs to be found in a large Python script over thousand lines of code long. Hence, modifying the generated code was challenging, especially if the overall algorithm needed to be modified, as was the case to add new PDE terms to the engine's early simplier version of its canonical PDE system.

Unlike other parts of the engine, the Kernel Generator will be modified in the future as new numerical schemes and optimization will need to be introduced to fulfill the requirements of future applications and hardware architectures. Thus, I introduced a new design goal for the Kernel Generator, implementing ExaHyPE's third design requirement from Section 3.1.3:

> 2. *Expandability* – The Kernel Generator's architecture facilitates its expansion and the addition of new features to it, by having the code generation process as close as possible to the produced C++ code.

An additional goal appeared when I properly defined the algorithm and optimization expert roles from the previous engine developer role. Not only do they require an easy to customize Kernel Generator, but they also focus on different tasks, with the formers creating and improving the kernels' algorithms implementing the numerical scheme, while the latters provide the architecture-specific optimizations necessary to achieve high performance on modern hardware. This resulted in the engine's fourth design requirement, which is translated as the following third design goal for the Kernel Generator:

> 3. *Separation of concerns* – The code generation process isolates the implementation of the algorithm from the low-level optimization and automatically integrates the latter into the former when generating the code.

	ExaHyPE's Kernel Generator	Web Application
Purpose	Generate customized C++ kernels	Generate customized HTML code
Input	An application expert's spec. file	A client's HTTP request
Dev. team	Algorithm and optimization experts	Front-end and back-end developers

Table 4.1: Comparison between ExaHyPE's code generation and a web application.

4.1.2 Parallels to web application development

Ignoring the context in which the Kernel Generator is used, the three identified goals are close to those of any code generation framework. In particular, as illustrated in Table 4.1, the Kernel Generator purpose, context and development process can be put in parallel with another area of software engineering that has very little overlap with HPC and numerical solvers: the development of a web application.

Here the application's purpose is to generate a customized webpage for the "client", e.g. a human viewing the webpage through a web browser processing it. A modern webpage itself is usually made of HTML, CSS and Javascript source code, and follows a predefined layout that is customized with the result of the client's query. For example, a webmail application displays some amount of emails always using the same given layout, but this layout is then enriched to display the specific emails of the current user.

The development of a web application combines multiple areas of expertise, usually divided between front-end and back-end. The front-end developers deal with the user interface that is seen by the client, they define the layout of the response. The back-end developers create the business logic that produces the data requested by the query and that will be then displayed in the response's layout. Their contributions are integrated into one output by the *web application framework* used. Both front-end and back-end developers use different programming languages and apply their domain specific knowledges. Thus, they are separate positions, filled by people possibly without the full expertise required for the other one.

From this broad stroke description, we can see that ExaHyPE's Kernel Generator's goal and requirements mirror closely the ones found in web application development. The division of labor between front-end and back-end developers also mirrors our algorithm and optimization experts user roles. A lot of research and cumulative experience in web application development have resulted in efficient solutions and processes to achieve their goals. I gained valuable insights by looking at the industry standard processes and the existing framework used to develop web applications.

4.1.3 MVC frameworks

A very common architectual pattern used to develop a web application is the *Model-View-Controller* (MVC) pattern. The MVC pattern was initially created by Trygve Reenskaug in 1979 to design user interfaces [57]. MVC was referenced by the Gang of Four in 1994 as a good usage scenario for multiple of their design patterns [58]. Since then, MVC itself

is recognized as a major architectural design pattern on its own. For over a decade, most leading web application frameworks, covering a large array of programming languages, have used this pattern either fully, partially or in a slightly modified form:

- AngularJS (JavaScript) [59].
- Django (Python3) in the slightly modified Model-View-Template form [60, 61].
- ASP.NET (.NET) includes it in its component ASP.NET MVC [62].
- Laravel, CodeIgniter, Symfony and CakePHP (PHP) [63].
- Spring (Java) has a MVC module [64, 65].
- Ruby On Rails (Ruby) [66].

Some of these are open source community projects, such as Laravel or Django, while others are supported by major companies in the information technology sector, such as ASP.NET by Microsoft or AngularJS by Google.

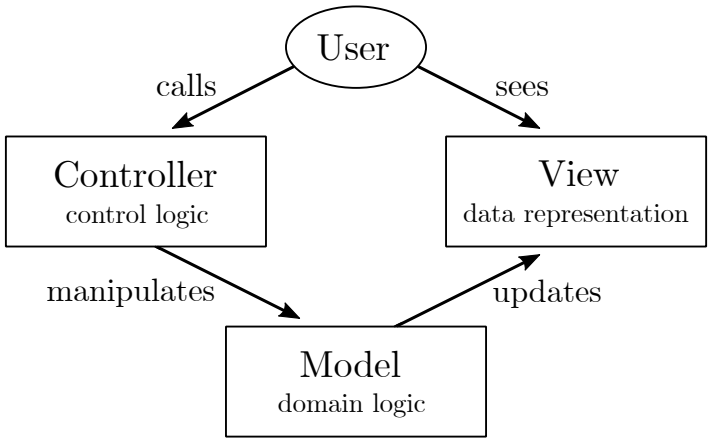


Figure 4.1: Simplified diagram of interactions within the MVC pattern

As the name implies, MVC divides an application into three components:

- **Model**: The core of the application managing its data and logic. The Model is often subdivided into independent subunits (*models*), each responsible for one part of the application or data. The Model is independent of the user API, isolated from it by the other two components.
- **View**: The representation of the data from the Model exposed to the user. Multiple *views* can exist for a same piece of data so that it can be expressed in multiple variants depending on the requirements. When the application is interactive, the user interacts with the View, e.g. clicking on a button on a web page with the web page being a view in this case.

- **Controller**: It handles the user actions and data flow, validates and translates the inputs as well as the data produced by the Model, and communicates them to the Model or the View to command them.

Figure 4.1 summarizes the interactions between these three components and the user. It is important to note that MVC is not a fixed architecture, but more often used as a blueprint to be adapted and reformulated depending on the use case's specificities and the philosophy of the engine implementing it. For example, some implementations allow the Model to directly update the View while others disallow any interaction between them, forcing them to go through the Controller acting as an abstraction layer.

4.1.4 Web template engines

To generate and adapt their views to various use cases, most MVC frameworks rely on a *template engine*. Some template engines are fully integrated into their frameworks, while others exist as standalone libraries. A very influencial template engine is the one integrated into the Django framework. It defines the *Django Template Language* (DTL), that was imitated by many other frameworks, using other underlying programming languages such as Symfony in PHP.

The DTL was designed explicitly to facilitate the segmentation of tasks between front-end and back-end developers [67]. To do so it defines *templates*, these must not be confused with C++ templates. A template is here an abstraction of a given text, in the use case of Django it is usually HTML source code. The template text is enriched with template tokens. These tokens can, for example, represent a variable that is to be replaced by its value in the final output. In the case of "full-logic" templating language, such as the DTL, these tokens can also be used to infuse some logic into a template, e.g. branching logic to enable or disable part of a template depending on a boolean evaluation. The template engine can then *render* a template with a *context* as input to resolve the template tokens using the values from the context and produce a text or file. For example, using the DTL the following string is a valid template:

```
1  Hello {{name}}{% if isHappy %}!{% else %}.{% endif %}
```

It uses two template variables: a string `name` and a boolean `isHappy`. Evaluated with the context `{"name"="World","isHappy"=True}` it produces "`Hello World!`", whereas with `{"name"="Alice","isHappy"=False}` it produces "`Hello Alice.`". From this trivial example, we can see how a template engine streamlines the production of customized outputs. Moreover, the template itself is written in the target's language with the addition of DTL's syntax tokens, making it more readable to someone only familiar with the target's language and not the framework's one.

Furthermore, thanks to template engines, the front-end developers can focus on writing the template itself using tokens while the back-end developers implement the business logic necessary to produce the context. In this example, it could be the logic required to find out the `name` variable to use, depending on a user id provided in the query and

using a database query to fetch the result, something that falls outside the expected area of expertise of a front-end developper. The possibilities offered by the DTL are further expanded by numerous more complex features that it supports, such as subtemplating and macros that allow for even higher levels of abstraction and metaprogramming using the DTL itself.

A template engine provides an efficient way to fulfill the Kernel Generator's three goals defined in Section 4.1.1. Despite this, template engines are not often directly used in HPC software involved with code generation. An exception is the MESA-PD particle dynamics code [68] developed within the waLBerla framework [69]. MESA-PD uses Jinja2 to generate the glue code steering its kernels using a custom Python library, in a fashion very similar to ExaHyPE's Toolkit. However, here the code generation process itself is not intended to be modifed by users.

4.1.5 Adapted MVC architecture for the Kernel Generator

Despite the similarities between the goals and requirements of web applications and the Kernel Generator in ExaHyPE, the latter is a much simpler use case with regard to the user interactions with the utility, and thus the MVC architecture can be adapted and simplified.

Indeed, in contrast to a web page, a rendered view from our code generation is not interactive: A user calls the Kernel Generator once, and not through previously generated files, but through a specific API. Thus the external API is handled by the Controller directly, and each view only needs to represent a pure non-interactive output. This greatly simplifies the data flow of the application, and the views are implemented as pure templates to be rendered by the template engine.

As the Kernel Generator is written in Python3, I chose the Jinja2 template engine (version 2.11) to implement and generate the views. Jinja2 is a standalone template engine that closely imitates the Django Template Language [49]. Most of the views as templates are abstractions of the C++ files to be generated, and thus have C++ as basis, extented by the DTL tokens to include some templating logic. On the other hand, using the advanced features of Jinja2, I implemented some other supporting templates and used them to provide usefull high level abstractions. As will be detailed in Section 4.4, I used this extensively to abstract away low-level optimizations from high-level algorithms, thus achieving the third goal of separating the work area of the algorithm and optimization experts.

Finally in the Model, each kind of file that can be generated is isolated as a separate output to be produced by its own model subunit. Thus, every kernel from ExaHyPE has its own model class, e.g. the SolutionUpdate kernel is handled by a `SolutionUpdateModel` class. As the views are simplified to templates, it would have added unnecessary complexity to go through the controller again to generate them. Thus each model class directly renders the view to format its data into a generated file. If a kernel has multiple variants, then its model chooses between multiple views to automatically select the appropriate one. This

is a major break from some MVC interpretations, where the Controller steers the View. However here, this break avoids some unecessary delegations, simplifying the application flow, and ensures that the generation of one kernel can be isolated to a single model class and its views once given the necessary inputs from the Controller, simplifying the development and debugging processes.

Some other logic required by the Kernel Generator, such as interacting with other utilities' APIs, is also handled by separate model classes. In this architecture, I made the choice to never have models interact directly with one another but instead always going back to the Controller if interactions were necessary, e.g. when a kernel model requires the generation of specialized functions by a third party library handled by another model. This way the Kernel Generator's components are entierly separated into independent subunits that can be directly matched with one type of output or operation.

With the adapted MVC architecture providing the necessary separation of concerns between the different steps involved in the code generation, the overall application flow of the Kernel Generator can be summarized as:

1. The controller object reads the input and validates it. It then selects and instantiates the required models, and calls each one of them with a specific *context* created from its input and containing the relevant translated data.

2. Each selected model applies its own internal logic to update its given context and selects the appropriate view to represent it. If necessary it may also return some data back to the controller for it to pass it down to other special models, e.g. when a third party library needs to be integrated.

3. Each selected view is a template, the Jinja2 template engine *renders* it using the context provided by its model.

Here, two of this architecture's advantages can already be seen for users working on the Kernel Generator.

First this architecture separates the user API from the proper code generation logic using the Controller as an abstraction layer. This allows the external API to be tuned to better match the vocabulary and understanding of application experts using it through the specification file, while having an internal representation in the Kernel Generator better matching the internal logic and combining multiple input parameters. For example, the application specifies the polynomial order used by the ADER-DG scheme as the value `order` in the specification file, but for the kernels the associated parameter used in the templates is `nDof`, the length of each spatial dimension equal to the order plus one, or `nDofPad`, which is the same parameter increased with a zero-padding to the next multiple of the architecture-specific vector register length.

Second, each variant of a generated kernel is represented by one view that is handled, with the other variants of this kernel, in one independent model. Thus modifying the code generation behavior of one specific kernel is greatly simplified, as all other parts of the code generation process can be safely ignored.

These two advantages will be discussed in the context of the development of the ExaSeis application in Sections 5.2 and 5.3.

4.2 Re-engineering of the Kernel Generator

The first version of the Kernel Generator, made at the beginning of the project, was a collection of Python3 scripts generating the kernels and controlled by a main frontend script. As explained in Section 4.1.1, while this first version provided the desired customizability, its lack of separation between the Python logic and the abstraction of the generated code made it difficult to further expand.

I rewrote the script collection to the current MVC-based version of the Kernel Generator in an incremental process, starting from the views to validate the idea of using a template engine and then updating the collection architecture to the MVC one.

4.2.1 Jinja2's templates

The first step was to validate that a template engine could be used to efficiently perform the code generation. In this initial step, the Jinja2 template engine [49] was made available directly through the local Python3 installation. Jinja2 is an open source standalone web template engine primarely designed to generate HTLM source code, but its versatility allows it to be repurposed to generate C++ source code.

For each script performing the generation of a kernel, I extracted the generated code representation and reformulated it into a template using the Jinja2's Template Language, which is close to the Django Template Language, to abstract the given kernel's source code. A template is a file representing the desired output enriched by syntax tokens, these are processed by the template engine and in the case of Jinja2 they come in three forms:

- **Variables**, delimited by `{{ ··· }}`, represent variables or template macros to be resolved and replaced by their value in the final output.
- **Statements**, delimited by `{\% ··· \%}`, are used to add logic to the template processing, e.g. branching, loops, and include statements.
- **Comments**, delimited by `{# ··· #}`, are ignored and removed from the output when rendering the template.

I used template variables to abstract known constants directly defined in the specification file, such as the number of degrees of freedom `nDof` and the qualified name of the `UserSolver` class, or derived from a combination of parameters, for example `nDofPad`, i.e. an array length increased by some padding, which takes into account the length of the hardware SIMD registers.

To include the Python logic from the previous script directly into the template, I used template statements, in particular branching. With them the rendered code can remove

some parts of the numerical scheme that are not needed, such as the computation of the gradient of the current state tensor when only a flux term not requiring it is used.

```
{% if not useCERKGuess %}
// 1. Trivial initial guess
for (int t = 0; t < {{nDof}}; t++) {
  for (int xyz = 0; xyz < {{nDof**nDim}}; xyz++) {
    std::copy_n(luh+{{nData}}*xyz, {{nData}}, lQi+{{nDataPad}}*(xyz
        ↪ +{{nDof**nDim}}*t));
  }
}
{% else %}
//[...] other initial guess scheme
{% endif %}
```

Listing 4.2: Templatized and modernized version of the code excerpt from Listing 4.1.

Listing 4.2 shows a modernized version of the code excerpt from Listing 4.1. It performs the same functionality, here copying the value from the input `luh` tensor into the state tensor `lQi` as a trivial initial guess for the upcoming Picard iterations. Unlike the previous version, the C++ code can easily be recognized and read, with template variables, such as `nDof` and `nDim`, abstracting constants to be provided by the context when rendering this template. Furthermore, logic is now included directly in the template, as illustrated by the `{% if not useCERKGuess %}` condition, which here ensures that if another initial guess scheme is selected in the context, then this part in the generated code is replaced by the part not shown here between the `{% else %}` and the `{% endif %}` statements.

Status after this step: The expected benefits from using a template engine were validated, the abstracted code is highly customizable as before, but also easily readable and modifiable, as long as the required context is provided. However at this point the templates were still basic as they did not use any macros or subtemplating, thus mixing the algorithm with some low-level optimizations.

4.2.2 Models standardization

With the code abstraction removed from the Python scripts, what was left provided the starting point for the model classes of the new MVC architecture. Model classes are instantiated and handled by the Controller. Hence, having a standardized interface simplifies this process. Also, they need to integrate the Jinja2 external library to be able to render the templates serving as views. To achieve both these goals efficiently, I introduced the `AbstractModelBaseClass` class, serving as an abstract parent class for all model classes.

The interface for its caller is standardized by the definition of its constructor and a virtual method `generateCode(self)`. The standardized contructor takes a context as argument and copies it internally using a deep copy method to ensure the absence of side effects from

other models. The `generateCode(self)` method is an abstract method to be overriden by the child classes to implement the required logic to prepare the local context, select the right view and render it into a file.

```
1 def render(self, templateName, outputFilename):
2     loader = jinja2.FileSystemLoader(Configuration.pathToTemplate)
3     env = jinja2.Environment(loader=loader, trim_blocks=True,
          ↪ lstrip_blocks=True)
4     if isinstance(templateName, str):
5         template = env.get_template(templateName)
6     else:
7         template = env.get_template(os.path.join(*templateName))
8     with open(os.path.join(context["pathToOutputDirectory"],
          ↪ outputFilename), "w") as output:
9         output.write(template.render(context))
```

Listing 4.3: The render method used to encapsulate Jinja2.

Listing 4.3 shows the `render(self, templateName, outputFilename)` method used to generate a file from a template using Jinja2. It takes the path to the template as first argument from the configured template main directory, either as a string or as a tuple of strings. The second argument is the desired filename. Jinja2 default rendering behavior needs to be modified to adapt it for C++ files generation instead of the default HTML. To do so a `jinja2.Environment` is defined and it requires a `jinja2.FileSystemLoader`. The template is then loaded as a `jinja2.Template` object using the previously defined environment and some Python3 logic to transform the tuple into a valid path if necessary. Finally the template is rendered to a string using `template.render(context)` with the model's internal context as input, and then outputed to a file using native Python3 logic. This method encapsulates all the required logic to render a template from a context in one place, therefore child classes to not need to directly include Jinja2 or reference it.

```
1 from .abstractModelBaseClass import AbstractModelBaseClass
2
3 class AdjustSolutionModel(AbstractModelBaseClass):
4
5     def generateCode(self):
6         self.render((self.context["kernelType"], "adjustSolution_cpp.
              ↪ template"), "solutionAdjust.cpp")
```

Listing 4.4: `AdjustSolutionModel` class from the Kernel Generator handling the creation of the `solutionAdjustment` kernel.

Using the abstract class defined previously, for the simplest kernels where the associated script was left almost empty after extracting the template, a model can be defined in a few lines of code as shown for the `AdjustSolutionModel` class in Listing 4.4. Here, the model looks at the desired `kernelType` (i.e. ADER-DG, FV or Limiter) to select the

correct template file, as a variant for each one exists. A given model might also render multiple templates to produce multiple files, usually an implementation file and its header file.

```
1  def generateCode(self):
2      self.context["sqrt_half_Pi"] = math.sqrt(math.pi/2)
3
4      if self.context["kernelType"] == "aderdg":
5          if(self.context["isLinear"]):
6              self.render(("aderdg", "riemannSolverLinear_cpp.template"
                   ↪ ), "riemannSolver.cpp")
7          else:
8              self.render(("aderdg", "riemannSolverNonLinear_cpp.
                   ↪ template"), "riemannSolver.cpp")
9      elif self.context["kernelType"] == "fv":
10         self.render(("fv", "riemannSolver_cpp.template"), "
               ↪ riemannSolver.cpp")
```

Listing 4.5: `RiemannSolverModel`'s `generateCode(self)` method setting up its model and selecting the correct template to produce the `RiemannSolver` kernel.

More complex model classes use more logic to select the template files to render and expand the default context with relevant data. An example of the behavior is presented in Listing 4.5 with the `RiemannSolverModel` class, where two variants of the RiemannSolver kernel exist for an ADER-DG solver, and the value of $\sqrt{\pi/2}$ is precomputed and added to the context.

Some other models perform multiple computations to set up value arrays to be hard-coded in their view. For example the `QuadratureModel` computes the Gauss-Legendre quadrature nodes and weights, as well as some precomputed weight combinations that are used to optimize kernels by replacing the corresponding computations.

Finally the model may return requests or useful informations to its caller as return value of the `generateCode()` method.

I also encapsulated the necessary code to call the command line API of the third party code generator LIBXSMM, used to generate highly optimized matrix multiplication code, inside a special model class `GemmsGeneratorModel`. While this class does not use Jinja, it follows the same external API signatures as the other model and requires a context specifying the configuration of the matrix multiplication functions to be generated by LIBXSMM. With this class, the third party library is isolated from the rest of the Kernel Generator and thus the integration of another BLAS library using code generation is greatly simplified.

Status after this step: The models and views of the new MVC architecture are properly defined and linked in independent subunits.

4.2.3 Controller

To implement the Controller, I rewrote the front-end script into a proper `Controller` class.

The initialization of this class takes the user given input as a Python dictionary and translates it to an internal context, also in the form of a dictionary. The context is then also expanded with other useful secondary parameters computed from the primary ones. For example, the number of degrees of freedom, `nDof`, is often used increased to the next multiple of the vector registers' length, which depends on the speficied architecture, as a new variable `nDofPad`.

From the context it built, the Controller uses its internal logic to determine which kernels, and thus models, are required. This process is greatly simplified by the standardized API of the models introduced earlier. Model objects are fully independent from one another and thus the order in which they are called is only constrained by the information needed to set up their contexts. In particular a model responsible for integrating generated code from a third party library, such as the previously introduced `GemmsGeneratorModel`, is called last after the other models defined the required third party code to be generated.

Status after this step: The Kernel Generator now follows the desired MVC architecture. However, its external API was still not formalized and did not match the previous command line interface.

4.2.4 External API

I isolated the external API of the front-end script into a separate `ArgumentParser` class. This allowed me to provide two external APIs. The first one is the existing command line API used at the time by the Toolkit to call the Kernel Generator. The second one is a pure Python API where a Python dictionary equivalent to the one the command line API would produce is directly given to the Controller. This new second API was made to facilitate the integration of the Kernel Generator into another Python application.

To unify the definition of both APIs, I defined arrays of tuples specifying the expected inputs and their types such as a mandatory integer argument or an optional boolean, as can be seen in the code excerpt in Listing 4.6. These tuples also contain a description to be shown in the command line interface with the usual `-h` or `--help` option. For example, the tuple `("numerics", ArgType.MandatoryString, "linear or nonlinear")` specifies that the Kernel Generator expects to be told to generate linear or nonlinear kernels with a mandatory `string` argument. In the command line interface this is a positional argument, whereas the Python interface expects it in an entry of the input dictionary called `numerics`.

The command line interface is constructed on the fly from these arrays of tuples using Python3's `argparse` module, which is the Pythonic way to define a command line interface for a Python3 application and parse its input into a Python dictionary. The Python

```
aderdgArgs = [
    # mandatory arguments
    ("kernelType",          ArgType.MandatoryString, "aderdg"),
    ("pathToApplication",   ArgType.MandatoryString, "[...]"),
    ("pathToOptKernel",     ArgType.MandatoryString, "[...]"),
    ("namespace",           ArgType.MandatoryString, "[...]"),
    ("solverName",          ArgType.MandatoryString, "[...]"),
    ("numberOfVariables",   ArgType.MandatoryInt,    "[...]"),
    ("numberOfParameters", ArgType.MandatoryInt,    "[...]"),
    ("order",               ArgType.MandatoryInt,    "[...]"),
    ("dimension",           ArgType.MandatoryInt,    "[...]"),
    ("numerics",            ArgType.MandatoryString, "[...]"),
    ("architecture",        ArgType.MandatoryString, "[...]"),
    # optional arguments
    ("useFlux",             ArgType.OptionalBool,    "enable flux"),
    ("useViscousFlux",      ArgType.OptionalBool,    "[...]"),
    ("useNCP",              ArgType.OptionalBool,    "[...]"),
    ("useSource",           ArgType.OptionalBool,    "[...]"),
    ("useMaterialParam",    ArgType.OptionalBool,    "[...]"),
    ("usePointSources",     ArgType.OptionalInt ,    "[...]", -1, "
        ↪ numberOfPointSources"),
    [...]
]
```

Listing 4.6: Excerpt of the Kernel Generator's `ArgumentParser` with some of the arguments for the generation of ADER-DG kernels. Most descriptions were removed from this excerpt.

dictionary produced by the command line interface, or directly provided as input from the other API, is then compared to the specification described by the tuples for validation. This design simplifies the modification of the external APIs as modifying the content of the arrays and tuples is all that is required to modify both API and this format keeps it easily readable. For example, adding a new optional boolean parameter can be done by simply adding a tuple with `ArgType.OptionalBool` type to the list of arguments.

Status after this step: The Kernel Generator's complete data flow from its external API to the code generation is handled using the new MVC architecture.

4.2.5 The Kernel Generator as a Python3 module

Finally, I added the necessary Python `__init.py__` files to turn the Kernel Generator application into a proper Python3 module that can be imported in other Python scripts or modules. Furthermore, I added a `__main.py__` file that implements the Python code to call and run the Kernel Generator as a script using its command line external API as seen in Listing 4.7. For example, the following command generates ADER-DG kernels in "`myApp/myKernels`" for a three dimensional ADER-DG linear application at polynomial

```
1  import sys
2  import os
3
4  def main():
5      sys.path.append(os.path.abspath(os.path.join(os.path.dirname(
           ↪ __file__),"..")))
6      from kernelgenerator import Controller
7      control = Controller()
8      control.generateCode()
9
10 if __name__ == "__main__":
11     # execute only if run as a script
12     main()
```

Listing 4.7: Kernel Generator's `__main.py__` to be able run the application as a script.

order 7 with a PDE system using only a flux term and 12 variables, optimized for Skylake architecture:

```
python3 kernelgenerator aderdg myApp myKernels kernelsNameSpace
    ↪ Elastic::ElasticWaveSolver 12 0 7 3 linear skx --useFlux
```

Status after this step: The Kernel Generator can be used as a Python3 module.

4.2.6 Dependency integration

Additionally, I refactored the integration of external dependency of the Kernel Generator. During the project development, we worked on multiple clusters only providing a minimal installation of Python, or one with outdated components. Thus, Python dependencies were removed when possible, or otherwise added from their source downloaded as a submodule of ExaHyPE to Python's `sys.path`. To do so the path to the dependency is required. It is hard-coded in a `configuration.py` file, so that the user can easily change the specific local installation used. By default it points to ExaHyPE's submodules directory where the dependencies should be present if ExaHyPE's installation steps are followed.

For example, as shows in Listing 4.8, I modified the `AbstractModelBaseClass` from Section 4.2.2 to use the Jinja2 local source code, with its dependency Markupsafe, instead of using the local installation in Python, which might be missing or using an outdated version of Jinja2.

The configuration file is also used to set up some code generation behaviors that are expected to be installation specific due to hardware or software specificities, such as the choice of BLAS library used by the Kernel Generator.

```
1 # add path to dependencies
2 from ..configuration import Configuration
3 sys.path.insert(1, Configuration.pathToJinja2)
4 sys.path.insert(1, Configuration.pathToMarkupsafe)
5
6 import jinja2
```

Listing 4.8: Importing Jinja2 in the `AbstractModelBaseClass` from its source code located in the submodule.

Replacing numpy

To reduce the number of external dependencies, I removed the numpy library from the Kernel Generator, as the code generation performance is not critical and numpy itself has too many dependencies to be easily used from its source code, as described earlier, if not provided with Python. I implemented the linear algebra operations needed by the Kernel Generator to compute some quadrature related coefficient matrices in pure Python in the `MathUtils.py` utility file.

It provides basic method to manipulate, transpose, pad, and flatten matrices. The matrix inversion is performed using a basic Gauss-Jordan elimination which is fast enough for the considered matrix sizes. Furthermore, the quadrature nodes and weights for both Gauss-Legendre and Gauss-Lobatto quadratures are hard-coded to very high precision, using numpy's values as reference. To ensure high numerical accuracy, I used Python's `decimal` type to perform the floating point computations with 50 decimal positions by default and the resulting values are hard-coded in the views with as many decimals. This way the computed coefficients are precise even if the engine is modified to use a quadruple-precision format instead of its current double-precision implementation.

Despite not using numpy and the increased precision used, the measured runtime of the Kernel Generator using the `MathUtils.py` utility stays below one second. Thus the benefits of it came at no noticable cost during the code generation step.

Status after this step: The number of external dependencies is reduced and all remaining external dependencies are properly included from their source codes included in ExaHyPE as a submodule.

4.2.7 Evaluation of the new design

The new Kernel Generator's use of Jinja2's templates greatly improved the abstraction of the generated code by making it very close to C++, while keeping all the customizability of the initial approach. Furthermore, using the templating logic and with the MVC architecture defining the context used by the template engine, the behavior of the code generation can easily be modified. The streamlined data flow from the external API to the

templates facilitates the addition of new features and provides a standardized workflow to perform this task, as will be illustrated by the use case in Chapter 5.

Therefore, this new architecture fulfills its first two design goals of customizability and expandability defined in Section 4.1.1. However, at this stage the separation of concerns is still missing as the templates mix the algorithms with the low-level optimizations.

4.3 Rewriting the Toolkit from Java to Python

The Toolkit is responsible for setting up a new application by generating its glue code and skeleton code, using the application's specification file as input, as well as calling the Kernel Generator. With the Kernel Generator proving the benefits of using templates and a MVC architecture, the Toolkit was fully rewritten to follow the same design principles. This work was performed by Sven Köppel, Dominic Charrier and myself.

4.3.1 Templatization of the initial Java application

The Toolkit was formerly a Java application pre-compiled to runnable jar file. The reason for this initial choice was to reuse code from a previous project, in particular the logic to process the custom made domain specific language (DSL) used at the time for the specification file. It initially used the same process as the script based Kernel Generator to generate files, with the java `write` method, e.g.

```
1  _writer.write("#include \"" + _solverName + ".h\"\n");
```

to write the C++ line responsible to include the solver header file.

As the benefits of using a template engine and templates to abstract the generated code were proved by the new Kernel Generator, we replicated the extraction of templates performed in Section 4.2.1 in the Toolkit. However, unlike with Python, adding a third party template engine to the Java application proved too difficult to do without increasing the installation complexity. Thus at first, instead of a template engine, simple logicless templates were written, i.e. using only templating variables and no statements to include logic, with the templating variable being replaced using Java's `replaceAll` method. Later, I wrote a custom basic template engine, Minitemp[1], to support basic logic by converting a template into an Abstract Syntax Tree (AST) to evaluate with the provided context to render it. By design Minitemp implements a template language close to the one used by Jinja2, with Pythonic idioms such as the boolean operators replaced by Java's ones to simplify the AST evaluation. Using Minitemp, basic logic was introduced to the templates.

The use of templates greatly increased the readability of the code abstraction and the customizability of the generate code, as was the case with the Kernel Generator.

[1] `https://github.com/gallardjm/minitemp`

4.3.2 External API with JSON Schema

The main reason for having the Toolkit be written in Java was the SableCC framework [45], which can generate parsers for any specified custom grammar. The grammar of the language to be parsed is expressed using a Backus-Naur form (BNF), a meta-language [70, 71], i.e. defining the tokens and the production rules of the language. This was used to define the specification file language, and then have SableCC produce a Java interpreter capable of validating and parsing it for the Toolkit. It allowed ExaHyPE to define its own Domain Specific Language for the specification file, making it easier to write and read. However, it also made it harder to modify the structure of the specification file and the options it included, as this requires modifying the language's grammar, and thus some non-trivial understanding the BNF's concepts and of the existing grammar.

With more and more options added to the grammar, the limitation of SableCC were reached and the interpreter could not be generated anymore, as it was becoming too large. As SableCC needed to be replaced, we made the decision to fully rewrite the Toolkit in Python to be able to reuse parts of the Kernel Generator's code and replace the custom Minitemp template engine with the more robust and powerfull Jinja2, that was already integrated to ExaHyPE for the Kernel Generator.

Since supporting a custom DSL for the specification file introduced unnecessary complexity for the users, we made the choice to change the specification file's language and after evaluating multiple options, we chose JavaScript Object Notation (JSON) for its ease of use, widespread usage, and third party library support, in particular with existing validation libraries. JSON was first specified in 2006 in RFC 4267 [72], updated 2017 in RFC 8259 [52] with its standardization as ISO/IEC 21778:2017 [51].

To read and validate a JSON specification file, the Toolkit relies on a JSON Schema [73]. A JSON Schema takes the form of a JSON file describing the grammar and values expected in another JSON file.

In a similar way to what was achieved by the templates, using a JSON Schema made the abstraction of a specification file in the form of its grammar much closer to instances of it. For example, the grammar excerpt in Listing 4.9 from the Toolkit's JSON Schema describes how the `solver` part of a specification file for an ADER-DG solver, such as the one shown in Listing 3.4, has to be an object that needs to include at least an entry `terms` and can have another entry `nonlinear`. The `terms` entry has to be an array containing some of the enumerated values to describe which parts of the canonical PDE (2.1) are used by the application, for example in Listing 3.4 it contains the `flux` and `ncp` values. Meanwhile, the optional `nonlinear` entry tells the Toolkit if the application requires nonlinear kernels or not, if not specified a nonlinear application is assumed. Other options are not shown here but are in the full JSON Schema that can be found in the project[2]. The JSON schema greatly simplified the modification of the specification file grammar to add new options, as will be demonstrated in Section 5.2.1 where it simply translates to adding a few lines of JSON code where the new option should appear in a specification file.

[2] at ExaHyPE-Engine/Toolkit/exahype-specfile.schema.json

```
"aderdg_kernel" : {
  "type" : "object",
  "scope" : "compile-time",
  "required" : [ "terms" ],
  "additionalProperties" : "false",
  "properties" : {
    [...]
    "terms" : {
      "type" : "array",
      "title" : "PDE terms to generate code for",
      "items" : {
        "type" : "string",
        "enum" : ["flux", "source", "ncp", "material_parameters",
            ↪ "point_sources", "viscous_flux"]
      }
    },
    "nonlinear" : {
      "type" : "boolean",
      "title" : "Is the PDE nonlinear?",
      "default" : true
    },
    [...]
  }
},
```

Listing 4.9: Excerpt of the Toolkit's JSON Schema showcasing how some components of the specification file from Listing 3.4 are specified and validated.

We used the JSON Schema third party library to obtain a validator object from the schema, to validate the Toolkit input. If successful, the validator returns a Python3 `json` object that can be read using native Python3 functions. Sven Köppel also added a conversion script to translate a specification file from the old DSL format to the new JSON one. Should ExaHyPE's users desire to use another unsupported format, e.g. YAML or XML, it would be sufficient to implement a conversion tool from this language to JSON. JSON being a widely used standardized data interchange format and natively supported by Python3, third party library, e.g. PyYAML [74], can be used to greatly simplify this conversion.

4.3.3 Kernel Generator's architecture reuse

With the input validation and translation to a Python dictionary in place, and the code to be generated already abstracted with templates, we simply reused and adapted the Controller and Model from the Kernel Generator to bind the two ends together.

Like its Kernel Generator counterpart, the Toolkit's `Controller` class takes the dictionary produced by its input validation, this time from the JSON Schema, and produces contexts with it to then instantiate and call models with their appropriate context. Unlike with

the Kernel Generator, multiple contexts are created as not all models require the same informations. For example, the model generating the `AbstractSolver` glue code requires information about the solver, while the one producing the Makefile requires the path to the Engine components.

For the Model we reused the `AbstractModelBaseClass` from the Kernel Generator, and we then implemented child classes of it to define the Toolkit's models, one for each type of file to be generated. The new models use the templates from the previous Java Toolkit, with only minor modifications to them to adapt the template language to the one of Jinja2.

To configure the paths to the submodules and other installation specific parameters, we copied the process used by Kernel Generator and defined a `configuration.py` file.

Finally we added the necessary Python's `__init.py__` files to make the new Toolkit a Python module. We then implemented a `__main.py__` file using the Kernel Generator's one as basis so that the Toolkit can also easily be called from the command line. To further simplify the way a user can call the Toolkit, we added a `toolkit.sh` bash script that checks that all the necessary dependencies, i.e. Python3 and the submodules, are present and then calls the Toolkit with the provided specification file and options.

Thanks to the previous re-engineering serving as an example, and the code reuse from the Kernel Generator for the architecture and from the old Java Toolkit for the templates, we were able to implement the new Python3 Toolkit over a very short period of time during a coding week event.

4.3.4 Binding with the Kernel Generator

After having rewritten the Toolkit from Java to Python3, we considered fusing the Toolkit and Kernel Generator into one single Python3 module. However, we decided against it to maintain the clear separation between the kernels, where a lot of performance critical optimization are required, and the glue code. Furthermore, keeping the Kernel Generator separated and with a command line API means that it can more easily be used as a standalone Python3 module by other projects implementing an ADER-DG scheme, where the ExaHyPE specific glue code and specification file would be irrelevant.

With the Toolkit now being a Python3 module, I modified its interface to the Kernel Generator to use the Python API instead of the command line one. Similarly to what I did with other third party libraries, I isolated the Toolkit's interface with the Kernel Generator in a special `KernelGeneratorModel` class. This class translates the context provided by the Toolkit's Controller from the specification file to a valid input (`KGContext`) for the Kernel Generator external Python API, described in Section 4.2.4. It then calls and runs the Kernel Generator's Controller imported as a Python module through its API:

```
1 KGController = kernelgenerator.Controller(KGContext)
2 KGController.generateCode()
```

Finally the `KernelGeneratorModel` returns the path to the optimized kernel it uses to set up the Kernel Generator to its controller so that this value can be added to the context given to the models generating the `AbstractSolver` glue code binding the generated kernels to ExaHyPE core.

4.3.5 Evaluation of the rewriting

The new Python3 Toolkit provides the same benefits as the new Kernel Generator, being more readable and easier to expand, especially with the migration of its input validation from SableCC to JSON Schema. In addition, by migrating the Toolkit from Java to Python, we reduced the number of programming languages used in ExaHyPE, and by reusing the Kernel Generator's dependencies, we also reduced the amount of external dependencies used overall. By replicating the Kernel Generator's architecture in the Toolkit, users modifying the Toolkit and Kernel Generator only need to familiarize themselves with one design philosophy and experience gained working on one utility can be applied when working on the other.

4.4 Algorithmic templates and optimization macros

The new MVC-based Kernel Generator using basic templates did not yet fulfill its third design goal, lacking the separation of concerns as the templates implemented the algorithm and contained the logic necessary for architecture-aware optimizations. Using Jinja2's subtemplating and custom macros, I modified the templates to isolate the optimization logic from the templates inside macros, in this way obtaining architecture-oblivious *algorithmic templates* using architecture-aware *optimization macros*.

4.4.1 Architecture-oblivious algorithmic templates

Listing 4.10 shows an excerpt of an algorithmic template used in the Kernel Generator. This piece of code processes the result of the flux terms in the first spatial direction in one of the linear SpaceTimePredictor kernel variant.

Here we can see the implementation of a customizable numerical scheme. Two cases are available with the "`if / else / endif`" template statements at line 1, 10 and 12. Depending on the value of the context variable `useMaterialParam`, only the first or second part of the branching is rendered. This variable represents the choice of using, or not, the *material parameter matrix* PDE term, to adapt the overall PDE solved to the application requirements. This choice is made in the specification file and following the MVC architecture it is translated by the Controller to this boolean in the context.

Some architecture-specific optimizations are already present here within the template variables and pragma used. The variable `nVarPad` abstracts the number of variables increased to the next multiple of the vector register's length, defined by the speficied architecture, e.g. 8 on Skylake with a double-precision floating point number format.

```
1  {% if useMaterialParam %}
2  {{ m.matmul('flux_with_mp_x', 'flux', 'negativeDudxT_by_dx', '
       ↪ tmpArray', '0', '0', '0') | indent(6) }}{##}
3  for (int x = 0; x < {{nDof}} ; x++){
4    solver.{{solverName}}::multiplyMaterialParameterMatrix(lPi + {{
         ↪ idxLPi(0,yz,x,0)}}, tmpArray + x*{{nVarPad}});
5    #pragma omp simd aligned(lQi_next,tmpArray:ALIGNMENT)
6    for (int n = 0; n < {{nVarPad}}; n++){
7      lQi_next[{{idx(0,yz,x,n)}}] += tmpArray[{{idx(0,0,x,n)}}];
8    }
9  }
10 {% else %}
11 {{ m.matmul('flux_x', 'flux', 'negativeDudxT_by_dx', 'lQi_next', '0',
       ↪  '0', idx(0,yz,0,0)) | indent(6) }}{##}
12 {% endif %}{# useMaterialParam #}
```

Listing 4.10: Simplified code excerpt from the fusedSPTVI_linear_split_ck_cpp template applying the result of the flux terms in the x direction to the current state with or without applying a multiplication with material matrix.

Line 5 with "`#pragma omp simd aligned(lQi_next,tmpArray:ALIGNMENT)`" is an optimization pragma to enable the auto-vectorization capability of the compiler. During compilation it uses the architecture defined in the generated Makefile to perform the loop more efficiently with the appropriate SIMD instruction set. Both these basic optimizations are set outside of this template itself, either in the Controller when setting the variable or in the Makefile at compilation, and thus this template does not directly implement architecture-aware logic.

A more important optimization comes from the use of the `matmul` optimization macro at line 2 and 11. This custom macro implements an optimized matrix multiplication. Depending on the configuration of the Kernel Generator, this could be a vectorized three loops implementation in pure C++ or a call to a highly-tuned function in assembly code generated by LIBXSMM, taking into account the target architecture and the matrix multiplication settings.

Finally, quality of life `idxLPi` and `idx` macros are used to abstract the fact that the tensors are stored as one-dimensional array and use instead a more easily readable multidimensional notation, that they automatically convert in the finalized code. It removes some potential for errors, e.g. using the wrong dimension length, and increases the readability of the template.

If the template in Listing 4.10 is rendered with `useMaterialParam = False` and the Kernel Generator configured to not use BLAS libraries, then the produced code could look like the excerpt in Listing 4.11. Here, we can recognize the matrix multiplication from the second case defined by the `matmul` macro at line 11 of the template.

```
1 for (int it_1 = 0; it_1 < 6; it_1++) {
2   for (int it_2 = 0; it_2 < 6; it_2++) {
3     #pragma omp simd aligned(lQi_next,tmpArray,negativeDudxT_by_dx:
          ↪ ALIGNMENT)
4     for (int it_3 = 0; it_3 < 12; it_3++) {
5       lQi_next[yz*72+it_1*12+it_3] += tmpArray[it_2*12+it_3] *
            ↪ negativeDudxT_by_dx[it_1*8+it_2];
6     }
7   }
8 }
```

Listing 4.11: Excerpt in Listing 4.10 rendered with `useMaterialParam = False` and no BLAS library.

```
1 gemm_12_6_6_x(tmpArray, negativeDudxT_by_dx, gradQ);
2 for (int x = 0; x < 6 ; x++){
3   solver.Elastic::ElasticWaveSolver::multiplyMaterialParameterMatrix(
        ↪ lPi+(yz*6+x)*0, gradQ+x*12);
4   #pragma omp simd aligned(lQi_next,gradQ:ALIGNMENT)
5   for (int n = 0; n < 12; n++){
6     lQi_next[(yz*6+x)*12+n] += gradQ[x*12+n];
7   }
8 }
```

Listing 4.12: Excerpt from Listing 4.10 rendered with `useMaterialParam = True` and using LIBXSMM for the matrix multiplication.

If instead `useMaterialParam = True` and LIBXSMM is used to generate an optimized function to perform the matrix multiplication, then the produced code with the other parameters being equal is shown in Listing 4.12. The matrix multiplication is performed by the function `gemm_12_6_6_x` and the rest of the variables are hard-coded values as expected. In the innermost loop at line 6 in the generated code, the index macro used in `lQi_next[{{idx(0,yz,x,n)}}]` was correctly transformed to a direct index computation in `lQi_next[(yz*6+x)*12+n]`.

Macros play a key role to remove the architecture-aware logic from the templates. Algorithm experts implement an algorithm in the form of an architecture-oblivious template without having to consider the low-level optimizations, instead using architecture-aware template variables set by the Controller and abstract optimization macros developed by optimization experts. The rendered code contains the correct variant of the algorithm described in the template, but with the architecture-aware optimizations that were moved to the variables and macros.

4.4.2 Macros for readability

Using macros, complex template logic can be hidden behind a single line macro call. Jinja2 comes with some macros built-in, but most of them are useful only in the context of HTML code generation, e.g. escaping HTML special characters. However, Jinja2 also allows the definition of custom macro using the `macro` statement.

```
{% macro index_2(x1, x2, L2) %}
{% if x1 == 0 or x1 == "0" %}
{{x2}}{##}
{% elif x2 == 0 or x2 == "0" %}{# fuse dimension #}
{{x1}}*{{L2}}{##}
{% else %}
{{x1}}*{{L2}}+{{x2}}{##}
{% endif %}
{% endmacro %}
```

```
{% macro index_3(x1, x2, x3, L2, L3) %}
{% if x1 == 0 or x1 == "0" %}
{{index_2(x2,x3,L3)}}{##}
{% elif x2 == 0 or x2 == "0" %}{# fuse dimension #}
{{index_2(x1,x3,L2*L3)}}{##}
{% else %}
{{index_2("("~x1~"*"~L2~"+"~x2~")",x3,L3)}}{##}
{% endif %}
{% endmacro %}
```

Listing 4.13: The index macros used to compute the index of a multidimensional tensor stored as a flattened and padded array.

Listing 4.13 shows some macros I introduced to compute tensor indexes, such as in the example in Listing 4.10. They follow a recursive design where each function adds another dimension and relies on the previous one with the simple two-dimensional case as starting point.

The first one `index_2(x1, x2, L2)` takes 3 template arguments, two position variables and the length of the fastest dimension, as the length of the slowest dimension is not needed to compute the position in the one-dimensional array storing the tensor. It then outputs the template code `{{x1}}*{{L2}}+{{x2}}` where each of the three variables has to be rendered. Using some logic it also simplifies this expression if the given `x1` or `x2` is the constant 0.

Building recursively from it `index_3(x1, x2, x3, L2, L3)` does the same for three-dimensional tensors by processing the two fastest dimensions into one expression and calling `index_2`. Once again, it also simplifies the expression if possible. Continuing this process, I defined macros up to `index_7(...)`.

To avoid having to specify the length of the dimensions each time, partial function application can be used to define new macros in the template using them, with these variables already defined as shown in the following code excerpt:

```
{% macro idx(z,y,x,n) %}{{index_4(z,y,x,n,nDof,nDof,nVarPad)}}{% end
    ↪ macro %}
```

A four arguments `idx(z,y,x,n)` macro is defined from the seven arguments `index_4` one, with the last 3 arguments specified in the new macro definition, here the context provided integers `nDof` and `nVarPad` used to define the size of the tensor's dimensions. The locally-defined `idx` macro computes a position in the one-dimensional array representation of a "?×`nDof`×`nDof`×`nVarPad`"-tensor. When rendered in the example from the previous section, the macro call `{{idx(0,yz,x,n)}}` at line 7 of Listing 4.10 produced the code `(yz*6+x)*12+n` with `yz`, `x` and `n` being local C++ loop integer variables in the generated source file.

The index macro only provides some quality of life improvement, one could also write `(yz*{{nDof}}+x)*{{nVarPad}}+n` instead of `{{idx(0,yz,x,n)}}`. However, the latter is easier to recognize and read while also being shorter. Using the index macro factorizes the logic in one place, should the length of some dimension of the tensor be modified, editing in one line the `idx` macro definition would propagate the change everywhere. More importantly, the macro removes the need to remember the dimensions' length every time an index is required, and reduce the possibility of introducing a bug when computing an index. Should one write the full expression to compute the index instead of using the macro, a simple error in the formula, e.g. using the wrong length, would be reflected in the generated code, and result in a potentially hard to find bug, wasting numerous hours of work.

4.4.3 Architecture-aware optimization macros

I introduced custom macros to hide complex low-level optimizations from the template using them. Listings 4.14 and 4.15 show the two macros used to allocate and free the temporary arrays used in the kernels. While the C++ standard way to allocate arrays is a one liner, someone not familiar with low-level optimization concepts might forget some optimization options. Furthermore, allocation on the stack is usually faster and thus prefered. However, operating system limitations might make it impossible, in particular on clusters where a standard user is not allowed to increase the stack size. Therefore, the `allocateArray` macro can abstract both kinds of allocations and use the one chosen in the specification file. The `freeArray` macro needs to be used for each array allocated with the macro in case heap allocation is chosen but it does not produce any code when rendered if stack allocation was used.

Furthermore, as will be discussed in Chapter 6, low-level optimizations require the arrays to be aligned. In the case of an allocation on the stack, this can be done using `__attribute__((aligned(ALIGNMENT)))`, with `ALIGNMENT` being a preprocessor macro defined in the Makefile. Whereas for heap allocation, alignment can be obtained from

```
1  {% macro allocateArray(name, size, type="double", setToZero=False,
       ↪ pointerExists=False, forceStack=False) %}
2  {% if tempVarsOnStack or forceStack %}
3  {{type}} {{name}}[{{size}}] __attribute__((aligned(ALIGNMENT))){% if
       ↪ setToZero %}  = {0.}{% endif %};
4  {% else %}
5  {% if not pointerExists %}{{type}}* {% endif %}{{name}} = (({{type}}
       ↪ *) _mm_malloc(sizeof({{type}})*{{size}}, ALIGNMENT));
6  {% if setToZero %}
7  std::memset({{name}}, 0, sizeof({{type}})*{{size}});
8  {% endif %}
9  {% endif %}
10 {% endmacro %}
```

Listing 4.14: The `allocateArray` macro.

```
1  {% macro freeArray(name) %}
2  {% if not tempVarsOnStack %}
3  _mm_free({{name}});
4  {% endif %}
5  {% endmacro %}
```

Listing 4.15: The `freeArray` macro.

the non-standard `_mm_malloc` function supported by the Intel Compiler and G++. This function needs to be matched with an `_mm_free` to free the allocated memory.

Finally, some optional parameters allow the templates using the macro to adapt it to their needs, such as chosing the number representation type used, by default `double`, or initializing the allocated array to zero.

This example illustrates how macros can be used to separate the work area of algorithm and optimization experts. The latter developing the *optimization macros* used by the former in the main *algorithmic templates* to integrate low-level optimizations. The template engine combines the two when rendering the templates into files having both an algorithm tailored toward the application need from the algorithmic templates, and with the low-level compiler- and architecture-specific optimizations from the macros.

4.4.4 Subtemplates

Jinja2 provides both import and include statements. The former renders the included template using the context of the parent template and copies the result where the statement is made, whereas the latter caches the imported template, which is useful to import macros' definitions from another file. Using these, I isolated complex templating logic in subtemplates, i.e. templates included or imported in other templates.

Importing the optimization macros

A first use case of this feature is to import the macros in algorithmic templates. I defined all indexes macros in a `index.template` file and the other optimization macros are written or included in a `macros.template` file. The macros can then be imported in an algorithmic template requiring them using:

```
1 {% import "subtemplates/macros.template" as m with context %}
2 {% import "subtemplates/index.template" as i with context %}
```

where a namespace for the imported macros is also defined, as seen in the `m.matmul` used in the template example in Listing 4.10.

Factorizing repeated complex part of an algorithm

A second use case is for highly optimized and repeated part of an algorithm. For example to enable the use of vectorized user functions in the STP kernel, I implemented a scheme where on-the-fly transposition is used to change the data layout when calling the user functions. This changed the template code used to call the user functions from a couple lines of code to over 100 lines with much templating logic involved. I also had to do it multiple times in the kernel's template, therefore isolating this logic in a subtemplate allowed the algorithmic template to stay easily readable and avoid code duplication. The subtemplate uses local variables, such as the name of the arrays involved or the name of some loop variable that changes depending on which of the calls is replaced by the subtemplate. It is then included using

```
1 {% with inputQ='lQi', inputQ_dataSize=nDataPad, outputF='lFi',
      ↪ timeInterleaved=False, time_var='i' %}
2 {% include 'subtemplates/flux_PDE_over_xyz.template' %}
3 {% endwith %}
```

The `with` statement defines a new scope for template variable with scope local variables to be used with the current context variable when rendering the subtemplate. This specific use case will be discussed in more detail in Section 5.3.1.

Isolating an important macro

Finally, I used subtemplating to isolate critical complex macros in their own files. This is particularly useful for macros that could be modified to expand the capabilities of the Kernel Generator. Listing 4.16 shows a simplified version of the `matmul` subtemplate. This subtemplate is used to perform an optimized matrix multiplication, with a *General Matrix Multiplication* (GEMM) from a BLAS libraries, if configured to do so. Thus, it can be expanded to add support to new BLAS libraries, as is illustrated in the use case in Section 5.3.2 where I added support for Eigen.

I start the subtemplate with some comments describing its behavior and expected template variables (shorten in the figure). A GEMM computes $C = \alpha(A \cdot B) + \beta C$ for three

```
{#    C = alpha *  A   *   B  + beta * C
   (M x N)      (M x K) (K x N)
  The gemm config (conf) contains M, N, K, LDA, LDB, LDC, alpha and
     ↪ beta
#}
{% with %}
{% set conf = matmulConfigs[key] %}
{#
// LIBXSMM case
//-------------
#}
{% if useLibxsmm %}
{{conf.baseroutinename}}({{A}}+{{A_shift}}, {{B}}+{{B_shift}}, {{C
   ↪ }}+{{C_shift}});
{#
// No BLAS case
//-------------
#}
{% else %}{# no BLAS library #}
{% if conf.beta == 0 %}
// reset {{C}}
for (int it_1 = 0; it_1 < {{conf.N}}; it_1++) {
  #pragma omp simd aligned({{C}}:ALIGNMENT)
  for (int it_3 = 0; it_3 < {{conf.M}}; it_3++) {
    {{C}}[{{C_shift}}+it_1*{{conf.LDC}}+it_3] = 0.;
  }
}
{% endif %}
for (int it_1 = 0; it_1 < {{conf.N}}; it_1++) {
  for (int it_2 = 0; it_2 < {{conf.K}}; it_2++) {
    #pragma omp simd aligned({{C}},{{A}},{{B}}:ALIGNMENT)
    for (int it_3 = 0; it_3 < {{conf.M}}; it_3++) {
      {{C}}[{{C_shift}}+it_1*{{conf.LDC}}+it_3] {{ '+' if conf.alpha
         ↪ == 1 else '-' }}= {{A}}[{{A_shift}}+it_2*{{conf.LDA}}+
         ↪ it_3] * {{B}}[{{B_shift}}+it_1*{{conf.LDB}}+it_2];
    }
  }
}
{% endif %}{# end BLAS case switch#}
{% endwith %}
```

Listing 4.16: Simplified `matmul` subtemplate to perform $C = \alpha(A \cdot B) + \beta C$.

matrices A, B, C, with α being -1 or 1 and β either 0 or 1. The subtemplate starts a new scope using a `with` statement and sets some local variables, just one in this simplified excerpt. The `conf` template variable is a dictionary containing the relevant parameters to define the GEMM such as the size of the matrices and the coefficient α and β. Then the BLAS library used is selected using branching, here only two possibilities are shown, either the LIBXSMM generated code or a standard pure C++ implementation.

In the first case the model using this macro also tells the Kernel Generator's Controller to generate a `gemm` function through the `GemmsGeneratorModel` introduced in Section 4.2.2. Thus, here the matrix multiplication is simply performed by calling the generated function. Other macros in the algorithmic templates ensure that the required `gemm` functions are properly included in the generated C++ file.

In the second case, the matrix multiplication is performed using three loops, optimized with a vectorized innermost loop. If β is zero, then C is set to zero before the matrix multiplication. This pure C++ implementation does not rely on any BLAS library. Thus, it can be used on architectures not supported by the integrated BLAS libraries or for debugging purposes.

Using an include statement in the main macro file, I then defined a proper macro from the subtemplate:

```
1 {% macro matmul(key, A, B, C, A_shift, B_shift, C_shift) %}
2 {% include 'subtemplates/matmul.template' %}
3 {% endmacro %}
```

The resulting macro can be used in algorithmic templates importing the macros, for example as is done in the template example in Listing 4.10:

```
1 {{ m.matmul('flux_x', 'flux', 'negativeDudxT_by_dx', 'lQi_next', '0',
      ↪  '0', idx(0,yz,0,0)) | indent(6) }}
```

In the macro call, other macros can be used, such as the index macros. The `indent` filter used when calling the macro is optional with no algorithmic impact, it is a default Jinja2 filter used to properly indent the macro rendered code with the rest of the template to increase the readability of the generated C++ source code.

4.4.5 Discussion of the template split

By isolating the architecture-aware optimization logic in optimization macros, I was able to separate, as much as possible, the numerical scheme implemented by algorithm experts from the low-level optimizations done by the optimization experts. Algorithm experts still need to use the provided macro and some optimized template variables, such as the padded dimensions, in the algorithmic templates. However, the logic behind these is properly encapsulated in the optimization macros for the former and in the Controller or the Model for the latter, where the optimization experts can work independently of the template using them.

Thanks to this split, the algorithmic template are architecture-oblivious and algorithm-aware, whereas the optimization macros and optimized variables are archtitecture-aware and algorithm-oblivious. The re-engineered Kernel Generator now also fulfills its last design goal of separation of concerns.

5 Use case: implementation of ExaSeis

In this chapter, we follow the implementation of ExaSeis, a simulation of waves propagation in heterogeneous isotropic and anisotropic elastic solids using ExaHyPE, taking the point of view of each of our three user roles in Section 5.1, 5.2 and 5.3. We then discuss the presented use case and evaluate the fulfillment of ExaHyPE's design requirements and the benefits of the user-role driven design in Section 5.4.

ExaSeis was developed by an interdisciplinary team composed of Alice-Agnes Gabriel[1], Kenneth Duru[1], Leonhard Rannabauer[2] et al., that took the roles of application and algorithm experts, with my support on the ExaHyPE side taking the roles of algorithm and optimization expert.

This application was developed incrementally, and in the end was used to simulate a regional wave propagation scenario in a geologically constrained 3D model including the topography of Mount Zugspitze, Germany [15], making it relevant to the field of computational seismology.

5.1 Application experts

In this section, the development of the application from the point of view of the application experts is presented. This role was taken by the ExaSeis team as they are the domain experts.

First, they developed an initial solver for wave propagation in elastic solids by implementing the relevant PDE system as an ExaHyPE application. This initial solver was validated using a small scale 3D WholeSpace problem, whose analytical solution is known.

Then, a curvilinear mesh was used to simulate a complex free surface topography on ExaHyPE's regular Cartesian mesh and ExaHyPE's default RiemannSolver kernel was replaced by a new one more adapted to the application specificities. The improved solver was again validated and benchmarked, this time using a small scale 3D layer over a half-space (LOH1) setup [75].

In a third step, the application was optimized to use more advanced numerical schemes provided by the Kernel Generator. With my support as an optimization expert, the application's user functions were further optimized using vectorization, i.e. performing

[1]Department of Earth and Environmental Sciences, Ludwig-Maximilians-Universität München, Germany
[2]Technical University of Munich, Germany

multiple floating point operations in one SIMD instruction, as will be presented in the next chapter.

Finally, the application experts expanded the application with a Perfectly Matched Layer (PML) scheme to remove instabilities caused by the boundary conditions [17, 18]. The curvilinear mesh was then set up to encode the topograpy of Mount Zugspitze, Germany's highest peak at 2962m, on a 80x80 km surface area and simulate seismic waves on this domain. Using ExaHyPE parallelization, the application was scaled to a multi-node configuration and was performed on the SuperMUC-NG cluster at the Leibniz Supercomputing Centre to produce relevant results to computational seismology.

The development, mathematical background, and results of this application are detailled in [15].

5.1.1 Incremental implementation of an elastic wave solver

Wave propagation in elastic solids, as described by Duru et al. [15], is modeled by

$$\mathbf{P}^{-1}\frac{\partial \mathbf{Q}}{\partial t} = \nabla \cdot \mathbf{F}(\mathbf{Q}) + \sum_{\xi=x,y,z} \mathbf{B}_\xi (\nabla \mathbf{Q}) \tag{5.1}$$

With a symmetric positive definite matrix $\mathbf{P}$ encoding the material parameters of the underlying medium,

$$\mathbf{Q}(x,y,z,t) = (v_x, v_y, v_z, \sigma_{xx}, \sigma_{yy}, \sigma_{zz}, \sigma_{xy}, \sigma_{xz}, \sigma_{yz})^T \tag{5.2}$$

the unknown wave field of the velocities and stresses, and for $\xi \in \{x, y, z\}$,

$$\mathbf{F}_\xi(\mathbf{Q}) = \begin{pmatrix} e_{\xi x}\sigma_{xx} + e_{\xi y}\sigma_{xy} + e_{\xi z}\sigma_{xz} \\ e_{\xi x}\sigma_{xy} + e_{\xi y}\sigma_{yy} + e_{\xi z}\sigma_{yz} \\ e_{\xi x}\sigma_{xz} + e_{\xi y}\sigma_{yz} + e_{\xi z}\sigma_{zz} \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}, \tag{5.3}$$

and

$$\mathbf{B}_\xi\left(\nabla\mathbf{Q}\right) = \begin{pmatrix} 0 \\ 0 \\ 0 \\ e_{\xi x}\frac{\partial v_x}{\partial \xi} \\ e_{\xi y}\frac{\partial v_y}{\partial \xi} \\ e_{\xi z}\frac{\partial v_z}{\partial \xi} \\ e_{\xi y}\frac{\partial v_x}{\partial \xi} + e_{\xi x}\frac{\partial v_y}{\partial \xi} \\ e_{\xi z}\frac{\partial v_x}{\partial \xi} + e_{\xi x}\frac{\partial v_z}{\partial \xi} \\ e_{\xi z}\frac{\partial v_y}{\partial \xi} + e_{\xi y}\frac{\partial v_z}{\partial \xi} \end{pmatrix}. \tag{5.4}$$

The vectors $\mathbf{e}_\xi = (e_{\xi x}, e_{\xi y}, e_{\xi z})^T$ represent the 3D canonical vector basis on a Cartesian mesh or in curvilinear coordinates arbitrary nonzero vectors, with $|\mathbf{e}_\xi| > 0$.

Defining the application

The first step was to write a specification file describing the desired application. Comparing the PDE system(5.1) to ExaHyPE's canonical PDE system (2.1),

$$\mathbf{P}\frac{\partial}{\partial t}\mathbf{Q} + \nabla\cdot\mathbf{F}(\mathbf{Q}) + \sum_{i=1}^{d}\mathbf{B}_i(\mathbf{Q})\frac{\partial \mathbf{Q}}{\partial x_i} = \mathbf{S}(\mathbf{Q}) + \sum_{i=1}^{n_{\text{ps}}}\delta_i,$$

we can identify that a material parameter term, a flux term, and a non-conservative product are required. Furthermore a single moment-tensor point source is used to provide the initial impulsion of an earthquake at a given epicenter.

As seen in (5.2), nine variables are required to define $\mathbf{Q}$, the three velocities of the particle velocity vector $\mathbf{v}$ and six terms of the stress vector σ . Also 16 constant material parameters are needed to encode the medium properties and the curvilinear mesh [15].

The application experts wrote a specification file similar to the simplified one shown in Listing 5.1. At this point, multi-node parallelization and specific engine tuning are ignored. These would be added later simply by adding the relevant block in the specification file, following ExaHyPE's Guidebook [76] recommendations in accordance with the target hardware specifications.

The specification file was then passed to the Toolkit with

```
./Toolkit/toolkit.sh specfile.exahype
```

As specified, the toolkit generated a new directory `ElasticWave` in which, among other generated files, the `UserSolver` implementation skeleton code `ElasticWaveSolver.cpp` is found.

```
{
  "project_name": "Elastic",
  "architecture": "noarch",
  "paths": {
    "peano_kernel_path": "./Peano",
    "exahype_path": "./ExaHyPE",
    "output_directory": "./ElasticWave"
  },
  "shared_memory": {
    "cores": 8,
    "properties_file": "sharedmemory.properties",
    "autotuning_strategy": "dummy",
    "background_job_consumers": 8
  },
  "computational_domain": {
    "dimension": 3,
    "end_time": 10.0,
    "offset": [   0.0, -7.5, -7.5 ],
    "width":  [  15.0, 15.0, 15.0 ]
  },
  "solvers": [
    { "type": "ADER-DG",
      "name": "ElasticWaveSolver",
      "order": 7,
      "maximum_mesh_size": 1.0,
      "time_stepping": "globalfixed",
      "aderdg_kernel": {
        "nonlinear": false,
        "terms": ["flux", "ncp", "material_parameters",
            ↪ "point_sources"],
        "point_sources": 1,
        "implementation": "generic"
      },
      "point_sources": 1,
      "variables": 9,
      "material_parameters": 16,
      "plotters": []
    }]
}
```

Listing 5.1: Basic specification file for the elastic wave application.

Implementing the user solver

The user solver implementation file `ElasticWaveSolver.cpp` contained the placeholder functions for the desired PDE system terms as well as the initial conditions, boundary condition and the eigenvalues required by the Riemann solver.

Listing 5.2 shows the implementation of the flux function given in (5.3). The `jacobian` term multiplied with the `q_x` makes up the $\mathbf{e}_{\xi x}$ term from (5.3), and so on for the other

```
1  void Elastic::ElasticWaveSolver::flux(const double* const Q,double**
       ↪ const F) {
2    double sigma_xx=Q[3 + 0];
3    double sigma_yy=Q[3 + 1];
4    double sigma_zz=Q[3 + 2];
5    double sigma_xy=Q[3 + 3];
6    double sigma_xz=Q[3 + 4];
7    double sigma_yz=Q[3 + 5];
8
9    double jacobian=Q[13];
10
11   double q_x = Q[14 + 0];
12   double q_y = Q[14 + 1];
13   double q_z = Q[14 + 2];
14   double r_x = Q[14 + 3];
15   double r_y = Q[14 + 4];
16   double r_z = Q[14 + 5];
17   double s_x = Q[14 + 6];
18   double s_y = Q[14 + 7];
19   double s_z = Q[14 + 8];
20
21   std::fill_n(F[0], 9, 0.);
22   F[0][0] = -jacobian*(q_x*sigma_xx+q_y*sigma_xy+q_z*sigma_xz);
23   F[0][1] = -jacobian*(q_x*sigma_xy+q_y*sigma_yy+q_z*sigma_yz);
24   F[0][2] = -jacobian*(q_x*sigma_xz+q_y*sigma_yz+q_z*sigma_zz);
25
26   std::fill_n(F[1], 9, 0.);
27   F[1][0] = -jacobian*(r_x*sigma_xx+r_y*sigma_xy+r_z*sigma_xz);
28   F[1][1] = -jacobian*(r_x*sigma_xy+r_y*sigma_yy+r_z*sigma_yz);
29   F[1][2] = -jacobian*(r_x*sigma_xz+r_y*sigma_yz+r_z*sigma_zz);
30
31   std::fill_n(F[2], 9, 0.);
32   F[2][0] = -jacobian*(s_x*sigma_xx+s_y*sigma_xy+s_z*sigma_xz);
33   F[2][1] = -jacobian*(s_x*sigma_xy+s_y*sigma_yy+s_z*sigma_yz);
34   F[2][2] = -jacobian*(s_x*sigma_xz+s_y*sigma_yz+s_z*sigma_zz);
35 }
```

Listing 5.2: Implementation of the flux function (5.3) using the default user API.

terms $\mathbf{e}_\xi$. Note that by convention, as specified in the Guidebook, the negative flux ($-\mathbf{F}(\mathbf{Q})$) needs to be computed.

The other functions were implemented similarly in their provided placeholders. Each method implemented here is purely application related and no engine or architecture specific considerations were required, only knowledge of the PDE system and the desired initial and boundary conditions.

Compiling and running the application

The application was compiled with the generated Makefile using the `make` command. The compiler produced an executable `ExaHyPE-Elastic` that was run with the specification file as input using

```
./ExaHyPE-Elastic specfile.exahype
```

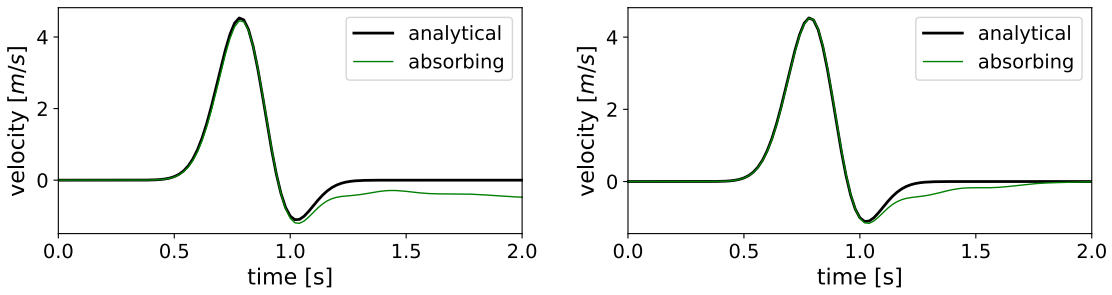
Results analysis

By setting the initial conditions to define a WholeSpace problem benchmark on a regular Cartesian mesh, the application in its current state was validated.

```
1 "plotters": [{
2   "type": "probe::ascii",
3   "name": "ProbeWriter1",
4   "time": 0.0,
5   "repeat": 0.005,
6   "output": "./seismogram_1km",
7   "variables": 9,
8   "select": {"x": 16.0,"y": 15.0,"z": 15.0}
9 }]
```

Listing 5.3: Example of a plotter block for the specification file in Listing 5.1.

To obtain outputs to analyze, plotters were added to the specification file in Listing 5.1 using the JSON object shown in Listing 5.3. The code excerpt shows only one of such plotters, others were defined at different positions using the `select` field. These plotters produce seismograms that can be compared to reference values known from the analytical solution to validate the application.



(a) Order 3 **(b)** Order 5

Figure 5.1: Seismograms produced by ExaSeis using the WholeSpace initial condition, using polynomials of order 3 and 5. Adapted from [15].

Running the Toolkit again to generate the plotters automatically and recompiling the application, the outputs shown in Figure 5.1 were obtained. The resulting seismograms closely follow the reference values from the analytical solution, validating the application implementation and in particular its PDE formulation to model wave propagation.

5.1.2 Physically motivated Riemann solver

By default ExaHyPE uses a Rusanov flux [77] (also called local Lax-Friedrichs flux) implementation in its RiemannSolver kernel, for its simplicity and robustness. However, the accuracy and stability of the DG method is highly dependent on the choice of Riemann solver scheme [78, 79, 80]. Other implementations, such as the Godunov flux [81] or Engquist-Osher flux [82], have also been used in other solvers. As the Rusanov flux scheme proved to be insufficient for ExaSeis, the ExaSeis team introduced a new physically motivated scheme for the Riemann solver [83].

For simplicity, this scheme was implemented as an application specific scheme by overriding the default binding to ExaHyPE's kernel implementation in the `AbstractSolver` glue code with the implementation of the new scheme directly wrapped in the `UserSolver`. Note that it would also have been possible to add this new scheme to ExaHyPE itself and offer it as a variant of the default RiemannSolver kernel, as will be discussed when introducing a SpaceTimePredictor kernel variant in Section 5.2.2. However, as the physically motivated Riemann solver is tailored toward this specific application, the additional workload required would not have been justified.

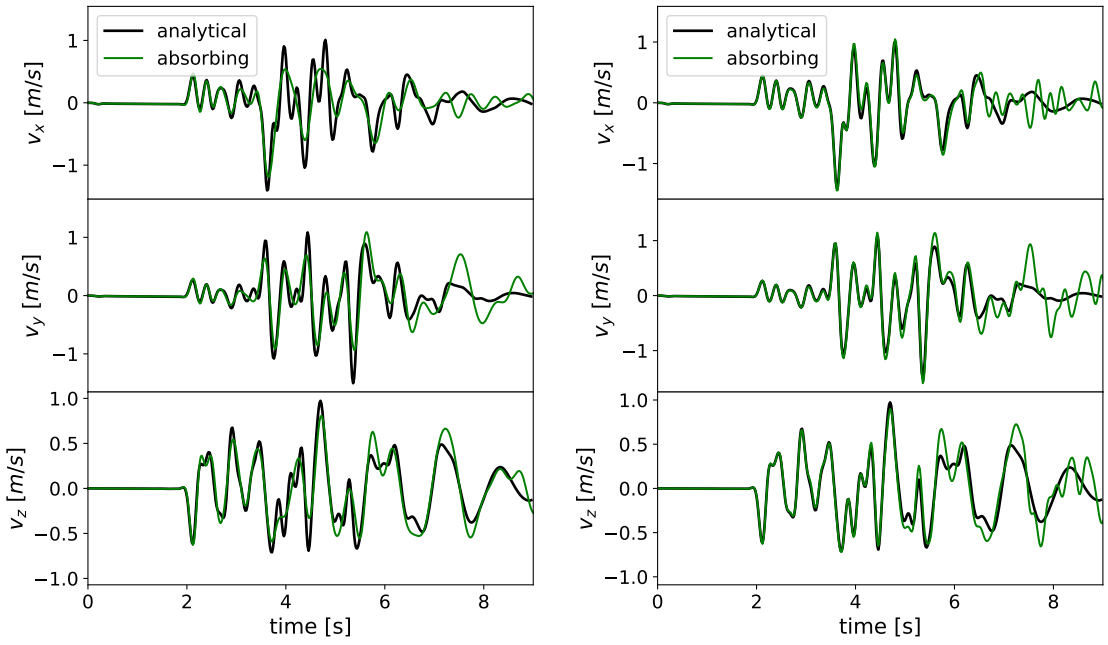
To test the new Riemann solver and the curvilinear mesh, the initial conditions were changed to a small scale three-dimensional layer over a half-space (LOH1) setup [75] that could not have been computed using the default Rusanov-based RiemannSolver kernel.

Figure 5.2 shows the resulting seismograms. We can see that it matches the analytical solution at first, better at higher order. However, some oscillations are present and increase with time, due to the imperfect absorbing boundary conditions used contaminating the solution [15].

5.1.3 Optimization of the application

With the `ElasticWave` solver implemented and validated, the application experts optimized the application by using the more performant variants of the numerical scheme implemented in the Kernel Generator's optimized kernels. These new variants required adapting the user functions to their specific APIs. The kernel variants will be presented in Chapter 8.

Assuming the role of an optimization expert, I supported the ExaSeis team when optimizing the application for more advanced kernel variants, with a low-level optimization of the user function using SIMD.



(a) Order 3 **(b)** Order 7

Figure 5.2: Seismograms produced by ExaSeis using the LOH1 initial condition, using polynomials of order 3 and 7. The absorbing boundary conditions introduce oscillations perturbing the results. Adapted from [15].

Basic optimized kernels

The first optimization step was simply to activate the generation of optimized kernels. Using the specification file from Listing 5.1 and targeting Intel's Skylake CPUs, line 3 was changed to "`"architecture": "skx"`" to specify the target architecture which is mandatory for optimized kernels, and line 26 to "`"implementation": "optimised"`"[3] to replace the default generic kernels with optimized generated ones in the ADER-DG scheme.

Running the Toolkit with this modified specification file automatically called the Kernel Generator. A new subdirectory `kernels` was created in the application directory and contained the optimized kernels. Furthermore, the glue code generated by the Toolkit was automatically refreshed to bind these new kernels instead of the generic ones. As the basic optimized kernels use the same API toward the `UserSolver` as the generic ones, the implemented application required no changes.

[3]ExaHyPE's specification file uses British English

SplitCK numerical scheme

The Kernel Generator offers more advanced numerical schemes. In the case of a linear application, the *SplitCK* scheme aims at reducing the overall memory footprint of the kernel as it causes performance loss at high order. To do so, the flux and ncp user functions in the `UserSolver` are each split into three functions, one for each spatial dimension. Also, the constant material parameters are now stored in a separate tensor `P` instead of after the variables in the tensor `Q`.

```
1  void  Elastic::ElasticWaveSolver::flux_x(const double* const Q,const
       ↪ double* const P,double* const F) {
2    double sigma_xx=Q[3 + 0];
3    double sigma_yy=Q[3 + 1];
4    double sigma_zz=Q[3 + 2];
5    double sigma_xy=Q[3 + 3];
6    double sigma_xz=Q[3 + 4];
7    double sigma_yz=Q[3 + 5];
8
9    double jacobian= P[4];
10
11   double q_x = P[5 + 0];
12   double q_y = P[5 + 1];
13   double q_z = P[5 + 2];
14
15   std::fill_n(F, 9 , 0);
16   F[0] = -jacobian*(q_x*sigma_xx+q_y*sigma_xy+q_z*sigma_xz);
17   F[1] = -jacobian*(q_x*sigma_xy+q_y*sigma_yy+q_z*sigma_yz);
18   F[2] = -jacobian*(q_x*sigma_xz+q_y*sigma_yz+q_z*sigma_zz);
19 }
```

Listing 5.4: Implementation of the flux function (5.3) in the x-direction using the SplitCK user API, by splitting the implementation shown in Listing 5.2.

Thus, to use this scheme, the application experts implemented the new flux user functions by splitting the existing `flux`, shown in Listing 5.2, into one for each dimension, and adapted the signature, as shown in Listing 5.4 for the first spatial dimension.

Likewise, user functions for the other two dimensions were implemented and the non-conservative product function was split into the three required by the scheme API. Other user functions did not require any changes.

Finally the specification file in Listing 5.1 was modified to set the proper boolean flag enabling this advanced scheme by adding a line in the `"aderdg_kernel"` block of the `"solver"` block:

```
"space_time_predictor": {"split_ck": true},
```

Running the Toolkit with this modified specification file triggered the Kernel Generator to regenerate the optimized kernels, in particular the STP kernel was updated from the

LoG variant to the new SplitCK one. The application was compiled, and the resulting executable used the advanced numerical scheme that still produced the same simulation, but 2 to 5 times faster on Skylake.

SIMD optimized application

To further optimize the application, the user functions using scalar instructions need to be vectorized to benefit from the SIMD capabilies of the SuperMUC-NG's Skylake CPUs. Optimized kernels can be configured to provide the more performance-tuned user functions with inputs using a *Structure-of-Arrays* (SoA) data layout, making them easily vectorizable by computing multiple quadrature nodes at the same time with SIMD instructions instead of one per function call with scalar instructions.

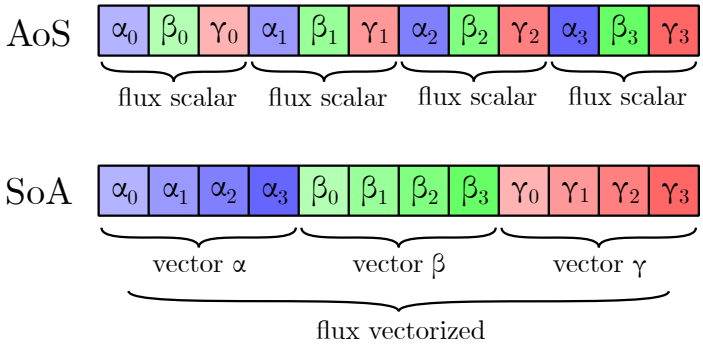


Figure 5.3: Using an SoA data layout, multiple nodes can be computed in one vectorized user function call (here flux), as it trivially extracts unit-stride vectors of each variable.

Figure 5.3 illustrates how an input vector is stored using the SoA data layout to contain multiple nodes at the same time, thus requiring only one call to a vectorized flux function instead of four to the scalar implementation to process the same amount of data. The parameters `VectLength` and `VectStride` are provided as `constexpr int` in the glue code, the former being the number of nodes stored in the input and output arrays and the latter the actual length of the fastest dimension used as array stride, that can be longer due to padding to optimize the SIMD instructions, in Figure 5.3 both are equal to 4. The n^{th} variable of the i^{th} node in an SoA array `A` is stored at `A[n*VectStride+i]`, e.g. β_2 is at index 6.

I used the split flux and split non-conservative product user function implementations from the previous step as a starting point for the new vectorized split versions. Listing 5.5 shows the vectorized `flux_x_vect` method obtained from the scalar method in Listing 5.4. In simple cases, such as this one, the vectorization of the user function is performed in 4 steps:

1. Changing the constants to arrays using pointers and multiplying the position by `VectStride`.

```
1  void  Elastic::ElasticWaveSolver::flux_x_vect(const double* const Q,
       ↪ const double* const P,double* const F) {
2    const double * const sigma_xx = Q + (6+0)*VectStride;
3    const double * const sigma_yy = Q + (6+1)*VectStride;
4    const double * const sigma_zz = Q + (6+2)*VectStride;
5    const double * const sigma_xy = Q + (6+3)*VectStride;
6    const double * const sigma_xz = Q + (6+4)*VectStride;
7    const double * const sigma_yz = Q + (6+5)*VectStride;
8    const double * const jacobian = P + 4    *VectStride;
9    const double * const q_x      = P + (5+0)*VectStride;
10   const double * const q_y      = P + (5+1)*VectStride;
11   const double * const q_z      = P + (5+2)*VectStride;
12
13   std::fill_n(F, 9*VectStride, 0.0);
14
15   #pragma omp simd
16   #pragma vector aligned
17   for(int i = 0 ; i < VectStride ; i++){
18     F[0*VectStride+i] = -jacobian[i]*(q_x[i]*sigma_xx[i]+q_y[i]*
         ↪ sigma_xy[i]+q_z[i]*sigma_xz[i]);
19     F[1*VectStride+i] = -jacobian[i]*(q_x[i]*sigma_xy[i]+q_y[i]*
         ↪ sigma_yy[i]+q_z[i]*sigma_yz[i]);
20     F[2*VectStride+i] = -jacobian[i]*(q_x[i]*sigma_xz[i]+q_y[i]*
         ↪ sigma_yz[i]+q_z[i]*sigma_zz[i]);
21   }
22 }
```

Listing 5.5: Implementation of the vectorized flux function in the x-direction using the vectorized SplitCK user API, by vectorizing the scalar implementation shown in Listing 5.4. The arrays `Q`, `P`, and `F` are implicitly matrices in the SoA data layout.

2. Adding a for loop around the whole implementation. The length of the loop should be `VectLength` if unsafe operations such as divisions are performed. Otherwise, if computing on random values does not risk causing a runtime error, then the padding can be included for better performance, as will be discussed in Section 6.2.2, by using `VectStride` instead.

3. Using an OpenMP pragma to instruct the compiler to use its auto-vectorization capabilities[4].

4. Changing the array indexes inside the loop to the SoA data layout by adding "`* VectStride+i`" and the constant to the new array using `i` as index.

With this the compiler used SIMD instructions instead of scalar ones, which I verified by looking at its optimization report:

[4]For brevity I used the Intel Compiler's `#pragma vector aligned` to specify that the all arrays are aligned instead of doing it in the OpenMP pragma where I would have to write each array.

```
1  Begin optimization report for: Elastic::ElasticWaveSolver::flux_x_vect(Elastic
       ::ElasticWaveSolver *, const double *, const double *, double *)
2  [...]
3  LOOP BEGIN at /[...]/ElasticWaveSolver.cpp(737,3)
4    remark #15388: vectorization support: reference F[i] has aligned access[...]
5    remark #15388: vectorization support: reference P[i+32] has aligned acc[...]
6    remark #15388: vectorization support: reference P[i+40] has aligned acc[...]
7    remark #15388: vectorization support: reference Q[i+48] has aligned acc[...]
8    remark #15388: vectorization support: reference P[i+48] has aligned acc[...]
9    remark #15388: vectorization support: reference Q[i+72] has aligned acc[...]
10   remark #15388: vectorization support: reference P[i+56] has aligned acc[...]
11   remark #15388: vectorization support: reference Q[i+80] has aligned acc[...]
12   remark #15388: vectorization support: reference F[i+8] has aligned acce[...]
13   remark #15388: vectorization support: reference P[i+32] has aligned acc[...]
14   remark #15388: vectorization support: reference P[i+40] has aligned acc[...]
15   remark #15388: vectorization support: reference Q[i+72] has aligned acc[...]
16   remark #15388: vectorization support: reference P[i+48] has aligned acc[...]
17   remark #15388: vectorization support: reference Q[i+56] has aligned acc[...]
18   remark #15388: vectorization support: reference P[i+56] has aligned acc[...]
19   remark #15388: vectorization support: reference Q[i+88] has aligned acc[...]
20   remark #15388: vectorization support: reference F[i+16] has aligned acc[...]
21   remark #15388: vectorization support: reference P[i+32] has aligned acc[...]
22   remark #15388: vectorization support: reference P[i+40] has aligned acc[...]
23   remark #15388: vectorization support: reference Q[i+80] has aligned acc[...]
24   remark #15388: vectorization support: reference P[i+48] has aligned acc[...]
25   remark #15388: vectorization support: reference Q[i+88] has aligned acc[...]
26   remark #15388: vectorization support: reference P[i+56] has aligned acc[...]
27   remark #15388: vectorization support: reference Q[i+64] has aligned acc[...]
28   remark #15305: vectorization support: vector length 8
29   remark #15427: loop was completely unrolled
30   remark #15301: SIMD LOOP WAS VECTORIZED
31   remark #15448: unmasked aligned unit stride loads: 21
32   remark #15449: unmasked aligned unit stride stores: 3
33   remark #15475: --- begin vector cost summary ---
34   remark #15476: scalar cost: 71
35   remark #15477: vector cost: 6.000
36   remark #15478: estimated potential speedup: 11.830
37   remark #15488: --- end vector cost summary ---
38 LOOP END
```

The loop was properly vectorized using aligned accesses and unrolled as the compiler recognized only one iteration is necessary in this case, with `VectStride` being 8. The compiler cost estimation for the speedup is, however, optimistic as it ignores effects, that will be discussed in Section 6.1.3.

Finally with the methods implemented, the specification file was modified to use them by setting up the appropriate optimization flag, which was done by changing

```
"space_time_predictor": {"split_ck": true},
```

to

```
"space_time_predictor": {"split_ck": true, "vectorise_terms": true},
```

This flag causes the generated SpaceTimePredictor kernel to use a hybrid data layout to resolve the data layout conflict caused by its optimization requirements conflicting with the SoA data layout required to vectorize the user functions, as will be discussed

in detail in Chapter 8. An alternative data layout is also available if the additional flag `"AoSoA2_layout": true` is set. It uses the same `UserSolver` API so no further modifications are required to use it, the `VectLength` and `VectStride` parameters automatically take larger values as this data layout aims at minimizing the padding size used by the kernel by making bigger SoA chunks, at the cost of some kernel operations' performance. The better data layout of the two depends on the application and the order at which it is run. Hence, both were tested on the benchmark setup.

Performance results

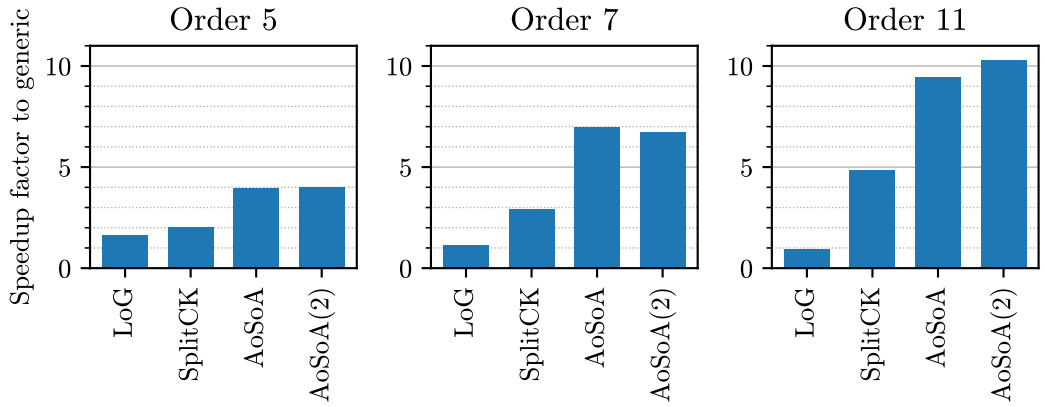


Figure 5.4: Speedup factors achieved relative to the generic kernel baseline, when running the LOH1 application with increasing optimizations at orders 5, 7 and 11, on SuperMUC-NG.

Figure 5.4 shows the speedup factor of the LOH1 benchmark application compared to the generic kernels, when using the optimized kernels on SuperMUC-NG (Skylake). We can see that each successive optimization reduced the total runtime of the application, with the optimal choice between the last two depending on the polynomial order used in the simulation. With the optimized kernels, we obtained a speedup of a factor 4.0 at order 5, 7.0 at order 7, and 10.3 at order 11.

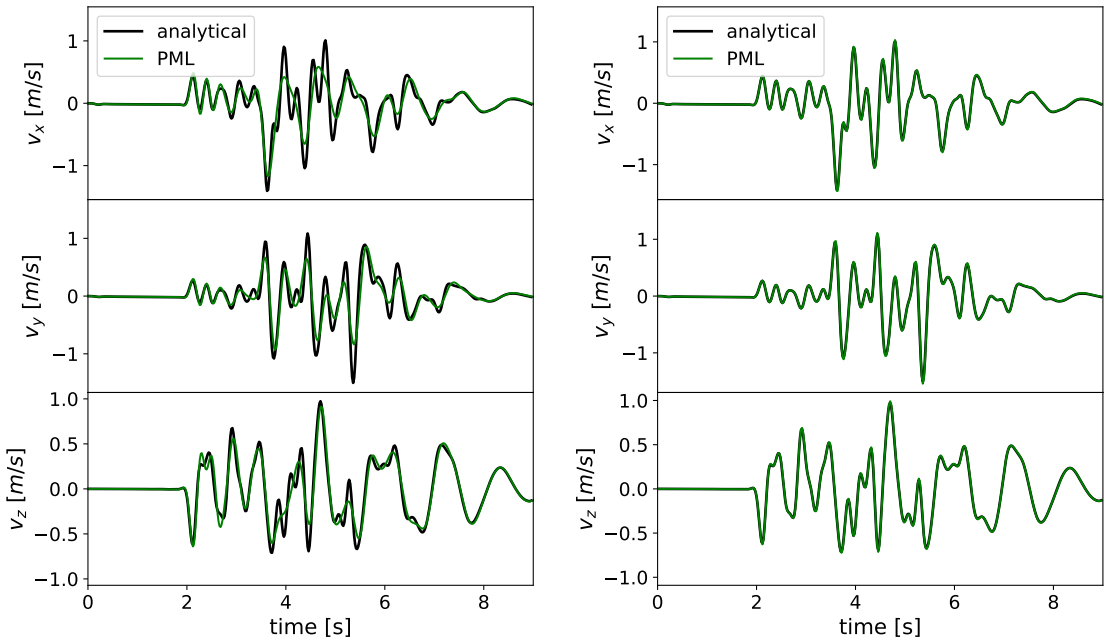
These successive speedups were achieved with only minor modifications to the application itself, and only the last one required some basic low-level optimization knowledge to introduce vectorization. Thus, from the point of view of an application expert, an application can easily be accelerated by selecting a more advanced numerical scheme in its specification file and adapting its user functions to the new kernels user API.

5.1.4 Simulation of wave progagation at Mount Zugspitze

In a final step, the application and its initial conditions were tuned with the goal to simulate seismic wave propagation on the topology of Mount Zugspitze.

Extending the application with perfectly matched layer

The LOH1 benchmark results in Figure 5.2 showed oscillation introduced by the imperfect boundary condition. Thus, before simulating Mount Zugspitze, the application experts solved this issue with the introduction in the application of a Perfectly Matched Layer (PML) scheme [17, 18]. In summary, this was achieved by introducing new variables and expanding the PDE system to take them into account, then using them in the Riemann solver when on a cell at the domain boundary to prevent artificial reflections. The full detail of this implementation can be found in [15].



(a) Order 3 **(b)** Order 7

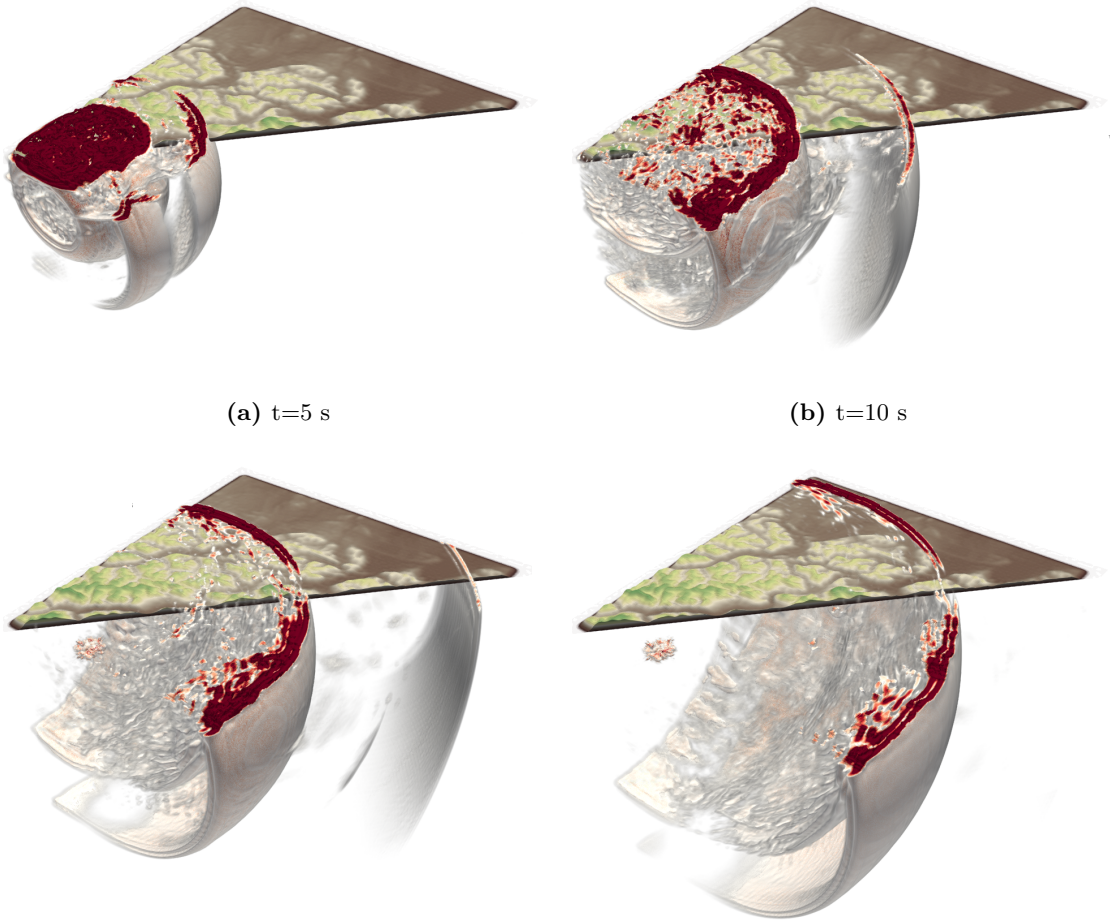
Figure 5.5: Seismogram produced by ExaSeis using the LOH1 initial condition with PML, using polynomials of order 3 and 7. PML removes the oscillation from the previous benchmark, and each result closely matches its analytical solution. Adapted from [15].

The new application with PML was then tested again using the same LOH1 setup as in Figure 5.2. This time, as can be seen on the seismograms in Figure 5.5, the application results closely matched the analytical solutions without any oscillation from the boundary conditions.

Large-scale simulation

The application initial conditions were then changed to load the topography of Mount Zugspitze. Furthermore, the application was set up to use Peano's parallelization by adding the `"shared_memory": {...}` and `"distributed_memory": {...}` blocks to the

specification file, regenerating the glue code with the Toolkit, and recompiling the application. The new executable was then run on 731 nodes of SuperMUC-NG.



(a) t=5 s **(b)** t=10 s

(c) t=15 s **(d)** t=20 s

Figure 5.6: 3D snapshots of the absolute velocity of the propagating seismic wavefield for the Zugspitze model at t=5 s, t=10 s, t=15 s and t=20 s simulated with ExaSeis. Adapted from [15].

Figure 5.6, adapted from [15], shows produced three-dimensional snapshots of the absolute velocity of the propagating seismic wavefield for the Zugspitze model. The results obtained with ExaHyPE were successfully validated by comparing them to the ones produced by a reference implementation using the finite difference code WaveQLab [14].

5.2 Algorithm experts

In this section, we discuss two modifications to the numerical scheme performed by algorithm experts to support the ExaSeis application. While these were made to support ExaSeis, they are now part of the Engine and can be used by any other linear application.

In the first one, I extended the canonical PDE supported at the time to add the moment-tensor point sources term now present in (2.1). This extension was done with the support of Kenneth Duru, from the ExaSeis team, to specify the API requirements and formulate the integration of the point sources' contributions to the prediction made in the SpaceTimePredictor (STP) kernel. In the second one, together with Leonhard Rannabauer, we introduced a new optimized variant of the STP kernel to fix the memory-related performance loss by reducing the overall memory footprint of the kernel. This new *SplitCK* kernel variant was used by the application experts when optimizing the application in Section 5.1.3. Its detailed motivation, effects on the overall performance, and the following optimizations made upon it, are discussed in Chapter 8, we here focus only on the workflow of its implementation.

5.2.1 Extension of the canonical PDE System with Point Sources

At the beginning of the project, the engine scheme could only solve canonical PDE using a flux, non conservative product, and a source terms:

$$\frac{\partial}{\partial t}\mathbf{Q} + \nabla \cdot \mathbf{F}(\mathbf{Q}) + \sum_{i=1}^{d} \mathbf{B}_i(\mathbf{Q}) \frac{\partial \mathbf{Q}}{\partial x_i} = \mathbf{S}(\mathbf{Q}) \tag{5.5}$$

To properly model the effects of seismic epicenters, we need one or multiple pointwise moment-tensor sources δ_i added to the right-hand side of the PDE (5.5). They constitute the $\sum_{i=1}^{n_{\text{ps}}} \delta_i$ point sources term of the current canonical PDE (2.1) where δ_i is a single moment-tensor point source and n_{ps} the total number of point sources. Therefore, at the request of Kenneth Duru from the ExaSeis team, and with his support for the integration of the new term to the numerical scheme with the development of a prototype, I added support for this new term to the engine at every level of optimization.

Defining the API

Before modifying the engine, together with the application expert, we defined the user API.

First, each point source has fixed coordinates that are set during the initialization of the solver. Thus, the glue code needs to set up a new member variable for the `UserSolver` to inherit:

```
1 double pointSourceLocation[NumberOfPointSources][DIMENSIONS];
```

A known `static constexpr int NumberOfPointSources` is required for the compiler to correctly allocate the `pointSourceLocation` array. Hence, the number of point sources needs to be known before the code generation and is defined in the specification file. The array can then be filled with each point source's coordinates by the application expert in the `init` method of the `UserSolver`. Using these locations, the solver is then able to automatically identify if a given cell contains some point sources. It uses this information to avoid unnecessary computations, as a point source only contributes directly to the cell it is located in.

The actual force tensor of a given point source is computed by a new user function:

```
1  void pointSource(const double t, double* const forceVector, int n);
```

This function is called on each node and point sources of the cell containing point sources and its arguments are:

- the time `t` in the simulation,
- the output vector to be written upon `forceVector`,
- the index `n` of the point source.

The time of the simulation is given as input to be able to define a point source term evolving in time, such as the one used in the Zugspitze simulation [15]:

```
1  double f =  M0*t/(t0*t0)*std::exp(-t/t0);
```

With the API defined, we directly modified the generated code of a test application to produce a working prototype validating our API and the modifications to the numerical scheme.

New Specification File PDE Term Option

With a prototype validating our approach, I implemented the API into the engine's workflow by modifying the specification file's JSON schema, described in Section 4.3.2, to include a new option shown in Listing 5.6. First, I added an option to specify the requirement of point source terms in the solved PDE system. As this was here an option for a PDE term, a `"point_sources"` parameter is added to the relevant list of allowed optional parameters. Then, as any number of point sources term may be requested, I defined a new variable specifying the number of point sources required similarly to how the number of variables is specified. Both options are required, the first one speficies that the application uses a point source term and the second one sets up a `numberOfPointSources` constant in the glue code.

```
"terms" : {
  "type" : "array",
  "title" : "PDE terms to generate code for",
  "items" : {
    "type" : "string",
    "enum" : ["flux", "source", "ncp", "point_sources"]
  }
},
```

```
"point_sources" : {
  "type" : "integer",
  "scope" : "compile-time",
  "default" : 0
},
```

Listing 5.6: New term flag `point_sources` and its associated variable added to the Toolkit's JSON Schema to define point sources in the specification file grammar.

Processing the new Option

The new "point_sources" PDE term option is processed in the Toolkit by its Controller with other term flags and added to the context for the models:

```
def buildKernelTermsContext(self,terms):
    context = {}
    context["usePointSources"] = "point_sources" in terms
    #[...]
    return context
```

Likewise, the number of point sources is processed with other quantities, such as the number of variables, and also added to the context.

To pass this new option from the Toolkit down to the Kernel Generator, I modified the `KernelGeneratorModel` to read the new flag from the provided context and add it to the API's input:

```
KGContext = {
    #[...]
    "usePointSources" : solverContext["numberOfPointSources"] if
        ↪ solverContext["numberOfPointSources"] > 0 else -1,
    #[...]
}
```

There it is defined as an optional integer parameter with the default value `-1` used to specify that no point source is used, as only positive values are valid. Then, I modi-

fied the Kernel Generator API to define this new term by adding a new tuple to the `ArgumentParser` definition, as seen in Listing 4.6:

```
aderdgArgs = [
    #[...]
    (("usePointSources", ArgType.OptionalInt , "enable
        ↪ numberOfPointSources point sources", -1, "
        ↪ numberOfPointSources"),
    #[...]
]
```

The tuple is then processed by the Kernel Generator's Controller and I added the translation into the contexts for the models and views in the same way as for the Toolkit.

Expanding the Model and View

Going back to the prototype and with all the relevant flags set up in the context of the code generation utilities, adapting the numerical scheme was done in three steps.

First, as expected from the initial design, I introduced a new kernel to determine the point sources contained in the current cell. As a simulation contains a very large number of cells and only a couple of point sources at most, it can be assumed that in most cases a cell does not contain any point source and optimize for this case. Thus, this kernel tests all point source's locations and only initializes a vector with the indexes of the concerned point sources if required, otherwise it returns `nullptr`.

```
{% if usePointSources %}
  // Add average PS to zero time derivative and lQh
  for (int t = 0; t < {{nDof}}; t++) {
    #pragma omp simd aligned(lQi,lQhi,PSi,weights1:ALIGNMENT)
    for (int it = 0; it < {{(nDof**nDim)*nVarPad}}; it++) {
      lQi [it] += dt*weights1[t]*PSi[{{idxPSi(t,0,0,0,it)}}];
      lQhi[it] += dt*weights1[t]*PSi[{{idxPSi(t,0,0,0,it)}}];
    }
  }
  //Initialize PS derivation operator
  double derPS[{{nDofPad}}] __attribute__((aligned(ALIGNMENT)));
  std::copy_n(FLCoeff, {{nDofPad}}, derPS);
{% endif %}
```

Listing 5.7: Contributions of the point source to the initial state tensor before the Cauchy-Kowalewsky procedure. Template branching ensures this code is only rendered when point sources are used.

Then, the linear STP kernel is the one that processes the eventual point sources. Similarly to what is done for other terms, I isolated point source operations inside template branching using the associated context parameter, for example in the code excerpt in

Listing 5.7, when adding the point sources contribution to the initial state tensor before the Cauchy-Kowalewsky procedure.

The branching ensures that the kernel can be generated without any point source computations. This is here particularly relevant as even if point sources are used, most cells do not have any of them to process. Thus to optimize the application, if point sources are defined in the specification file, I modified the model to then render the template twice with a slightly modified context, producing one version of the kernel with point source computation and one without it.

```
1  {% if usePointSources %}
2    std::vector<int>* pointSources = {{optNS}}::getPointSources([..]);
3    if(pointSources != nullptr) {
4      // compute pointsource
5      {{optNS}}::fSTPVI([...], pointSources);
6    } else {
7      // no point source, skip term
8      {{optNS}}::fSTPVI_WithoutPS([...], pointSources);
9    }
10 {% else %}
11   {{optNS}}::fSTPVI([...], nullptr);
12 {% endif %}
```

Listing 5.8: Binding of the kernels in the `AbstractSolver` glue code generated by the Toolkit. When point sources are used, two versions of the kernels are rendered, and the Toolkit only calls the one with a point source term if the cells contains some.

Finally, I modified the glue code in the Toolkit to properly bind everything together if point sources are used. The proper arrays, constant and skeleton code are produced, and the function binding the STP kernel is expanded as can be seen in the simplifed code excerpt in Listing 5.8. When point sources are used, the function first calls the new kernel to determine if point sources are in the current cell. Then if the returned array is not `nullptr`, it calls the version of the STP kernel with point source computations. Otherwise, it calls the one generated as if no point term is used, as there are none for this specific cell. If no point source term is requested by the specification file, the default binding is rendered, calling directly an STP kernel generated without a point source term.

Some other minor modifications to other templates were also necessary to ensure that all the proper headers are defined and that the required temporary memory is allocated. Furthermore, the generation of the `UserSolver` skeleton code for the application expert starting a new application was modified to implement the correct paceholder user functions if point sources are used.

Evaluation

Thanks to the modular architecture of the engine and the MVC architecture of the code generation utilities, extending the numerical scheme to add a new optional term to the canonical PDE was performed efficiently, following a clear workflow to modify the specification file grammar and pass down the information to the relevant parts of the code generation utilities. The code generation itself was easily adapted with the new capability made fully optional by the template logic. The use of glue code to bind the new components together ensured that the user API stays simple from the point of view of application experts, that can now request this term and process it similarly to the already existing ones.

The canonical PDE was further expanded at a later point for another application on cloud simulations by Lukas Krenz [21]. This time the existing flux term was modified into a viscous flux term $\mathbf{F}(\mathbf{Q}, \nabla\mathbf{Q})$, that depends on $\nabla\mathbf{Q}$, required for the simulation of the compressible Navier-Stokes equations [22]. This similar use case was performed following the same workflow and was presented in a previous publication [23].

5.2.2 New SpaceTimePredictor Algorithm

Benchmarks of the linear PDE solver at high polynomial orders showed significant loss of performance due to memory stalls inside the STP kernel. This use case presents the implementation of a new variant of the linear STP aiming to reduce the memory footprint used by the kernel. The reasoning behind this new variant, its mathematical background as well as its full performance evaluation are discussed in Section 8.3.

To develop this new kernel variant, integrate it into the code generation utilities and optimize it, Leohnard Rannabauer and I used an iterative approach.

Prototyping the new Algorithm

One key idea of this new kernel is to compute each spatial dimension in isolation to save the need to store intermediate results of all two or three dimensions at the same time. Doing so implies modifying the application API, as the default one processes all dimensions at once in the flux and non-conservative product user functions.

Therefore, we started with the small-scale WholeSpace problem benchmark developed in the previous section to simplify testing and debugging on a single core. To further simplify the development process, the application was simplified and formulated at first using only a pointsource and nonconservative product term. The specification of the application was then used to generate optimized kernels for it, with the BLAS library disabled to have a generated STP in pure C++ as basis to work on.

We modified the application's non-conservative product method to aggregate three new independent methods, each method computing only one of the spatial dimensions. This allowed us to ensure that the new formulation in the application was correctly working. Then, we modified the STP generated kernel to use the new algorithm, developed by

Leonhard Rannabauer, with the new one-dimensional methods. By keeping a copy of the default kernel and performing each modification step by step, we were able to easily compare intermediate values to the reference implementation by swapping the kernel binding in the glue code. We iterated upon the prototype to incrementally add new optimizations to the algorithm, as tests revealed new bottlenecks and possible areas of improvement.

Inclusion in the Kernel Generator

Once the prototype was finished and validated, I incorporated the new kernel prototype directly into the Kernel Generator. The prototype source code was used without any modification as the first iteration of a new template, since a template can also be explicit code without any template logic.

```
1  "space_time_predictor" : {
2    "type" : "object",
3    "title" : "Turn certain features on/off for the space time
       ↪ predictor",
4    "properties" : {
5      "split_ck" : {
6        "type" : "boolean",
7        "title" : "Use a split Cauchy-Kowalewsky for the space time
           ↪ predictor (linear only)",
8        "default" : false
9      },
10   #[...]
11   }
12 },
```

Listing 5.9: New term flag `split_ck` added to the Toolkit's JSON Schema to use kernel variants implementing this algorithm.

Similarly to the previous use case, I added a new boolean flag `split_ck` to the specification file grammar, this time as a new optional member of the `space_time_predictor` object, as shown in Listing 5.9. The MVC data flow was then adapted as in the previous use case to pass down this flag to the relevant models in the Kernel Generator as a boolean context parameter `useSplitCK`. In the Kernel Generator, the model responsible for generating the STP kernel was modified to chose the new template as the view to render if the boolean flag is set.

Template generalization and optimization

Finally, I generalized the template using template variables and statements, so that it can be used with other settings or by other applications. This step can be seen as reversing the rendering operation from the template engine, and thus the work in progress can

easily be tested at any stage by rendering the template to check that it produces the desired output.

```
1  for (int yz = 0; yz < 36; yz++) {
2    // reset gradQ
3    std::memset(gradQ+yz*72, 0, 72*sizeof(double));
4    //gradQ in x
5    for (int x1 = 0; x1 < 6; x1++) {
6      for (int x2 = 0; x2 < 6; x2++) {
7        for (int n = 0; n < 12; n++) {
8          gradQ[yz*72+x1*12+n] += lQi[yz*72+x2*12+n] * dudxT_by_dx[x1
               ↪ *8+x2];
9        }
10     }
11   }
12 }
```

Listing 5.10: Generalization of a template: prototype's code excerpt to be generalized.

```
1  for (int yz = 0; yz < {{nDof*nDof3D}}; yz++) {
2    // reset gradQ
3    std::memset(gradQ+{{idx(0,yz,0,0)}}, 0, {{nDof*nDof3D}}*sizeof(
       ↪ double));
4    //gradQ in x
5    for (int n = 0; n < {{nDof}}; n++) {
6      for (int k = 0; k < {{nDof}}; k++) {
7        for (int m = 0; m < {{nVarPad}}; m++) {
8          gradQ[{{idx(0,yz,n,m)}}] += lQi[{{idx(0,yz,k,m)}}] *
               ↪ dudxT_by_dx[n*{{nDofPad}}+k];
9        }
10     }
11   }
12 }
```

Listing 5.11: Generalization of a template: first generalization using templates variables to generalize the hard-coded constants.

```
1  for (int yz = 0; yz < {{nDof*nDof3D}}; yz++) {
2    {{ m.matmul('gradQ_x_sck', 'lQi', 'dudxT_by_dx', 'gradQ', idx(0,yz
       ↪ ,0,0), '0', idx(0,yz,0,0)) | indent(6) }}{##}
3  }
```

Listing 5.12: Generalization of a template: final template with the matrix multiplication abstracted by a `matmul` macro call.

For example, let us consider the code excerpts in Listings 5.10, 5.11, and 5.12.

The first excerpt is from the prototype. Knowing that it used 9 variables, padded to 12, and was computed at with `nDof=6`, padded to 8, it is generalized to the second excerpt by using template variables and a newly introduced index macro. At this point the code excerpt is generalized to any order and number of variables. Furthermore, the use of `nDof3D` and the index macro allows it to be generalized for two dimensions as then the outer loop would correspond to a `y` loop with `nDof3D = 1`.

From there, we recognize a matrix multiplication, $C = A \cdot B$ ($\alpha = 1$ and $\beta = 0$), with the matrix dimensions given by the loop boundaries and matrix leading dimensions by the loop index computation ignoring the index offset. The configuration of this matrix multiplication is added to the one produced by the model under the id `gradQ_x_sck`, and the template is modified to use it with a `matmul` macro, resulting in the last generalized template. Now that a `matmul` macro is used, a BLAS library is automatically used to optimize and perform the matrix multiplication instead of the C++ for loops if the Kernel Generator is configured back to using one.

During this step, I transformed the prototype template into an algorithmic template, as described in Section 4.4, with the optimization macros and template variables providing the architecture-aware optimizations. Thus, this new kernel variant was optimized toward all the supported architecture without needing any optimization knowledge.

Further Kernel Variants and Evaluation

At a later stage, we expanded the template to include other terms from the canonical PDE (2.1), such as a flux term or a material parameter matrix. For the simplest cases, such as the material parameter matrix, we directly modified the template to include the necessary logic. It was then rendered and tested on a reference application for validation. For the more complex ones, we used an iterative process again, where the template was rendered and the resulting file modified to be tested on an application. The modification were then added to the template directly and then generalized. In both cases, we added templating branching logic to avoid regression by ensuring that the same code as before was generated in the absence of the new terms in the specification file.

Like in the previous use case, we also modified the generation of the user solver header and skeleton implementation to ensure that the right `UserSolver` API was presented to an application expert using this kernel variant.

As will be discussed in Chapter 8, benchmarks showed that this new kernel variant greatly improved performance at high order, enabling me to introduce further variants of this scheme to solve its new bottlenecks. The high performance was expected early in de development process, as it started with a prototype to validate and tune our initial intuitions before any modification to the Kernel Generator. Furthermore, this iterative process greatly reduced the time spend debugging, as validation could be performed at each step and even substeps by comparing the work in progress output to a previously validated reference. Finally, all available low-level optimizations are obtained without

effort when generalizing the template and applying the optimization macros, as desired from the Kernel Generator third design goal introduced in Section 4.1.1.

5.3 Optimization experts

In this Section, two architecture-targeted optimizations I implemented are presented. Both were developed assuming the role of an optimization expert and required no understanding of the underlying numerical scheme taking place.

The first one is the initial idea tried out to solve the data layout conflict emerging from conflicting optimization requirements between the kernels and the user functions, using on-the-fly transposition. This optimization was initially necessary to the application optimization using vectorization performed in Section 5.1.3. The second one is the expansion of the Kernel Generator to rely upon the BLAS library Eigen [41] for the matrix multiplications if desired, instead of only having the choice between LIBXSMM and a direct C++ implementation.

5.3.1 Vectorized PDE with on-the-fly transpositions

Concisely, to fully exploit the capabilities of modern CPUs, the code needs to be vectorized, i.e. use SIMD vector instructions and vector registers instead of the standard scalar ones, as will be discussed in Chapter 6. This comes into conflict with having pointwise user functions processing only a single node, as to vectorize them multiple nodes would need to be processed per function call. Furthermore, due to other optimization concerns, the kernels use an Array-of-Structures (AoS) data layout, which is the opposite of the Structure-of-Arrays (SoA) one required to vectorize user functions efficiently, as was shown in Figure 5.3, where the variable dimension was the slowest one and the node index the fastest in the input matrices.

In a first attempt, I solved this conflict using on the fly transpositions to swap the data layout of the input and outputs of the user functions. This optimization is now deprecated and only supported as a legacy feature by some kernel variants, as this use case showed some significant limitations of this method and a better solution using a hybrid data layout, shown in Section 8.4, was later implemented to solve the conflict. I presented this use case in a previous publication [23] and it is here updated to its latest status before it became deprecated.

Creating a new macro

To be able to perform the transposition, I introduced the new optimization macro shown in Listing 5.13. This `transpose` macro transposes a matrix, thus swapping its layout between AoS and SoA. It uses pointers' offsets to be able to work in chunk instead of processing the whole matrix at once.

```
1 {% macro transpose(in, out, in_offset, out_offset, A, B) %}
2 {# Transpose base, input is a [A][B] matrix #}
3 for(int it_1=0; it_1<{{B}}; it_1++) {
4   #pragma omp simd aligned({{out}},{{in}}:ALIGNMENT)
5   for(int it_2=0; it_2<{{A}}; it_2++) {
6     {{out}}[it_1*{{A}}+it_2+{{out_offset}}] = {{in}}[it_2*{{B}}+it_1
        ↪ +{{in_offset}}];
7   }
8 }
9 {% endmacro %}
```

Listing 5.13: `transpose` macro to transpose a matrix.

I also introduced a more complex `transpose_rest` macro to be able to process incomplete submatrices from the last remaining chunk, where not enough nodes may be left to produce a full transposed matrix with the right dimensions required by the optimization. In this case, the first node is copied to fill the missing entries and ensure that no numerical error occurs when appliying the user function on this padding, as a zero-padding may result for example in divisions by zero.

For the tests of this use case, only a standard C++ implementation of the transposition was implemented. However, should I have kept this solution, the macro would have been expanded like the `matmul` one with more optimized implementations and inclusion of third party optimized library. These optimizations would have immediatly impacted all the existing code due to the code factorization with the macro. One such optimization, using AVX2-specific intrinsic operations like `_mm256_permute2f128_pd` [84] and `_mm256_shuffle_pd` [85], was tested as a proof of concept for the specific problem settings were it could be used efficiently [23].

Adding new options in the specification file

I added new optional optimization flags to the specification file's JSON Schema in a similar fashion as in Section 5.2.1. An application expert can then use these flags to indicate a vectorized implementation of a given user function exists and should be used to improve performance, as was done in Section 5.1.3. I added multiple flags, one for each vectorizable user function:

- Flux, `flux_vect` in the specification file, translated to `useFluxVect` in the context and Kernel Generator API.
- Non-Conservative Product, `ncp_vect` translated to `useNCPVect`.
- Source terms combined with the non-conservative product term, as it was a relevant optimization for the astrophysic applications FO-CCZ4, `fusedsource_vect` translated to `useFusedSourceVect`.

- Multiplication by a material parameter matrix, `material_parameters_vect` translated to `useMaterialParamVect`.

These options were added to a new specification file array: `optimised_terms` and can only be used in combination with the corresponding entry in the `term` array.

Isolating the logic from the template

```
1  { // Compute the fluxes
2    double* F[{{nDim}}];
3    for (int xyz = 0; xyz < {{nDof**nDim}}; xyz++) {
4      F[0] = lFi+{{idxLFi(0,t,0,0,xyz,0)}};
5      F[1] = lFi+{{idxLFi(1,t,0,0,xyz,0)}};
6  {% if nDim == 3 %}
7      F[2] = lFi+{{idxLFi(2,t,0,0,xyz,0)}};
8  {% endif %}
9      solver.{{solverName}}::flux(lQi+{{idxLQi(t,0,0,xyz,0)}}, F);
10   }
11 }
```

Listing 5.14: Flux function call loop isolated in a subtemplate.

I then went through the kernel templates and isolated the user function call loops in subtemplates, for example Listing 5.14 shows the flux function call loop.

In an early iteration, presented in [23], these subtemplates were then hidden behind macros, this however made it harder to use other optimization macros inside the subtemplate, in particular the index macro that was introduced later. Thus in the later versions, I used direct inclusion instead.

```
1  {# Calling the flux function to fill the lFi tensor#}
2  {% with %}
3  {% filter indent(width=6, first=True) %}
4  {% include 'subtemplates/flux_PDE_over_xyz.template' %}
5  {% endfilter %}
6  {% endwith %}
```

Listing 5.15: Inclusion of the extracted subtemplate from Listing 5.14.

For example, in Listing 5.15 the previous code excerpt is included where it was extracted from. The `with` statement creates a new scope and allows the declaration of scope local template variables if required. A `filter` statement is used to properly indent the included code for readability.

```
1  {% with %}
2  {# /** Set up helper template values */ #}
3  {% set chunkRest=((nDof**nDim)%chunkSize) %}
4  {% set restStart=nDof**nDim-chunkRest %}
5  {# /** Subtemplate */ #}
6  {% if useFluxVect %}{# Vectorized flux #}
7  { // Compute the fluxes in chunks
8    {{m.allocateArray('Fx', nVarPad*chunkSize, forceStack=True)}}{##}
9    {{m.allocateArray('Fy', nVarPad*chunkSize, forceStack=True)}}{##}
10   {{m.allocateArray('Fz', nVarPad*chunkSize, forceStack=True)}}{##}
11   {{m.allocateArray('lQiT', nVarPad*chunkSize, forceStack=True)}}{##}
12   double* F[3] = {Fx, Fy, Fz};
13   for (int xyz = 0; xyz < {{restStart}}; xyz+={{chunkSize}}) {
14     {{m.transpose('lQi', 'lQiT', idxLQi(t,0,0,xyz,0), 0, chunkSize,
         ↪ nVarPad) | indent(4) }}{##}
15     solver.{{solverName}}::flux_vect(lQiT, F);
16     {{m.transpose('Fx', 'lFi', 0, idxLFi(0,t,0,0,xyz,0), nVarPad,
         ↪ chunkSize) | indent(4) }}{##}
17     {{m.transpose('Fy', 'lFi', 0, idxLFi(1,t,0,0,xyz,0), nVarPad,
         ↪ chunkSize) | indent(4) }}{##}
18 {% if nDim == 3 %}
19     {{m.transpose('Fz', 'lFi', 0, idxLFi(2,t,0,0,xyz,0), nVarPad,
         ↪ chunkSize) | indent(4) }}{##}
20 {% endif %}{# 3D #}
21   }
22 {% if chunkRest > 0 %}
23   {  // process the last non complete chunk
24     {{m.transpose_rest('lQi', 'lQiT', idxLQi(t,0,0,restStart,0), 0,
         ↪ chunkSize, nVarPad, chunkRest, nVarpad, safe=True) | indent
         ↪ (4) }}{##}
25     solver.{{solverName}}::flux_vect(lQiT, F);
26     {{m.transpose_rest('Fx', 'lFi', 0, idxLFi(0,t,0,0,restStart,0),
         ↪ nVarPad, chunkSize, nVarPad, chunkRest) | indent(4) }}{##}
27     {{m.transpose_rest('Fy', 'lFi', 0, idxLFi(1,t,0,0,restStart,0),
         ↪ nVarPad, chunkSize, nVarPad, chunkRest) | indent(4) }}{##}
28 {% if nDim == 3 %}
29     {{m.transpose_rest('Fz', 'lFi', 0, idxLFi(2,t,0,0,restStart,0),
         ↪ nVarPad, chunkSize, nVarPad, chunkRest) | indent(4) }}{##}
30 {% endif %}{# 3D #}
31   }
32 {% endif %}{# chunkRest > 0 #}
33 }
34 {% else %}{# useFluxVect #}
35 {#[...] default scalar case#}
36 {% endif %}{# useFluxVect#}
37 {% endwith %}
```

Listing 5.16: Modified subtemplate of the call to the `flux` adding the on the fly transpose case.

Adding the new branching option

With the function calls to be optimized isolated into subtemplates, such as the flux function in Listing 5.14, I modified them to add the new logic as shown in Listing 5.16 for the `flux` function. First some local template variables are defined at line 3 and 4. A branching at line 6 using the flags introduced earlier ensures that by default the standard scalar version is rendered in the else case.

If vectorization is used, required temporary arrays are allocated on the stack with the `allocateArray` macro, introduced in Listing 4.14. Then the for loop proceeds in chunks. At each iteration a chunk of the input is transposed, using the `transpose` macro from Listing 5.13. Then the vectorized user function is called and the outputs stored in the previously allocated temporary arrays are transposed back to the expected kernel tensor. At line 22, a branching using the previously define `chunkRest` variable tests if some remaining data needs to be processed in an incomplete chunk. If so the same logic as before is used for a last iteration, this time using the `transpose_rest` macro to replace the missing data by copying existing data in the forward transposition to complete the chunk and ignoring this excess data when transposing back the result.

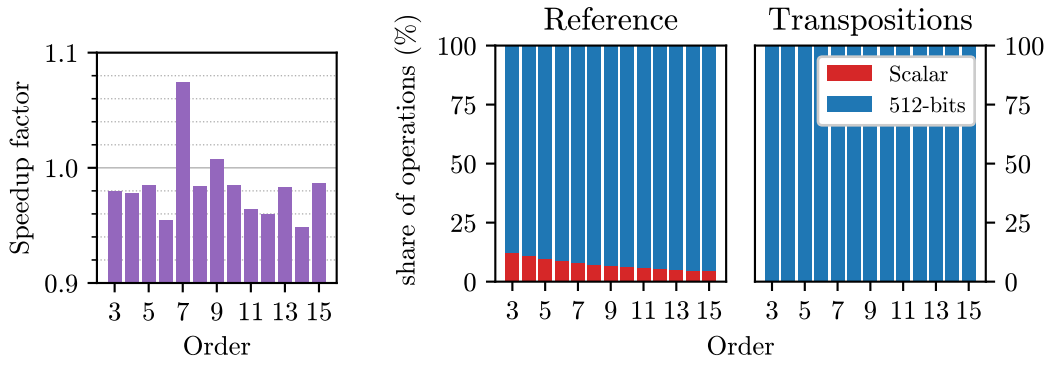
I modified other user function's subtemplates in a similar fashion. Using subtemplates allowed me to hide this new complexity from the main algorithmic template.

Performance evaluation

This optimization showed good performance increases on complex nonlinear application such as the FO-CCZ4 astrophysics model, as reported in a previous publication [23], with a speedup up to a factor 1.27 on the Haswell architecture (AVX2). This application's runtime is dominated by its costly PDE system, and thus the additional costs of the transpositions are more than compensated by the increased performance there.

However, when I applied the same ideas to the linear scheme for ExaSeis, the results, shown in Figure 5.7 were unsatisfactory. Only a small 7% speedup was observed at order 7, with small slowdown of a couple percentage points at other orders. While the user functions were now properly vectorized, with over 99.9% of the kernel floating-point arithmetic operations being packed in SIMD instructions, more time was lost performing the transpositions. The exception of order 7 is due to it falling on a sweet spot: In this case the number of degrees of freedom in each direction is 8, which is exactly the vector register length.

A better transposition implementation in the `transpose` macro could slightly improve this, and maybe help achieve a speedup in ExaSeis at all orders. However, other improvements to the numerical scheme allowed me to try out another solution using a hybrid data layout, that will be presented in Section 8.4. It showed much better results on linear applications than this scheme, and after some algorithmic modification to the nonlinear scheme, it could also be replicated there, again showing better performance than the existing on the fly transposition scheme. Thus, while the on the fly transposition macros were kept as a proof of concept, they were not used in the latest kernel variants.



(a) Performance

(b) Floating point operations packing

Figure 5.7: Performance benchmark of the on the fly transposition scheme on ExaSeis LOH1 setup, at order 3 to 15. **(a)**: Runtime speedup factor compared to the reference with scalar user function. **(b)**: Distribution of the packing size used to perform the floating point operations.

5.3.2 Support for the BLAS library Eigen

In this use case, I extended the Kernel Generator to be able to render its `matmul` macro using the BLAS library Eigen [41]. Eigen is a C++ library of template headers for linear algebra, matrix and vector operations. It aims at presenting operations such as matrix multiplications with a high-level API while maintaining high performance through expression template metaprogramming, i.e. having the compiler produce the optimized code from the included template header at compilation.

To perform a matrix multiplication with Eigen, the arguments have to be expressed as `Eigen::Matrix` objects. If raw C++ arrays are used, an `Eigen::Map` can be used instead to proceed with minimal overhead. Once properly set up, the matrix multiplication $C+ = A \cdot B$ can simply be written as "`C += A * B;`". Some options can be used to improve the performance, the most important one, according to the documentation[5], being using `C.noalias()` when no aliasing is present to allow Eigen to directly evaluate the matrix multiplication into the matrix `C`.

Adding the submodule

Eigen is a third party library available through a git repository and does not have any dependency. Thus I added it to ExaHyPE's submodules list as shown in the following `.gitmodules` excerpt

```
1 [submodule "Submodules/eigen"]
2     path = Submodules/eigen
3     url = https://gitlab.com/libeigen/eigen.git
```

[5] `https://eigen.tuxfamily.org/dox/group__TutorialMatrixArithmetic.html`

I also updated the `updateSubmodules.sh` script to properly download and set up the Eigen submodule.

Linking to the Kernel Generator

The Kernel Generator already provided an option to chose to use LIBXSMM or a pure C++ implementation, and therefore Eigen's integration was also made through the `configuration.py` file. First the path to the library was added in the configuration:

```
1 # path to eigen, symlinked into the directory if eigen is used
2 pathToEigen = os.path.abspath(os.path.join(pathToExaHyPERoot, "
    ↪ Submodules", "eigen"))
```

Then to decide which BLAS library to use, the Kernel Generator uses a hard-coded string variable. I made this design choice with the expectation that chosing a BLAS library is not "application-dependent", but rather "installation-dependent", as chosing the best BLAS library mostly depends on the hardware. Thus, I added Eigen as a possible value for the `matmulLib` variable:

```
1 # choose the BLAS library for the matmul:
2 # matmulLib = "Libxsmm"
3 matmulLib = "Eigen"
4 # matmulLib = "None"
```

When building the context, the Controller sets up the boolean `useEigen` dependending on the `matmulLib` value in the configuration.

Expanding the `matmul` macro

Matrix multiplications are abstracted behind the `matmul` macro that reads configuration parameters to set up the proper matrix multiplication for its input. A simplified version of the macro implementation capable of chosing between using LIBXSMM or a pure C++ implementation was presented in Listing 4.16. Therefore, I expanded this macro to add a third case for Eigen. The excerpt shown in Listing 5.17 is a simplifed version of the Eigen case that is added to Listing 4.16 between the two existing cases.

Listing 5.18 shows a possible rendered code of the `matmul` call performing the computation of the gradient of the state vector. In the first line we can see that an `Eigen::Map` is applied to the shifted pointer of the tensor to setup the extraction of a matrix slice of type `Eigen::Matrix<double,12,6>`, specifying the floating point format used as well as the size of the matrix. Furthermore, this matrix is defined as being aligned and an outerstride is set with `Eigen::OuterStride<12>`, this is particularly relevant for the second matrix as the outerstride is here bigger than the used dimension size, indicating that the matrix used some padding and the relevant data is not contiguous.

```
{% elif useEigen %}
#pragma forceinline recursive
{
  new (&{{conf.baseroutinename}}_A_map) Eigen::Map<Eigen::Matrix<
      ↪ double,{{conf.M}},{{conf.K}}>, Eigen::{{"Aligned" if conf.
      ↪ alignment_A == 1 else "Unaligned"}}, Eigen::OuterStride<{{
      ↪ conf.LDA}}> >({{A}}+{{A_shift}});
  new (&{{conf.baseroutinename}}_B_map) Eigen::Map<Eigen::Matrix<
      ↪ double,{{conf.K}},{{conf.N}}>, Eigen::{{"Aligned" if conf.
      ↪ alignment_B == 1 else "Unaligned"}}, Eigen::OuterStride<{{
      ↪ conf.LDB}}> >({{B}}+{{B_shift}});
  new (&{{conf.baseroutinename}}_C_map) Eigen::Map<Eigen::Matrix<
      ↪ double,{{conf.M}},{{conf.N}}>, Eigen::{{"Aligned" if conf.
      ↪ alignment_C == 1 else "Unaligned"}}, Eigen::OuterStride<{{
      ↪ conf.LDC}}> >({{C}}+{{C_shift}});
  {{conf.baseroutinename}}_C_map.noalias() {{ '+' if conf.beta == 1
      ↪ }}= {{ '-1. * ' if conf.alpha == -1 }}{{conf.baseroutinename
      ↪ }}_A_map * {{conf.baseroutinename}}_B_map;
}
```

Listing 5.17: Extension to the `matmul` macro, in Listing 4.16, to add the Eigen case.

```
#pragma forceinline recursive
{
  new (&gradQ_x_A_map) Eigen::Map<Eigen::Matrix<double,12,6>, Eigen::
      ↪ Aligned, Eigen::OuterStride<12> >(lQi+yz*72);
  new (&gradQ_x_B_map) Eigen::Map<Eigen::Matrix<double,6,6>, Eigen::
      ↪ Aligned, Eigen::OuterStride<8> >(dudxT_by_dx);
  new (&gradQ_x_C_map) Eigen::Map<Eigen::Matrix<double,12,6>, Eigen::
      ↪ Aligned, Eigen::OuterStride<12> >(gradQ+yz*72);
  gradQ_x_C_map.noalias() = gradQ_x_A_map * gradQ_x_B_map;
}
```

Listing 5.18: Example of a matrix multiplication performed with Eigen using the `matmul` macro.

Also here the `Eigen::Map` variables are not created but instead the C++ "placement new" syntax is used to change the pointer used to avoid having to create the Map variable every time the same matrix multiplication is performed on a different matrix slice of the tensor. This however requires for each matrix multiplication configuration to initialize three Map variables before using the `matmul` macro.

Adding support macros

Listing 5.19 shows a simplified version of the new macro `setupMatmul(matmulKey)`, that I introduced to initialize the `Eigen::Map` variables. If Eigen is used, it creates the three variables using the matrix configuration and initializes them with the `nullptr`. Otherwise

```
{% macro setupMatmul(matmulKey) %}
{% if matmulKey in matmulConfigs %}
{% with %}
{% set conf = matmulConfigs[matmulKey] %}
{#
// Eigen case
#}
{% if useEigen %}
// setup Map for {{conf.baseroutinename}}
Eigen::Map<[...]> {{conf.baseroutinename}}_A_map(nullptr);
Eigen::Map<[...]> {{conf.baseroutinename}}_B_map(nullptr);
Eigen::Map<[...]> {{conf.baseroutinename}}_C_map(nullptr);
{% endif %}
{% endwith %}
{% endif %}{# matmulKey in matmulConfigs #}
{% endmacro %}
```

Listing 5.19: New `setupMatmul` macro to initialize the `Eigen::Map` when Eigen is used by the `matmul` macro.

it does nothing. This macro is then called in each template using the `matmul` macro. On the same example as previously for the `matmul` code, the call is simply :

```
{{ m.setupMatmul('gradQ_x_sck') | indent(2) }}{##}
```

```
{% macro matmulInclude() %}
{% if useEigen %}
// include Eigen for matmul
#include <{{pathToOptKernel}}/Eigen/Dense>
{% endif %}
{% if useLibxsmm %}
// include libxsmms' gemms for matmul
#include "{{pathToOptKernel}}/gemmsCPP.h"
{% endif %}
{% endmacro %}
```

Listing 5.20: Extension of the `matmulInclude` macro, used to include LIBXSMM's generated gemms when used, to properly include Eigen's header when Eigen is used by the `matmul` macro.

Furthermore, Eigen is a pure C++ header library, the required header needs to be included by each file using Eigen. To do so I expanded the `matmulInclude()` macro, that was used to include the header to the `gemm` functions produced by LIBXSMM, with a new case if `useEigen` is set, as shown in Listing 5.20.

Evaluation and future development

With these short steps, ExaHyPE now fully integrates Eigen as a new BLAS library option that all applications can use when producing optimized kernels of any numerical scheme using matrix multiplication. The only modifications I did to the actual algorithmic templates were to add the calls to the new `setupMatmul` macro as Eigen was the first BLAS library requiring some setup. Otherwise, all modifications were done either in the controller and configuration files of the Kernel Generator, or in existing optimization macros. This greatly simplified the process and ensured that all the code generation is upgraded at the same time.

As will be discussed in Section 7.2.3, performance tests using the LOH1 application showed that Eigen is slower than LIBXSMM on SuperMUC-NG's Skylake CPUs. This was to be expected as LIBXSMM is highly tuned toward this type of architecture and the small matrices we are using, whereas Eigen is a more generalist library introducing overheads to reformulate the matrix multiplication in its user-friendly API.

However, the options now exists should ExaHyPE be ported to other architecture not supported by LIBXSMM. Furthermore, the same steps can be replicated to add support for another BLAS library to ExaHyPE. This ensures that ExaHyPE can be upgraded to stay performant if a better BLAS library is introduced for a given future relevant hardware.

5.4 Evaluation of the role-oriented design

Looking back at the design requirements made in Section 3.1, the ExaSeis use case showed that they were all fulfilled by the modular design of ExaHyPE and the use of code generation with Jinja2 templates:

1. The application experts were able to implement the full simulation without directly working on the engine, only having to consider the exposed user API to implement model relevant user functions.

2. The Kernel Generator produced optimized kernels tuned toward the application and the Skylake architecture. When the application was further optimized with split and vectorized user functions, more optimized numerical schemes were used in the kernels.

3. The algorithm and optimization experts were able to modify the kernels to respond to new applications' requirements and optimization opportunities. These new features were fully integrated to the Engine and are now available to future applications.

4. The algorithm and optimization experts were able to perform their work independently of one another with the produced kernels integrating both contributions.

The design of the Kernel Generator, in particular the use of the Jinja2 template engine and the separation of the views in algorithmic templates and optimization macros, was fully exploited to satisfy the last requirements.

Furthermore, the implementation of the overarching ADER-DG scheme was provided by ExaHyPE core and the Peano framework dealt with the parallelization of the code. No work was necessary on these complex tasks and they were safely ignored when implementing the application or modifying the Engine's kernels.

With the fulfillement of these requirements, the workflow of each of our user roles was streamlined and each role could focus on its area of expertise. The contribution of each user roles was automatically integrated to the compiled executable used to perform the simulation. As each user working on ExaSeis was able to assume one user role when performing its tasks, the role-oriented approach facilitated the implementation and optimization of the ExaSeis application.

Part III

Optimizations toward modern Intel CPU architectures

6 CPU optimization considerations

In its initial project proposal, ExaHyPE targeted the Haswell architecture and the novel Xeon Phi processors with the upcoming Knight Landing release, which aimed to be a bridge between CPU and GPU. As Intel discontinued the Xeon Phi line in 2018 to refocus on more classical CPU architectures, ExaHyPE included the new Skylake products to its targets. Thus, while the modularity introduced previously would allow the support of more diverse architectures, we focused on modern CPU optimizations.

This chapter introduces the key concepts relevant to code optimization on modern CPU architectures in the context of ExaHyPE. In Section 6.1, we introduce SIMD and its requirements. Then in Section 6.2, we present how padding can be used on multidimensional tensors to improve the performance gain from vectorization by ensuring data alignment. Finally in Section 6.3, we discuss the effects of cache behavior on performance.

6.1 SIMD

Single-Instruction-Multiple-Data, SIMD for short, is a paradigm for data-level parallelism in which the same operation is applied simultaneously on multiple elements in one thread.

Supercomputers operating on a vector of data with a single instruction can be traced back as far as the 1960's with the ILLIAC IV supercomputer [86]. Later others, such as the CDC Star-100 [87] and Cray-1 [88], improved upon ILLIAC IV's limitations. However, it is only in recent years that this concept became increasingly relevant to the performance of CPU based supercomputers with the introduction of the AVX2, and later AVX-512, instruction sets. These instruction sets rely on *vector registers*, 256-bits long for the YMM registers of AVX2, and 512-bits long for AVX-512's ZMM registers, on which the SIMD vector instructions are performed. For example with AVX-512, a SIMD Fused-Multiply-Add instruction can be applied to ZMM vector-registers containing 8 double-precision floating point numbers, thus performing 16 double-precision floating point operations in one CPU cycle.

In essence, it is a form of single core parallelism, which like regular parallelism can greatly improve the code performance at the cost of some constraints to enable its usage. The optimization of a codebase with SIMD instructions is commonly referred as *vectorization*.

```
1  void foo(double* A, double* B, double* C) {
2    for(int i=0; i<8; i++) {
3      C[i] += A[i]*B[i];
4    }
5  }
```

Listing 6.1: Small example function suitable for vectorization.

6.1.1 Code vectorization

There are two main ways to vectorize a C++ codebase. To illustrate the different approaches, let us consider the function snippet in Listing 6.1, which represents a very suitable target for vectorization.

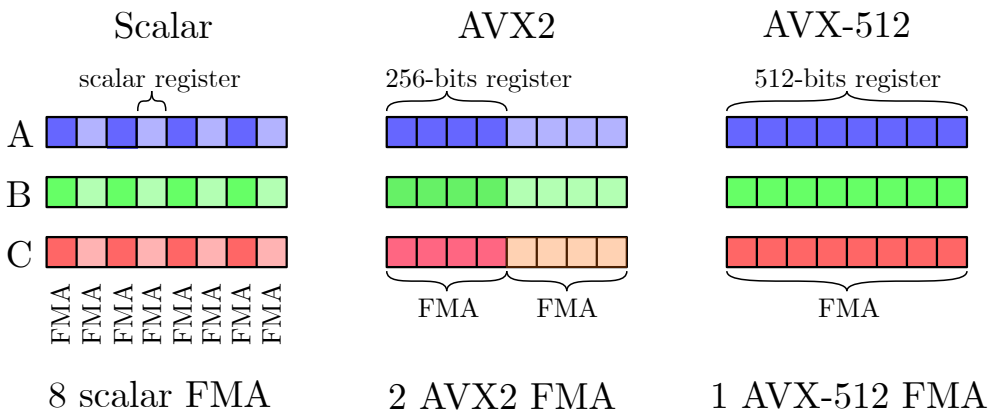


Figure 6.1: The code excerpt in Listing 6.1 requires 8 scalar FMA instructions (left). AVX2 packs 4 double together on its vector registers and performs the same computation with two vectorized FMA (middle), whereas AVX-512 packs 8 double and only needs one vectorized FMA (right).

As can be seen in Figure 6.1, the scalar version of this function requires 8 Fused-Multiply-Add (FMA) instructions, whereas only two packed AVX2 FMA instructions, or a single AVX-512 FMA one, perform the same computations.

The first method is to include assembly code in the C++ code using the chosen SIMD instruction set. However, this is very rarely used as another equivalent way is to use *intrisics*, which are C-style functions providing a direct access to the related SIMD instructions without the need to explicitly write assembly code [89].

Listing 6.2 shows a vectorized implementation of the code from Listing 6.1 using intrisics for AVX2 instructions. While effective at achieving vectorization, the new code is less readable and its development requires an expertise of intrinsics. Furthermore, this code uses explicitly AVX2, and thus it is unusable on older hardware not supporting the

```
1  #include <immintrin.h>
2
3  void foo(double* A, double* B, double* C) {
4    __m256d ymm0, ymm1, ymm2;
5    ymm0 = _mm256_load_pd(A);
6    ymm1 = _mm256_load_pd(B);
7    ymm2 = _mm256_load_pd(C);
8    ymm2 = _mm256_fmadd_pd(ymm0,ymm1,ymm2);
9    _mm256_store_pd(C, ymm2);
10   ymm0 = _mm256_load_pd(A+4);
11   ymm1 = _mm256_load_pd(B+4);
12   ymm2 = _mm256_load_pd(C+4);
13   ymm2 = _mm256_fmadd_pd(ymm0,ymm1,ymm2);
14   _mm256_store_pd(C+4, ymm2);
15 }
```

Listing 6.2: Vectorization of the function from Listing 6.1 using intrinsics for AVX2 instructions.

instruction set, and it cannot use the more efficient AVX-512 instructions when available on newer CPUs. The latter is particularly detrimental as in this case AVX-512 would be very suitable, since everything could be done in one step instead of two.

The second approach is to rely on the compiler's *auto-vectorization* capabilities. A modern compiler can automatically vectorize a loop when its heuristics detect it being suitable. Pragmas are often required to guide these heuristics, telling the compiler that it can safely vectorize the loop, and the assumptions regarding the data it can make when doing so.

The C++ standard does not define specific pragmas for this and thus, until recently, most were compiler specific such as `#pragma simd` which was an Intel Compiler pragma not recognized by G++ for example. Thankfully, the OpenMP API introduced the standardized `#pragma omp simd`, that is now supported by all the latest versions of the most commonly used compilers. This pragma also supports multiple optional clauses, allowing the developer to further help the compiler to optimally vectorize the loop, for example one can specify the memory alignment of the arrays.

```
1  void foo(double* A, double* B, double* C) {
2    #pragma omp simd aligned(A,B,C:64)
3    for(int i=0; i<8; i++) {
4      C[i] += A[i]*B[i];
5    }
6  }
```

Listing 6.3: Vectorization of the function from Listing 6.1 using an OMP pragma and compiler auto-vectorization.

With this the example can be trivially vectorized, as shown in Listing 6.3, assuming the correct compiler flags are used. The vectorization can be verified on most compilers using their optimization reporting features. Here the source code stays in C++ and the compiler vectorizes it in accordance with the target architecture defined at compilation time. Thus the codebase is not bound to a specific SIMD instruction set and can be ported to newer architectures.

Therefore, I used auto-vectorization when vectorizing code directly. Direct vectorization in assembly is also present in ExaHyPE in some highly optimized subroutines that are generated by LIBXSMM, a third party code generator, the code generation negating the approach's drawbacks.

6.1.2 Data layout's optimization toward vectorization

Like regular parallelism, vectorization requires a suitable codebase and works best when specific constraints are fulfilled [89].

The hard constraints that can prevent vectorization are mostly algorithmic in nature. The absence of data flow dependencies in the vectorized loop is required as the simultaneous nature of SIMD instruction breaks the dependency and introduces numerical errors. Likewise data-dependent loop exit conditions, function calls inside the loop, and non-vectorizable instructions (e.g. I/O access) prevent vectorization. In ExaHyPE's kernels, most of these constraints are easily satisfied.

Vectorization can work on any array stride, but contiguous memory spares the need for costly *gather* and *scatter* operations. Thus, when vectorizing operations on multidimensional arrays, it is better to vectorize the innermost loop that should be, when possible, on the unit-stride fastest running index. In ExaHyPE, this causes some data layout constraints and conflicts that will be discussed in Section 8.4.1.

Similarly, *alignment* of the data, i.e. having the starting address of an array lying at a specific bytes-boundary dependant on the SIMD instruction set, e.g. 64-bytes alignment for AVX-512 instructions, improves performance. Alignment needs to be ensured at data allocation. Therefore, it requires code adaptations outside of the vectorized loop.

To allocate aligned memory, there is, again, no C++ standard defined way to do so. For stack memory allocation, the de-facto standard is to use the `__attribute__` keyword that is supported by most compilers. For example,

```
double A[128] __attribute__((aligned(64)));
```

to allocate an array `A` aligned on a 64-bytes boundary. For heap memory allocation, the Intel compiler provides the `_mm_malloc` and `_mm_free` functions that are similar to the usual `malloc` and `free` C functions, but need to be used together to avoid memory corruption.

6.1.3 Performance considerations

With AVX-512 on a codebase using double-precision floating-point, a single SIMD instruction can perform the equivalent of 8 scalar instructions. Thus, a speedup of a factor 8 can naively be expected. However, multiple factors make this naive expectation hard or even impossible to reach.

First, on most architectures, using SIMD instructions lowers the base clock speed to prevent overheating, which decreases the overall speedup. For example, on SuperMUC-NG's Intel Skylake Xeon Platinum 8174, the base clock frequency on a scalar codebase is 2.7GHz and drops down to 1.9GHz when using AVX-512[1]. In this case, the overall maximum speedup is only $8 \dot{1}.9/2.7 = 5.62$. This slowdown persists for some microseconds after the last seen SIMD instruction, and thus the Intel Compiler's heuristics by default favor AVX2 over AVX-512, as the slow down from AVX2 is less significant making it better for lightly vectorized code[89].

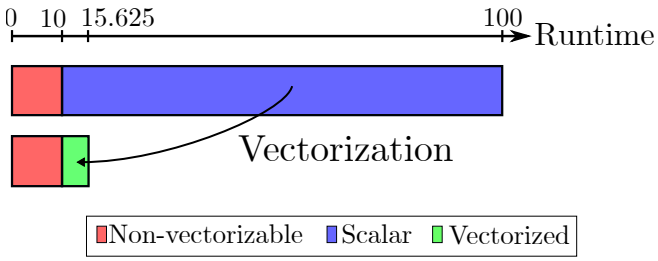


Figure 6.2: According to Amdahl's law, 90% of the runtime being vectorized and becoming 16 times faster only results in an overall speedup of 6.4 at best.

Second, Amdahl's law needs to be taken into consideration. As most codebase cannot be fully vectorized, the scalar portion of the code limits the theoretical speedup. While this effect was negligible with the small vector length of early SIMD instruction sets, the now larger 512-bits vector length means that even a small fraction of the codebase still using scalar operations results in a ceiling to the overall speedup way below expectations. For example, as illustrated in Figure 6.2, a codebase in single-precision where 10% of the runtime cannot be vectorized can only reach a speedup of a factor 6.4[2], instead of 16, when replacing the scalar operations with AVX-512 ones, as the non-vectorized portion of the code becomes the dominant one.

Third, as the floating point operations are performed faster, new bottlenecks may reveal themselves and become the dominant factor limiting the overall performance. Looking at a simplified roofline model in Figure 6.3, we see that a program with an arithmetic intensity placing it in the blue part of the model is compute-bound using the scalar roof, but becomes memory-bound with the higher SIMD roof, as its arithmetic intensity stays unchanged by vectorization. Small dense matrix multiplications in double-precision

[1] `https://doku.lrz.de/display/PUBLIC/Details+of+Compute+Nodes`
[2] Applying Amdahl's law formula, the theoretical maximum speedup is $1/(0.1 + 0.9/16) = 6.4$

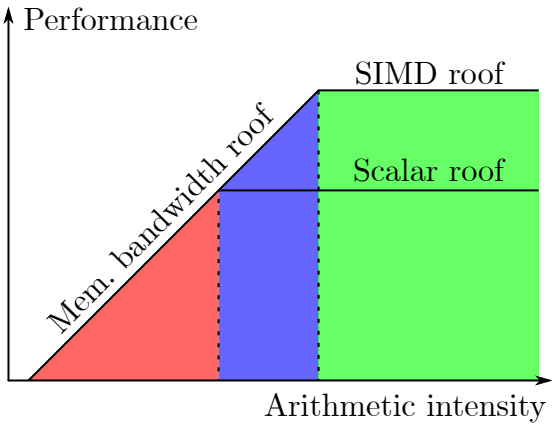


Figure 6.3: Simplified roofline model with a roof for scalar instructions and for vectorized ones. A program in the red domain is memory-bound and compute-bound in the green one, with or without vectorization. However, in the blue domain it is compute-bound without vectorization and becomes memory-bound with it.

(DGEMM) can be affected by this, should they not properly exploit the cache to move the memory bandwidth roof to the left.

Finally, another memory-related bottleneck often arises from cache misses and cache stalls, as will be discussed in Section 6.3.

Therefore, while vectorization increases performance, fully exploiting the SIMD capabilities of modern CPU requires more than a basic vectorization of the dominant part of the codebase. The vectorization needs to be aggressive and be applied to as much of the code as possible, to reduce the ceiling gap implied by Amdahl's law and minimize the cost of the clock speed's slowdown. Furthermore, memory related bottlenecks that arise after a first vectorization pass can only be solved by rethinking the whole algorithm to also optimize memory accesses and cache behavior.

6.2 Memory alignment and padding

To fully exploit the capability of SIMD vectorization, the data needs to be aligned [89]. However, this becomes more difficult to ensure when working with multidimensional tensors, when numerical operations are only applied to a subset of the tensors' dimensions.

6.2.1 Multidimensional array

In C++, tensors such $Q_{z,y,x,n}$ can be realized in two main ways:

- A pointer-based array, e.g. realized via `double**** Q` and with the tensor value $Q_{z,y,x,n}$ = `Q[z][y][x][n]`
- A flattened array, e.g. `double* Q` and $Q_{z,y,x,n}$ = `Q[((z*yL+y)*xL+x)*nL+n]`

As our numerical scheme works on tensors' slices, to optimize the vectorization we need to ensure that the fastest dimension stays aligned in each slice. In other words, using the pointer-based array notation, $\forall z, y, x$, `Q[z][y][x]` (a one-dimensional array) should be an aligned array.

For pointer-based arrays allocated with the naive for loop method, this can trivially be ensured using aligned array allocation, as illustrated by the following example:

```
1 double** A = ((double**) _mm_malloc(sizeof(double)*dim2, 64);
2 for(int i=0; i<dim2; i++) {
3   A[i] = ((double*) _mm_malloc(sizeof(double)*dim1, 64));
4 }
```

However, using this code results in arrays that are not contiguous in memory. This can lead to poor performance due to suboptimal cache behaviors. Most methods that can be used to get a contiguous pointer-based array are working with a flattened array in the background.

While the flattened array ensures a contiguous memory, alignment is lost in most cases. Looking at the implementation of a flattened array, e.g. `Q[((z*yL+y)*xL+x)*nL+n]`, the only way to ensure that all its subarrays, for any `z`, `y` and `x`, are aligned is to have the array itself be aligned and the leading dimension's size (`nL` in the example) be a multiple of the vector length, e.g. a multiple of 4 with double-precision AVX2. Aligning the array is trivial, but the leading dimension's size is an application-specific parameter and the vector length an architecture-specific one. To solve this, zero-padding is used to increase the leading dimension's size to the next architecture-specific value.

In ExaHyPE, multidimensional tensors and matrices are implemented as flattened one-dimensional zero-padded arrays. As flattened array can be cumbersome with the conversion of the tensor index to the array index, the Jinja2 `index` macro, presented in Section 4.4.2, can be used to generate the relevant computation while keeping the code simple and readable in the templates.

6.2.2 Padding's free lunch

Padding ensures that all tensor slices are aligned for optimal vectorization, while keeping the memory contiguous to improve cache behavior, although it can come at some costs.

The first one is the obviously increased memory footprint. This increase is mostly negligible performance-wise, as long as the numerical scheme is compute-bound. Furthermore, the padding is done on the fastest running dimension and only increases its sizes to the next multiple of the vector length (usually 4 or 8). Hence, the relative impact of padding

diminishes with a larger base length as it lowers the ratio of padding to useful data. Padding can also be totally avoided if the base length is already at a sweet spot.

The second cost comes if some computations are performed on the padding, which increases the total amount of floating-point operations executed while not being scientifically relevant. However, with SIMD taken into account this is not always a true performance cost, as these extra operations can be computed for free or even at a speedup compared to not performing them. To illustrate this let us consider a simple vectorizable loop with a SIMD vector length of 4, e.g. AVX2, and aligned floating point arrays `A` and `B` of size 7:

```
1 for(int i=0; i<7; i++) {
2   A[i] += B[i];
3 }
```

Without padding and SIMD, this loop requires 7 scalar additions. With SIMD it can be performed in either 2 SIMD instructions, the second one with masking, or one SIMD instruction and three scalar ones. If both arrays are padded to the next multiple of the vector length, here 8, and the loop's boundary increased to include the padding, then it now requires 8 scalar additions, so one extra floating point operation without scientific significance. However with SIMD, these 8 scalar operations can be performed in 2 full SIMD instructions without masking[3], which is faster than what was possible without the padding. Thus, while the padding added some irrelevant floating point operations to perform, computing them with SIMD can actually decrease the amount of instructions required and increase performance, even without considering the additional gain from the guaranteed alignment.

Therefore, as long as the code is compute-bound, increasing the memory footprint with padding can accelerate a vectorized code.

6.3 Cache behavior considerations

So far, optimizations have been discussed under the assumption of a compute-bound algorithm, i.e. the performance is determined by the CPU speed when processing the algorithm's floating point operations. However, the simple compute-bound vs memory-bound dichotomy often does not give the full picture. One also has to consider the memory hierarchy, depicted in Figure 6.4, with the slow but large main memory and the progressively faster but smaller CPU cache levels. For example, on a Skylake core, the latency when fetching data from the L3 cache is 50 to 70 CPU cycles, compared to only 4 to 6 cycles from L1 and 14 cycles for L2 [89], with a main memory access being much slower than any cache access.

An *memory stall* is when a CPU core idles while waiting for data. This can happen even when the data is in cache, thus having the data in a higher cache level can improve

[3]SIMD instructions with masking are slower than ones without [89].

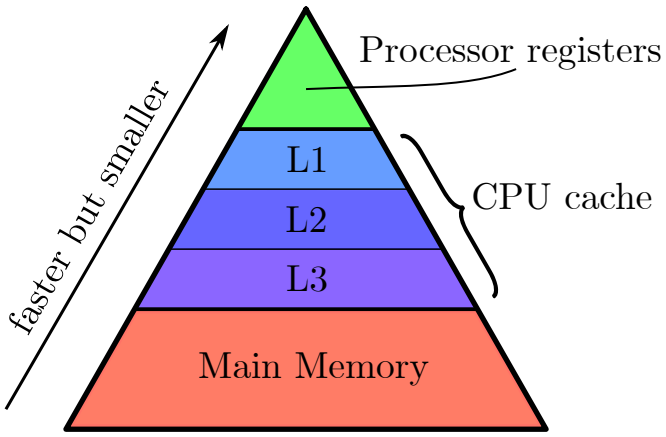


Figure 6.4: Simplified memory hierarchy on modern CPU such as Intel Xeon. Higher levels have a higher bandwith and reduced latency but are smaller in size.

performance. A *cache miss* occurs when a piece of data cannot be read or written in the cache, requiring a main memory access and usually resulting in an memory stall. Hence nowadays, due to the processor-memory performance gap [90], the CPU performance is very reliant on the proper use of its cache, in particular the higher L1 and L2 caches.

Furthermore, as vectorization increases the number of floating point operations performed per CPU cycle, when a memory stall occurs on a vectorized code, the time lost waiting for the data represents more potential floating point operations not performed. This is worsened as vectorization makes the core process the data faster and in bigger batches, increasing the frequency of cache misses, both in time and per instruction.

Thus, as will be discussed in Section 8.3, improving the cache behavior, in particular minimizing the amount of cache misses, is a critical step to improve the overall performance as cache stalls may become the dominant performance bottleneck in a vectorized application. Caching works in large data chunks: cache lines. Therefore, cache behavior can be improved by increasing data locality to better exploit cache lines and by reducing the overall memory footprint to ensure all the data fits on the limited cache capacity. This may go against performance related optimizations with, for example, recomputing some data instead of storing them for later to be able to reduce the memory footprint. Hence, optimizing an algorithm is a balancing act between contradicting guidelines.

7 Optimization of ExaHyPE

As described when discussing the ADER-DG scheme in Section 2.2, the kernels are the most critical components of ExaHyPE performance-wise, and are isolated in single-thread functions.

This chapter presents my general optimization of the kernels as well as a tuning of the compilation process. Section 7.1 details how I used SIMD and other optimization techniques, made possible by the use of code generation, with a template engine to fine tune the generated kernels toward a target CPU architecture. To fully exploit SIMD, I used a Loop-over-GEMM scheme to perform tensor contractions with highly optimized matrix multiplication functions produced by the LIBXSMM BLAS library, as is presented in Section 7.2. Finally, as I enabled the compiler auto-vectorization capabilities, I fine-tuned the compiler's heuristics to improve ExaHyPE's performance without any code modification, as is discussed in Section 7.3.

7.1 Simple code optimizations

To optimize all ExaHyPE kernels, I took the role of an optimization expert and relied on the code generation architecture described in Chapter 4 to work on Jinja2 templates and macros. In particular, I used optimization macros and architecture-aware template variables, as described in Section 4.4.

7.1.1 SIMD and data allocation

My main focus was to vectorize the code as much as possible using the concepts described in the previous chapter.

Vectorization

```
1 #pragma omp simd aligned(lQi_next,tmpArray:ALIGNMENT)
2 for (int n = 0; n < {{nVarPad}}; n++) {
3   lQi_next[{{idx(0,0,zyx,n)}}] -= tmpArray[n];
4 }
```

Listing 7.1: Optimization of a tensor operation in a template using vectorization with alignment and padding.

I vectorized most tensor operations by ensuring that the fastest dimension is traversed in the innermost loop, and by using the OpenMP API's SIMD pragmas to enable the compiler auto-vectorization. I also specified the alignment of the subarrays in the pragma to indicate to the compiler that it can safely use the faster aligned intructions. Likewise, if all subarrays have a padded fastest dimension, I included the padding in the loop range to accelerate its computation by removing the need for masking or scalar loop spilling as discussed in Section 6.2.2. The code excerpt from a template in Listing 7.1 illustrates these optimizations.

Tensor contractions can be better optimized by other techniques, that will be discussed in Section 7.2, and therefore are excluded from this simple optimization.

Data alignment and padding

```
1  constexpr int totalSize = kernels::getFusedSTPVISize();
2  double memory[totalSize] __attribute__((aligned(ALIGNMENT)));
3
4  double* lQi  = memory + kernels::getlQiShift();
5  double* lFi  = memory + kernels::getlFiShift();
6  constexpr double* lSi   = nullptr;
7  double* lQhi = memory + kernels::getlQhiShift();
8  double* lFhi = memory + kernels::getlFhiShift();
9  constexpr double* lShi  = nullptr;
10 constexpr double* gradQ = nullptr;
11 double* rhs  = memory + kernels::getrhsShift();
```

Listing 7.2: Allocation in the glue code of the temporary tensors used by the STP kernel.

I allocated all temporary tensors used by the kernels as aligned flattened one-dimensional zero-padded arrays. To ensure data locality and save allocation instructions, they are allocated as one memory block, and I then defined each tensor with pointer arithmetic as a subarray of this block, as illustrated in Listing 7.2 with an example of the STP kernel's temporary tensors allocation. Unused tensors are defined as `nullptr` to preserve the kernel signature and allow the compiler to optimize them away safely. Using the `allocateArray` and `freeArray` macros, shown in Listings 4.14 and 4.15, in Section 4.4.3, by default the memory block is allocated on the stack using `__attribute__((aligned(ALIGNMENT)))`. However, if the stack size is too limited and cannot be set by the user, the macros are used to allocate it instead on the heap with `_mm_malloc` and free it with `_mm_free` afterwards. The size of the memory block and the pointer shifts required to get the tensors' subarrays are defined as `constexpr int` in a header file generated by the Kernel Generator to be tailored toward the kernel specification.

All other tensors that persist between time steps, or need to be communicated between cells, most notably the local solution, its update, and the projections at the cell boundary necessary for the RiemannSolver kernel, are also allocated as aligned flattened one-dimensional zero-padded arrays, with the exception of the local solution `luh` that is kept

without padding. They are allocated at the start of the program using Peano's allocator, as they need to be handled by it. The lack of padding in `luh` is to maintain the API with the other parts of the engine such as the plotter, AMR or the limiter, that do not expect `luh` to be padded. It does not cost too much performance as it is used in the STP kernel only once to initialize the local temporary arrays that are themselves padded, and when updating the solution using a simple vectorized loop.

7.1.2 Precomputation and numerical optimization

Exploiting the benefits of code generation, I hard-coded all variables known at the time of the code generation, thus allowing the compiler to better tune its heuristics. This includes user defined parameters such as the order of the problem or its dimensionality, as well as derived parameters taking the target architecture into account, such as the padded tensors' leading dimension. Furthermore, using template branching, unused parts of the kernels are automatically removed from the generated source code. For example, if the application only uses a flux term in its PDE, then in the STP kernel, computing the gradient of the state is useless and therefore spared.

Then, I performed simple numerical optimizatizations by replacing costly floating point operations with cheaper mathematically equivalent ones. For example, I replaced all divisions by using a multiplication with the inverse, either hard-coded, precomputed or computed once at runtime depending on when the parameter is defined. The same is done for other simple optimizations, such as the precomputation at the start of the kernel of some coefficient matrices multiplied by a runtime variable, e.g. the cell length dx or time step increment dt. I also performed relevant loop fusion when possible, mostly on tensor traversal along the spatial coordinates.

Moreover, as the optimized code is generated for a given order, I used the Kernel Generator to precompute and generate all coefficient vectors and matrices used by the numerical scheme, e.g. the quadrature nodes and weights or the stiffness matrix. I then used them in the kernels, instead of the equivalent generic multidimensional arrays that used their slowest dimension to specify the order, thus saving the kernels one indirection each time they are used. Furthermore, they are allocated as aligned, and the matrices are zero-padded to match the layout of the tensors and allow better vectorization as described earlier.

Finally, I also precomputed common products and inverses. For example, the cross product of the quadrature weights w in the two or three dimensions used by the application can be precomputed as a w^3 tensor without padding (`weights3`) such that $w^3_{z,y,x} = w_z \cdot w_y \cdot w_x$ in three dimensions and $w^3_{0,y,x} = w_y \cdot w_x$ in two dimensions. This way, I not only spare redundant multiplications, but also have one array that encodes products of weights independently of the application's dimensionality to be able to write dimension-agnostic templates.

7.2 Loop-over-GEMM

Some tensor operations can be reformulated into sequential matrix multiplications on tensor slices. Thus, they are better optimized by using an efficient *General Matrix Multiplication function*, or GEMM, from a dedicated BLAS library, than with the simple innermost loop vectorization. This *Loop-over-GEMM* (LoG) scheme was explored among others by Di Napoli et al. [91] and by Shi et al. [92], and is used in other PDE solvers such as SeisSol [93].

7.2.1 Extraction of matrix slices from a tensor

Tensors are stored as flattened one-dimensional arrays. I take advantage of this data structure to extract a slice from the tensors without memory operations by using an offset and a non-unit stride. In particular, two-dimensional slices, or *matrix slices* can easily be extracted.

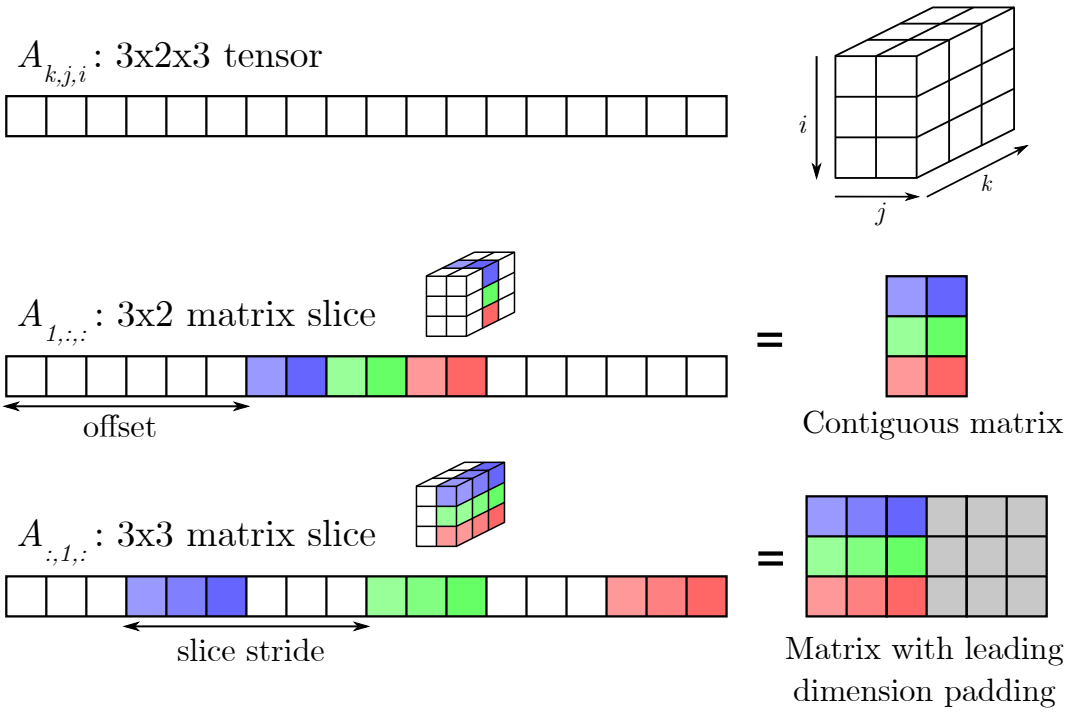


Figure 7.1: Extraction of matrix slices from a three-dimensional tensor without memory operations, using offsets and non-canonical outer strides.

Figure 7.1 illustrates such processes in a given three-dimensional tensor $A_{i,j,k}$. The slicing is encoded using the Fortran sub-array manipulation notation with ":" denoting that this dimension is extracted. For example, $A_{1,:,:}$ represents a matrix slice from the tensor A along its j and k dimensions at $i = 1$.

To extract $A_{1,:,:}$, the slice is simply encoded in the subarray with unit stride that can be accessed from the tensor pointer with the offset corresponding to $A_{1,0,0}$. If instead the slice $A_{:,0,:}$ on i and k is required, then it is encoded in the tensor as a matrix with a non-canonical outer stride, or leading dimension size. Instead of the actual size of the k dimension, here it is using its size multiplied by the size of the j dimension, when converting the matrix two-dimensional coordinates to a one-dimensional index. Finally, if the fastest dimension, here k, is not included in the slice, one could use a non-unit inner stride but as this heavily degrades SIMD vectorization, I avoided this situation entierly in ExaHyPE.

Most BLAS libraries support selecting a bigger outer stride for a matrix than the default one defined as the dimension's size. For example with Eigen's `OuterStride` template parameter or LIBXSMM's leading dimensions `LDA`, `LDB` and `LDC` parameters. Thus by using this to specify a non-canonical outer stride and with pointer arithmetic for the slices' offsets, no extra memory operation is required to manipulate and work on matrix slices of a tensor.

7.2.2 Loop-over-GEMM

The computation-heavy tensor operations performed in the kernels take the form of the multiplication of a matrix along one spatial dimension of the tensors. This tensor contraction is analog to a matrix multiplication, but with higher dimensional tensors. Mathematically these contractions represent, for example, computing the derivative of a tensor with the discrete derivative operator obtained by using the mathematical properties of the Gauss-Legendre quadrature nodes.

To illustrate this, let us consider a four-dimensional tensor `Q` with three spatial dimensions x, y and z of size `nDof`, and a variable dimension n of size `nVar`, i.e. $Q_{z,y,x,n}$. The computation of its gradient's component along the x dimension `gradQx`, also a four-dimensional tensor of the same size ($\nabla Q^x_{z,y,x,n}$), can be performed using a tensor contraction with a constant discrete derivative operator matrix `dudx` ($\Delta_{x,l}$) that is determined by the quadrature properties.

$$\forall z, y, x, n,\ \nabla Q^x_{z,y,x,n} = \sum_l \Delta_{x,l} \cdot Q_{z,y,l,n}$$

Using the template index macro introduced in Section 4.4.2 this is done with 5 nested loops:

```
1 for (int z = 0; z < {{nDof}}; z++)
2   for (int y = 0; y < {{nDof}}; y++)
3     for (int x = 0; x < {{nDof}}; x++)
4       for (int l = 0; l < {{nDof}}; l++)
5         for (int n = 0; n < {{nVar}}; n++)
6           gradQx[{{idx(z,y,x,n)}}] +=   dudx[x*{{nDof}}+l]
7                                       * Q[{{idx(z,y,l,n)}}];
```

This can be rewritten as operations on matrices by taking slices of the tensors along the x dimension with the variable dimension as second one for the slices since it is the fastest one in the tensor. As discussed earlier, I can obtain these for free with an offset and pointer arithmetic:

```
1  for (int z = 0; z < {{nDof}}; z++)
2    for (int y = 0; y < {{nDof}}; y++) {
3      double* Qs   = Q     +{{idx(z,y,0,0)}};
4      double* gQxs = gradQx+{{idx(z,y,0,0)}};
5      for (int x = 0; x < {{nDof}}; x++)
6        for (int l = 0; l < {{nDof}}; l++)
7          for (int n = 0; n < {{nVar}}; n++)
8            gQxs[x*{{nVar}}+n] +=   dudx[x*{{nDof}}+l]
9                                  * Qs[l*{{nVar}}+n];
10   }
```

In the three innermost loops on the slices, we recognize a matrix multiplication on row-major ordered matrices. With a function `matmul_rm(A,B,C)` to perform $C+ = A \cdot B$ with row-major order, the tensor contraction can be rewritten as:

```
1  for (int z = 0; z < {{nDof}}; z++)
2    for (int y = 0; y < {{nDof}}; y++)
3      matmul_rm(dudx, Q+{{idx(z,y,0,0)}}, gradQx+{{idx(z,y,0,0)}});
```

Thus the tensor contraction along the x dimension can be performed with matrix multiplications on all matrix slices of the tensors along the x dimension and the variable dimension,

$$\forall z, y,\ \nabla Q^x_{z,y,:,:} = \Delta_{:,:} \cdot Q_{z,y,:,:}.$$

However, most BLAS libraries implement their GEMM in column-major order. As a matrix in row-major order is equivalent to the transposed matrix written in column-major order, and $(A \cdot B)^T = B^T \cdot A^T$, a function `gemm(A,B,C)` that performs a GEMM with colum-major order can still be used here by swapping the inputs A and B, and ensuring that all matrices are written in row-major order.

```
1  for (int z = 0; z < {{nDof}}; z++)
2    for (int y = 0; y < {{nDof}}; y++)
3      gemm(Q+{{idx(z,y,0,0)}}, dudx, gradQx+{{idx(z,y,0,0)}});
```

On the other hand, if we wanted to get `dQy`, the gradient component along the y dimension, then we can follow the same steps and get:

```
1  for (int z = 0; z < {{nDof}}; z++)
2    for (int x = 0; x < {{nDof}}; x++) {
3      double* Qs   = Q     +{{idx(z,0,x,0)}};
4      double* gQys = gradQy+{{idx(z,0,x,0)}};
```

```
5       for (int y = 0; y < {{nDof}}; y++)
6         for (int l = 0; l < {{nDof}}; l++)
7           for (int n = 0; n < {{nVar}}; n++)
8             gQys[y*{{nVar*nDof}}+n] +=    dudy[y*{{nDof}}+l]
9                                         * Qs[l*{{nVar*nDof}}+n];
10    }
```

Once more, a matrix multiplication can be recognized, and a column-major GEMM used, but this time the leading dimensions of the matrices A and C do not match the lengths of the corresponding `y`- and `l`-loops. The corresponding slices can be obtained without extra memory manipulation as described previously by configuring non-canonical outer strides for A and C in the GEMM, here `nVar*nDof` for both instead of `nVar`. Thus the tensor contraction can be performed as:

```
1  for (int z = 0; z < {{nDof}}; z++)
2    for (int x = 0; x < {{nDof}}; x++)
3      gemm(Q+{{idx(z,0,x,0)}}, dudy, gradQy+{{idx(z,0,x,0)}});
```

Likewise, we would get a similar reformulation along the z dimension.

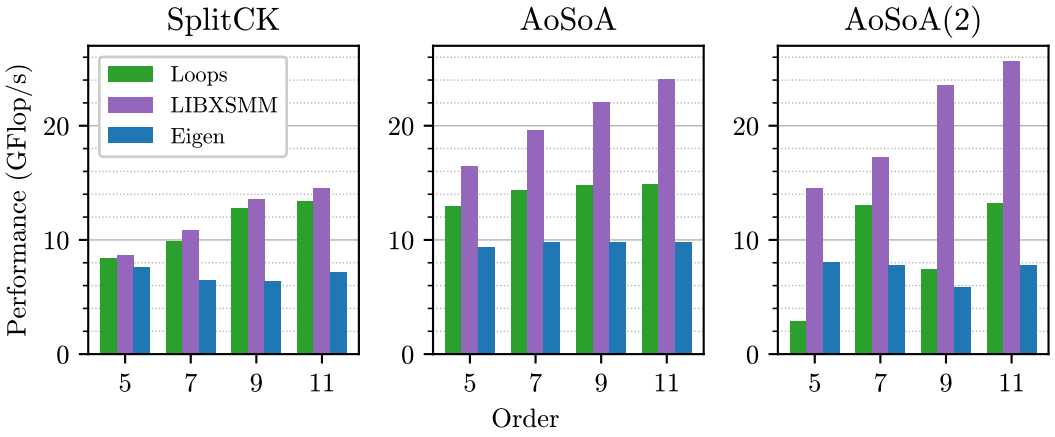
> Tensor contractions along a given dimension can be performed with loops over matrix multiplications of slices of the tensors in the given dimension.

This technique is called *Loop-over-GEMM* (LoG). It was shown to increase performance over the naive tensor contractions by Di Napoli et al. [91], by how much depending on quality the GEMM used.

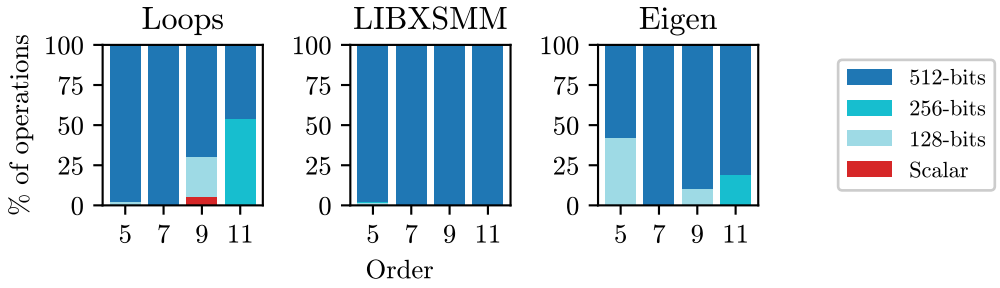
7.2.3 Optimizing the GEMM

To optimize the matrix multiplication, I used the `matmul` macro presented in Section 4.4.4. This macro can render the operation using a vectorized three loops C++ implementation of the matrix multiplication. Moreover, if configured to do so, it can also delegate the matrix multiplication to a BLAS library. I implemented support for two BLAS libraries: LIBXSMM [44] and Eigen [41]. LIBXSMM produces a `gemm` function encapsulating highly-tuned assembly code, whereas Eigen relies on C++ templates expression and metaprogramming to have the compiler generate the matrix multiplication during compilation.

Figure 7.2 compares the performance achieved by the same benchmark application for each of the three options and three STP kernel variants, introduced in Chapter 8. Eigen performed worse in almost all cases, as it is not specifically designed for our use case of small dense highly vectorizable GEMM, that both the loops implementation and LIBXSMM can fully exploit. In the SplitCK benchmarks, the default vectorized loops implementation achieved results close to those with LIBXSMM, as the padding ensured that the subarrays in the innermost loop are always aligned. However, the AoSoA(2) case changes the padding scheme, preserving the subarrays' alignment only at order 7,



(a) Performance



(b) Floating point operations packing

Figure 7.2: **(a)**: Comparison of the performance achieved by each matrix multiplication implementations supported by the `matmul` macro, performed using the linear benchmark at order 5, 7, 9 and 11 with the SplitCK (left), AoSoA (middle) or AoSoA(2) (right) STP kernel variant. **(b)**: Distribution of the packing size used to perform the floating point operations with AoSoA(2) for each implementation.

and thus greatly degrading the performance at the other orders tested. The distribution of the packing sizes used to perform the floating point operations reflects this, with only LIBXSMM being fully vectorized with AVX-512, the two other implementations using less efficient 128- and 256-bits SIMD instructions outside of order 7.

LIBXSMM produced the best results in all cases, hence justifying its choice as the default BLAS library used by the Kernel Generator. As discussed with the addition of Eigen in Section 5.3.2, an optimization expert could add support for another BLAS library if desired.

7.3 Compiler related optimizations

The concepts used to write structured and easily readable source code often go against the ways of directly getting a highly optimized machine code. To avoid this conflict, modern compilers are very efficient at optimizing source code during compilation, with, for example, their auto-vectorization capability.

Therefore, I activated and enabled multiple compiler's optimization capabilities to get better performance without much modifications to the source code itself. However, doing so can be highly compiler specific. In this work, I focused on the Intel C++ Compiler (ICPC) version 19 [94].

7.3.1 Compiler's flags for auto-vectorization

I relied on the compiler auto-vectorization capabilities to vectorize the code, by marking the relevant loops with OpenMP API's SIMD pragmas. However, some compiler's flags also need to be set up for the compiler to properly vectorize the code. Hence, I adapted the Toolkit's generated configuration Makefile and the global ExaHyPE's Makefile that are presented in Section 3.3.5.

First, the optimization level needs to be set at least at level 2 with `-O2`. As using one level higher enables more aggressive loop transformations and data analysis when combined with other flags used, and since ExaHyPE's performance is dominated by floating point operations in loops, I set the optimization level to the third one with `-O3`.

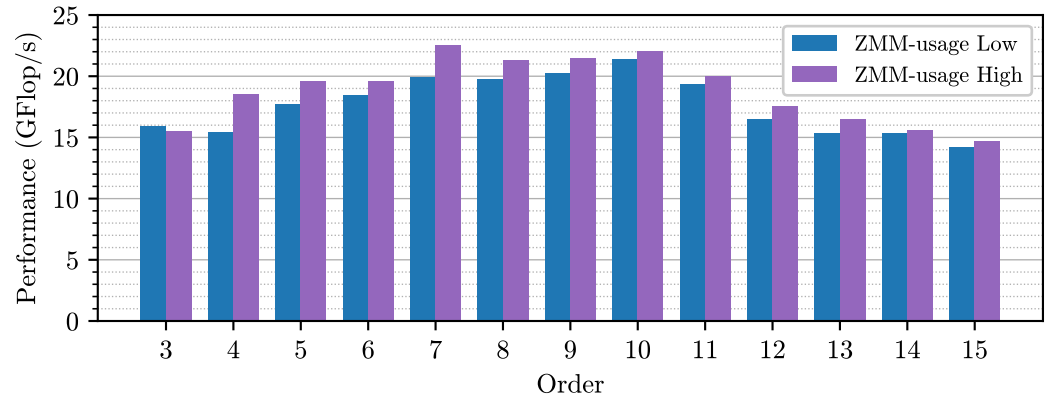
Furthermore, as the OpenMP API is used, it needs to be loaded by the compiler using the `-qopenmp-simd` flag.

Then, the compiler needs to know which SIMD instruction set is supported by the target architecture. This information is given using the `-x` flag appended with the architecture code. For an Haswell CPU where the newest supported SIMD instruction set is AVX2, the flag is then `-xCORE-AVX2`. Whereas, for a newer Skylake CPU that supports AVX-512 instructions, the flag is `-xCORE-AVX512`.

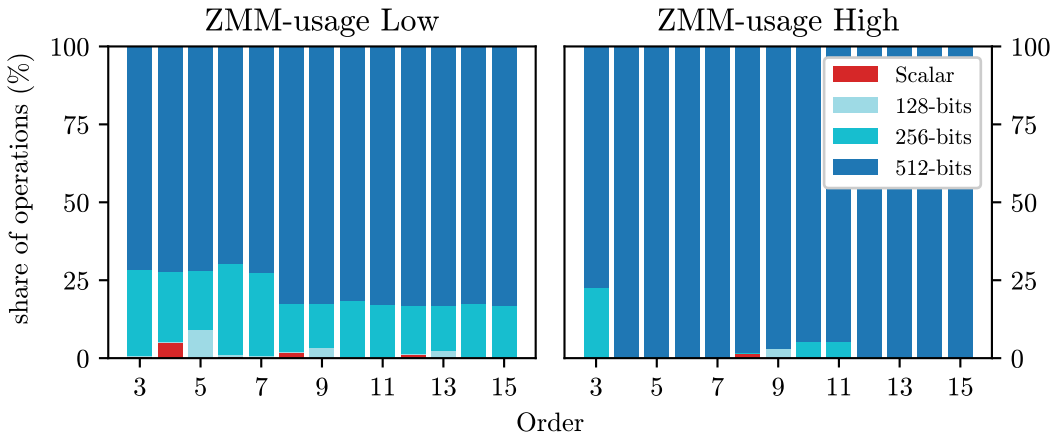
Finally as described in Section 6.1.2, I use data alignment and the specific byte boundary required depends on the target architecture. As this also impacts code that is not generated in ExaHyPE's core, it is set on a custom preprocessor macro `ALIGNMENT`, that is defined and set at compilation with the correct value, e.g. on Haswell with 32 bytes alignment the flag is `-DALIGNMENT=32`.

7.3.2 Compiler heuristics with AVX-512

As mentioned in Section 6.1.3, on Skylake the ICPC compiler tends to favor vectorization with AVX2 over the more efficient AVX-512 to lessen the CPU frequency slowdown associated with SIMD instructions. Intel's Optimization Manual recommends disabling this feature on heavily vectorized applications using `-qopt-zmm-usage=high` [89].



(a) Performance



(b) Floating point operations packing

Figure 7.3: Comparison of the low and high settings for the `-qopt-zmm-usage` compiler flag using a benchmark with the nonlinear AoSoA kernel variant, introduced in Section 8.7. **(a)**: Measured performance in GFlop/s for both settings at order 3 to 15. **(b)**: Distribution of the packing size used to perform the floating point operations.

Figure 7.3 shows metrics from the same benchmark application compiled with both the default `low` setting and the `high` one. With the exception of order 3, where AVX2 is particularly suitable as the application is vectorized over 4 elements, the `high` setting results in almost all floating point operations being performed with AVX-512 instructions. On the other hand the `low` setting performs 17% to 22% of them with AVX2. This difference comes from the auto-vectorization of the user functions. The AVX-512 instructions in the `low` ZMM-usage case are caused by the optimization of the kernel's matrix multiplications using LIBXSMM, which relies on generated assembly code always using AVX-512 instructions independently of the compiler setting.

This translates to an increase in performance with the `high` setting, as the CPU frequency in this benchmark was fixed at 2.5GHz. As the code is fully vectorized with this kernel variant, and with LIBXSMM using AVX-512, the speedup using the high setting is expected to stay even without a fixed CPU frequency.

8 Optimization of the SpaceTimePredictor kernel

The SpaceTimePredictor (STP) kernel is where most of the runtime of an application is spent. This chapter presents a sequence of optimizations I performed on the linear version of the STP kernel. Each introduced a variant tackling the main bottlenecks from the previous one. Of the five variants introduced here, the first four variants were presented in a previous publication [95].

First, in Section 8.2, the linear STP kernel and its generic implementation are introduced and optimized following the methods described previously, in particular using Loop-over-GEMM (LoG) to compute the costly tensor contractions. The optimized LoG kernel variant suffered from a performance bottleneck caused by a high amount of memory stalls. To solve it, together with Leohnard Rannabauer, we introduced a new formulation of the algorithm aiming at reducing the memory footprint of the kernel and improve its cache awareness, in Section 8.3. Then, after the good results of the new formulation, I used a hybrid data layout to solve the data layout conflict preventing the vectorization of the user functions. This new variant is presented and benchmarked in Section 8.4. Finally, I pushed the idea behind the hybrid data layout further to greatly reduce the required padding for the data alignment and also to improve the vectorization of the user functions, at the cost of removing some performance related optimizations in the kernel itself. The performance of the five variants is analyzed in Section 8.6.

Following this optimization of the linear STP kernel, I adapted and replicated the concepts used there to the nonlinear STP kernel. This process is outlined in Section 8.7, with a focus on the differences compared to the linear case.

8.1 Experimental setup

I compared every introduced STP kernel variant on the same identical experimental setup, which is outlined in this section.

To measure the performance of the STP kernel variants precisely, I introduced a dummy solver benchmark setup to run a given STP kernel in isolation, with user functions and input data taken from real three-dimensional applications:

- **Linear kernel** - the ExaSeis application LOH1 setup without PML, presented in Section 5.1.

- **Nonlinear kernel** - the compressible Navier-Stokes model used for cloud simulation with the two air bubbles collision scenario, outlined in Section 2.1.3.

Using the initial data provided, the dummy solver prepares a given amount of cells, here 800, to emulate the workload of a single-thread in a hybrid MPI+TBB ExaHyPE application. It then applies its given STP kernel variant on each cell to emulate a time step, and performs multiple time steps as would be the case in ExaHyPE. Unlike in a true simulation, the cells are not evolved, but this is not relevant to the performance metrics measured.

The STP kernel being a single-threaded function, the benchmark executable runs on a single core. To emulate a true ExaHyPE application and realistically reproduce the memory bandwidth usage occuring in this case, the benchmark executable was started multiple times in parallel, with each process pinned to a separate core to saturate the CPU. I then measured the performance metrics of one of these processes by counting specific hardware performance events with the performance tool LIKWID [96].

To validate the observations made during the benchmarks of the kernels, I also performed benchmarks of the two proper applications emulated by the dummy solver setups, using ExaHyPE. They used shared-memory parallelization only, to avoid a sweet spot issue caused by Peano's multi-node parallelism. However, Peano's task-based shared-memory parallelization of ExaHyPE deteriorates when the tasks are not costly enough, which occurs in low-order simulations. In large production runs, this can be avoided by running multiple ranks on each node, each using a fraction of the available cores, in Peano's intended hybrid MPI+TBB approach. These issues and the hybrid approach are discussed by Charrier [36]. Therefore to be able to analyze the performance of low-order simulations, they were performed on 4 or 8 cores each, whereas high-order simulations used all 24 cores of a socket. In a similar fashion to what was done with the single-thread benchmarks, to emulate multiple ranks per node of a large ExaHyPE run, I saturated the node by starting multiple processes in parallel, and then measured the performance metrics on the whole node.

I carried out all benchmarks on the SuperMUC-NG cluster at Leibniz Supercomputing Centre. SuperMUC-NG uses two Intel Xeon Platinum 8174 CPUs[1] per node, with 24 cores per socket running at a fixed 2.5GHz in benchmark mode with the Energy Aware Runtime tool off at the time of the tests[2]. Each core has two AVX-512 FMA units [89], and thus the available performance per core is estimated at $2.5 \cdot 2 \cdot 2 \cdot 8 = 80$ double-precision GFlop/s.

Tests were compiled using the Intel Compiler (version 19.0.5) with the optimization flags `-O3`, `-std=c++11`, `-ip`, `-xCORE-AVX512`, `-qopenmp-simd`, and `-qopt-zmm-usage=high`, as discussed in Section 7.3. The kernels were generated with LIBXSMM to perform the matrix multiplications abstracted by the `matmul` macros.

[1] `https://doku.lrz.de/display/PUBLIC/Details+of+Compute+Nodes`
[2] This frequency was also measured by LIKWID.

8.2 Generic kernel and LoG optimized variant

8.2.1 Linear SpaceTimePredictor kernel

In the ADER-DG scheme described in Section 2.2, the SpaceTimePredictor (STP) kernel is responsible for evolving a cell independent of the others. Therefore, this kernel requires no communication, and thus multiple cells are computed in parallel using the Peano framework's TBB parallelization to keep all cores busy. Hence, the goal of the optimization process is to exploit as much performance as possible from a single core.

The STP kernel itself is made of three successive steps, to which we add the former VolumeIntegral kernel as an integrated fourth step.

The inputs of the the STP kernels are:

- the local solution tensor `luh`,
- the coordinates of the cell's center,
- its dimensions, we assume a cubic cell, i.e. $dx = dy = dz$,
- the current simulation time T,
- the current time step increment ΔT,
- multiple temporary tensors to store intermediate results.

Its outputs are:

- the prediction projected at the cell boundary, used later by the RiemannSolver: `lQhbnd` for the state tensor and `lFhbnd` for the flux one,
- the first part of the correction term `lduh` produced by the VolumeIntegral using previous intermediate results.

For simplicity, the notation in the following algorithms ignores the dimension of the tensor storing the variables of the application, as it is always the fastest running index and the same operations are applied to each variable. Input and output tensors are written in typewriter font, e.g. `luh`, whereas temporary tensors for intermediate results use mathematical notations, e.g. Q.

(1) Taylor expansion in time with a Cauchy-Kowalewsky procedure

The Cauchy-Kowalewsky procedure outlined in Algorithm 2 is used to compute a local Taylor expansion in time (tensor Q) of the solution (`luh`). It also stores the time derivatives of the associated flux F and source S as intermediate results. The last iteration of the time-loop is only used to compute the last time derivative in F and S.

Algorithm 2 Cauchy-Kowalewsky procedure

for $x, y, z \leftarrow 0$ to $N-1$ **do** // Initialize Q with input luh
  $Q_{0,z,y,x} \leftarrow \texttt{luh}_{z,y,x}$
end for

for $t \leftarrow 0$ to $N-1$ **do** // CK time-loop

  for $x, y, z \leftarrow 0$ to $N-1$ **do** // Flux
    $F_{:,t,z,y,x} \leftarrow \texttt{flux}(Q_{t,z,y,x})$
  end for
  for $d \leftarrow 0$ to 2 **do** // Differentiate flux F in place
    $F_{d,t,:,:,:} \leftarrow \frac{\partial F_{d,t,:,:,:}}{\partial x_d}$
  end for

  for $d \leftarrow 0$ to 2 **do** // Compute gradient of Q
    $\nabla Q_{d,:,:,:} \leftarrow \frac{\partial Q_{t,:,:,:}}{\partial x_d}$
  end for
  for $x, y, z \leftarrow 0$ to $N-1$ **do** // NCP, added to F
    $F_{:,t,z,y,x} \leftarrow F_{:,t,z,y,x} + \texttt{ncp}(Q_{t,z,y,x}, \nabla Q_{:,z,y,x})$
  end for

  for $x, y, z \leftarrow 0$ to $N-1$ **do** // Source, stored in S
    $S_{t,z,y,x} \leftarrow \texttt{source}(Q_{t,z,y,x})$
  end for

  if $t < N-1$ **then** // skip update in the last iteration
    for $x, y, z \leftarrow 0$ to $N-1$ **do** // Compute the next iteration's Q
      $Q_{t+1,z,y,x} \leftarrow -S_{t,z,y,x} - \sum_{d=0}^{2} F_{d,t,z,y,x}$
    end for
  end if

end for

(2) Predictor

Using the time derivatives stored in Q, a prediction of the solution $\overline{Q}$ is made by performing a time average of the derivatives, as described in Algorithm 3. The same is done to obtain the predicted flux $\overline{F}$ and source $\overline{S}$ terms.

(3) Extrapolator

Using the predictions and projection vector r and l, the predicted values of the solution at the boundary `lQhbnd` of the cell are computed, as shown in Algorithm 4. Likewise the

Algorithm 3 Predictor: make a prediction by averaging the tensor Q, F and S over time ($\overline{A}$ is the time-averaged tensor A)

for $x, y, z \leftarrow 0$ to $N-1$ **do**

$\overline{Q}_{x,y,z} \leftarrow \sum_{t=0}^{N-1} \frac{\Delta T^t}{(t+1)!} Q_{t,z,y,x}$ // Average Q

for $d \leftarrow 0$ to 2 **do**
$\overline{F}_{d,z,y,x} \leftarrow \sum_{t=0}^{N-1} \frac{\Delta T^t}{(t+1)!} F_{d,t,z,y,x}$ // Average each direction of F
end for

$\overline{S}_{z,y,x} \leftarrow \sum_{t=0}^{N-1} \frac{\Delta T^t}{(t+1)!} S_{t,z,y,x}$ // Average S (for VolumeIntegral)

end for

Algorithm 4 Extrapolator: project the averaged tensor on the cell boundary for the faceIntegral kernel later

for $y, z \leftarrow 0$ to $N-1$ **do** // Projection in x
$\texttt{lQhbnd}_{0,z,y} \leftarrow \sum_{x=0}^{N-1} \left(\overline{Q}_{z,y,x} \cdot l_x\right)$
$\texttt{lQhbnd}_{1,z,y} \leftarrow \sum_{x=0}^{N-1} \left(\overline{Q}_{z,y,x} \cdot r_x\right)$
$\texttt{lFhbnd}_{0,z,y} \leftarrow \sum_{x=0}^{N-1} \left(\overline{F}_{0,z,y,x} \cdot l_x\right)$
$\texttt{lFhbnd}_{1,z,y} \leftarrow \sum_{x=0}^{N-1} \left(\overline{F}_{0,z,y,x} \cdot r_x\right)$
end for

for $x, z \leftarrow 0$ to $N-1$ **do** // Projection in y
$\texttt{lQhbnd}_{2,z,x} \leftarrow \sum_{y=0}^{N-1} \left(\overline{Q}_{z,y,x} \cdot l_y\right)$
$\texttt{lQhbnd}_{3,z,x} \leftarrow \sum_{y=0}^{N-1} \left(\overline{Q}_{z,y,x} \cdot r_y\right)$
$\texttt{lFhbnd}_{2,z,x} \leftarrow \sum_{y=0}^{N-1} \left(\overline{F}_{1,z,y,x} \cdot l_y\right)$
$\texttt{lFhbnd}_{3,z,x} \leftarrow \sum_{y=0}^{N-1} \left(\overline{F}_{1,z,y,x} \cdot r_y\right)$
end for

for $x, y \leftarrow 0$ to $N-1$ **do** // Projection in z
$\texttt{lQhbnd}_{4,y,x} \leftarrow \sum_{z=0}^{N-1} \left(\overline{Q}_{z,y,x} \cdot l_z\right)$
$\texttt{lQhbnd}_{5,y,x} \leftarrow \sum_{z=0}^{N-1} \left(\overline{Q}_{z,y,x} \cdot r_z\right)$
$\texttt{lFhbnd}_{4,y,x} \leftarrow \sum_{z=0}^{N-1} \left(\overline{F}_{2,z,y,x} \cdot l_z\right)$
$\texttt{lFhbnd}_{5,y,x} \leftarrow \sum_{z=0}^{N-1} \left(\overline{F}_{2,z,y,x} \cdot r_z\right)$
end for

flux at the boundary `lFhbnd` is also calculated. These two boundary values are outputs of the kernel and used in the next step of ADER-DG in the RiemannSolver.

Algorithm 5 VolumeIntegral: compute the volume integral part of the correction tensor *lduh*

for $x, y, z \leftarrow 0$ to $N-1$ **do**
  $\texttt{lduh}_{z,y,x} \leftarrow -w_z w_y w_x \left(\overline{S}_{z,y,x} + \sum_{d=0}^{2} \overline{F}_{d,z,y,x}\right)$
end for

(4) Integrated VolumeIntegral kernel

Furthermore, I fused the STP kernel with the VolumeIntegral kernel described in Algorithm 5, i.e. the first step of the correction step, as it uses the time averaged flux and source tensors computed by the predictor that are not needed afterward as input. It produces the first part of the correction tensor `lduh` that is the last output of the STP kernel.

8.2.2 Customizable generic kernel with template metaprogramming

The generic kernel is provided by the engine without code generation involved, and thus needs to be able to support the whole canonical PDE (2.1) system. However, in many cases an application only requires a simpler PDE system, that can be obtained by using only some of the PDE terms. Therefore, it is necessary to be able to disable the rest to avoid unnecessary computations. To do so without having access to code generation for the kernel itself, I used C++ templating with template metaprogramming, as presented in Section 3.3.2. This ensures a fair and relevant comparison between the generic kernel and generated kernels, where the unused parts of the numerical scheme are omitted during the rendering of the source code.

8.2.3 Optimization of the generic kernel

Unlike a generic kernel, an optimized one is produced on demand by the Kernel Generator. Thus, it is further tailored toward both the application and the target architecture.

As described in Section 4.4, the optimized kernel takes the form of an algorithmic template in the Kernel Generator and is optimized using Jinja2's templating functionalities and optimization macros. To optimize this kernel, I applied the technique described in Sections 7.1 and 7.2:

- Known constants are hard-coded.
- Loop fusion is performed where applicable.
- Basic coefficient multiplications are replaced by precomputed values.
- Qualified-name lookup is used to bypass the vtable.
- Tensor and vector operations are vectorized automatically using OpenMPI's pragmas.

- Tensors contractions are reformulated as a Loop-over-GEMM with an optimized implementation of matrix multiplication abstracted by the `matmul` macro.

Using the loop fusions, index macros, and template branching logic, I also made the algorithmic template dimension-agnostic to only have one version of it for both two and three-dimensional applications instead of the two generic versions.

Due to the kernel being optimized with the LoG formulation of tensor contractions, this optimized variant is refered as the *LoG* variant.

8.2.4 Results

Using the dummy solver setup described in the previous section, I compared both the generic kernel and the LoG optimized variant on the same application.

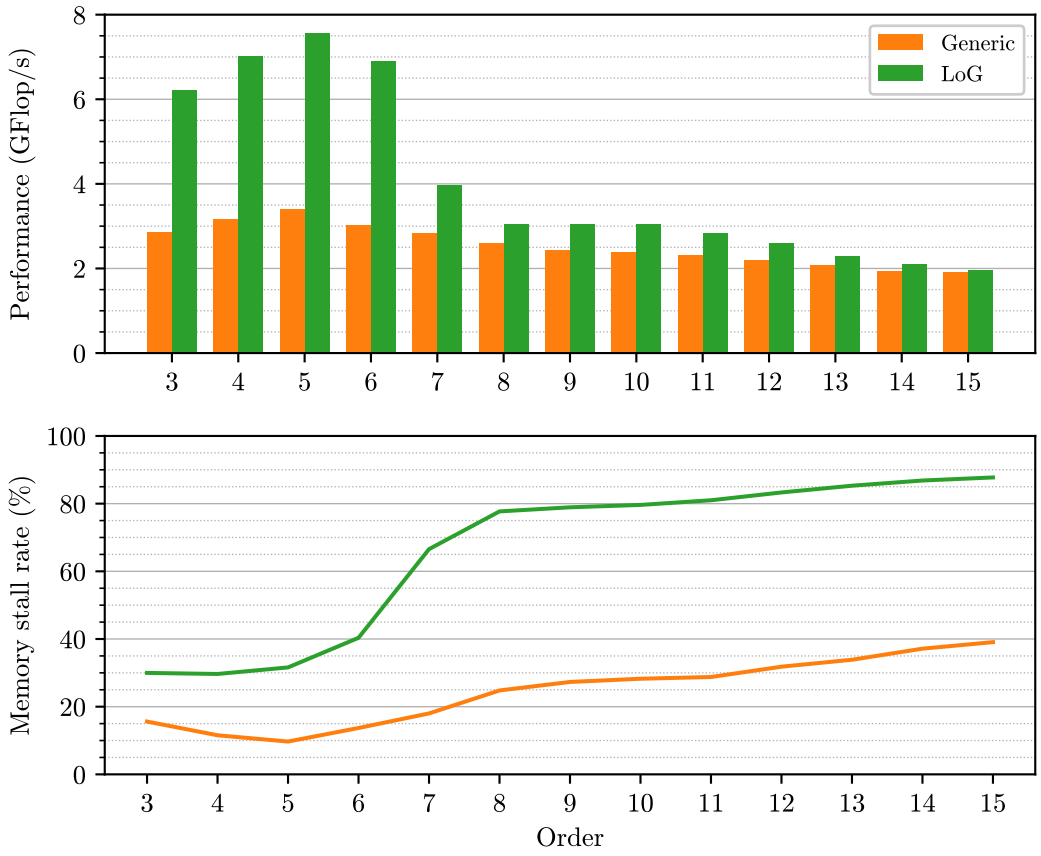


Figure 8.1: Performance and memory stall rate measurements of the Generic and LoG kernels with the LOH1 dummy solver benchmark for order 3 to 15.

Figure 8.1 shows the reported performance. As expected the performance of the generic kernel is quite low, peaking at 3.4 GFlop/s, and slowly decreasing to 1.9 GFlop/s. This

low performance was expected, due to the code being largely not vectorized by the compiler, and matches reported results in other HPC softwares [97, 46].

In contrast, in the LoG variant, over half of the arithmetic instructions are AVX-512 ones, which translates to over 90% of the floating point operations being performed using SIMD, with the scalar operations mostly coming from the scalar user functions. Compared to the tensor contraction optimized with LIBXSMM, the computational cost of the user function increases at a slower rate with the order, improving the vectorization ratio as the order increases. Therefore, the performance starts higher and increases with the order at first up to 7.6 GFlop/s, close to 10% of peak performance. However, despite the improving vectorization as the order increases further, the performance sharply declines after order 6 and converges toward the performance of the generic kernel.

This shows us that the code is not compute-bound at high order, and it is confirmed by measurements of the memory stall rate also shown in Figure 8.1. It increases past order 5, rapidly in the LoG case and in direct correlation with its loss in performance. At high order, the memory stall rate in the LoG benchmark is above 80%, making the code memory-bound.

8.3 Reduced memory footprint: SplitCK

With memory stalls being the main bottleneck of the LoG variant, together with Leonhard Rannabauer, we introduced a new formulation of the linear STP kernel to reduce its memory footprint using a sum factorization approach. Similar approaches are used in other PDE solver frameworks, such as [98, 99, 100, 101].

This new kernel variant is implemented as a new view, i.e. algorithmic template, that can be selected in the `SpaceTimePredictorModel` instead of the previous one to produce the kernel. The development process of this new variant is presented in Section 5.2.2, whereas this section focuses instead on the algorithm re-engineering itself and the evaluation of its performance.

8.3.1 Cause of the previous bottleneck

During the previous benchmarks, the LoG variant showed a high amount of memory stalls once the order increased beyond 7. It was already observed in previous performance studies of the full engine by Charrier et al.[102], that the high memory footprint of the kernels can impede performance.

Table 8.1 shows all tensors potentially used in the STP kernel, leaving out coefficient matrices negligible compared to the tensors. The formulas to compute their size are given using N the number of degrees of freedom equal to the polynomial order of the application plus one, d the dimension of the problem, ν the number of variables, and p the number of parameters, with $\lceil x \rceil$ denoting that x is increased to the next multiple of the vector register length, here 8, due to padding requirements.

Tensor		Memory footprint	Description
`luh`		$(\nu+p)N^d$	input solution
`lduh`		$\lceil\nu\rceil N^d$	output correction
`lQhbnd`		$2d\lceil\nu+p\rceil N^{d-1}$	output face-projected prediction
`lFhbnd`		$2d\lceil\nu\rceil N^{d-1}$	output face-projected flux
`lQi`,	Q	$\lceil\nu+p\rceil N^{d+1}$	state tensor with derivatives in time
`gradQ`,	∇Q	$d\lceil\nu\rceil N^d$	only of the current time derivative
`lQhi`,	$\overline{Q}$	$\lceil\nu+p\rceil N^d$	time-averaged prediction
`lFi`,	F	$d\lceil\nu\rceil N^{d+1}$	flux tensor with derivatives in time
`lFhi`,	$\overline{F}$	$d\lceil\nu\rceil N^d$	time-averaged flux prediction
`lSi`,	S	$\lceil\nu\rceil N^{d+1}$	source with derivatives in time
`lShi`,	$\overline{S}$	$\lceil\nu\rceil N^d$	time-averaged source prediction

Table 8.1: Input, output, and temporary tensors used in the linear STP kernel with flux, non-conservative product, and source terms. N is the number of degrees of freedom, d the dimension of the problem, ν its number of variables, and p its number of parameters. $\lceil x\rceil$ indicates that x is increased to the next multiple of the vector register length due to padding.

Our test application uses all of them with the exception of both source term related tensors. Summing all tensors used and ignoring padding, a lower bound of the kernel's memory footprint is:

$$\nu\left((d+1)N^{d+1}+(3+2d)N^d+4dN^{d-1}\right)+p\left(N^{d+1}+2N^d+2dN^{d-1}\right).$$

At order 7, when a large performance loss was measured previously, the values for our test application are $N=8$, $d=3$, $\nu=9$, and $p=16$, and thus the total memory footprint of the array is above 283904 double-precision numbers which represents over 2.2 MB. With the padding virtually increasing ν or $\nu+p$ to the next multiple of 8 in most tensors, in the LoG case the memory footprint of the kernel at order 7 is above 3.4 MB, growing from around 2.1 MB at order 6.

The benchmarks were performed on Intel Xeon Platinum 8174 CPUs, i.e. Skylake. The details on Skylake's cache sizes and performance are given in Intel's Architectures Optimization Reference Manual [89]. Each Skylake core has 1 MB of L2 cache available, and on average 1.375 MB of L3 cache per core, assuming the socket is saturated. Thus, at order 7, the kernel's total memory footprint exceeds the available cache size and cache overflow is to be expected beyond order 6.

Therefore, at high order the previous optimizations cannot be fully exploited due to cache misses. The code stops being compute-bound and is instead dominated by memory stalls.

This motivated a reformulation of the algorithm to reduce its memory footprint with the goal of making it fit in the cache at higher orders. As the code is memory-bound, slightly increasing the amount of computations is an acceptable cost if it allows a sharp decrease in memory footprint.

8.3.2 Algorithm reformulation

The reformulation of the kernel's algorithm focused mostly on the Cauchy-Kowalewsky procedure, as it is the one using the biggest tensors identified in Table 8.1: Q (`lQi`), F (`lFi`), and S (`lSi`). We used three key ideas to remove or reduce the size of the intermediary result tensor required.

Exploiting the linearity of the user functions

First, we exploited the linearity of the applications computed by this kernel and used a sum factorization approach. As the source is linear, $\lambda\texttt{source}(Q_1) + \mu\texttt{source}(Q_2) = \texttt{source}(\lambda Q_1 + \mu Q_2)$. Thus, using the notations from Algorithms 2 and 3:

$$\begin{aligned}
\overline{S}_{:,:,:} &= \sum_{t=0}^{N-1} \frac{\Delta T^t}{(t+1)!} S_{t,:,:,:} \\
&= \sum_{t=0}^{N-1} \frac{\Delta T^t}{(t+1)!} \texttt{source}(Q_{t,:,:,:}) \\
&= \texttt{source}\left(\sum_{t=0}^{N-1} \frac{\Delta T^t}{(t+1)!} Q_{t,:,:,:} \right) \\
&= \texttt{source}\left(\overline{Q}_{:,:,:}\right)
\end{aligned}$$

The source prediction $\overline{S}$ can be recomputed in one step like a regular source contribution using the predicted solution $\overline{Q}$ as input of the source user function. The same is true for the flux prediction using also the linearity of the differentiation performed when adding its contribution to the current state derivative.

Hence, the whole `lSi` tensor used to store the evaluated fluxes of the state tensor `lQi` in the predictor is not required anymore. Instead we directly recomputed the time-averaged `lShi` from the time-averaged `lQhi`, and since `lShi` only contributes to the update tensor `lduh` we can directly add its contribution, removing both source tensors. Likewise, we can spare `lFi` and use only `lFhi` as temporary tensor in the modified Cauchy-Kowalewsky procedure, as the flux still has to be evaluated in the predictor to evolve the state tensor. This recomputation removes some of the biggest tensors used.

Evaluating each direction independently

Both the flux and non-conservative product user functions evaluate all three spatial directions at once, thus using inputs and outputs storing all three dimensions. By splitting each function into three separate functions, each responsible for one direction, we reformulated the scheme into a split one, where each direction is evaluated one after the other. With this the "direction" dimension (index d in Algorithm 2) of the related tensors, here `lFhi` and `gradQ`, was not needed anymore and we reused the same smaller tensor to store the intermediary results of each direction.

On the fly time integration in Cauchy-Kowalewsky

Finally, in the Cauchy-Kowalewsky procedure, the time-loop used to compute the time derivatives only needs the current time derivative to compute the next. Thus, instead of storing them all in a big tensor `lQi` (Q) to then compute the time-average in the next step, we performed the time averaging into the `lQhi` tensor on the fly at the end of each time loop iteration. Only two tensors, both without a time dimension, are required, one for the current time derivative, the now smaller `lQi`, and a new one for the next time derivative, `lQi_next` (Q^*). Having these two smaller tensors requires less memory than the previous bigger one, as its a time dimension is always bigger than two.

8.3.3 Algorithm of the SplitCK scheme

Using the three ideas, we introduced a three step *SplitCK* scheme. The first step is a modifed Cauchy-Kowalewsky procedure. The second is the extrapolator on the predicted solution only. The last is a fusion of the previous last three to recompute the contribution of the predicted flux and source from the predicted solution.

As the user API was changed by the split user function anyway, we also took the opportunity to separate the parameters from the variables, both being previously stored together in the same dimension of `lQi` and its associated tensors. The parameters being constant, they can be stored once in a smaller tensor `lPi` (P) passed as argument to the user functions instead of being copied in all time derivatives of the state tensor Q and its associated tensors, that now only store variables, further reducing the memory footprint of the kernel.

Algorithm 6 shows the new split Cauchy-Kowalewsky procedure in the first step of the kernel. It uses a reduced memory footprint to directly compute the prediction $\overline{Q}$, but does not keep in memory the intermediate results it computed. We only need to perform the time-loop $N-1$ times, as the last time-loop performed in the former version of the algorithm was only used to compute the associated flux and source derivatives, since the zeroth derivative is known from the initialization.

The predicted solution is then used to compute the `lQhbnd` output in the second step similar to the one in Algorithm 4.

Algorithm 6 Split Cauchy-Kowalewsky procedure, predict $\overline{Q}$ by integrating on the fly and do not keep intermediary results in memory

for $x, y, z \leftarrow 0$ to $N-1$ **do** // Initialize Q, $\overline{Q}$ and P with input *luh*
  $Q_{z,y,x} \leftarrow \texttt{luh}_{z,y,x}|_\nu$
  $\overline{Q}_{z,y,x} \leftarrow \texttt{luh}_{z,y,x}|_\nu$
  $P_{z,y,x} \leftarrow \texttt{luh}_{z,y,x}|_p$
end for

for $t \leftarrow 0$ to $N-2$ **do** // CK time-loop

  for $d \leftarrow 0$ to 2 **do** // Split dimensions for Flux and NCP
    for $x, y, z \leftarrow 0$ to $N-1$ **do** // Flux, use $\overline{F}$ to store raw flux
      $\overline{F}_{z,y,x} \leftarrow \texttt{flux}_d(Q_{z,y,x}, P_{z,y,x})$
    end for
    $Q^*_{:,:,:} \leftarrow \frac{\partial \overline{F}_{:,:,:}}{\partial x_d}$ // Add the flux derivative contribution

    $\nabla Q_{:,:,:} \leftarrow \frac{\partial Q_{0,:,:,:}}{\partial x_d}$ // ∇Q only stores one dimension
    for $x, y, z \leftarrow 0$ to $N-1$ **do** // Non Conservative Product, added to Q^*
      $Q^*_{z,y,x} \leftarrow Q^*_{z,y,x} + \texttt{ncp}_d(Q_{z,y,x}, P_{z,y,x}, \nabla Q_{z,y,x})$
    end for
  end for

  for $x, y, z \leftarrow 0$ to $N-1$ **do** // Source, added to Q^*
    $Q^*_{z,y,x} \leftarrow Q^*_{z,y,x} + \texttt{source}(Q_{z,y,x}, P_{z,y,x})$
  end for

  for $x, y, z \leftarrow 0$ to $N-1$ **do** // Add next derivative Q^* to time average $\overline{Q}$
    $\overline{Q}_{z,y,x} \leftarrow \overline{Q}_{z,y,x} + \frac{\Delta T^t}{(t+1)!} Q^*_{z,y,x}$
  end for
  $Q_{z,y,x} \leftarrow Q^*_{z,y,x}$ // Replace current Q with next derivative Q^*

end for

Then using this prediction, the last step in Algorithm 7 computes directly the flux prediction $\overline{F}$ for one direction using directed user functions for the flux and the ncp terms. The flux prediction contribution to the outputs is immediatly computed and then the tensors are reused for the next direction. Finally the source term is recomputed and directly added to the update tensor.

Algorithm 7 Recomputation of flux and source

$\texttt{lduh} \leftarrow 0$
for $d \leftarrow 0$ to 2 **do** // Split dimensions

  for $x, y, z \leftarrow 0$ to $N-1$ **do** // Flux
    $\overline{F}_{z,y,x} \leftarrow \texttt{flux}_d(\overline{Q}_{z,y,x}, P_{z,y,x})$
  end for
  $\overline{F}_{:,:,:} \leftarrow \frac{\partial \overline{F}_{:,:,:}}{\partial x_d}$ // Differentiate flux in place

  $\nabla \overline{Q}_{:,:,:} \leftarrow \frac{\partial \overline{Q}_{0,:,:,:}}{\partial x_d}$ // $\nabla \overline{Q}$ only stores one dimension
  for $x, y, z \leftarrow 0$ to $N-1$ **do** // Non Conservative Product, added to $\overline{F}$
    $\overline{F}_{z,y,x} \leftarrow \overline{F}_{z,y,x} + \texttt{ncp}_d(\overline{Q}_{z,y,x}, P_{z,y,x}, \nabla \overline{Q}_{z,y,x})$
  end for

  for $i, j \leftarrow 0$ to $N-1$ **do** // Projection in d
    $\texttt{lFhbnd}_{2d,i,j} \leftarrow \sum_{k=0}^{N-1} \overline{F}_{0,\sigma(i,j,k)} l_k$
    $\texttt{lFhbnd}_{2d+1,i,j} \leftarrow \sum_{k=0}^{N-1} \overline{F}_{0,\sigma(i,j,k)} r_k$
  end for
  for $x, y, z \leftarrow 0$ to $N-1$ **do** // Volume Integral contribution
    $\texttt{lduh}_{z,y,x} \leftarrow \texttt{lduh}_{z,y,x} - w_z w_y w_x \overline{F}_{z,y,x}$
  end for

end for

for $x, y, z \leftarrow 0$ to $N-1$ **do** // Source directly added to volume integral
  $\texttt{lduh}_{z,y,x} \leftarrow \texttt{lduh}_{z,y,x} - w_z w_y w_x \texttt{source}(\overline{Q}_{z,y,x}, P_{z,y,x})$
end for

8.3.4 Memory footprint reduction and computational cost

Table 8.2 shows all tensors used by the reformulated SplitCK variant. If a source term is used, then it is computed without requiring a temporary tensor to store intermediate results. This reduced the lower bound of the memory footprint without padding to:

$$\nu \left(7N^d + 4dN^{d-1}\right) + p \left(2N^d + 2dN^{d-1}\right).$$

Table 8.3 shows the memory footprint of the tensors used in our benchmarks by the LoG and SplitCK variants at orders 3 to 15. In the LoG variant, the cache becomes full after order 6. Whereas, in the SplitCK variant, when including inputs and outputs tensors that are only used at the beginning and end of the kernel, the 1 MB limit of the L2 cache is only crossed at order 9, and the full cache becomes too small only after order 12, assuming each core gets the same share of it.

Tensor		Memory footprint	Description
`luh`		$(\nu+p)N^d$	input solution
`lduh`		$\lceil\nu\rceil N^d$	output correction
`lQhbnd`		$2d\lceil\nu+p\rceil N^{d-1}$	output face-projected prediction
`lFhbnd`		$2d\lceil\nu\rceil N^{d-1}$	output face-projected flux
`lPi`,	P	$\lceil p\rceil N^d$	constant parameters
`lQi`,	Q	$\lceil\nu\rceil N^d$	state tensor current derivatives in time
`lQi_next`,	Q^*	$\lceil\nu\rceil N^d$	state tensor next derivatives in time
`gradQ`,	∇Q	$\lceil\nu\rceil N^d$	only of the current time derivative
`lQhi`,	$\overline{Q}$	$\lceil\nu\rceil N^d$	time-averaged prediction
`lFhi`,	$\overline{F}$	$\lceil\nu\rceil N^d$	time-averaged flux prediction

Table 8.2: Input, output, and temporary tensors used in the reformulated SplitCK linear STP kernel variant, with N the number of degrees of freedom, d the dimension of the problem, ν its number of variables, and p its number of parameters.

Order	3	4	5	6	7	8	9
LoG (kB)	288	627	1205	2,114	3,462	5,372	7,983
SplitCK (kB)	108	195	320	489	709	986	1,327

Order	10	11	12	13	14	15
LoG (kB)	11,449	15,940	21,639	28,748	37,482	48,071
SplitCK (kB)	1,738	2,226	2,798	3,460	4,218	5,080

Table 8.3: Memory footprint in kB of the linear STP kernel main tensors, as used in the benchmark application, for the LoG and SplitCK variants.

The difference in computational cost between the two variants depends on the application. Inside the kernel, the recomputations in SplitCK in Algorithm 7 are equivalent in cost to the last time-loop not performed anymore in the Cauchy-Kowalewsky procedure. However, more calls to user functions are made, and thus if some variable transformations are required inside them, for example to change from the conserved quantities given as input to prime quantities, then these transformations need to be duplicated in each split user function, increasing the overall application cost[3].

8.3.5 Results

Figure 8.2 compares the performance achieved by the LoG and SplitCK variants, as well as their memory stall rates. As seen earlier, the LoG variant starts to degrade past order 6 with increasing memory stall rates as the kernel memory footprint grows beyond the

[3]For example, the nonlinear astrophysic model FO-CCZ4 requires costly variable conversions [103].

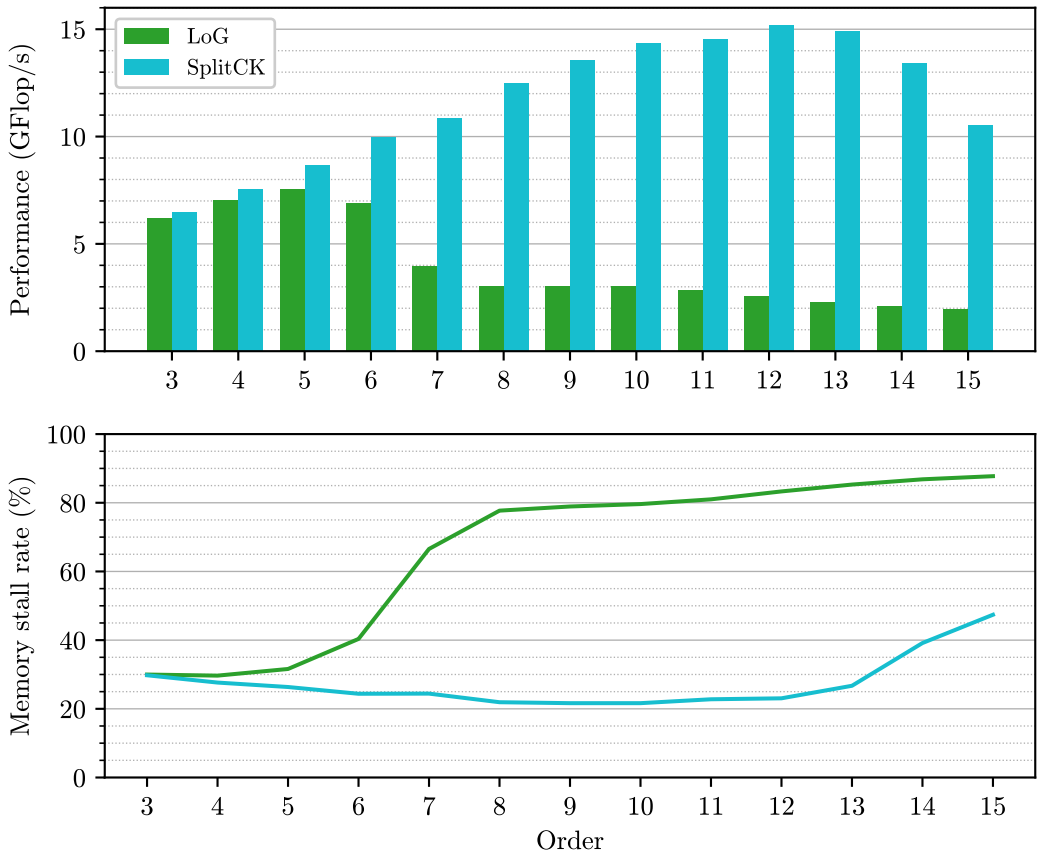


Figure 8.2: Performance and memory stall rate measurements of the LoG and SplitCK kernels with the LOH1 dummy solver benchmark for order 3 to 15.

available cache size. In constrast to this, the SplitCK variant's performance continues to increase up to order 12, at which point it starts to decrease, due to the memory stall rate increasing in parallel.

At its peak at order 12, the SplitCK variant reaches 15.2 GFlop/s, which represents 19% of peak performance.

8.4 Hybrid data layout: AoSoA

With the memory stall bottleneck removed by the SplitCK scheme, further performance increases can be obtained by vectorizing the remaining scalar operations taking place in the user functions. Thus, with this new kernel variant, I introduced a new user API to allow the application experts to write vectorized user functions, as illustrated in Section 5.1.3.

8.4.1 Data Layout Conflict

By default, the user functions are scalar functions processing one single quadrature node. The first reason for this is to have a simpler user interface for the application experts implementing the user functions. As this new variant is optional and expands the SplitCK scheme, that already modified the user interface, this was not a concern for this new variant.

The second reason lies in the classical *Array-of-Structures* (AoS) versus *Structure-of-Arrays* (SoA) data layout conflict.

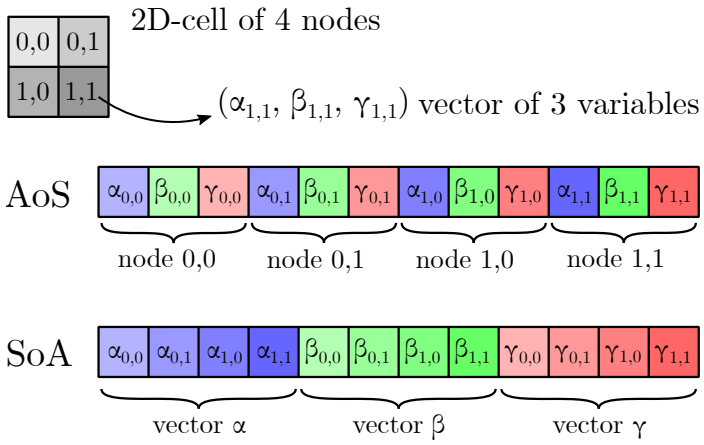


Figure 8.3: Illustration of the AoS vs SoA data layout. AoS groups the variable by position in the cell, whereas SoA write groups the nodes by variable.

A given tensor array A in ExaHyPE stores multiple quadrature nodes, each holding multiple variables used to describe the simulated physical phenomenon, e.g. local water column height and its momentum in each direction in a shallow water simulation, or compute the numerical scheme, i.e. the flux of each of the stored variables for the flux tensor. Figure 8.3 illustrates how this tensor A can be stored in an array. In an AoS data layout, each node is stored after the other, thus the tensor index order is $A_{z,y,x,n}$ with the spatial coordinates (z, y and x) being the slowest and the variable dimension (n) being the fastest. An SoA data layout stores first all the first variable's instances, then the second variable's instances and so on, resulting in the order $A_{n,z,y,x}$.

In the kernels, all operations only take the coordinates into account and are applied in the same way to all variables, thus the natural data layout is AoS, allowing the vectorization on the variable dimension with unit-stride performed in the previous kernel variants. However, the opposite happens in the user functions, where the variable dimension is the critical one and the operations are repeated on all spatial coordinates, as is seen in the user function of ExaSeis in Section 5.1. Therefore, to vectorize them efficiently, unit-stride arrays of each variable should be provided, which corresponds to an SoA data layout. The kernels are the dominant part in the total the runtime for most applications,

and since the user functions are written by users, so not always optimized, an AoS data layout for all tensors was chosen at the start of the project.

8.4.2 On the fly transpositions

A straightforward way to get around this conflict is to perform on the fly transpositions of the tensors when calling a user function to switch the inputs to an SoA data layout, and then transpose back the SoA outputs to the expected AoS data layout, as discussed in Section 5.3.1.

However, the cost of performing the necessary transpositions is often too high, making this only worthwhile for applications with costly PDE system like the FO-CCZ4 astrophysics simulations where a speedup of up to 1.27 on Haswell architecture (AVX2) was reported [23]. For simpler systems, a performance loss can even be observed, as was the case with ExaSeis.

8.4.3 Hybrid data layout

This data layout conflict is commonly seen in many numerical solvers. In the PyFR framework, it is solved by a hybrid data layout [104]. Likewise the optimization of tensor operations in YATeTo uses hybrid data layouts [48].

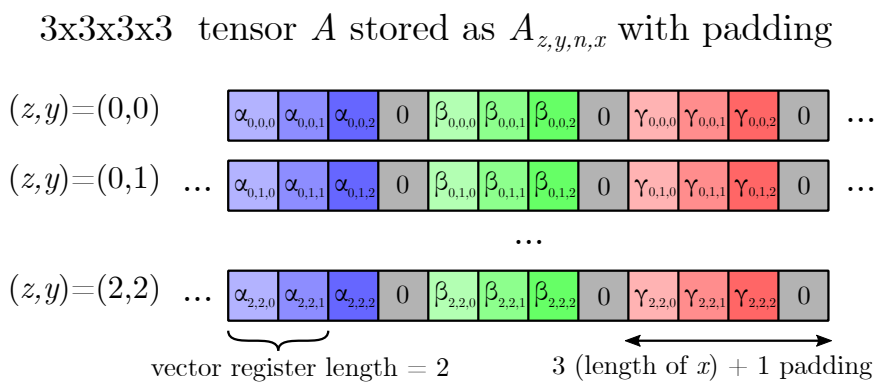


Figure 8.4: Illustration of the AoSoA data layout for a tensor A with three spatial dimensions, each of length 3, and three variables α, β, and γ. AoSoA stores in AoS, on two spatial dimensions, slices of the tensor stored in SoA on the remaining spatial dimension. The fastest dimension is zero-padded to the next multiple of the vector register length, in this example 2.

Such a data layout, illustrated in Figure 8.4, is often called *Array-of-Structures-of-Arrays* (AoSoA), and is sometimes used to optimize SIMD operations when taking cache behavior into account [105, 106]. In its simplest form, it uses one spatial dimension to store data slices in SoA and then stores these in AoS on the remaining spatial dimensions. This translates to having the variable dimension in between the spatial coordinate ones. Using

the previous example of a tensor A, in AoSoA it is stored as $A_{z,y,n,x}$, with one spatial dimension being the fastest, then the variable one and finally the other spatial ones. In ExaHyPE, the fastest dimension is zero-padded to the next multiple of the vector register length to ensure alignment. While this layout is slightly less intuitive to work with, it allows the kernels to keep working with a pseudo-AoS layout and to trivially extract small SoA slices for the user functions.

AoSoA in the user functions

With the AoS data layout, calling a user function translates to looping over all spatial coordinates to then call it on a single quadrature node. I performed this in the previous variants using pointer arithmetics to obtain the subarray representing a single quadrature node. The following code excerpt shows the call of the non-conservative product in the x direction user function with the here fused nested loops over the spatial coordinates:

```
1 for (int xyz = 0; xyz < 216; xyz++) {
2   solver.Elastic::EWSolver::nonConservativeProduct_x(lQi+xyz*16,
      ↪ nullptr, gradQ+xyz*16, result);
3   #pragma omp simd aligned(lQi_next,result:ALIGNMENT)
4   for (int n = 0; n < 16; n++) {
5     lQi_next[xyz*16+n] -= result[n];
6   }
7 }
```

Instead with the AoSoA data layout, I only loop on the two slower spatial coordinates and call the user function at the start of a subarray storing two dimensions, the variable ones like before but also the first spatial dimension. The following code excerpt shows the same part of the kernel code, this time with an AoSoA data layout:

```
1 for (int yz = 0; yz < 36; yz++) {
2   solver.Elastic::EWSolver::nonConservativeProduct_x_vect(lQi+yz*72,
      ↪ nullptr, gradQ+yz*72, result);
3   #pragma omp simd aligned(lQi_next,result:ALIGNMENT)
4   for (int nx = 0; nx < 72; nx++) {
5     lQi_next[yz*72+nx] -= result[nx];
6   }
7 }
```

Each subarray used as argument of the user function is implicitly encoding a matrix as a flattened array instead of a single vector, and since its fastest dimension is a spatial one, it is stored using an SoA data layout. Therefore, using the AoSoA data layout, I trivially extract SoA slices of the tensor to pass to the user functions. The user functions can then be optimized to efficiently process their SoA inputs using vectorized operations, as was shown in Section 5.1.3 .

AoSoA in the kernel

In the kernel, tensor operations are performed using Loop-over-GEMM on matrix slices of the tensors. A matrix multiplication on the slices takes the form $C = \alpha A \cdot B + \beta C$, with A and C matrix slices from the tensors, and B a constant coefficient matrix. The dimension of the variable (n) is always one of the two dimensions in the slices. Thus, one of two cases occurs.

The first one is when the second slice dimension is the first spatial one (x). Compared to the AoS data layout on a 4 dimensional tensor, with slices in the form $(A|_{z,y})_{x,n}$, the slices in the AoSoA data layout are now transposed taking the form $(A|_{z,y})_{n,x}$. Using the linearity of the transposition operation and $(A \cdot B)^T = B^T \cdot A^T$, the matrix multiplication can be rewritten as $C^T = \alpha B^T A^T + \beta C^T$. Hence, as A and C are extracted from tensors and therefore already transposed, the same matrix multiplication operation is performed by using a transposed version of the coefficient matrix B, which I can precompute already transposed, and swapping A and B in the `matmul` macro call.

In the second case, the second slice dimension is instead another slower one. Then, as long as A and C have the same dimension lengths for their two fastest ones, dimension fusion can be used to fuse them together when taking the tensor slices for the matrices A and C. By doing so, the fact that x and n were transposed is irrelevant when considering the new nx fused dimension of the slices in the LoG scheme, and I can use the same matrix multiplication as with an AoS data layout.

Thus, when applying LoG to perform the tensor contractions on tensor stored using an AoSoA data layout, the AoSoA data layout falls back to an AoS data layout by either trivially transposing the matrix multiplication on the slice, or by dimension fusion in the slice extractions, both without any extra computation or memory operation.

8.4.4 New kernel variant

The new AoSoA variant follows the same algorithm as the SplitCK variant. Hence, I started from a copy of its template. As with the SplitCK variant I introduced a new optimization flag in the specification file JSON Schema and followed the code generation utilities data flow to pass down a corresponding context boolean to the relevant model, so that it uses the new template if the AoSoA variant is selected by the application specification file.

To avoid having to introduce AoSoA variants of all the kernels, and to preserve the compatibility with other parts of the engine, only the SpaceTimePredictor kernel uses the AoSoA data layout. Thus its inputs and outputs are in the AoS data layout.

As can be seen from Algorithm 6, in the SplitCK scheme the input is only read at the beginning to initialize the initial guess. Its transposition is performed on the fly when performing the initialization.

Likewise, the outputs are produced at the end of the kernel from the intermediary tensors and are now computed with the reverse transposition included. The boundary tensors

are a projection of the intermediary ones, by contracting one spatial dimension with a tensor vector operations, optimized using a vectorized innermost loop on the fastest variable dimension. With the transposition this cannot be done anymore as the ordering of the dimensions changes. Benchmarks showed that for the projections on the second and third spatial dimensions, still the slowest for both tensors, the best loop vectorization is to vectorize on the output's fastest dimension and let the compiler introduce a gather function on the now non-unit stride array.

```
1  for (int yz = 0; yz < {{nDof*nDof3D}}; yz++) {
2    for (int n = 0; n < {{nVar}}; n++) {
3      double tmpL = 0.;
4      double tmpR = 0.;
5      #pragma omp simd aligned(lFhi,FLCoeff,FRCoeff:ALIGNMENT)
           ↪ reduction(+:tmpL,tmpR)
6      for (int x = 0; x < {{nDofPad}}; x++) {
7        // left
8        tmpL += lFhi[{{idx(0,yz,n,x)}}] * FLCoeff[x];
9        // right
10       tmpR += lFhi[{{idx(0,yz,n,x)}}] * FRCoeff[x];
11     }
12     lFhbnd[{{idxLFhbnd(0,0,yz,n)}}] = tmpL;
13     lFhbnd[{{idxLFhbnd(1,0,yz,n)}}] = tmpR;
14   }
15 }
```

Listing 8.1: Projection of the AoSoA flux tensor on the first spatial dimension using a SIMD reduction operation.

However, for the first spatial dimension, the right-hand side tensor being projected (`lFhi`) and the coefficient vectors (`FLCoeff` and `FRCoeff`) now have the same fastest dimension. Benchmarks showed that using a SIMD reduction operation was slightly faster. An excerpt of the reduction is shown in Listing 8.1, it uses two temporary variables `tmpL` and `tmpR` to store the reduction results. The reduction is then performed using the `reduction (+:tmpL,tmpR)` argument of the OpenMP SIMD pragma, and the result is stored on the output boundary tensor.

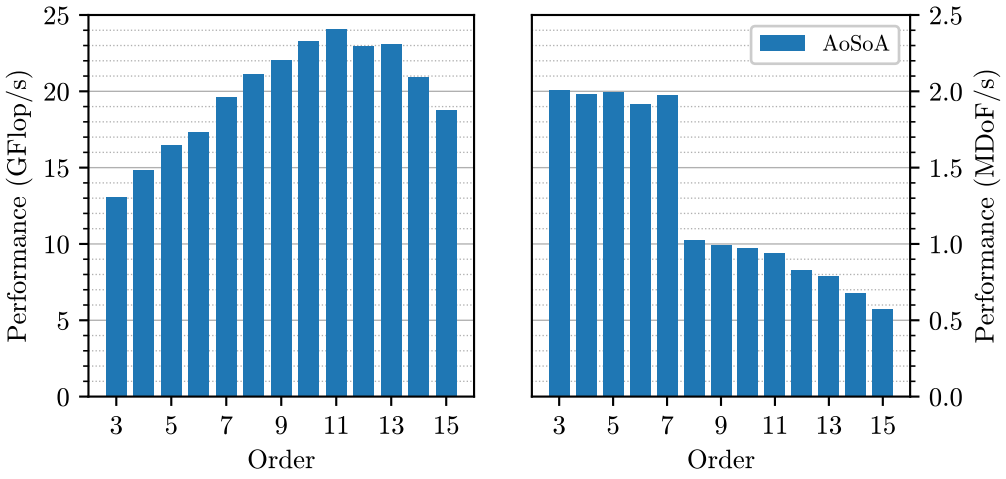
With the transpositions in place, I modified the internal components of the kernel to use the AoSoA data layout as described earlier, so that vectorized user functions can be used and the LoG optimization preserved. An important modification is also made on the tensor's padding. The fastest dimension is padded to a multiple of the SIMD register size if needed, in the AoS data layout this was usually the variable dimension, as seen in Tables 8.1 and 8.2, making padding purely application dependent and not easily modifiable as the number of variables is determined by the application's PDE system's requirements. However, for tensors in the AoSoA data layout, the fastest dimension is now the first spatial one, whose length N is equal to the polynomial order of the ADER-DG scheme plus one. Thus, with the AoSoA variant, all applications can be computed

at a sweet spot where no padding is required as long as architecture-specific polynomial orders are used

8.4.5 Results

To match the new user API, new vectorized variants of the application's user functions were implemented, as discussed in Section 5.1.3. In ExaSeis, both the vectorized flux and non conservative product user functions can safely compute on the padding of their SoA input to increase their performance, with the vectorization loop using `VectStride` as upper bound. However, the multiplication by the material parameter matrix requires the inverse of a variable, and thus the vectorized loop only iterates on non-padded data with `VectLength`. Both `VectLength` and `VectStride` are provided as compile-time known parameters in the glue code, with the former set to the length of the spatial dimension and the latter to the padded length used a outer stride for the matrices in the SoA data layout.

Using the new vectorized user functions, the benchmarks report a close to perfect vectorization, with a vectorization ratio[4] increasing with the order, from 97.4% to 99.5%.



(a) GFlop/s **(b)** MDoF/s

Figure 8.5: Performance measurements of the AoSoA kernel with the LOH1 dummy solver benchmark for order 3 to 15. **(a)**: Number of floating point operations per second (in GFlop/s) achieved. **(b)**: Number of degrees of freedom (DoF), i.e. quadrature nodes, processed per second (MDoF/s = 10^6 DoF/s).

Figure 8.5 shows the performance of the AoSoA variant both in terms of number of floating operations performed per second (Flop/s), and with the number of degrees of freedom, i.e. quadrature nodes, processed per second (DoF/s). The former is a pure

[4]Packed arithmetic instructions over all (packed and scalar) arithmetic instructions.

performance metric, that includes the computation on the padding. Whereas the latter is closer to a "science-per-second" metric.

As expected the first metric follows the same trend as with the SplitCK variant, increasing with the order before cache limitations start degrading it beyond order 12. Compared to SplitCK, the performance reached in terms of GFlop/s are even higher thanks to the close to perfect vectorization, peaking at 24.1 GFlop/s (30% of peak performance).

In theory, the nDoF/s metric is expected to be decreasing as the order increases, as each DoF is computed with an increased numerical accuracy, here seen with the longer time-loop in the Cauchy-Kowalewsky procedure described in Algorithm 6, thus counteracting the performance gain made possible by the higher arithmetic intensity. Here, this metric is at first stable, as the raw performance increases quickly enough between order 3 and 7 to compensate the additional computations required for each DoF. However, a drop occurs at order 8, where this metric is halved compared to its value at order 7. This order corresponds to a discontinuity in the evolution of the padded leading tensor dimension, jumping from a sweet spot at 8 with 0 padding for order 7 to 16, i.e. 9 plus 7 padding, at order 8. Therefore, the raw performance increase does not translate perfectly to the application runtime, as some operations are performed on the padding.

8.5 Reduced padding: AoSoA(2)

8.5.1 Issues with longer SIMD lengths

While the AoSoA kernel variant achieved very high raw performance and exploited all vectorization capabilities of the Skylake CPU, the padding required to achieve this resulted in wasting a significant amount of the performance achieved computing on the zero-padding in some cases.

Table 8.4 shows to which size the leading dimension of a tensor using the AoSoA data layout needs to be padded. Due to the AoSoA data layout, this is a spatial dimension, whose base size is the *number of degrees of freedom* (nDoF), i.e. the polynomial order used by the ADER-DG scheme increased by one. This base size, between 4 and 16, is padded to the next multiple of the size of the vector register used by the architecture's SIMD instruction set. Hence, with double-precision floating point numbers (64 bits), these are multiple of 4 for AVX2's 256-bits registers and multiple of 8 for AVX-512's 512-bits ones. If the base size is already on a sweet spot, such as nDoF being 8, then no padding is required. However, if nDoF is chosen to be 9, i.e. order 8, as much as 44% of the tensor elements is actually zero-padding on AVX-512. This jump in padding ratio between orders 7 and 8 explains the drop in performance in terms of DoF/s observed previously.

Furthermore, should an "AVX-1024" SIMD instruction set with 1024-bits vector registers be introduced, then all commonly used values for nDoF would be padded to 16, making the current padding technique unusable. This exact issue already occurs on AVX-512 when single-precision floating point numbers are used, as is explored in the experiment

	AVX2		AVX-512	
nDoF	padded nDoF	padding ratio	padded nDoF	padding ratio
4	4	0%	8	50%
5	8	38%	8	38%
6	8	25%	8	25%
7	8	13%	8	13%
8	8	0%	8	0%
9	12	25%	16	44%
10	12	17%	16	38%
11	12	8%	16	31%
12	12	0%	16	25%
13	16	19%	16	19%
14	16	13%	16	13%
15	16	6%	16	6%
16	16	0%	16	0%

Table 8.4: Size of the zero-padded leading dimension and ratio of padding over the total memory footpring in tensor using the AoSoA data layout for various leading dimension base size (nDoF) and SIMD instruction sets (AVX2 and AVX-512).

presented in Section 9.3. Single-precision floating point numbers are stored on 32 bits instead of 64, thus doubling the amount of number fitting in a vector register, emulating on AVX-512 a 1024-bits SIMD.

Removing or changing the padding is not enough to solve all the issues, as to vectorize user functions, I used the AoSoA data layout to extract slices of size nDoF plus padding stored in an SoA data layout. On an 1024-bits SIMD, the slices would always be too small to fill the vector register, thus wasting some vectorization potential of the user function.

8.5.2 New data layout

To solve this issue, I introduced a new data layout, AoSoA(2), modified from AoSoA in two ways:

1. I pushed further the ideas behind the AoSoA data layout and moved 2 spatial dimensions instead of only one. Thus a tensor $A_{z,y,n,x}$ in AoSoA becomes $A_{z,n,y,x}$ in AoSoA(2).

2. Instead of padding the leading dimension, the first two dimensions are padded as a block, so that each slice on the leading two dimensions is aligned.

Figure 8.6 illustrates the new data layout using a simple tensor with spatial dimension of size 3 being padded for a vector length of 2. It requires only 1 padding every 9 useful

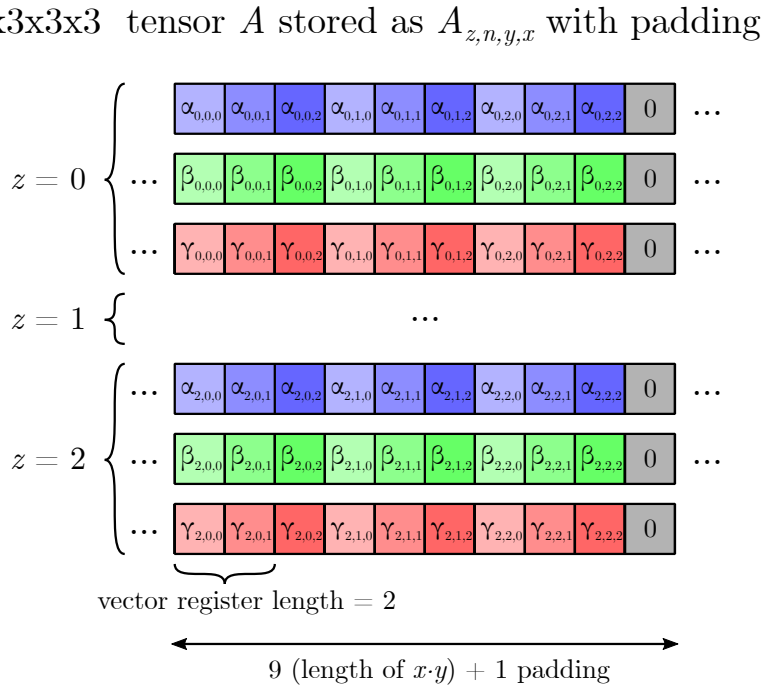


Figure 8.6: Illustration of the AoSoA(2) data layout for a tensor A with three spatial dimensions, each of length 3, and three variable α, β, and γ. Compared to AoSoA, one more dimension is swapped and the padding is performed on the fused first two dimensions, hence a reduced amount of padding here.

data, a padding ratio of 10%, less than the 1 every 3, or 25%, when using the AoSoA data layout, as previously shown in Figure 8.4.

Compared to AoS, or even AoSoA, the new AoSoA(2) data layout is more complex, which introduces some challenges. First, the order of the dimensions complicates the vectorization of kernel operations that were previously vectorized on the dimension expressing the variable, i.e. the second dimension in AoSoA, but must now be vectorized on another spatial dimension in some cases. Then, the non trivial padding complicates LoG operations and makes it impossible to extract slices on the first dimension without considering the second one. Finally, slices on the two leading dimensions are aligned but not padded thus BLAS operations may be suboptimally vectorized, as was seen in the loop implementation of the matrix multiplication in Section 7.2.3.

Nevertheless, the benefits regarding the padding ratio are significant, as can be seen in Table 8.5. With the padded size being now the square of nDoF, since two dimensions are padded together, the padding becomes much less relevant at high order. In particular for nDoF being 9, the padding ratio on AVX-512 goes from 44% with AoSoA to just 8% with AoSoA(2), which means that the total size of tensors using this layout shrinks by 25%, and the amount of floating point operations performed overall is reduced by the same

		AVX2		AVX-512		"AVX-1024"	
nDoF	$nDoF^2$	padded	pad ratio	padded	pad ratio	padded	pad ratio
4	16	16	0%	16	0%	16	0%
5	25	28	11%	32	22%	32	22%
6	36	36	0%	40	10%	48	25%
7	49	52	6%	56	13%	64	23%
8	64	64	0%	64	0%	64	0%
9	81	84	4%	88	8%	96	16%
10	100	100	0%	104	4%	112	11%
11	121	124	2%	128	5%	128	5%
12	144	144	0%	144	0%	144	0%
13	169	172	2%	176	4%	176	4%
14	196	196	0%	200	2%	208	6%
15	225	228	1%	232	3%	240	6%
16	256	256	0%	256	0%	256	0%

Table 8.5: Size of the zero-padded leading dimension and ratio of padding over the total memory footpring in tensor using the AoSoA(2) data layout for various leading dimension base size (nDoF), the first two dimensions ($nDoF^2$) being padded together, and SIMD instruction sets (AVX2, AVX-512 and "AVX-1024" representing a hypotethical future 1024-bits SIMD instruction set).

percentage. Thus despite the performance hit caused by degrading some optimizations, an overall runtime gain is to be expected on these previous worst case orders. Furthermore, the padding ratio is now acceptable even when padding is done in multiples of 16, as in a 1024-bits SIMD instruction set or in single-precision on AVX-512. Finally, the user functions are now given larger slices of data to process at once, the padded size, thus even at low order they can fill the vector registers with scientifically relevant data in most iterations and run optimally.

8.5.3 New kernel variant and user function vectorization

As with the AoSoA variant, I implemented the new AoSoA(2) STP kernel variant as a new template. To this end, I started with the AoSoA template as basis and performed the same steps to adapt the algorithm to the new data layout.

The AoSoA(2) variant uses the same user functions as the AoSoA variant, but it changes both `VectLength` and `VectStride` parameters to account for the bigger SoA slices.

8.5.4 Results

Figure 8.7 shows the performance measured on the new AoSoA(2) variant, with the results from the previous one as a reference. The effects observed are application dependent. However, we can first see that, as expected, the new variant is slightly slower at lower order

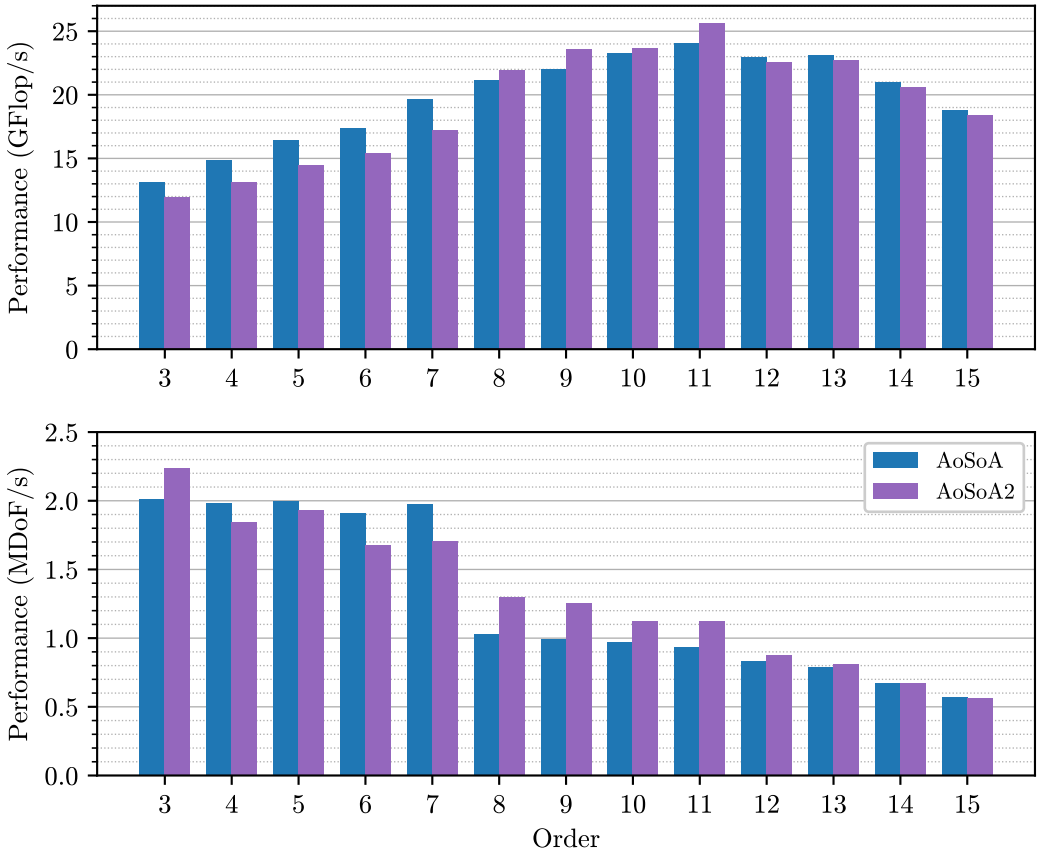


Figure 8.7: Comparison of the AoSoA and AoSoA(2) kernel variants with the LOH1 dummy solver benchmark for order 3 to 15, both in terms of floating point operations per second (GFlop/s) and degrees of freedom processed per second (MDoF/s).

as the effect of the new data layout on the padding is still minimal and the impact on the kernel LoG optimization is significant due the small slices' sizes involved. Nevertheless, as the order goes past 7, the benefits of the new data layout to the padding increases and the performance hit on the LoG optimization diminishes, resulting in overall slightly better performance.

More importantly, the new data layout shows its benefits when looking at the number of degrees of freedom processed per second, which measures the useful computations. They are less wasted on the zero-padding, and so instead of the drop at order 8, we see a more regular decreasing trend, achieving better results between order 8 and 13. At order 3, we have the same issue as at order 8 with a large padding requirement for the AoSoA data layout but a sweet spot where no padding is used with AoSoA(2). As this metric is directly correlated to the runtime of the application, in the case of the ExaSeis application, the AoSoA data layout is better at order between 4 and 7, but the AoSoA(2) one is faster after order 7 and for very low order simulations at order 3.

8.6 Evaluation of the variants

During this optimization work on the linear SpaceTimePredictor kernel, I introduced five successive variants of the kernel:

- **Generic**: Non optimized variant provided with the engine and requiring no code generation.
- **LoG**: Generated variant of the generic scheme with vectorization and Loop-over-GEMM formulation of the tensor contraction operations.
- **SplitCK**: Reformulation of the Cauchy-Kowalewsky procedure to reduce the kernel's memory footprint and increase its cache awareness.
- **AoSoA** (SplitCK): SplitCK scheme with a new hybrid data layout to solve the layout conflict between the vectorization of kernel operations and user functions.
- **AoSoA(2)** (SplitCK): Variant of the previous one pushing the new data layout further to reduce the use of padding.

Each of these variants was benchmarked with LIKWID, using a dummy solver setup for orders 3 to 15.

8.6.1 Instruction mix

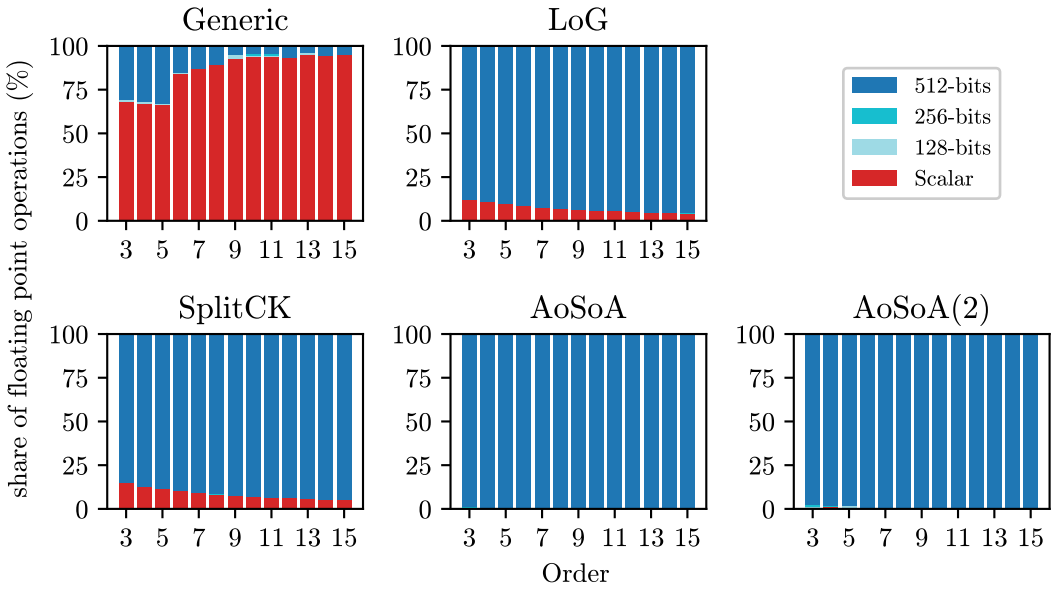


Figure 8.8: Distribution of the packing size used to perform the floating point operations with each linear kernel variant, at order 3 to 15.

Figure 8.8 shows how the floating point operations were performed in the five kernel variants at increasing orders. Scalar means that a given operation was computed using

a scalar instruction, whereas 512-bits means it was packed with 7 other operations in an single AVX-512 instruction. As compilation was performed with `-qopt-zmm-usage=high`, 128- and 256-bit packings were either not use at all, or in very low amount.

As expected, the generic variant is mostly scalar, with the compiler finding some minor[5] vectorization opportunities, especially at low order.

LoG and splitCK show similar trends, as in both cases the kernel operations are fully vectorized but not the user functions. The cost of the kernel itself increases faster with the order than the amount of user functions to call, thus the vectorization ratio improves at high order.

Finally, as expected from their design, both hybrid data layout variants exhibit an almost perfect vectorization with less than 1% of their floating point operations still being performed in scalar instructions.

8.6.2 Memory stalls

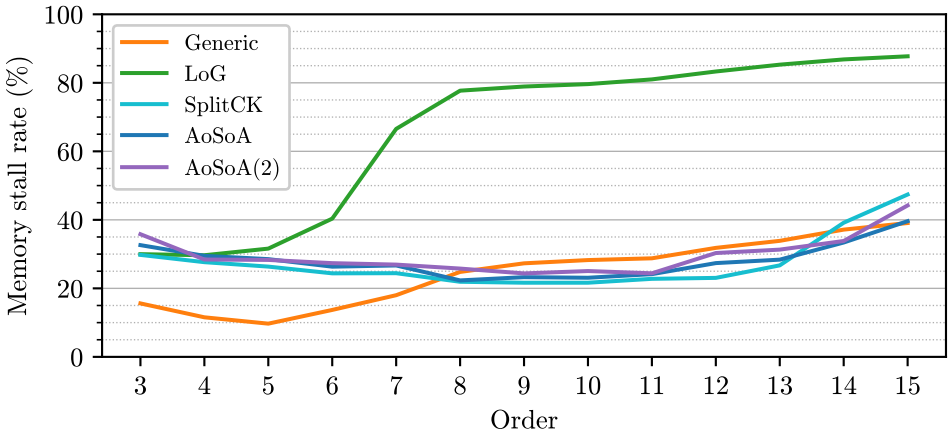


Figure 8.9: Measured memory stalls of the five kernel variants at order 3 to 15.

Figure 8.9 shows the reported memory stall rate of each setup. The generic variant being a scalar code, it shows less memory stalls despite its memory footprint largely exceeding the cache size at high order. On the contrary, requiring a similar memory footprint, but being vectorized, the LoG variant sees a sharp increase in stalls past order 5 and quickly becomes dominated by them. Hence, applications using this kernel variant at high order are not compute-bound. The splitCK reformulation of the STP algorithm proved its effectiveness, as the three variants using it follow the same trend with a steady rate of memory stalls, that only start increasing past order 11-12, as anticipated from the analysis of their memory footprint and the properties of the Intel Xeon Platinum 8174 CPUs used for these measurements.

[5] 33% of the floating point operations being performed in AVX-512 instructions means they only represent 5% of the total amount of arithmetic instructions.

8.6.3 Performance measurements

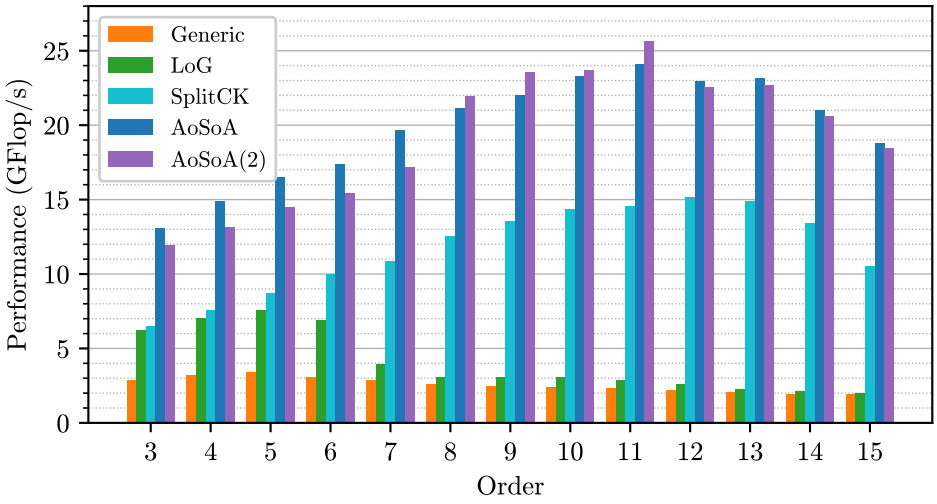


Figure 8.10: Measured performance of the five kernel variants at order 3 to 15.

The previously reported vectorization and memory stall rate measurements translate to the performance observed during the benchmarks. Figure 8.10 shows the raw performance, in GFlop/s, of all setups.

The generic kernel stays slow, reaching its peak at 3.4 GFlop/s at order 5 and then steadily losing performance, ending up at 1.9 GFlop/s at order 15. The LoG kernel is faster at first, but after order 5, memory stalls cause its performance to degrade. On the other hand, splitCK starts like the LoG variant, but thanks to its reduced memory footprint, it keeps improving its performance up to 15.2 GFlop/s (19% of peak performance) at order 12, before being affected by memory stalls too. With its hybrid data layout, the AoSoA variant manages to achieve even better results. However, its padding issues prevent it from performing optimally at its expected peak. This is solved by the AoSoA(2) variant, that reaches the best performance of all at order 11 with 25.6 GFlop/s, or 32% of peak performance.

To show how each setup uses its performance, Figure 8.11 shows the speedup in runtime it achieves compared to equivalent generic setup, which is equivalent to a normalized DoF/s graph. Despite always showing more performance than the generic kernel, the LoG variant is actually slower at high order as it also computes on padding and is not compute-bound anymore. At low order, it provides a speedup of a factor 1.6 to 1.8. Likewise, despite achieving over 7 times the performance, the splitCK variant only reaches a speedup of a factor 5.3 at its peak, which is again caused by computations being performed on the padding.

Due to their padding varying with the order, each hybrid data layout variant can be the better one at a given order. Because of the trade-offs made in the AoSoA(2) variant, this is application-dependent, and therefore, quick benchmarks should be made on new appli-

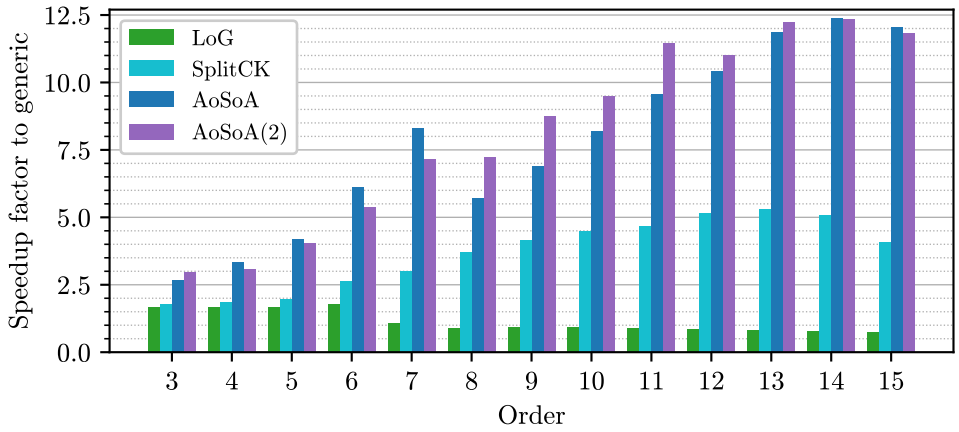


Figure 8.11: Runtime speedup of the generated kernel variants compared to the generic one at order 3 to 15.

cations to determine which setup is better at the desired order. In the case of ExaSeis, at very low order, the AoSoA variant requires a large padding and thus the AoSoA(2) variant achieves a better speedup. The AoSoA variant is faster at orders between 4 and 7 due to its superior performance, however padding issues beyond order 7 cause it to fall behind in terms of speedup more than what the raw performance results would indicate, before catching up at very-high orders when its padding becomes reasonable again.

We can see that speedup factors above 8 are achieved at high order by both hybrid data layout variants. A factor of 8 would be the theoretical limit achievable by fully vectorizing with AVX-512 a compute-bound scalar code. This proves that the generic setup is not fully compute-bound and is penalized by its large memory footprint as is clearly observed in the LoG variant magnifying this shared issue with its vectorization. It also illustrates the importance of considering the cache behavior, as the simple vectorization done in the LoG variant is unable to reach high performance, but solving hidden underlying bottlenecks of the code, by reformulating its algorithm, unlocked much more speedup.

8.6.4 Comparison of the variants using proper ExaHyPE applications

Figure 8.12 compares the performance, in terms of DoF processed per second per core, between full ExaSeis applications using different STP kernel variants, at interesting orders:

- Order **5**, before the memory footprint of the LoG variant degrades its performance.
- Order **7**, the padding sweetspot for both hybrid data layout variants.
- Order **11**, the best order for the AoSoA(2) variant in terms of GFlop/s.

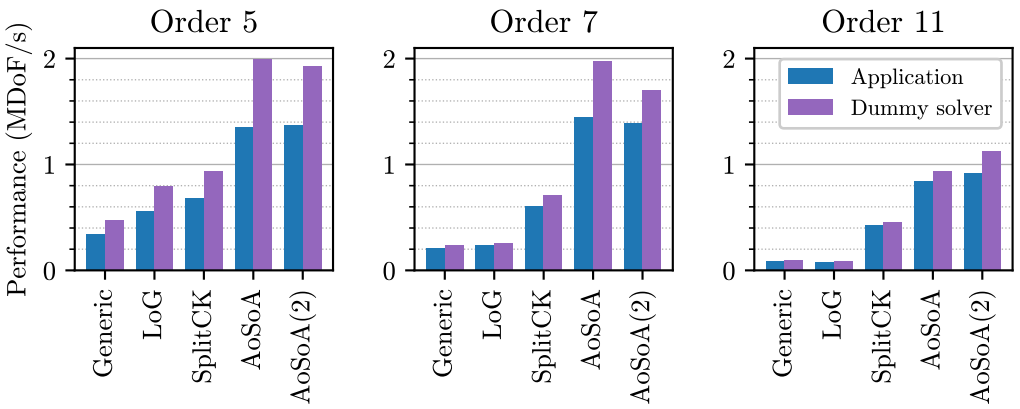


Figure 8.12: Performance in MDoF/s of the full application, with the value from the benchmarks using dummy solvers given as reference.

At each order, the generic setup uses generic kernels and the four others optimized kernels, with only the STP kernel variant used changing between the setups. For reference, the values obtained in the dummy solver benchmarks, testing only the STP kernel variants without the overhead of ExaHyPE and the other kernels, are also given.

We see exactly the same differences between the setups, both in the proper applications and in the dummy solvers setups, thus validating the use of a dummy solver to precisely benchmark the STP kernel. At order 11 the LoG variant is memory-bound, hence achieving close to the same performance in the application as in isolation, the other kernels having only a minor impact on the memory and bandwidth use compared to the STP kernel. On the other hand, the AoSoA and AoSoA(2) setups show a bigger gap in results between the application and the dummy solver benchmarks. However, this is to be expected, as they are the most performant variants, and therefore, the more impacted by a constant lower performance overhead in the form of the other kernels and ExaHyPE's parallelization overhead. They are still, by far, the most performant variants available, here accelerating the application by a factor between 3.9 at order 5 to 10 at order 11, compared to the generic kernels. Therefore, ExaHyPE's linear applications should be adapted to use them, when focusing on performance.

This speedup is especially relevant to production runs on large clusters involving thousands of CPUs, where runtimes to produce scientifically signicant results are measured in hours and the associated energy consumption in MWh. A faster application saves both scarce compute time and a significant amount of energy.

Using the 48 cores of a single node of SuperMUC-NG with 2 parallel processes, ExaSeis LOH1 benchmark at order 11 with the AoSoA(2) STP kernel reached 1066 GFlop/s, 27.8% of the node's peak performance.

8.7 Extension towards the nonlinear SpaceTimePredictor

8.7.1 Numerical scheme: Picard iterations

The nonlinear formulation of the STP kernel follows the same layout as its linear counterpart and is done in the same three steps: preparation of the prediction, prediction and extrapolation, with the added volume integral correction fused to the kernel as a fourth step.

The nonlinearity of the PDE system prevents the simple use of a Taylor expansion in time to make the prediction. Instead, a Picard iteration scheme is used to solve a fixpoint problem, as shown by Dumbser et al. [1], with the current state taken as an initial guess to start the first iteration. Empirically, we measure that in the best case only one Picard iteration is required, in cells without significant evolution, and in the worse ones the amount of iterations required to converge is around the polynomial order used by the ADER-DG scheme.

The prediction itself is then obtained from the converged guess by integrating it in time using the properties of the Gauss-Legendre quadrature, which is very similar to the averaging in time in the linear case. The extrapolation step is identical to the linear formulation. The volume integration is more complex than its linear counterpart, as this time each spatial dimension of the time averaged flux tensor has to be processed individually with a coefficient matrix and then added to the update. Nonetheless, as each is performed with a tensor contraction, this last step is easily optimized using multiple Loop-over-GEMM formulations.

As in the linear case, the first step is the dominant one with regard to the runtime and amount of computation required. However here, these cannot be predicted in advance accuratly as the number of Picard iterations for a given cell can vary as the simulation evolves in time. With respect to distributed memory parallelization, Peano's reactive load balancing techniques presented by Samfass et al. [107] can tackle these unpredictable load imbalances coming from the numerical scheme, as well as the ones caused by hardware variability.

8.7.2 LoG and basic optimizations

The first optimization step is the same as with other kernels. Using Jinja2's variable and statements the generic code is optimized and the tensor contractions are performed as Loop-over-GEMM with the `matmul` macro.

8.7.3 Reducing the memory footprint

Similarly to the linear case, the nonlinear STP memory footprint exceeds the cache size at high order and memory stalls are observed during benchmarks. Due to the Picard iterations, the full state tensor in time `lQi` cannot be integrated on the fly, as it is required for the next iteration should the current one not have converged. Splitting the

PDE function to only process one spatial dimension at a time would not help either, as the nonlinear nature of the PDE system implies that the inputs of all spatial directions are required every time.

Despite these limitations, I reduced the overall memory footprint by not storing the full flux tensor and the algebraic source tensor computed during each Picard iteration, but instead, recomputing and integrating them on the fly after the last iteration using the converged state tensor, `lQi`, resulting from the iterations. Unlike in the linear case, this recomputation of the predictor always adds floating points operations and user function calls to the overall scheme, here approximatively the equivalent of half a Picard iteration. However, this *Predictor-Recompute* (PR) variant reduces the memory footprint by around a factor 2, which can improve the overall performance enough to more than compensate the extra computations.

8.7.4 Vectorizing the user function

The next optimization step is the vectorization of the user functions using an AoSoA data layout. Before being able to use the new data layout, some specific numerical adaptations were required to the direct adaptation of the mathematical formulation used in the previous variant.

In the nonlinear case, when processing the user functions output to compute the system right-hand side tensor (R), it needs to be multiplied with weight coefficients w dependent on the coordinate used to select the slice. For example, for the contribution of the flux term, the flux component in the first spatial direction ($F_{0,:,:,:,:}$) is multiplied with the stiffness matrix K (dependent on the cell size dx) and is also multiplied by coefficients w in the other two spatial dimensions and time. Thus, the contribution of the whole flux term to the right-hand side tensor is:

$$\begin{aligned} R_{t,z,y,x,n} \leftarrow \quad & w_t w_z w_y \sum_k K_{x,k} F_{0,z,y,k,n} \\ & + w_t w_z w_x \sum_k K_{y,k} F_{1,z,k,x,n} \\ & + w_t w_y w_x \sum_k K_{z,k} F_{2,k,y,x,n} \end{aligned}$$

Each of the tensor contractions in the AoS data layout are performed using LoG with slices only in the dimension contracted. These coefficients prevent the dimension fusion trick used to perform LoG despite the AoSoA data layout for the second and third spatial dimension dimensions.

I bypassed the issue by exploiting the linearity of the operations used to compute the right-hand side tensor with a modified coefficient matrix $K'_{i,j} = w_i^{-1} K_{i,j}$, so that

$$\sum_k K_{x,k} F_{0,z,y,k,n} \;=\; w_x \sum_k K'_{x,k} F_{0,z,y,k,n},$$

and likewise for the other spatial dimentions. With the inverse weight coefficient, the tensor contractions are computed first without explicit coefficients, and then, the final tensor is multiplied by the product of all coordinates coefficient $w_t w_z w_y w_x$ to get the correct result:

$$\begin{aligned} R_{t,z,y,x,n} &\leftarrow \sum_k K'_{x,k} F_{0,z,y,k,n} + \sum_k K'_{y,k} F_{1,z,k,x,n} + \sum_k K'_{z,k} F_{2,k,y,x,n} \\ R_{t,z,y,x,n} &\leftarrow w_t w_z w_y w_x R_{t,z,y,x,n} \end{aligned}$$

As no weight coefficient is involved in the first step, I was then able to use the dimension fusion trick in the tensor contractions. To efficiently compute the matrix K', I precomputed the inverse weights and hard-coded them during the code generation, as I did to precompute products of the weights.

8.7.5 Decreasing the padding requirement

Finally, I used the AoSoA variant as a basis to implement the AoSoA(2) variant, using exactly the same method as described in Section 8.5.

8.7.6 Results

Using a dummy solver based upon the Navier-Stokes model, I replicated the benchmarks performed in the linear case for the nonlinear kernel. To be able to compare different orders, every dummy solver benchmark performed exactly four Picard iterations per execution of the STP kernel.

Instruction mix

Figure 8.13 shows how the floating point operations were performed. It matches closely what was observed in the linear case. This time the user functions are more complex and costly, hence the higher ratio of scalar operations for LoG and PR. Furthermore, as they contain some divisions, they were vectorized on `VectLength` to not compute on the zero-padding. This can be seen with the 128- and 256-bits intructions used in the AoSoA variant, as the compiler chose to use them to more efficiently vectorize the smaller loop spills.

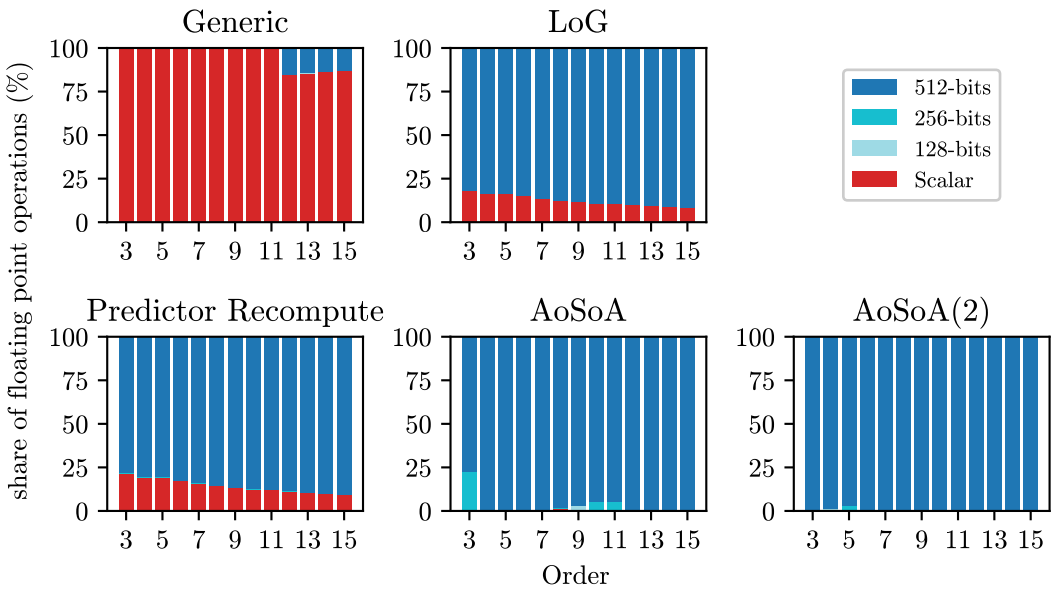


Figure 8.13: Distribution of the packing size used to perform the floating point operations with each nonlinear kernel variant, at order 3 to 15.

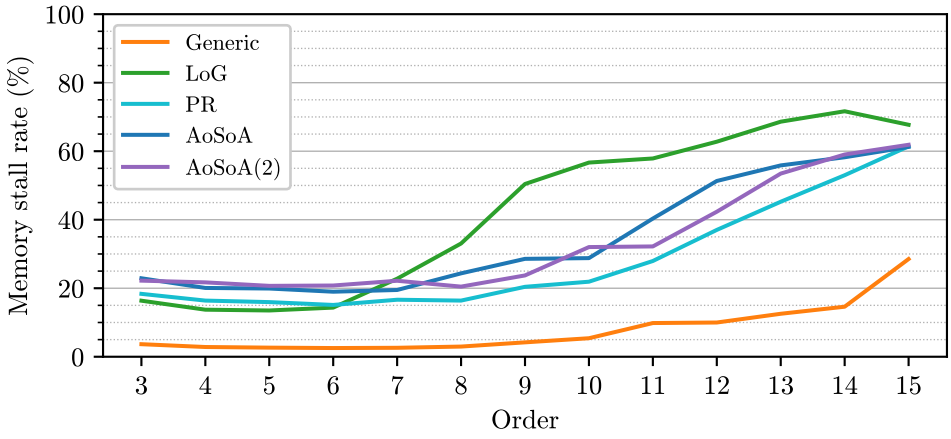


Figure 8.14: Measured memory stalls of the five nonlinear kernel variants at order 3 to 15.

Memory stall rates

The memory stall rates of each variant are presented in Figure 8.14. Once more, LoG is quickly affected by its large memory footprint. Nevertheless, the nonlinear Picard iterations are more arithmetic intensive than the linear CK procedure, and thus, the memory stall rates do not rise as fast here.

The reduction of the memory footprint with the PR reformulation is less effective here than the SplitCK reformulation was in the linear STP. Therefore, in the PR, AoSoA, and

AoSoA(2) variants using it, the memory stalls are only delayed by a few orders and start increasing as soon as order 9.

Performance measurements

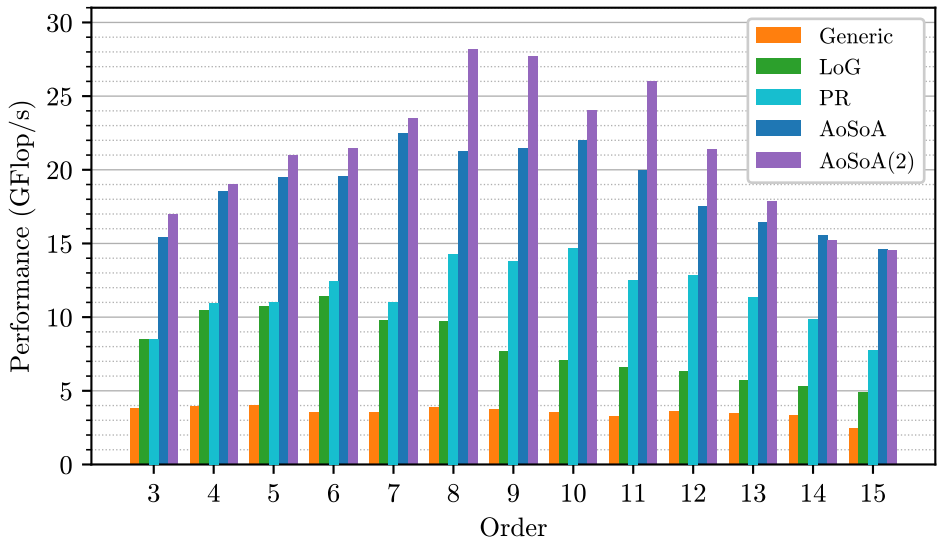


Figure 8.15: Measured performance of the five nonlinear kernel variants at order 3 to 15.

Due to the earlier memory footprint issues, the performance measurments reported in Figure 8.15 peak at order 8 before stagnating for some orders and then decreasing. This time, with the application having a lower number of variables and more complex user functions benefitting more from a better vectorization at low order, the AoSoA(2) data layout performs better than the AoSoA one at every order. The higher arithmetic intensity of the nonlinear scheme results in a higher peak performance reached, with 28.1 GFlop/s, or 35% of peak performance, for the AoSoA(2) variant at order 8.

Figure 8.16 shows the speedup in runtime each variant achieves compared to equivalent generic setup. The AoSoA(2) data layout reaches a speedup around factor 8 as soon as order 7 and exceeds this theoretical limit at higher orders due to reduced memory stalls. Once more ExaHyPE's nonlinear applications can greatly benefit from using the more advanced optimization available.

Finally, I measured the performance of the complete Navier-Stokes solver ExaHyPE application, at order 8 with the AoSoA(2) STP kernel. With 2 processes on a single node, it reached 1217 GFlop/s, 31.7% of the node's peak performance.

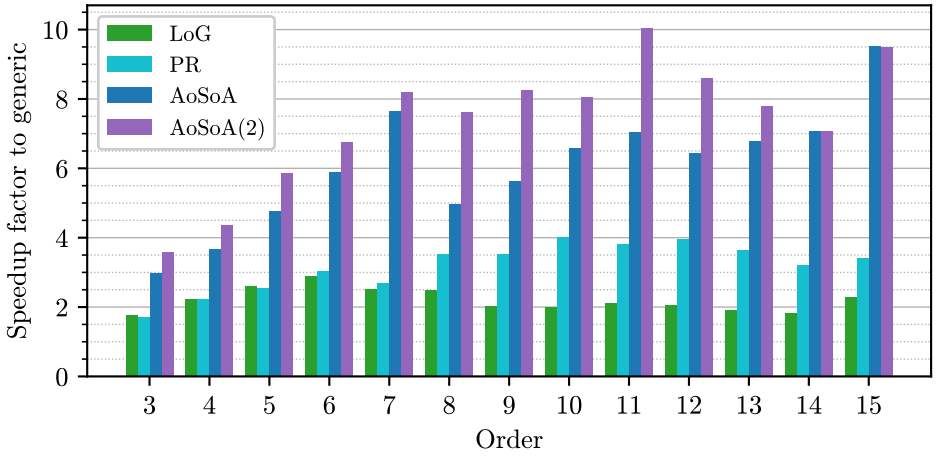


Figure 8.16: Runtime speedup of the generated nonlinear kernel variants compared to the generic one at order 3 to 15.

9 Experimentations for future works

This chapter presents three experimentations I performed to explore other optimization opportunities. Due to time constrains and development scope limitations, they were performed as proof of concepts only.

In Section 9.1, the first experimentation examines if software prefetching can be applied to further improve the Loop-over-GEMM reformulation of the tensor contractions by reducing L1 cache misses. The second one, in Section 9.2, was performed early in the project to improve the nonlinear STP kernel for low-order applications, by using a more advanced initial guess for the Picard iterations. It was abandonned with the new focus toward high-order applications, but still provides valuable insights for possible algorithmic optimizations. Finally, in Section 9.3, the third one implements a single-precision variant of the linear AoSoA(2) STP kernel to test the potential of a mixed-precision engine, and also to validate the optimization techniques on expected future hardware with vector registers twice as big as the current ones in AVX-512.

9.1 Improving Loop-over-GEMM with prefetching

Improving the cache behavior was a key component for the optimization of both linear and nonlinear STP kernels. As the Loop-over-GEMM can be impeded by cache misses, it can sometimes be further optimized by instructing the CPU core to load in advance, or *prefetch*, the next slices into its higher cache levels. In the best case, this loading occurs during the compute-bound computation of the current slices, using the otherwise idling bandwidth capacities, so that the next slices are ready in the L1 cache when their turn comes. However, in the worse case, prefetching represents extra operations that can be redundant or even concurrent with other compiler or runtime optimizations already performed automatically, and thus cause a performance hit. Prefetching is used to this end in YATeTo's LoG to improve its performance, it proved effective, in particular, on the discontinued co-processor Intel Xeon Phi (Knights Landing architecture, KNL) [93, 108].

In this experiment, I tested prefetching with the LoG using LIBXSMM, to see if it increases performance on Skylake CPUs. I used the nonlinear AoSoA STP kernel variant for the tests as previous variants suffer from other memory related bottlenecks solved by the hybrid data layout, and the AoSoA(2) variant sacrifices some LoG optimizations.

Listing 9.1 shows a modified LoG with prefetching to load the next slice of its input tensor. Prefetching is not defined in the C++ standard, thus I used the Intel's intrisinc

```
1 for (int zy = 0; zy < 64; zy++) {
2   _mm_prefetch(lFhi+0*8+(zy+1)*40, _MM_HINT_T0);
3   _mm_prefetch(lFhi+1*8+(zy+1)*40, _MM_HINT_T0);
4   _mm_prefetch(lFhi+2*8+(zy+1)*40, _MM_HINT_T0);
5   _mm_prefetch(lFhi+3*8+(zy+1)*40, _MM_HINT_T0);
6   _mm_prefetch(lFhi+4*8+(zy+1)*40, _MM_HINT_T0);
7   gemm_8_5_8_rhs_x(rhsCoeff_T, lFhi+zy*40, rhs+(t*64+zy)*40);
8 }
```

Listing 9.1: Loop-over-GEMM using prefetching to load the next slice of its input tensor before calling the `gemm` on the current slice, as generated by the expanded `matmul` macro.

function `_mm_prefetch`. Each instruction prefetches a cache line, which on Skylake has a size of 64 Bytes [89], hence a double-precision array is prefetched in chunks of 8. Using the flag `_MM_HINT_T0`, `_mm_prefetch` is translated to the `prefetcht0` instruction, loading the requested cache line in all cache levels, including L1 [85]. I used the code generation capabilities by expanding the `matmul` macro to be able to indicate the next slices as additional parameters, and automatically generate the prefetching instructions on the Kernel Generator configuration.

I tested multiple variants only prefetching some kernels or only the input slices. However, none of the prefetching variants improved the kernel's runtime on Skylake. Unlike in YATeTo, ExaHyPE's matrix slices are small and the tensors fit in the L2 cache, which could explain the lack of improvement with the crude prefetching used in these tests compared to YATeTo's positive results, where L2 prefetching was used as the tensors were expected not to be in the cache. Furthermore, these tests were performed on a classical CPU architecture, Skylake, whereas YATeTo's ones were on a less classical co-processor architecture, KNL, both having different cache and memory capabilities.

The more recent version of LIBXSMM provides an advanced API, that could be used to perform LoG with batched GEMMs, where prefetching can be natively included in the generated code. However, using it would require a heavy re-engineering of the templates as the whole LoG would need to be included in a macro, not just the matrix multiplications, and the new version of LIBXSMM, using just-in-time compilation instead of a gemm generator, would need to be supported. As it could provide some small performance increases, it should be explored at least in future small scale proof of concepts, despite this tests being unsuccessful.

9.2 Continuous Extension Runge-Kutta initial guess

As outlined in Section 8.7.1, the nonlinear STP kernel makes a prediction by solving a fix-point problem using Picard iterations, using the formulation from Dumbser et al. [1]. By default the current state tensor is used as the initial guess for the Picard iterations.

An alternative to Picard iterations are the previously used Continuous Extension Runge-Kutta methods (CERK), as explored by Owren et al. [109]. While they directly compute the desired solution, instead of converging toward it like the Picard iteration scheme, they are multistage methods and the number of stages required as well as their complexity increases drastically with the order of the scheme [110]. Thus CERK methods are only appropriate at low order.

Despite this, a low-order CERK method can be used to produce a low-order initial guess for the Picard iterations to evolve into ExaHyPE's required high-order prediction. When running ExaHyPE at a low order, this CERK guess is usually better than the previously used trivial initial guess, which can result in less Picard iterations required, thus more than compensating for the additional cost of calculating the CERK guess. This idea was proposed by Fambri et al. [11], and together with Michael Dumbser, we implemented the CERK guess as an optimization option for the Kernel Generator in the LoG variant of the nonlinear STP kernel, which at the time was the only optimized variant available. We used a second order CERK method, requiring the computation of two coefficients, which is roughly equivalent in cost to half a Picard iteration at low-order. I employed subtemplating to reuse existing code, and avoid code duplication when computing the CERK's coefficients. The optimized nonlinear STP kernel with the CERK guess was used to compute astrophysicial simulations in ExaHyPE more efficiently as reported in ExaHyPE's final report [4].

However, at high order, we noticed that the low-order CERK guess would often degrade performance compared to the the naive initial guess reusing the high-order current state tensor. With the shift in focus from the Haswell hardware architecture (AVX2) to Skylake (AVX-512), running application at order 3 changed from being a sweet spot for the optimizations with regard to AVX2's vector length to a very poor spot for AVX-512 ones, as can be seen by the lacking performance at order 3 compared to higher order by all benchmarks performed on Skylake and presented in Chapter 8. Therefore, as optimizations were tuned for high-order settings, and running an application at high order with the naive guess is empirically more efficient than running it a low order with the CERK guess, the CERK guess optimization was abandonned, and I did not implement it in the latter introduced kernel variants.

Nevertheless, the CERK guess approach was a successful proof of concept that showed the potential benefits of investing time in computing a better initial guess for the Picard scheme, to later save more in spared Picard iterations. The low-level code optimizations are now achieving high performance with the more advanced kernel variants, and do not offer much room for further improvements in this direction. Instead, improving the initial guess is an algorithmic approach that could further reduce the runtime of the nonlinear STP kernel by reducing the amount of floating point operation required to achieve the same result. Therefore, it should be explored in future works.

9.3 Single-precision SpaceTimePredictor kernel

9.3.1 Motivation

An intuitively simple way to speedup an application using a double-precision floating-point format is to compute it in single-precision instead. In single-precision, the memory footprint is halved and all SIMD instructions can be performed on twice as many numbers, which optimistically would result in a speedup of a factor up to 2. Single-precision could be used in the whole scheme or only partially in a mixed-precision scheme, depending on the application tolerance for the associated loss in numerical accuracy.

As ExaHyPE was designed only with double-precision in mind, modifying the whole engine to single-precision or to allow mixed-precision would require a consequent development effort. Thus in this proof of concept, I took advantage of the code generation and only adapted the linear STP kernel to single-precision, converting its inputs and outputs back and forth.

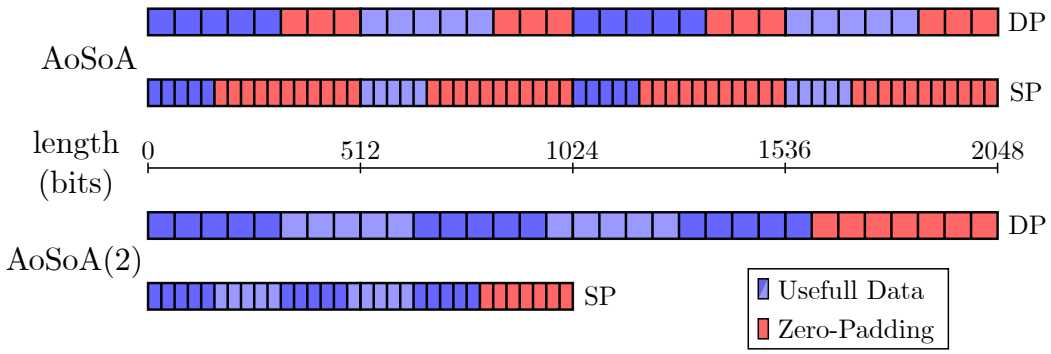


Figure 9.1: Tensors with nDoF = 5 stored with zero-padding using the AoSoA and AoSoA(2) data layouts. A vector register holds 8 double-precision (DP) numbers or 16 in single-precision (SP). With AoSoA, the fastest dimension is padded from 5 to 8 in DP and 5 to 16 in SP, negating the benefits of changing precision. With AoSoA(2) a block of the two fastest dimensions is padded from $5^2 = 25$ to 32 in DP (4 registers) and SP ($\lceil \frac{4}{2} \rceil = 2$ registers), the memory footprint is here halved.

However, this only makes sense for the AoSoA(2) data layout, because with other data layouts the use of padding for optimization negates all the benefits from the lower precision in most of the kernel variants, as illustrated in Figure 9.1 with nDoF = 5. Padding is used to increase the length of each tensor's fastest dimension, i.e. the number of degrees of freedom (nDoF) in the hybrid data layout variants, to the next multiple of the vector length. In single-precision the size of the vector registers virtually doubles, so that if n vector registers are used to store a chunk without padding in double-precision, then $\lceil \frac{n}{2} \rceil$ are required in single-precision. Thus, on AVX-512, for every setting of nDoF below 8, computing in single-precision is pointless as the memory footprint would be doubled back by the additional zero-padding required, as seen in the upper part of Figure 9.1. Moreover, the total amount of operations to perform would not decrease as the vector

registers would be filled with the zero-padding instead of performing two instructions in one as desired. It should be noted that this issue would also occur in double-precision on an instruction set using 1024-bits vector registers.

Reducing the padding ratio by applying it to bigger data chunks was one of the motivations for the AoSoA(2) data layout introduced in Section 8.5. Here, the first two spatial dimensions are padded as a single block, thus it is the square of nDoF that is padded to the next sweet spot. In Figure 9.1, a block of 25 double-precision numbers is stored on 4 vector registers, hence only 2 are needed in single-precision. Therefore, with this data layout, converting from double to single-precision has the expected effects, or close to it[1], on the memory footprint and the number of SIMD instructions required.

For this reason, this proof of concept only makes sense with th AoSoA(2) data layout. It aims at both validating the potential benefits of a single-precision or mixed-precision scheme in ExaHyPE, and ensuring that the optimization will stay relevant on a future architecture using 1024-bits vector registers.

9.3.2 Implementation of the single-precision kernel variant

To implement the proof of concept, I started with the introduction of a new kernel variant by simply duplicating the existing AoSoA(2) template and adding a new specification file parameter to trigger this new option, as was done for the other kernel variants.

I then modified the template to adapt the kernel to compute in single-precision while having double-precision inputs and outputs. The input is only used to initialize the temporary array, thus the convertion is performed on the fly. Temporary arrays for the outputs were created and used in the kernels, they are used to fill out the proper output at the end of the kernel. I then modified the generation of the coefficient arrays to also setup a single-precision version of them to be used in the kernel.

The kernels' padding is automatically included in the algorithmic template by using the padded template variables, e.g. `nVarPad` for the padded `nVar`. The definition of these variables is done in the Kernel Generator's Controller using a configuration variable `vectSize` derived from the architecture. To simplify the implementation, I multiplied `vectSize` by two if single-precision is used. This shortcut affects all the padded temporary variables in every kernel, but it does not matter here, as this proof of concept benchmarks only the STP kernel in isolation. However, in a proper implementation of single-precision kernels, the temporary variables such as `nVarPad` should be split into one per kernel and properly set, which would have required too much development effort here.

Finally, I adapted the `matmul` macro to consider a new parameter `precision` in its configuration, and used it in place of the instance of `double` in the macro implementation, as well as in the model encapsulating LIBXSMM to generate single-precision GEMM when requested. Likewise I adapted the Toolkit's glue code generation to use the correct kernel signature and allocate single-precision temporary memory for it.

[1] For values of nDoF above 4, the worse case is at nDoF = 6 where 36 variables requires 5 vector registers in DP and 3 in SP, resulting in a reduction factor of 0.6 instead of 0.5.

9.3.3 Results

To test the kernel, I modified the ExaSeis application used in Chapter 8 to implement a single-precision version of all the required user functions and used the dummy solver setup to benchmark the single-precision STP kernel.

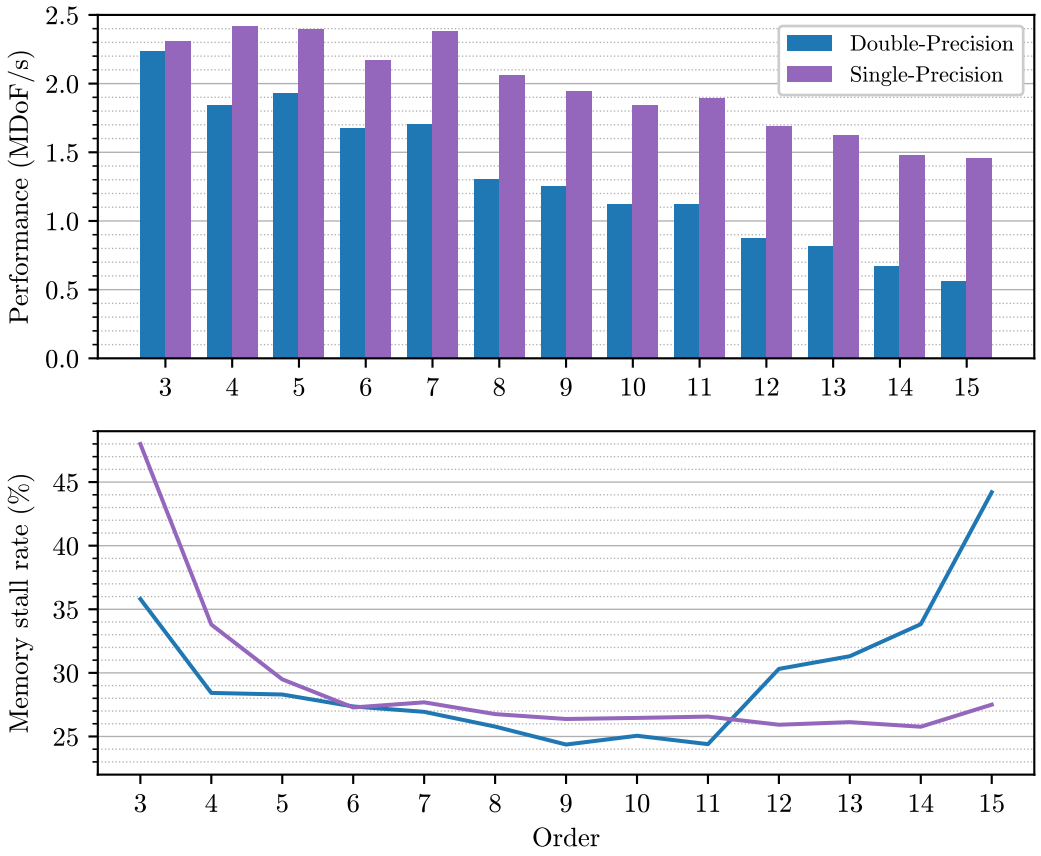


Figure 9.2: Performance measurements of the sinpgle-precision benchmark for order 3 to 15, compared to the double-precision reference. The upper panel shows the number of degrees of freedom processed per second (MDoF/s), and the bottom one the reported memory stalls (%).

Figure 9.2 shows the performance achieved when using the single-precision kernel compared to the double-precision one. At low order, the speedup stays small as the smaller matrix slices in single-precision degrade the performance of the tensor contraction with LIBXSMM, as can be seen with the higher rate of memory stalls. However, as the order increases, the speedup approaches the expected factor 2. It even surpasses it at order 14 and 15, reaching 2.58 at order 15, due to the kernel memory footprint being halved in single-precision, and thus still fitting in the core's cache, compared to the double-precision implementation, as can be seen with the memory stall rates staying almost constant beyond order 11 in the single-precision case compared to the increasing rates in double-precision.

This proof of concept validates that single-precision could be used to speedup ExaHyPE. The results at very high orders might not be scientifically relevant, because of the expected loss in numerical accuracy when computing in single-precision. However, for a large application using more variables than the one used here, the single-precision halving the memory footprint might be the most important factor as such application performance would be degraded by its larger memory footprint earlier than the one used in this benchmark, at orders where computing in single-precision could still be as accurate as doing it in double-precision.

Furthermore, this proof of concept validated that the AoSoA(2) scheme keeps its high performance when a single vector instruction aggregates 16 scalar instructions. Thus, having access to 1024-bits vector registers would speed up double-precision applications using the AoSoA(2) kernel variant and the anticipated padding-related issue is solved by this data layout.

Part IV

Conclusion

10 Conclusion and Outlook

As an engine, ExaHyPE relies on its Kernel Generator to achieve its desired customizability. Furthermore, the Kernel Generator itself can be customized, enabling ExaHyPE's users to add both new numerical schemes to fulfill their models' requirements and low-level optimizations targeting novel hardware architectures. To streamline this process, I refined the usual distinction between domain experts and engine developers and formalized three user roles: application, algorithm, and optimization experts. Each role has its own area of expertise and the latter two introduce a separation between the algorithms implemented by the kernels and the low-level optimizations used to achieve high-performance. Taking inspiration from web application development practices, I designed the Kernel Generator to follow a MVC architectural pattern and to rely on the Jinja2 template engine to generate the kernels, abstracted using its templating language. MVC structures the Kernel Generator and isolates the logic used to generate each kernel into its own independent subunit, with separate templates for each variants. Jinja2 enables the split of the code abstraction into architecture-oblivious algorithmic templates and architecture-aware optimization macros, fulfilling the desired separation of concerns for the user roles.

This design proved its effectiveness during the development of multiple ExaHyPE applications, ExaSeis being presented in this thesis as an example. During the development of ExaSeis, the domain experts assumed the role of application experts and focused on the development of the numerical model used to simulate earthquakes with ExaHyPE. In parallel to their work, the required expansions of the numerical scheme were implemented by the algorithm experts, and at the same time optimization experts worked on further optimizing the kernels toward the Skylake architecture of SuperMUC-NG. Owing to the Kernel Generator's design and its use of a template engine, each user role was able to work efficiently and independently of the other two, with the combined work of the team being automatically integrated in the application.

The Kernel Generator is also critical to ExaHyPE performance. Thus, in the second half of this thesis, I presented my optimization of the generated kernels toward the Skylake CPU architecture used in SuperMUC-NG. Relying on the code generation, I implemented state-of-the-art optimization techniques, with the vectorization of the kernels and the reformulation of the tensor contractions as Loop-over-GEMM using the LIBXSMM BLAS library to obtain high-performance assembly gemms. However, new bottlenecks limited the performance gains. Therefore, I presented successive variants of the performance-critical SpaceTimePredictor kernel, each tackling the main performance bottleneck identified in the previous variant. In particular, after reducing the memory footprint of the kernel by re-engineering its algorithm, I introduced two hybrid data layouts, AoSoA and

AoSoA(2), to first solve the AoS-vs-SoA data layout conflict preventing the vectorization of the application, and then to also reduce the padding requirements.

I benchmarked the kernels variants using the ExaSeis application and a Navier-Stokes solver for the nonlinear implementation. Comparison between variants validated the successive optimizations and showed steady performance and runtime improvements, with the AoSoA(2) ExaSeis setup being up to 10 times faster than the same application with generic kernels. At high-order, with the hybrid data layout, ExaSeis achieved 27.8% of peak performance on a single node of SuperMUC-NG, and the Navier-Stokes solver 31.7%.

Finally, I explored possible paths to further improve ExaHyPE's performance. In particular, the proof of concept of a single-precision version of the STP kernel not only validated my optimization methods for future hardware, but it will be further developed in an upcoming work to enable mixed-precision applications.

Overall, the Kernel Generator presented in this thesis has significantly improved the capabilities of ExaHyPE as an engine. The scientific insights, gained in the process, can be used in other PDE engines. First to optimize them toward modern CPU architectures by optimizing the cache behavior, as well as maximizing vectorization with hybrid data layouts. Second to develop code generation utilities, where the use of a template engine streamlines its development process and facilitates future expansions.

Bibliography

[1] Michael Dumbser, Olindo Zanotti, Raphaël Loubère, and Steven Diot. A posteriori subcell limiting of the discontinuous Galerkin finite element method for hyperbolic conservation laws. *Journal of Computational Physics*, 278:47–75, 2014.

[2] Olindo Zanotti, Francesco Fambri, Michael Dumbser, and Arturo Hidalgo. Space–time adaptive ADER discontinuous Galerkin finite element schemes with a posteriori sub-cell finite volume limiting. *Computers & Fluids*, 118:204–224, 2015.

[3] Anne Reinarz, Dominic E Charrier, Michael Bader, Luke Bovard, Michael Dumbser, Kenneth Duru, Francesco Fambri, Alice-Agnes Gabriel, Jean-Matthieu Gallard, Sven Köppel, et al. ExaHyPE: an engine for parallel dynamically adaptive simulations of wave problems. *Computer Physics Communications*, 254:107251, 2020.

[4] ExaHyPE Consortium. ExaHyPE final project report, 2019. Available at: `https://cordis.europa.eu/project/id/671698/results`.

[5] ExaHyPE Consortium. ExaHyPE second project report, 2018. Available at: `https://cordis.europa.eu/project/id/671698/results`.

[6] Nigel T Bishop and Luciano Rezzolla. Extraction of gravitational waves in numerical relativity. *Living reviews in relativity*, 19(1):1–117, 2016.

[7] Olindo Zanotti and Michael Dumbser. Efficient conservative ADER schemes based on WENO reconstruction and space-time predictor in primitive variables. *Computational astrophysics and cosmology*, 3(1):1–32, 2016.

[8] Francesco Fambri, Michael Dumbser, and Olindo Zanotti. Space–time adaptive ADER-DG schemes for dissipative flows: Compressible Navier-Stokes and resistive mhd equations. *Computer Physics Communications*, 220:297–318, 2017.

[9] Matthias Hanauske, Kentaro Takami, Luke Bovard, Luciano Rezzolla, José A Font, Filippo Galeazzi, and Horst Stöcker. Rotational properties of hypermassive neutron stars from binary mergers. *Physical Review D*, 96(4):043004, 2017.

[10] Benjamin P Abbott, Rich Abbott, TD Abbott, Fausto Acernese, Kendall Ackley, Carl Adams, Thomas Adams, Paolo Addesso, RX Adhikari, VB Adya, et al. GW170817: observation of gravitational waves from a binary neutron star inspiral. *Physical Review Letters*, 119(16):161101, 2017.

[11] Francesco Fambri, Michael Dumbser, Sven Köppel, Luciano Rezzolla, and Olindo Zanotti. ADER discontinuous Galerkin schemes for general-relativistic ideal magne-

tohydrodynamics. *Monthly Notices of the Royal Astronomical Society*, 477(4):4543–4564, 2018.

[12] David Radice and Luciano Rezzolla. Discontinuous Galerkin methods for general-relativistic hydrodynamics: Formulation and application to spherically symmetric spacetimes. *Phys. Rev. D*, 84:024010, Jul 2011.

[13] Sven Köppel. Towards an exascale code for GRMHD on dynamical spacetimes. In *Journal of Physics: Conference Series*, volume 1031, page 012017. IOP Publishing, 2018.

[14] Kenneth Duru and Eric M Dunham. Dynamic earthquake rupture simulations on nonplanar faults embedded in 3d geometrically complex, heterogeneous elastic solids. *Journal of Computational Physics*, 305:185–207, 2016.

[15] Kenneth Duru, Leonhard Rannabauer, Alice-Agnes Gabriel, On Ki Angel Ling, Heiner Igel, and Michael Bader. A stable discontinuous Galerkin method for linear elastodynamics in 3d geometrically complex media using physics based numerical fluxes, 2021.

[16] Kenneth Duru and Gunilla Kreiss. Boundary waves and stability of the perfectly matched layer for the two space dimensional elastic wave equation in second order form. *SIAM Journal on Numerical Analysis*, 52(6):2883–2904, 2014.

[17] Kenneth Duru, Leonhard Rannabauer, Alice-Agnes Gabriel, Gunilla Kreiss, and Michael Bader. A stable discontinuous Galerkin method for the perfectly matched layer for elastodynamics in first order form. *Numerische Mathematik*, 146(4):729–782, 2020.

[18] Kenneth Duru, Alice-Agnes Gabriel, and Gunilla Kreiss. On energy stable discontinuous Galerkin spectral element approximations of the perfectly matched layer for the wave equation. *Computer Methods in Applied Mechanics and Engineering*, 350:898–937, 2019.

[19] Maurizio Tavelli, Michael Dumbser, Dominic Etienne Charrier, Leonhard Rannabauer, Tobias Weinzierl, and Michael Bader. A simple diffuse interface approach on adaptive Cartesian grids for the linear elastic wave equations with complex topography. *Journal of Computational Physics*, 386:158–189, 2019.

[20] Leonhard Rannabauer, Stefan Haas, Dominic Etienne Charrier, Tobias Weinzierl, and Michael Bader. Simulation of tsunamis with the exascale hyperbolic PDE engine ExaHyPE. In *Environmental Informatics: Techniques and Trends. Adjunct Proceedings of the 32nd edition of the EnviroInfo.*, 2018.

[21] Lukas Krenz, Leonhard Rannabauer, and Michael Bader. A high-order discontinuous Galerkin solver with dynamic adaptive mesh refinement to simulate cloud formation processes. *Lecture Notes in Computer Science*, page 311–323, 2020.

[22] Gregor Gassner, Frieder Lörcher, and C-D Munz. A discontinuous Galerkin scheme based on a space-time expansion II. viscous flow equations in multi dimensions. *Journal of Scientific Computing*, 34(3):260–286, 2008.

[23] Jean-Matthieu Gallard, Lukas Krenz, Leonhard Rannabauer, Anne Reinarz, and Michael Bader. Role-oriented code generation in an engine for solving hyperbolic PDE systems. In *Tools and Techniques for High Performance Computing*, pages 111–128, Cham, 2020. Springer International Publishing.

[24] Lukas Krenz. Cloud simulation with the ExaHyPE-engine. *Masterarbeit*, 2019.

[25] William H Reed and Thomas R Hill. Triangular mesh methods for the neutron transport equation. Technical report, Los Alamos Scientific Lab., N. Mex.(USA), 1973.

[26] Bernado Cockburn and Chi-Wang Shu. The Runge-Kutta local projection P^1 - discontinuous-Galerkin finite element method for scalar conservation laws. *ESAIM: Mathematical Modelling and Numerical Analysis*, 25(3):337–361, 1991.

[27] Bernardo Cockburn and Chi-Wang Shu. TVB Runge-Kutta local projection discontinuous Galerkin finite element method for conservation laws II: General framework. *Mathematics of Computation*, 52(186):411–435, 1989.

[28] Bernardo Cockburn, San-Yih Lin, and Chi-Wang Shu. TVB Runge-Kutta local projection discontinuous Galerkin finite element method for conservation laws III: One-dimensional systems. *Journal of Computational Physics*, 84(1):90 – 113, 1989.

[29] Bernardo Cockburn, Suchung Hou, and Chi-Wang Shu. The Runge-Kutta local projection discontinuous Galerkin finite element method for conservation laws. IV. The multidimensional case. *Mathematics of Computation*, 54:545–581, April 1990.

[30] Bernardo Cockburn and Chi-Wang Shu. The Runge–Kutta discontinuous Galerkin method for conservation laws V: Multidimensional systems. *Journal of Computational Physics*, 141(2):199 – 224, 1998.

[31] Vladimir A Titarev and Eleuterio F Toro. ADER: Arbitrary high order Godunov approach. *Journal of Scientific Computing*, 17(1):609–618, 2002.

[32] Michael Dumbser, Francesco Fambri, Maurizio Tavelli, Michael Bader, and Tobias Weinzierl. Efficient implementation of ADER Discontinuous Galerkin schemes for a scalable hyperbolic PDE engine. *Axioms*, 7(3):63, 2018.

[33] Michael Dumbser, Cedric Enaux, and Eleuterio F Toro. Finite volume schemes of very high order of accuracy for stiff hyperbolic balance laws. *Journal of Computational Physics*, 227(8):3971–4001, 2008.

[34] Gregor Gassner, Michael Dumbser, Florian Hindenlang, and Claus-Dieter Munz. Explicit one-step time discretizations for discontinuous Galerkin and finite volume schemes based on local predictors. *Journal of Computational Physics*, 230(11):4232–4247, 2011.

[35] Richard Courant, Kurt Friedrichs, and Hans Lewy. Über die partiellen Differenzengleichungen der mathematischen Physik. *Mathematische annalen*, 100(1):32–74, 1928.

[36] Dominic Etienne Charrier. *Communication-avoiding algorithms for a high-performance hyperbolic PDE engine*. PhD thesis, Durham University, 2020.

[37] Anne Reinarz, Jean-Matthieu Gallard, and Michael Bader. Influence of a-posteriori subcell limiting on fault frequency in higher-order DG schemes. In *2018 IEEE/ACM 8th Workshop on Fault Tolerance for HPC at eXtreme Scale (FTXS)*, pages 79–86. IEEE, 2018.

[38] Dominic E Charrier and Tobias Weinzierl. Stop talking to me–a communication-avoiding ADER-DG realisation. *arXiv preprint arXiv:1801.08682*, 2018.

[39] Florian Rathgeber, David A. Ham, Lawrence Mitchell, Michael Lange, Fabio Luporini, Andrew T. T. Mcrae, Gheorghe-Teodor Bercea, Graham R. Markall, and Paul H. J. Kelly. Firedrake: Automating the finite element method by composing abstractions. *ACM Trans. Math. Softw.*, 43(3), December 2016.

[40] Robert C. Kirby and Lawrence Mitchell. Code generation for generally mapped finite elements. *ACM Trans. Math. Softw.*, 45(4), 2019.

[41] Gaël Guennebaud, Benoît Jacob, et al. Eigen v3. http://eigen.tuxfamily.org, 2010.

[42] Tobias Weinzierl and Miriam Mehl. Peano – A Traversal and Storage Scheme for Octree-Like Adaptive Cartesian Multiscale Grids. *SIAM J. Sci. Comput.*, 33(5):2732–2760, 2011.

[43] Tobias Weinzierl. The Peano software—parallel, automaton-based, dynamically adaptive grid traversals. *ACM Trans. Math. Softw.*, 45(2):14:1–14:41, 2019.

[44] 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*, pages 981–991. IEEE, 2016.

[45] Etienne M Gagnon and Laurie J Hendren. SableCC, an object-oriented compiler framework. In *Proceedings. Technology of Object-Oriented Languages. TOOLS 26 (Cat. No.98EX176)*, pages 140–154, 1998.

[46] Dominic Kempf, René Heß, Steffen Müthing, and Peter Bastian. Automatic code generation for high-performance discontinuous Galerkin methods on modern architectures. *ACM Trans. Math. Softw.*, 47(1), December 2020.

[47] Peter Bastian, Markus Blatt, Andreas Dedner, Nils-Arne Dreier, Christian Engwer, René Fritze, Carsten Gräser, Christoph Grüninger, Dominic Kempf, Robert Klöfkorn, Mario Ohlberger, and Oliver Sander. The dune framework: Basic concepts and recent developments. *Computers & Mathematics with Applications*, 81:75–112, 2021. Development and Application of Open-source Software for Problems with Numerical PDEs.

[48] Carsten Uphoff and Michael Bader. Yet another tensor toolbox for discontinuous Galerkin methods and other applications. *ACM Trans. Math. Softw.*, 46(4), October 2020.

[49] Armin Ronacher. Jinja2 documentation. *Welcome to Jinja2 — Jinja2 Documentation (2.11)*, 2020.

[50] William H Brown, Raphael C Malveau, Hays W" Skip" McCormick, and Thomas J Mowbray. *AntiPatterns: refactoring software, architectures, and projects in crisis*. John Wiley & Sons, Inc., 1998.

[51] ISO. Information technology – the JSON data interchange syntax. Standard, International Organization for Standardization, Geneva, CH, November 2017.

[52] Tim Bray. The JavaScript Object Notation (JSON) Data Interchange Format. RFC 8259, December 2017.

[53] Niels Lohmann et al. JSON for modern C++, version 3.1.2. *GitHub Repository*, 2018.

[54] Michael Bader. *Space-filling curves: an introduction with applications in scientific computing*, volume 9. Springer Science & Business Media, 2012.

[55] Wolfgang Eckhardt and Tobias Weinzierl. A blocking strategy on multicore architectures for dynamically adaptive PDE solvers. In *International Conference on Parallel Processing and Applied Mathematics*, pages 567–575. Springer, 2009.

[56] Dominic Etienne Charrier, Benjamin Hazelwood, and Tobias Weinzierl. Enclave tasking for DG methods on dynamically adaptive meshes. *SIAM Journal on Scientific Computing*, 42(3):C69–C96, 2020.

[57] Trygve Mikjel H Reenskaug. The original MVC reports, 1979.

[58] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, and Design Patterns. Elements of reusable object-oriented software. *Reading: Addison-Wesley*, 1995.

[59] Ken Williamson. *Learning AngularJS: a guide to AngularJS development.* " O'Reilly Media, Inc.", 2015.

[60] Nigel George. *Mastering Django: Core.* Packt Publishing Ltd, 2016.

[61] Arun Ravindran. *Django Design Patterns and Best Practices.* Packt Publishing Ltd, 2015.

[62] Andrew Troelsen and Philip Japikse. Introducing ASP. NET MVC. In *Pro C# 7*, pages 1179–1221. Springer, 2017.

[63] Rashidah F Olanrewaju, Thouhedul Islam, and NA Ali. An empirical study of the evolution of PHP MVC framework. In *Advanced Computer and Communication Engineering Technology*, pages 399–410. Springer, 2015.

[64] Geoffroy Warin. *Mastering Spring MVC 4.* Packt Publishing Ltd, 2015.

[65] Paul Deck. *Spring MVC: a tutorial*. Brainy Software Inc, 2016.

[66] Sam Ruby, David B Copeland, and Dave Thomas. *Agile Web Development with Rails 6*. Pragmatic bookshelf, 2020.

[67] Adrian Holovaty and Jacob Kaplan-Moss. *The Django Template System*, pages 31–58. Apress, Berkeley, CA, 2008.

[68] Sebastian Eibl and Ulrich Rüde. A modular and extensible software architecture for particle dynamics. In *8th International Conference on Discrete Element Methods*, 2019. arXiv:1906.10963.

[69] Martin Bauer, Sebastian Eibl, Christian Godenschwager, Nils Kohl, Michael Kuron, Christoph Rettinger, Florian Schornbaum, Christoph Schwarzmeier, Dominik Thönnes, Harald Köstler, and et al. waLBerla: A block-structured high-performance framework for multiphysics simulations. *Computers and Mathematics with Applications*, 81:478–501, Jan 2021.

[70] John W Backus, Robert J Beeber, Sheldon Best, Richard Goldberg, L Mitchell Haibt, Harlan L Herrick, Robert A Nelson, David Sayre, Peter B Sheridan, H Stern, et al. The FORTRAN automatic coding system. In *Papers presented at the February 26-28, 1957, western joint computer conference: Techniques for reliability*, pages 188–198, 1957.

[71] Daniel D McCracken and Edwin D Reilly. Backus-Naur form (bnf). In *Encyclopedia of Computer Science*, pages 129–131. AJohn Wiley and Sons Ltd., Chichester, UK, 2003.

[72] Douglas Crockford. The application/json Media Type for JavaScript Object Notation (JSON). RFC 4627, July 2006.

[73] Felipe Pezoa, Juan L. Reutter, Fernando Suarez, Martín Ugarte, and Domagoj Vrgoč. Foundations of JSON Schema. In *Proceedings of the 25th International Conference on World Wide Web*, WWW '16, page 263–273, Republic and Canton of Geneva, CHE, 2016. International World Wide Web Conferences Steering Committee.

[74] Kirill Simonov. PyYAML, 2014. Available at: `http://pyyaml.org`.

[75] Steven M Day and Christopher R Bradley. Memory-efficient simulation of anelastic wave propagation. *Bulletin of the Seismological Society of America*, 91(3):520–531, 06 2001.

[76] ExaHyPE developers. *ExaHyPE Guidebook*. The ExaHyPE consortium, 2021.

[77] Viktor Vladimirovich Rusanov. The calculation of the interaction of non-stationary shock waves with barriers. *Zhurnal Vychislitel'noi Matematiki i Matematicheskoi Fiziki*, 1(2):267–279, 1961.

[78] Robert M Kirby and George Em Karniadakis. Selecting the numerical flux in discontinuous Galerkin methods for diffusion problems. *Journal of Scientific Computing*, 22(1):385–411, 2005.

[79] David A Kopriva, Jan Nordström, and Gregor J Gassner. Error boundedness of discontinuous Galerkin spectral element approximations of hyperbolic problems. *Journal of Scientific Computing*, 72(1):314–330, 2017.

[80] Jianxian Qiu. Development and comparison of numerical fluxes for LWDG methods. *Numerical Mathematics, Theory, Methods and Application*, 1:1–32, 2008.

[81] Christian Pelties, Josep De la Puente, Jean-Paul Ampuero, Gilbert B Brietzke, and Martin Käser. Three-dimensional dynamic rupture simulation with a high-order discontinuous Galerkin method on unstructured tetrahedral meshes. *Journal of Geophysical Research: Solid Earth*, 117(B2), 2012.

[82] Raimund Bürger, Kenneth H. Karlsen, and John D. Towers. An Engquist–Osher-type scheme for conservation laws with discontinuous flux adapted to flux connections. *SIAM Journal on Numerical Analysis*, 47(3):1684–1712, 2009.

[83] Kenneth Duru, Leonhard Rannabauer, Alice-Agnes Gabriel, and Heiner Igel. A new discontinuous Galerkin spectral element method for elastic waves with physically motivated numerical fluxes, 2019.

[84] R Intel. Intel® 64 and IA-32 architectures software developer's manual volume 2C: Instruction set reference, V-Z, 2021.

[85] R Intel. Intel® 64 and IA-32 architectures software developer's manual volume 2B: Instruction set reference, M-U, 2021.

[86] George H Barnes, Richard M Brown, Maso Kato, David J Kuck, Daniel L Slotnick, and Richard A Stokes. The ILLIAC IV computer. *IEEE Transactions on computers*, 100(8):746–757, 1968.

[87] Paul B Schneck. The CDC STAR-100. In *Supercomputer Architecture*, pages 99–117. Springer, 1987.

[88] Richard M Russell. The CRAY-1 computer system. *Communications of the ACM*, 21(1):63–72, 1978.

[89] R Intel. Intel® 64 and IA-32 architectures optimization reference manual, 2021.

[90] Zvi Or-Bach. *A 1000× Improvement of the Processor-Memory Gap*, pages 247–267. Springer International Publishing, Cham, 2020.

[91] Edoardo Di Napoli, Diego Fabregat-Traver, Gregorio Quintana-Ortí, and Paolo Bientinesi. Towards an efficient use of the BLAS library for multilinear tensor contractions. *Applied Mathematics and Computation*, 235:454–468, 2014.

[92] Yang Shi, Uma Naresh Niranjan, Animashree Anandkumar, and Cris Cecka. Tensor contractions with extended BLAS kernels on CPU and GPU. In *2016 IEEE 23rd*

International Conference on High Performance Computing (HiPC), pages 193–202, 2016.

[93] 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*, pages 343–362. Springer, 2016.

[94] R Intel. Intel® C++ compiler 19.0 developer guide and reference, 2019.

[95] Jean-Matthieu Gallard, Leonhard Rannabauer, Anne Reinarz, and Michael Bader. Vectorization and minimization of memory footprint for linear high-order discontinuous Galerkin schemes. In *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 711–720, Los Alamitos, CA, USA, may 2020. IEEE Computer Society.

[96] Jan Treibig, Georg Hager, and Gerhard Wellein. LIKWID: Lightweight performance tools. In *Competence in High Performance Computing 2010*, pages 165–175. Springer, 2011.

[97] Simon D. Hammond, Courtenay T. Vaughan, and Clay Hughes. Evaluating the Intel Skylake Xeon processor for HPC workloads. *2018 Int. Conf. on High Perf. Comp. & Sim. (HPCS)*, pages 342–349, 2018.

[98] Martin Kronbichler and Katharina Kormann. A generic interface for parallel cell-based finite element operator application. *Comp. Fluids*, 63:135–147, 2012.

[99] Miklós Homolya, Robert C. Kirby, and David A. Ham. Exposing and exploiting structure: optimal code generation for high-order finite element methods. *arXiv e-prints*, 2017. arXiv:1711.02473.

[100] Steffen Müthing, Marian Piatkowski, and Peter Bastian. High-performance implementation of matrix-free high-order discontinuous Galerkin methods. *Int. J. High Perf. Comp. App.*, 2018.

[101] Joachim Schöberl. C++11 implementation of finite elements in NGSolve. *Institute for Analysis and Scientific Computing, Vienna University of Technology*, 30, 2014.

[102] Dominic Etienne Charrier, Benjamin Hazelwood, Ekaterina Tutlyaeva, Michael Bader, Michael Dumbser, Andrey Kudryavtsev, A. Moskovskiy, and Tobias Weinzierl. Studies on the energy and deep memory behaviour of a cache-oblivious, task-based hyperbolic PDE solver. *Int. J. High Perf. Comp. App.*, 33(5):973–986, 2019.

[103] Michael Dumbser, Federico Guercilena, Sven Köppel, Luciano Rezzolla, and Olindo Zanotti. Conformal and covariant Z4 formulation of the Einstein equations: Strongly hyperbolic first-order reduction and solution with discontinuous Galerkin schemes. *Phys. Rev. D*, 97:084053, Apr 2018.

[104] Freddie D Witherden, Antony M Farrington, and Peter E Vincent. PyFR: an open source framework for solving advection–diffusion type problems on stream-

ing architectures using the flux reconstruction approach. *Comp. Phys. Comm.*, 185(11):3028–3040, 2014.

[105] Ingo Wald. Fast construction of SAH BVHs on the Intel many integrated core (MIC) architecture. *IEEE Transactions on Visualization and Computer Graphics*, 18(1):47–57, 2010.

[106] Florian Lemaitre, Benjamin Couturier, and Lionel Lacassagne. Small SIMD matrices for CERN high throughput computing. In *Proceedings of the 2018 4th Workshop on Programming Models for SIMD/Vector Processing*, WPMVP'18, New York, NY, USA, 2018. Association for Computing Machinery.

[107] Philipp Samfass, Tobias Weinzierl, Dominic E. Charrier, and Michael Bader. Lightweight task offloading exploiting MPI wait times for parallel adaptive mesh refinement. *Concurrency and Computation: Practice and Experience*, 32(24):e5916, 2020.

[108] Carsten Uphoff, Sebastian Rettenberger, Michael Bader, Elizabeth H Madden, Thomas Ulrich, Stephanie Wollherr, and Alice-Agnes Gabriel. Extreme scale multiphysics simulations of the tsunamigenic 2004 Sumatra megathrust earthquake. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '17, New York, NY, USA, 2017. Association for Computing Machinery.

[109] Brynjulf Owren and Marino Zennaro. Derivation of efficient, continuous, explicit Runge–Kutta methods. *SIAM journal on scientific and statistical computing*, 13(6):1488–1501, 1992.

[110] Brynjulf Owren and Marino Zennaro. Order barriers for continuous explicit Runge–Kutta methods. *Mathematics of computation*, 56(194):645–661, 1991.