Performance Evaluation of Acoustic Model Implementation in SeisSol Using Roofline Model

Jinwen Pan
TUM School of Computation, Information and Technology
Technical University of Munich
Garching, Germany
jinwen.pan@tum.de

***Abstract*—SeisSol is a high-performance software framework widely used for simulating seismic wave propagation and dynamic rupture processes. The incorporation of the acoustic wave model, which is crucial for accurately representing the natural processes in SeisSol, presents new performance challenges. The acoustic wave model, consisting of four partial differential equations (PDEs), is a special case of the elastic wave model, which comprises nine PDEs. Based on this, in the current version of SeisSol, the acoustic model is implemented by setting the second Lamé parameter to zero within the elastic model, which leads to unnecessary resource usage. We implement a standalone acoustic model for SeisSol's proxy application, evaluate its performance improvements using the roofline model, and analyze its cache utilization. Although the performance bottlenecks have not significantly changed, the runtime and memory data volume have been reduced by approximately half under several typical test conditions, while the last-level cache (LLC) miss rate has increased by up to about 17%. Additionally, the process of building the roofline model on the Leibniz Supercomputing Center (LRZ)'s CoolMUC-2 cluster, along with its advantages and limitations, is discussed.**

***Index Terms*—acoustic wave, computational seismology, roofline model, high-performance computing**

I. Introduction

Seismic wave propagation simulations have become an essential tool for understanding earthquake dynamics, predicting ground motion, and informing hazard assessments. These simulations are critical for engineers, geophysicists, and disaster management authorities in their efforts to mitigate earthquake risks. As earthquake phenomena involve highly complex physical processes, accurate simulations must capture intricate interactions between geological layers, fault lines, and various material types in the subsurface environment. One of the leading software frameworks in this domain is SeisSol, a high-performance computing (HPC) application that uses the Arbitrary high-order DERivative Discontinuous Galerkin (ADER-DG) method to simulate seismic wave propagation and dynamic rupture processes [1]. The ADER-DG method combines the DG spatial discretization with the ADER temporal discretization. The DG method handles complex geometries with high-order accuracy in an explicit semi-discrete form while ensuring local conservation, making it well-suited for simulating conservation laws, which are the primary physical models in seismic simulations [2]. The ADER method, on the other hand, predicts time evolution through local Taylor expansions and corrects neighboring elements, allowing it to achieve arbitrary accuracy in both time and space as a one-step method [3]. Because of these properties SeisSol is capable of delivering highly accurate results at large scales, making it a valuable resource in both academic research and practical applications.

The accuracy of seismic simulations also depends heavily on the physical models used to represent the materials through which seismic waves travel. SeisSol currently supports various materials, such as isotropic elastic [4], poroelastic [5], viscoelastic [6], off-fault plastic [7], and elastic-acoustic coupled materials [8]. In this context, the incorporation of acoustic materials has become increasingly important. Acoustic materials, such as air, water, and certain geological layers, have distinct properties that affect the speed and attenuation of seismic waves. These materials are especially relevant in regions with heterogeneous subsurface structures, such as sedimentary basins or areas with varying rock densities. By integrating acoustic material models into SeisSol, the simulation can more accurately capture the complex behavior of seismic waves, leading to more reliable predictions of ground motion and hazard scenarios.

In practice, since acoustic waves are a special case of elastic waves, the current version of SeisSol models acoustic materials by setting the second Lamé parameter in the elastic model to zero while still retaining the 9-dimensional PDEs description. Although this approach ensures compatibility and flexibility, it does not take full advantage of the simplified nature of the acoustic model, which can be described by only 4 PDEs. Consequently, there is room to improve both runtime and memory usage by decoupling the acoustic model from the elastic formulation, thus allowing for a more efficient standalone implementation.

To assess these performance limitations, we implement a standalone acoustic wave model for SeisSol's proxy application. We then evaluate the performance improvements of this implementation using the roofline model, one of the most effective tools for performance analysis and optimization in HPC applications [9]. The roofline model provides a clear, visual representation of a program's performance by balancing two critical metrics: computational performance (measured in floating-point operations per second, or FLOPS) and memory bandwidth (the rate at which data are moved

between memory and processing units). By plotting these two metrics, the roofline model helps identify whether a program's performance is constrained by the processing power of the CPU (computation-bound) or by the memory access speed (memory-bound). This insight is particularly valuable when optimizing scientific software like SeisSol, as different simulation scenarios may have varying computational and memory demands. Through performance testing on the CoolMUC-2 cluster at the LRZ, we demonstrate that the standalone acoustic implementation reduces both the runtime and memory data volume by approximately 50%, without significantly altering the performance bottlenecks. Additionally, a cache usage analysis reveals that the cache miss rate of this implementation fluctuates under different test conditions, with a maximum increase of approximately 17%.

II. Physical Models

SeisSol supports various physical models, such as elastic, poroelastic, and viscoelastic models. Since the primary focus of this project is to evaluate and compare the performance of elastic and acoustic models with varying degrees of freedom (DOFs), this section will only cover the elastic and acoustic models and their relationship. Due to space limitations, certain details, such as boundary conditions and seismic sources, will be omitted. A complete discussion can be found in [10]. Throughout this report, we will describe the three-dimensional space using a Cartesian coordinate system $\boldsymbol{x} = (x, y, z)^T$, along with a time coordinate t.

A. Elastic Wave Model

In seismic modeling, particle perturbations are typically considered small. Therefore, wave propagation in elastic solids can be described using the linear elastic wave equations. In this subsection, we will derive these equations following [10], [11], and [12].

At a given time t, the displacement of a particle located at $\boldsymbol{x_0}$ in three-dimensional space can be expressed as

$$\boldsymbol{u}(\boldsymbol{x_0}, t) = \begin{pmatrix} u_1(\boldsymbol{x_0}, t) \\ u_2(\boldsymbol{x_0}, t) \\ u_3(\boldsymbol{x_0}, t) \end{pmatrix}. \tag{1}$$

After a small perturbation $\boldsymbol{\delta x}$, the new displacement of the particle can be linearly approximated as

$$\boldsymbol{u}(\boldsymbol{x_0} + \boldsymbol{\delta x}, t) \approx \boldsymbol{u}(\boldsymbol{x_0}, t) + \boldsymbol{J}\boldsymbol{\delta x}, \tag{2}$$

where $\boldsymbol{J}$ is the Jacobian matrix with entries $J_{ij} = \frac{\partial u_i}{\partial (x_0)_j}$. The Jacobian matrix can be split into the symmetric strain matrix $\boldsymbol{\varepsilon}$ and the skew-symmetric rotation matrix $\boldsymbol{\Omega}$ with $\varepsilon_{ij} = \frac{1}{2}(J_{ij} + J_{ji})$ and $\Omega_{ij} = \frac{1}{2}(J_{ij} - J_{ji})$ respectively.

Consider an infinitesimal cube near any point within a material. The diagonal components of the strain matrix represent normal strains, which indicate the relative extent of stretching or compression of the cube along the coordinate axes (dimensionless). The off-diagonal components represent shear strains, which indicate the relative extent of shear deformation within the plane defined by the corresponding subscript (dimensionless).

Strain within a material leads to the generation of internal forces that maintain the material's stability, known as stress (measured in Pa). Similarly, at any point in the material, we can define a stress matrix $\boldsymbol{\sigma}$. The diagonal components of this matrix represent normal stresses, indicating the tensile or compressive forces acting on the cube along the coordinate axes. The off-diagonal components represent shear stresses, which act within the planes defined by the corresponding subscript. Based on the assumption of static equilibrium in solids, the stress matrix is also symmetric.

According to the generalized Hooke's law, under the assumption of small perturbations, stress and strain are linearly related through a fourth-order tensor $\boldsymbol{C}$. Based on symmetry and thermodynamical considerations, $\boldsymbol{C}$ actually only has 21 independent entries [13], [14]. Here, we do not present the stress-strain relationship in tensor form. Instead, for convenience, we express this relationship under the assumption of isotropy as

$$\sigma_{ij} = \lambda \delta_{ij} \sum_{k=1}^{3} \varepsilon_{kk} + 2\mu \varepsilon_{ij}, \tag{3}$$

where λ and μ are the Lamé parameters which are material-specific and δ_{ij} is the Kronecker delta. In addition, the bulk modulus $K = \lambda + \frac{2}{3}\mu$ can be defined, which measures the incompressibility of a material. By differentiating both sides of (3) with respect to time, and incorporating the definitions of the strain matrix and velocity, we can derive six of the nine elastic wave equations, given by

$$\frac{\partial \sigma_{ij}}{\partial t} = \lambda \delta_{ij} \sum_{k=1}^{3} \frac{\partial v_k}{\partial x_k} + \mu \left(\frac{\partial v_i}{\partial x_j} + \frac{\partial v_j}{\partial x_i} \right), \tag{4}$$

because the stress matrix has 6 independent entries due to symmetry.

Consider an arbitrary three-dimensional domain Ω within a material, along with its boundary $\partial\Omega$. According to Newton's second law, the rate of change of momentum equals the net force acting on the domain. This net force consists of the traction forces $\boldsymbol{T}(\boldsymbol{n})$ (Pa) applied on the domain boundary and the body forces $\boldsymbol{f}$ (N/m^3) acting within the domain. Therefore, we have

$$\frac{\partial}{\partial t} \int_\Omega \rho \frac{\partial \boldsymbol{u}}{\partial t} \, dV = \int_{\partial\Omega} \boldsymbol{T}(\boldsymbol{n}) \, dS + \int_\Omega \boldsymbol{f} \, dV, \tag{5}$$

where $\boldsymbol{n}$ is the unit normal vector of $d\boldsymbol{S}$ (note that in the integral, dS is a scalar instead), and ρ is the density of the material. The components of the traction force can be expressed in terms of stress as $T_i = \sum_{j=1}^{3} \sigma_{ji} n_j$. Applying the divergence theorem, the surface integral in (5) can be converted into a volume integral. Therefore, in any dimension i, (5) can be written as

$$\int_\Omega \rho \frac{\partial^2 u_i}{\partial t^2} \, dV = \int_\Omega \sum_{j=1}^{3} \frac{\partial \sigma_{ji}}{\partial x_j} + f_i \, dV, \tag{6}$$

where we assume the density is constant. In seismic simulations, we can typically rewrite (6) into a stronger differential form as

$$\rho \frac{\partial v_i}{\partial t} = \sum_{j=1}^{3} \frac{\partial \sigma_{ij}}{\partial x_j} + f_i, \tag{7}$$

where we utilize both the definition of velocity and the symmetry of the stress matrix. We ultimately obtain the remaining three of the nine elastic wave equations.

By combining (4) and (7), the system of elastic wave equations can be written in matrix form as

$$\frac{\partial \boldsymbol{q}}{\partial t} + \boldsymbol{A}(\boldsymbol{x}) \frac{\partial \boldsymbol{q}}{\partial x_1} + \boldsymbol{B}(\boldsymbol{x}) \frac{\partial \boldsymbol{q}}{\partial x_2} + \boldsymbol{C}(\boldsymbol{x}) \frac{\partial \boldsymbol{q}}{\partial x_3} = 0. \tag{8}$$

Here, $\boldsymbol{q}$ is defined as $(\sigma_{11}, \sigma_{22}, \sigma_{33}, \sigma_{12}, \sigma_{23}, \sigma_{13}, v_1, v_2, v_3)^T$, which is a vector function of $\boldsymbol{x}$ and t, meaning the elastic model is described by nine quantities at any spacetime coordinates. $\boldsymbol{A}(\boldsymbol{x})$, $\boldsymbol{B}(\boldsymbol{x})$, and $\boldsymbol{C}(\boldsymbol{x})$ are flux matrices in the directions of the three coordinate axes respectively, which may vary with position since the Lamé parameters and density of the material may also vary spatially. Under the assumption of isotropy, these matrices have the same set of eigenvalues $(-c_p, -c_s, -c_s, 0, 0, 0, c_s, c_s, c_p)$, where $c_p = \sqrt{\frac{\lambda+2\mu}{\rho}}$ and $c_s = \sqrt{\frac{\mu}{\rho}}$ are defined as the velocities of primary waves (P-waves) and secondary waves (S-waves), respectively. Therefore, the set of eigenvalues correspond to two S-waves and one P-wave in each direction. P-waves and S-waves are two common types of waves in earthquakes, both classified as body waves. P-waves are compressional waves, where particle motion is in the same direction as wave propagation, and they travel faster. S-waves, on the other hand, are shear waves, where particle motion is perpendicular to the direction of wave propagation, and they travel slower. For brevity, using the definition $\hat{\boldsymbol{A}}(\boldsymbol{x}, \boldsymbol{n}) = n_1 \boldsymbol{A}(\boldsymbol{x}) + n_2 \boldsymbol{B}(\boldsymbol{x}) + n_3 \boldsymbol{C}(\boldsymbol{x})$, we only present the result of the linear combination of the flux matrices:

$$\hat{\boldsymbol{A}}(\boldsymbol{x}, \boldsymbol{n}) = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & -n_1(\lambda+2\mu) & -n_2\lambda & -n_3\lambda \\ 0 & 0 & 0 & 0 & 0 & 0 & -n_1\lambda & -n_2(\lambda+2\mu) & -n_3\lambda \\ 0 & 0 & 0 & 0 & 0 & 0 & -n_1\lambda & -n_2\lambda & -n_3(\lambda+2\mu) \\ 0 & 0 & 0 & 0 & 0 & 0 & -n_2\mu & -n_1\mu & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -n_3\mu & -n_2\mu \\ 0 & 0 & 0 & 0 & 0 & 0 & -n_3\mu & 0 & -n_1\mu \\ -\frac{n_1}{\rho} & 0 & 0 & -\frac{n_2}{\rho} & 0 & -\frac{n_3}{\rho} & 0 & 0 & 0 \\ 0 & -\frac{n_2}{\rho} & 0 & -\frac{n_1}{\rho} & -\frac{n_3}{\rho} & 0 & 0 & 0 & 0 \\ 0 & 0 & -\frac{n_3}{\rho} & 0 & -\frac{n_2}{\rho} & -\frac{n_1}{\rho} & 0 & 0 & 0 \end{pmatrix}, \tag{9}$$

which is the flux matrix of the infinitesimal plane defined by any unit normal vector $\boldsymbol{n}$ near $\boldsymbol{x}$.

B. Acoustic Wave Model

In seismic simulations, in addition to modeling the motion of solid particles, it is also necessary to simulate the flow of fluids, which is typically described using the acoustic wave model. In this subsection, we derive the model following [10], [15], and [16].

Consider an arbitrary three-dimensional domain Ω with its boundary $\partial\Omega$. The mass of the fluid within the domain is a conserved quantity, meaning its rate of change is equal to the surface integral of the mass flux across the boundary of the domain, given by:

$$\int_\Omega \frac{\partial \rho}{\partial t}\, dV + \int_{\partial\Omega} \rho \boldsymbol{v} \cdot \boldsymbol{n}\, dS = 0. \tag{10}$$

Similarly to the previous subsection, we apply the divergence theorem and rewrite (10) in its differential form:

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \boldsymbol{v}) = 0. \tag{11}$$

In addition, momentum is also a conserved quantity. Its rate of change is not only directly contributed by the surface integral of the momentum flux on the boundary but also by the net force acting on the fluid. This net force consists of the integral of the pressure p (Pa) exerted on the domain's surface and the integral of the body forces $\boldsymbol{f}$ (N/m^3) acting within the domain, expressed as

$$\frac{\partial \rho \boldsymbol{v}}{\partial t} + \nabla \cdot (\boldsymbol{v} \otimes \rho \boldsymbol{v} + \boldsymbol{I} p) - \boldsymbol{f} = 0, \tag{12}$$

where we omit the application of the divergence theorem and directly present the equation in its differential form to avoid repetition. Here, $\boldsymbol{I}$ is a 3-dimensional identity matrix, and the divergence of a $3 \times n$ matrix $\boldsymbol{A}$, denoted as $\nabla \cdot \boldsymbol{A}$, is defined as producing a n-dimensional vector with

$$(\nabla \cdot \boldsymbol{A})_i = \frac{\partial a_{1i}}{\partial x_1} + \frac{\partial a_{2i}}{\partial x_2} + \frac{\partial a_{3i}}{\partial x_3}, \tag{13}$$

where a_{ij} are the entries of $\boldsymbol{A}$. The symbol $\otimes$ represents the Kronecker product, which takes two vectors $\boldsymbol{a}$ and $\boldsymbol{b}$, and outputs a matrix with $(\boldsymbol{a} \otimes \boldsymbol{b})_{ij} = a_i b_j$. Furthermore, it can also take a $m \times n$ matrix $\boldsymbol{A}$ and a $p \times q$ matrix $\boldsymbol{B}$, and produce a $pm \times qn$ block matrix:

$$\boldsymbol{A} \otimes \boldsymbol{B} = \begin{pmatrix} a_{11}\boldsymbol{B} & \cdots & a_{1n}\boldsymbol{B} \\ \vdots & \ddots & \vdots \\ a_{m1}\boldsymbol{B} & \cdots & a_{mn}\boldsymbol{B} \end{pmatrix}. \tag{14}$$

Finally, in our case, $\boldsymbol{f} = (0, 0, -\rho g)^T$, where g is the gravitational acceleration on Earth and the pressure p is a function of a single variable ρ (density) under the assumption of isentropic flow [17].

We use letters with a subscript 0 to denote physical quantities in the model under steady-state conditions. In equilibrium, $\frac{\partial p_0}{\partial x_1} = \frac{\partial p_0}{\partial x_2} = \frac{\partial p_0}{\partial t} = \frac{\partial \rho_0}{\partial x_1} = \frac{\partial \rho_0}{\partial x_2} = \frac{\partial \rho_0}{\partial t} = 0$ and $\boldsymbol{v_0} = 0$. Therefore, (12) reduces to

$$\frac{\partial p_0}{\partial x_3} = -\rho_0 g. \tag{15}$$

In addition, we define the bulk modulus in equilibrium as $K_0 = \rho_0 \frac{dp_0}{d\rho_0}$. Using the chain rule, we have

$$\frac{\partial p_0}{\partial x_3} = \frac{K_0}{\rho_0} \frac{\partial \rho_0}{\partial x_3}, \tag{16}$$

which results in

$$\frac{\partial \rho_0}{\partial x_3} = \frac{\rho_0}{K_0} \frac{\partial p_0}{\partial x_3}. \tag{17}$$

In non-equilibrium conditions, the physical quantities $(\boldsymbol{v}(\boldsymbol{x}, t), \rho(\boldsymbol{x}, t), p(\boldsymbol{x}, t))^T$ can be considered as the sum of

the equilibrium quantities $(\boldsymbol{v_0}(\boldsymbol{x}), \rho_0(\boldsymbol{x}), p_0(\boldsymbol{x}))^T$ and the time-dependent perturbations $(\boldsymbol{v'}(\boldsymbol{x},t), \rho'(\boldsymbol{x},t), p'(\boldsymbol{x},t))^T$. Similarly, $K = \rho_0 \frac{dp}{d\rho}$ is defined and as in [15], we assume $K = K_0$.

Now, we can rewrite (11) as

$$\frac{\partial \rho'}{\partial t} + \nabla \cdot ((\rho_0 + \rho')\boldsymbol{v'}) = \frac{\partial \rho'}{\partial t} + \nabla \cdot (\rho_0 \boldsymbol{v'}) + \nabla \cdot (\rho' \boldsymbol{v'}) = 0. \tag{18}$$

Next, we try to simplify this equation. Using a Taylor expansion, we have

$$p(\rho) \approx p(\rho_0) + \rho' \frac{dp}{d\rho}(\rho_0) = p_0 + \frac{\rho'}{\rho_0} K, \tag{19}$$

which results in

$$\rho' = p' \frac{\rho_0}{K}, \tag{20}$$

because $p = p_0 + p'$. Due to $\frac{\partial \rho_0}{\partial x_1} = \frac{\partial \rho_0}{\partial x_2} = 0$, using the definition of divergence and the chain rule, we obtain

$$\nabla \cdot (\rho_0 \boldsymbol{v'}) = \rho_0 \nabla \cdot \boldsymbol{v'} - \frac{\rho_0}{K_0} \rho_0 g v_3', \tag{21}$$

where we apply (15) and (17). $\nabla \cdot (\rho' \boldsymbol{v'})$ can be considered negligible because, compared to the other two terms in (18), it involves the multiplication of two small perturbations. Finally, by inserting (20) and (21) into (18), we arrive at

$$\frac{\partial p'}{\partial t} + K \sum_{k=1}^{3} \frac{\partial v_k'}{\partial x_k} - \rho_0 g v_3' = 0, \tag{22}$$

where we divide both sides of the equation by $\frac{\rho_0}{K}$ and drop the subscript of K because of the previous assumption $K = K_0$.

Similarly, (12) can be rewritten with perturbations as

$$\frac{\partial (\rho_0 + \rho')\boldsymbol{v'}}{\partial t} + \nabla \cdot (\boldsymbol{v'} \otimes ((\rho_0 + \rho')\boldsymbol{v'})) + \nabla \cdot (\boldsymbol{I}(p_0 + p')) + (\rho_0 + \rho') g \boldsymbol{e_3} = 0, \tag{23}$$

where $\boldsymbol{e_3} = (0, 0, 1)^T$. Because $\rho' \boldsymbol{v'} \approx 0$ and $v_i' v_j' \approx 0$, the first term can be approximated as $\rho_0 \frac{\partial \boldsymbol{v'}}{\partial t}$ and the second term can be approximated as 0. Using the definitions $\frac{\partial p_0}{\partial x_1} = \frac{\partial p_0}{\partial x_2} = 0$ and (15), the third term can be simplified as $\nabla \cdot (\boldsymbol{I} p') - \rho_0 g \boldsymbol{e_3}$. By inserting (20), the fourth term can be rewritten as $\rho_o g (1 + \frac{p'}{K}) \boldsymbol{e_3}$. Finally, by combining these, (23) can be simplified as

$$\rho_0 \frac{\partial \boldsymbol{v'}}{\partial t} + \nabla \cdot (\boldsymbol{I} p') + \frac{\rho_0 g p'}{K} \boldsymbol{e_3} = 0, \tag{24}$$

where we replace the approximate equality sign with an equality sign.

Since $\boldsymbol{v'}$ is a three-dimensional vector, (22) and (24) together form the acoustic wave system consisting of four PDEs. Considering the material parameters of the Earth's oceans, the terms involving gravitational acceleration g in the system can be neglected [8]. Thus, the system can be written in a matrix form similar to (8):

$$\frac{\partial \boldsymbol{q^{ac}}}{\partial t} + \boldsymbol{A^{ac}}(\boldsymbol{x}) \frac{\partial \boldsymbol{q^{ac}}}{\partial x_1} + \boldsymbol{B^{ac}}(\boldsymbol{x}) \frac{\partial \boldsymbol{q^{ac}}}{\partial x_2} + \boldsymbol{C^{ac}}(\boldsymbol{x}) \frac{\partial \boldsymbol{q^{ac}}}{\partial x_3} = 0, \tag{25}$$

where $\boldsymbol{q^{ac}} = (p', v_1', v_2', v_3')^T$ and $\boldsymbol{A^{ac}}(\boldsymbol{x})$, $\boldsymbol{B^{ac}}(\boldsymbol{x})$, and $\boldsymbol{C^{ac}}(\boldsymbol{x})$ are flux matrices. They also share the same set of eigenvalues $(-c_{ac}, 0, c_{ac})$, where c_{ac} is the acoustic wave speed given by $c_{ac} = \sqrt{\frac{K}{\rho_0}}$. The linear combination $\hat{\boldsymbol{A}}^{\boldsymbol{ac}}(\boldsymbol{x}, \boldsymbol{n}) = n_1 \boldsymbol{A^{ac}}(\boldsymbol{x}) + n_2 \boldsymbol{B^{ac}}(\boldsymbol{x}) + n_3 \boldsymbol{C^{ac}}(\boldsymbol{x})$ is given by

$$\hat{\boldsymbol{A}}^{\boldsymbol{ac}}(\boldsymbol{x}, \boldsymbol{n}) = \begin{pmatrix} 0 & n_1 K & n_2 K & n_3 K \\ \frac{n_1}{\rho_0} & 0 & 0 & 0 \\ \frac{n_2}{\rho_0} & 0 & 0 & 0 \\ \frac{n_3}{\rho_0} & 0 & 0 & 0 \end{pmatrix}. \tag{26}$$

The acoustic wave model is essentially a special case of the elastic wave model. In the elastic wave model, if we set $\mu = 0$, the normal stresses in all three directions become equal, and the shear stresses reduce to zero. Although the solution vector still has nine dimensions, only four of them are independent and necessary. By retaining only these four independent variables, (8) reduces to (25). This aligns with the physical interpretation: in a fluid, the pressure (corresponding to the negative of the stress in a solid) is equal in all directions, and the fluid's ability to flow means there are no shear strains or shear stresses. It is important to note that normal stress and pressure are opposites, as compressive stress in solids is defined as negative, while compressive pressure in fluids is positive.

In the current version of SeisSol, the acoustic wave model is implemented by setting $\mu = 0$ in the elastic wave model (with nine DOFs). To create a standalone acoustic model for SeisSol's proxy application, we duplicate the source code of the elastic implementation and modify the matrices used for numerical computations and their dimensions, as well as the functions responsible for reading and writing these matrices, to accommodate the new physical model with four DOFs. However, we do not modify the functions related to the computations themselves, as the form of the PDEs describing both models is the same. Finally, the CMake system is also updated to reflect the configuration and build process for the acoustic implementation. To verify the correctness of the acoustic model, we extract the modified functions and compare their outputs with those from the elastic implementation with $\mu = 0$. In the next section, we will evaluate the performance improvements of the standalone acoustic wave model with only four DOFs.

III. Performance Evaluation

Many large-scale simulation software packages often come with a proxy application or mini application. These applications are typically simple and lightweight but capture the core computational characteristics of the main simulation program. They allow for performance evaluation and tuning of the main application to some extent without requiring complex inputs or consuming significant computational resources. In this section, we evaluate and compare the performance of SeisSol's proxy application based on the elastic (with $\mu = 0$) and acoustic models.

A. Test System and Software Environment

Since the target application is a shared-memory application and, throughout this work, it runs only on a single node, network-related system parameters are omitted, and unless otherwise specified, the following parameters refer to those only on a single node (the complete information can be obtained at the LRZ):

- Cluster Name: CoolMUC-2 of Linux Cluster at LRZ
- CPU Name: Intel(R) Xeon(R) CPU E5-2697 v3
- CPU Base Frequency: 2.60 GHz
- CPU Type: Intel Xeon Haswell EN/EP/EX Processor
- Number of Sockets: 2
- Number of NUMA Domains: 4
- Number of Cores per Socket: 14
- Number of Threads per Core: 2
- Cache Topology: L1d (32 KBytes per core), L1i (32 KBytes per core), L2 (256 KBytes per core), and L3 (8.75 MBytes per NUMA domain)
- Memory Capacity: 128 GByte
- Memory Bandwidth: 120 GByte/s (stream)
- Operating System: SUSE Linux Enterprise Server 15 SP1

Due to platform limitations, we did not fix the clock frequency. However, across all experiments, no significant imbalance or substantial deviation from the base frequency among the cores was observed. On a single node, although there are four NUMA domains, when considering NUMA effects, the first two domains share the same local and remote memory access latencies, as do the other two domains, because the memory is physically distributed across two sockets.

The full dependencies for SeisSol can be found in its documentation. Due to space limitations, we only list the specific compiler and MPI implementation used, as well as the external software employed for the performance analysis:

- intel-oneapi-compilers/2021.4.0
- intel-oneapi-mpi/2021.4.0-intel
- likwid/5.2.2-intel21

The names of these packages correspond to the names of the environment modules on the system. Although the MPI implementation is not used by the shared-memory target application, it is required by many other modules of SeisSol, making it essential for a complete build.

B. Target Application

The complete source code for SeisSol can be accessed in this GitHub repository. After properly installing all dependencies, the entire project is built using CMake and Make (with the option to build only the proxy application). A variety of build flags and their predefined options are supported. Selections such as the physical model, the order of the numerical method, the host architecture, and the data precision all need to be specified at this stage. A higher order of the numerical method usually results in greater accuracy, but it also necessitates smaller time steps and more basis functions, which demand additional computational time and resources. Theoretically, the increase in computation time due to the reduced time step and the increased number of basis functions follows the factors $\frac{2O-1}{2o-1}$ and $\frac{O(O+1)(O+2)}{o(o+1)(o+2)}$ respectively, where O and o represent the higher and lower orders of numerical accuracy, respectively. For order O, in the elastic model, the number of numerical unknowns is given by $\frac{O(O+1)(O+2)}{6} \times 9$, where the 9 in the equation represents the dimension of the PDEs describing the elastic model. It is important to note that the unknowns here refer to the numerical unknowns that need to be solved for each discrete cell at every time step, not the unknowns in the theoretical system of PDEs. Therefore, for a self-contained acoustic model, the 9 in the equation should be replaced with 4. Informing the compiler about the architecture of the host is also crucial for optimized compilation. The compiler can generate optimized machine code based on the SIMD instruction set, pipelining characteristics, cache architecture, and memory access patterns specific to the host architecture. Finally, unless otherwise specified, all experiments in this report were conducted using double precision.

The target application is a shared-memory C++ program parallelized with OpenMP, and it can be executed like any typical OpenMP program. Besides configuring the number of threads and their pinning, the target application requires only three inputs: the number of timesteps (`timesteps`), the number of cells (`cells`), and the kernel choice (`kernel`). These can be specified on the command line when running the program. Since all experiments were conducted on a cluster, a SLURM script is used to submit jobs. The full script is provided by the LRZ. The proxy application does not provide an option to configure material parameters, so it is necessary to modify the source code and recompile it when setting μ for the elastic model.

In this work, we only consider selecting the kernel `all` for the application because it contains all available kernels, such as `neigh`, `local`, and `ader`. In this case, the main logic of the program is to perform `timesteps` iterations, where in each iteration, functions `void computeLocalIntegration()` and `void computeNeighboringIntegration()` are executed sequentially. For simplicity, we can assume that in these two functions, some numerical operations reflecting the full computational characteristics of SeisSol are executed in parallel on each cell using OpenMP with static scheduling. Given the main purpose of this work, we will no longer delve further into the details of the source code or the implementation of numerical methods, but will instead focus more on program performance from a hardware perspective.

The target application records and reports the kernel's execution time itself, and this was used for all experiments rather than the total program runtime or the time measured by external software. Prior to starting the timing, the kernel is run for one time step on all cells to exclude the overhead of the operating system initializing the OpenMP thread pool and the first loading of the last-level cache (LLC). This ensures more accurate timing, which is especially important for shorter runtimes. In addition to the computation time, the proxy application also reports a non-zero FLOP number and

a hardware FLOP number. The former refers to the number of floating-point operations (FLOPs, note the distinction from FLOPS, which refers to the number of FLOPs per second) in numerical computations that are unavoidable because of the non-zero operands (theoretically, if the operands of a FLOP are known to be zero before the execution, the FLOP is unnecessary). The latter refers to the actual number of FLOPs executed by the machine. Therefore, the hardware FLOP count is always greater than or equal to the non-zero FLOP count and is used to construct the roofline models. Finally, the theoretical amount of data required for the numerical computation is output in bytes, but for accuracy considerations, this metric is not used to calculate the actual bandwidth because of caching.

C. Roofline Models

The construction of the roofline models utilizes LIKWID with reference to this tutorial. LIKWID is a toolkit for performance analysis and monitoring, specifically designed for the hardware performance counters of modern processors. The building of a roofline model involves two parts. First, the roofline is drawn based on the peak performance and peak bandwidth of the node. The roofline is specific to the machine, so experiments with different configurations running on the same node can reuse the same roofline. Second, points representing the actual performance and bandwidth of the application are added to the roofline. Whether benchmarking the machine or conducting the performance analysis on the target application, we always utilize all 28 physical cores available on a single node.

In a roofline model, performance specifically refers to the computational performance of a processor. This can be characterized by various metrics, such as FLOPS, CPI (cycles per instruction), clock frequency, throughput, and latency. When constructing roofline models, one of these metrics can be flexibly chosen to represent the single-node computational performance of a system. However, in the field of scientific computing, FLOPS is the most commonly used metric, typically expressed in units like GFLOPS, TFLOPS, or PFLOPS. The vertical axis of the roofline model represents performance, so the peak performance is shown as a horizontal line in the model. The simplest way to determine the peak performance is by consulting the machine's documentation. For example, the peak FLOPS can be obtained by multiplying the number of cores, instruction width (introduced by vectorization), the number of fused multiply-add (FMA) operations, the number of instructions per cycle, and the maximum clock frequency. However, compared to self-measured or actual performance during use, this is an overly ideal and theoretical estimate. Many factors limit the practicality of using this result to build a roofline model. For instance, no real-world application or benchmark can be fully parallelized and vectorized. Additionally, the maximum number of FMA operations supported by a single machine instruction may not align with the computational patterns of the application. Furthermore, when calculating the ideal number of instructions processed per cycle, issues such as cache misses, branch prediction failures, and pipeline underutilization are often overlooked. Finally, due to power consumption and thermal constraints, processors typically cannot sustain maximum frequency over extended periods. Therefore, obtaining the peak FLOPS by running appropriate benchmarks is a more practical approach.

The `likwid-bench` command in LIKWID provides various streaming-access benchmarks, some of which can be used to measure the peak FLOPS. This access pattern is quite common in scientific applications, such as sequential reading and writing of a matrix. The naming convention for these kernels follows this format: `peakflops[_precision][_vectorization][_fma]`. The kernel precision can be either single or double (with double precision as the default if not specified). The vectorization options include scalar (default if not specified), SSE, AVX, and AVX512, corresponding to instruction widths of 64 bits (or 32 bits for single precision), 128 bits, 256 bits, and 512 bits, respectively. The test system supports up to the 256-bit AVX2 instruction set. When FMA is enabled, each instruction can handle both a multiply and an add operation simultaneously; otherwise, it defaults to handling only one operation. The kernels load a specified amount of data from contiguous memory into the L1 cache of each core averagely (the striding allocation feature of `likwid-bench` is used to avoid utilizing logical cores), perform numerous multiply and add operations, and measure the FLOPS. As a result, factors like load operations and loop mechanics are ignored, and the time is almost exclusively due to pure processor computation. In all the experiments conducted in this paper, we disabled simultaneous multithreading (SMT), which is common in scientific computing. This is because the software threads in most scientific applications typically have balanced workloads and share the same pipeline resources. As a result, SMT often fails to fully utilize idle hardware resources to improve throughput. Instead, it can introduce additional latency and power consumption due to hardware scheduling. Furthermore, SMT does not provide additional cache or memory bandwidth at the hardware level for a single core, which leads to contention for cache and memory bandwidth when accessing large datasets or matrices in parallel.

Another input for constructing a roofline model is the maximum data throughput, with the most typical metric being the maximum memory bandwidth of a single node. Similarly, depending on the perspective of interest, data throughput can also be measured by cache bandwidth at various levels, network bandwidth, or I/O bandwidth. Since the horizontal axis of the roofline model represents computational intensity (measured in FLOP/Byte), the maximum data throughput uniquely defines a line passing through the origin. Together with the horizontal line mentioned earlier, they form the piecewise function that represents the complete roofline model. The performance and bandwidth of an application running on this machine are represented by points that lie below the curve of this function. Similarly, the theoretical maximum memory bandwidth can be obtained from the machine's data sheet, which is calculated by the manufacturer based on parameters such as memory

type, channel configuration, and clock frequency. This figure is often overly idealized, as actual memory bandwidth can be influenced by factors like memory access patterns, cache effects, load contention, and system configuration. Therefore, the maximum memory bandwidth measured by benchmarking under specific workloads or test conditions is more practical and reproducible.

`likwid-bench` also provides various benchmarks for measuring memory bandwidth, such as `load (scalar = A[i])`, `copy (A[i] = B[i])`, `stream (A[i] = B[i] + scalar * C[i])`, and `triad (A[i] = B[i] + C[i] * D[i])`. The differences between these benchmarks and their various versions lie in factors such as read-write ratios, operation types, data precision, and support for vectorization. Notably, each kernel has a corresponding non-temporal version (indicated by `_mem` in the name), which ensures that data are written directly to memory during write operations, rather than being temporarily stored in the cache or registers for reuse. In addition, `likwid-bench` supports parallel first-touch initialization before starting the benchmarks to avoid the impact of NUMA effects on memory access performance, which is crucial when measuring maximum memory bandwidth. Finally, referring to the tutorial mentioned earlier, we always set the stream size of the benchmarks that measure bandwidth to 2 GByte, which naturally avoids the influence of cache effects on measurement accuracy.

Using `likwid-bench`, we conducted multiple benchmarks, repeating each test five times and averaging the results to reduce uncertainty caused by system performance fluctuations. The corresponding computational performance (represented by horizontal lines) and memory bandwidth (represented by lines passing through the origin) are plotted together in Fig. 1. These pairs form multiple roofline models. For FLOPS, there is no significant difference between single-precision and double-precision results. This could be due to the processor's hardware being well-optimized for double precision computations or similar throughput for double and single precision in the floating-point units. For the kernels `stream` and `stream_sp`, there is also almost no difference in bandwidth. Additionally, the introductions of AVX ($\times 4$) and FMA ($\times 2$) yield performance gains that are almost in line with theoretical expectations, indicating that the benchmarks nearly fully utilize these hardware features, which is unlikely to be the case in real-world applications. For memory bandwidth, the read-write ratios of `load`, `stream`, and `copy` are 1:0, 2:1, and 1:1, respectively. While reads and writes share the same memory bandwidth, due to the hardware design, the bandwidth is positively correlated with the read-write ratio, although not linearly. The difference between `stream_mem_avx_fma` and `stream_avx_fma` lies in that the former always ensures data is written back to memory during write operations, significantly increasing the measured bandwidth and providing a more accurate reflection of the memory system performance. Additionally, by comparing the read-write ratios of both, it becomes evident that `stream_avx_fma` rarely writes data back to the memory. Finally, it is clear that even on the same machine, selecting different benchmarks can have a significant impact on roofline model construction and performance analysis results.

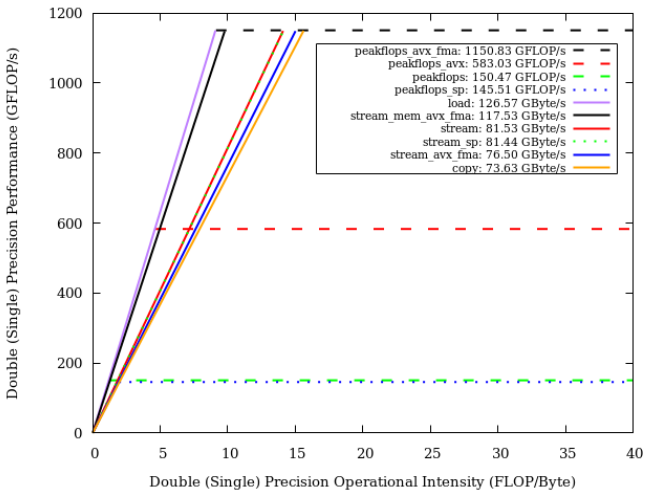

Fig. 1. Roofline models based on LIKWID streaming-access benchmarks. Except for the kernels `peakflops_sp` and `stream_sp`, whose data and FLOPs are in single precision, all the other kernels use double precision. When constructing a roofline model for a specific machine, it's uncommon to plot performance parameters for both single and double precision on the same graph. Here, it's done merely for comparison of the benchmark results. Specifically, the blue dotted line representing `peakflops_sp` cannot form a roofline model with the solid lines from other kernels; it can only combine with the green dotted line representing `stream_sp`, and vice versa.

After understanding that we should utilize benchmark results rather than theoretical values, we still need to address another question: Which benchmark results can be considered as the maximum computational performance and maximum memory bandwidth of the machine? The most straightforward approach is to conduct all available benchmarks on the test system and select the combination with the highest results. However, this may not always be meaningful for performance analysis; it often also requires considering the characteristics of the target application. For FLOPS, we have already discussed the rationale behind the results of the benchmark `peakflops`. Additionally, due to the hardware support of the machine, we choose the version optimized for AVX with FMA enabled. When measuring the memory bandwidth, an important parameter is the read-write ratio. For example, on x86-64 CPUs, the test `load` with purely sequential read operations can typically achieve the maximum memory bandwidth, thanks to optimizations like data prefetching. However, selecting a kernel that more closely matches the read-write ratio of the target application is usually more meaningful. Therefore, we choose the kernel `stream` to match the read-write ratio of approximately 2:1 (which will be discussed later) of SeisSol-proxy. Furthermore, kernels that utilize sequential access often achieve higher bandwidth compared to those that use random access. Lastly, we select the non-temporal version of the kernel. Although more frequent memory writes may

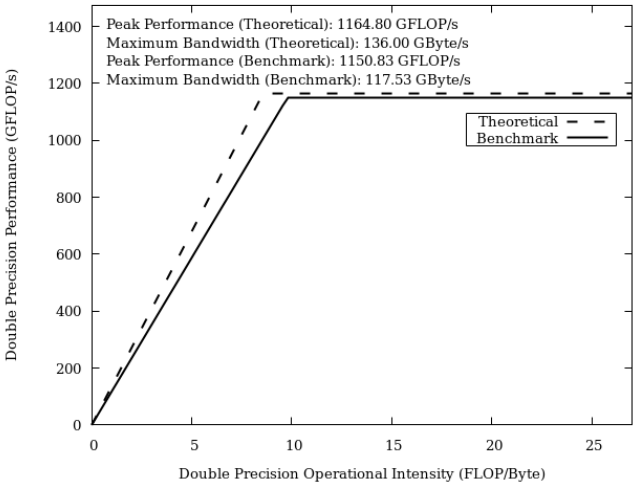

Fig. 2. Roofline models based on benchmarks and theoretical calculations. The benchmarks used to measure the maximum performance and bandwidth are `peakflops_avx_fma` and `stream_mem_avx_fma`, respectively. The theoretical maximum performance is calculated by multiplying the number of cores, the instruction width, the number of FMA operations, the number of instructions per cycle, and the maximum clock frequency, while the theoretical maximum bandwidth is obtained from Intel's documentation.

increase testing time, this choice better reflects the actual memory performance of the system, particularly for analyzing applications that involve a significant amount of data write-back operations. The temporal version of the kernel can also be useful in certain situations, such as when evaluating the data locality of an application or the impact of the machine's cache hierarchy on performance. The comparison of the roofline models constructed based on theoretical calculations and benchmark tests is illustrated in Fig. 2. For FLOPS, the benchmark result is very close to the theoretical value because the test conditions are highly idealized, with nearly 100% parallelization, vectorization, and use of FMA operations. For bandwidth, the test result differs significantly from the value provided by Intel, as bandwidth is also affected by factors such as specific memory modules, motherboard, and system configuration.

Finally, to incorporate the target application into the roofline model, its FLOPS and computational intensity need to be determined. The former is calculated from the FLOP and runtime output by SeisSol-proxy itself, while the latter is derived from the memory data volume reported by `likwid-perfctr` divided by the runtime provided by the application. `likwid-perfctr` can monitor and collect various performance metrics through hardware performance counters, organizing them into predefined event groups related to common performance analysis scenarios such as cache access, memory bandwidth, and floating-point performance. We select the `MEM` event group, which provides numerous statistics related to memory access. Specifically, it outputs the amount of data read and written between each core and memory, as well as the total data volume. This can be used to calculate the actual read-write ratio of the application, which differs from the theoretical data volume reported by the program itself. Although the `MEM` group also reports runtime and bandwidth, the recorded time includes overhead from LIKWID's configuration, which introduces significant absolute error, especially when measuring shorter runtimes. Therefore, we do not use this time or the bandwidth calculated from it. Additionally, `likwid-perfctr` supports pinning threads, which fixes the mapping of software threads to physical cores, avoiding the activation of SMT and preventing performance uncertainties due to thread floating. It is worth noting that we do not use LIKWID's Marker API to analyze specific regions of the application. Since the proxy application is written specifically for performance tuning, it does not contain many non-performance-related code blocks. Thus, it is reasonable to collect performance metrics for the entire program.

The complete roofline model for the elastic (with $\mu = 0$) implementation is shown in Fig. 3. The model can be divided into two regions based on the horizontal axis. The first region lies below the line passing through the origin, representing memory bandwidth. Applications that fall within this region are limited by memory bandwidth because they require a large amount of data to be read from memory while performing relatively few floating-point operations. The second region lies below the horizontal line, representing computational performance. Applications in this region are limited by the processor's computational capability, as they perform many floating-point operations but require relatively little data from memory. The roofline model primarily addresses two key questions regarding application performance: (1) Is the performance bottleneck due to the memory bandwidth or the processor? In other words, is the application memory-bound or compute-bound? This question can be answered even without running the program, as the computational intensity is determined solely by the code. However, it's important to note that the parameters used when constructing the model can affect the horizontal threshold dividing the regions. (2) What is the maximum achievable performance for the application, and how can it be optimized? The maximum performance of the application is defined by the roofline in each region. From the graph, it is evident that compute-bound applications can always achieve higher peak performance than memory-bound ones. Therefore, optimizing the program to push the application's performance point toward the upper-right corner is desirable (though further right movement has diminishing returns when beyond the threshold). Moving the point to the right typically involves optimizing memory access patterns, reducing unnecessary memory accesses, making better use of caching, efficiently utilizing the cache hierarchy, or compressing data to reduce the amount of data transferred. Moving the point upward usually involves increasing parallelism, vectorization, using more efficient mathematical algorithms, or leveraging hardware features such as SIMD and FMA instructions to improve computational efficiency. When the application's performance point is already close to the roofline, regardless of the region, it indicates that the performance is approaching the machine's limits. In this case, upgrading the hardware becomes a better option (e.g.,

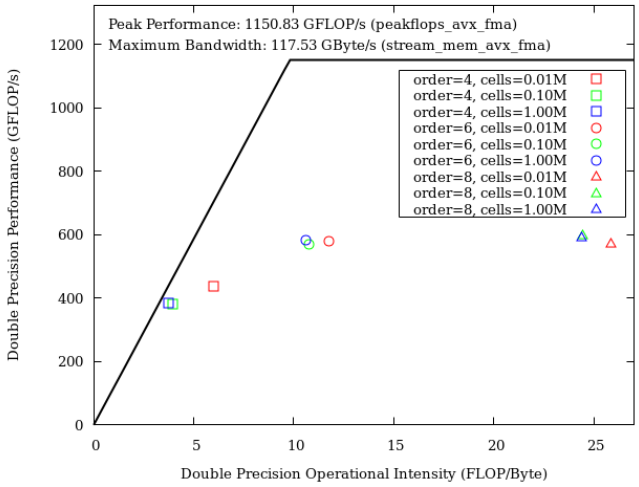

Fig. 3. The roofline model for SeisSol-proxy based on the elastic implementation with $\mu = 0$. A higher `order` indicates greater numerical accuracy and a larger computational workload. `cells` refers to the number of cells that need to be processed in parallel for each time step. Number of time steps: 100; kernel: `all`; number of threads: 28 (SMT disabled).

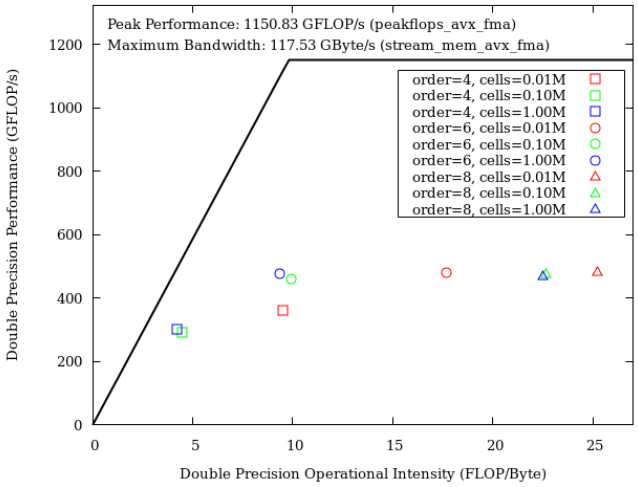

Fig. 4. The roofline model for SeisSol-proxy based on the acoustic implementation. A higher `order` indicates greater numerical accuracy and a larger computational workload. `cells` refers to the number of cells that need to be processed in parallel for each time step. Number of time steps: 100; kernel: `all`; number of threads: 28 (SMT disabled).

upgrading the memory for memory-bound applications or upgrading the processor for compute-bound ones). Even for the same application, as the numerical accuracy (order) increases, SeisSol-proxy shifts from being memory-bound to compute-bound. The performance differences between orders 6 and 8 are not significant, both being compute-bound, and further performance improvements can be achieved by optimizing the computation patterns. In contrast, for order 4, the performance is slightly lower due to the memory bandwidth being nearly maxed out, and further improvements would require upgrading the memory or optimizing the memory access patterns. The number of cells does not lead to significant performance differences, but when the number of cells is 0.01M, the computational intensity is slightly higher than in the other two cases. This is because, with a smaller data size, the cache effect becomes more pronounced, reducing the amount of data read from memory.

The complete roofline model for the acoustic implementation is shown in Fig. 4. Its characteristics do not significantly differ from those of the roofline model for the elastic implementation, but this does not mean the performance of the two implementations is identical in every aspect—this is one of the limitations of the roofline model. In fact, due to the reduced number of DOFs, both the runtime and memory data volume of the acoustic implementation are reduced by about half, as shown in Fig. 5. Note that this figure does not display the dynamic changes in the program's memory data volume over time but rather the relationship between the total memory data volume and the total runtime, which is why it is represented by data points instead of lines. Logarithmic axes are used to include all test cases in a single plot. The black dashed line mathematically passes through the origin with a slope equal to the maximum memory bandwidth. To compare memory bandwidth, parallel dashed lines can be drawn through the data points, where the bandwidth is reflected by the intercept (not mathematically equivalent) of these lines in the plot. Hence, all data points fall below the dashed line, consistent with the fact that the measured memory bandwidth of the target application is lower than the machine's maximum memory bandwidth. Compared to the elastic implementation, the acoustic implementation shows lower memory bandwidth under various test conditions, with this difference decreasing as the number of cells increases and the order rises. Additionally, as the number of cells increases and the order decreases, the memory bandwidth of the application tends to approach the maximum. Similarly, as shown in Fig. 6, compared to the elastic implementation, the acoustic implementation consistently achieves lower FLOPS across various test conditions, since the acoustic implementation is adapted from the elastic version by merely adjusting the relevant tensors and their dimensions, without dedicated optimization or independent design, but the variation in FLOPS is much smaller than that of memory bandwidth. Furthermore, the differences in FLOPS caused by varying the number of cells and the order are also smaller compared to those seen in memory bandwidth. These characteristics ultimately make the computational intensity difference between the two implementations not very significant. This implies that not all performance differences between the two implementations can be fully captured in the roofline model. Finally, due to the smaller data size in the acoustic implementation, the relative cache effect is more pronounced in the case of 0.01M cells compared to the elastic implementation, causing these points to be farther apart from the other two cases.

The roofline model, as a visual tool for performance analysis, provides a unified standard for evaluating performance

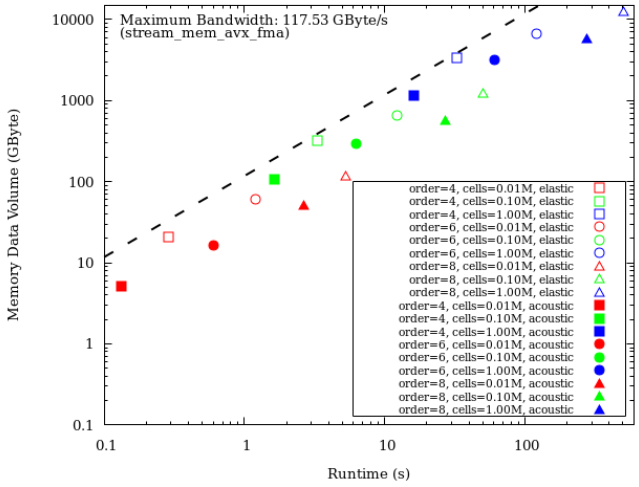

Fig. 5. Memory data volume vs runtime for SeisSol-proxy based on the elastic ($\mu = 0$) and acoustic implementations. A higher `order` indicates greater numerical accuracy and a larger computational workload. `cells` refers to the number of cells that need to be processed in parallel for each time step. Mathematically, the dashed line passes through the origin and has a slope equal to the maximum bandwidth. Number of time steps: 100; kernel: `all`; number of threads: 28 (SMT disabled).

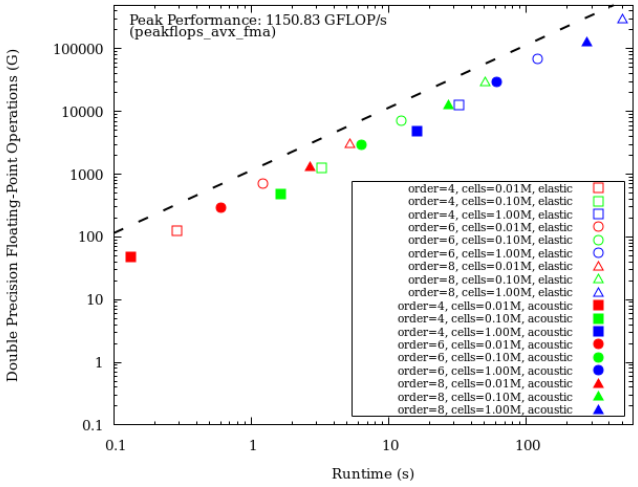

Fig. 6. Number of double precision FLOPs vs runtime for SeisSol-proxy based on the elastic ($\mu = 0$) and acoustic implementations. A higher `order` indicates greater numerical accuracy and a larger computational workload. `cells` refers to the number of cells that need to be processed in parallel for each time step. Mathematically, the dashed line passes through the origin and has a slope equal to the peak performance. Number of time steps: 100; kernel: `all`; number of threads: 28 (SMT disabled).

across different platforms and applications in an easily understandable way. It helps identify performance bottlenecks and guides optimization efforts. However, the roofline model also has its limitations: (1) It is often based on simplified assumptions, such as ideal memory access patterns and computational models, which may lead to inaccurate performance predictions in real-world scenarios; (2) The model may fail to capture subtle performance characteristics like memory access latency, cache effects, or thread contention, which can significantly impact performance in certain cases; (3) The model's effectiveness depends on accurate performance measurements and system parameters. If the input data is incorrect, it can result in misleading analysis; (4) While the roofline model is applicable to many compute- and memory-intensive tasks, its applicability may be limited in some contexts, such as GPU computing, heterogeneous systems, or large-scale, network-based multi-node systems; (5) In some dynamic application scenarios, the performance may fluctuate over time or with changes in input data, while the roofline model is typically static and may not effectively capture this dynamic behavior.

D. Cache Usage Analysis

The roofline models we construct primarily focus on the interaction between the main memory and processor. However, when evaluating performance, the caches within the memory subsystem are unavoidable. The LLC (L3 cache in the test system) is the final and longest-latency level in the memory hierarchy before the main memory. Any memory requests that miss in the LLC will be serviced by local or remote main memory, resulting in significant latency. Fig. 7 shows the L3 cache miss rates, calculated as the total number of L3 cache misses across all cores divided by the total number of memory requests, for the two implementations of SeisSol-proxy on the test system by using the `L3CACHE` event group of `likwid-perfctr`. When the order is 4, the L3 cache miss rates of the acoustic implementation increase. At this point, the L3 cache miss rate is highest for both models, and it continues to rise with the number of cells, reaching a maximum of 62% for the acoustic model. The gap between the two models also widens, with a maximum difference of 17%. When the order is 6, the acoustic implementation still shows a positive correlation with the number of cells, while the elastic implementation fluctuates around 6%. As the order increases, the L3 cache miss rates rapidly decline for both implementations and by the time the order reaches 8, they are nearly zero.

Similarly, the L2 cache miss rates are shown in Fig. 8. When the order is 4, the L2 cache miss rates remain the highest, peaking at approximately 22% for the elastic implementation. However, these rates show little variation with increasing cell count, and the values for the acoustic implementation are lower than those for the elastic one. At orders 6 and 8, except for the case of order 6 and cell count 0.10M, where the L2 miss rate for the acoustic implementation significantly decreases, the miss rates generally increase.

In the memory subsystem, typically only adjacent levels of hardware can interact directly. The data paths between different cache levels serve multiple functions and thus closely influence performance. Fig. 9 illustrates the bandwidth between the L3 and L2 caches. A black dashed line representing the maximum memory bandwidth is also added to the figure. Although cache bandwidth generally exceeds memory bandwidth, only when the order is 8 does the cache bandwidth between L3 and L2 surpass this maximum value. A significant

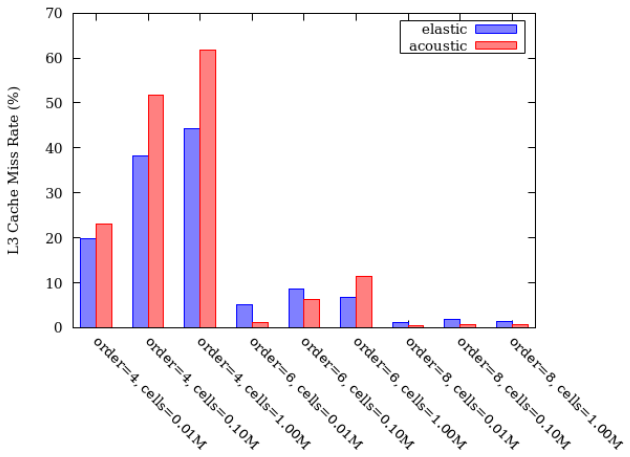

Fig. 7. L3 cache miss rates of SeisSol-proxy based on the elastic ($\mu = 0$) and acoustic implementations. A higher `order` indicates greater numerical accuracy and a larger computational workload. `cells` refers to the number of cells that need to be processed in parallel for each time step. Number of time steps: 100; kernel: `all`; number of threads: 28 (SMT disabled).

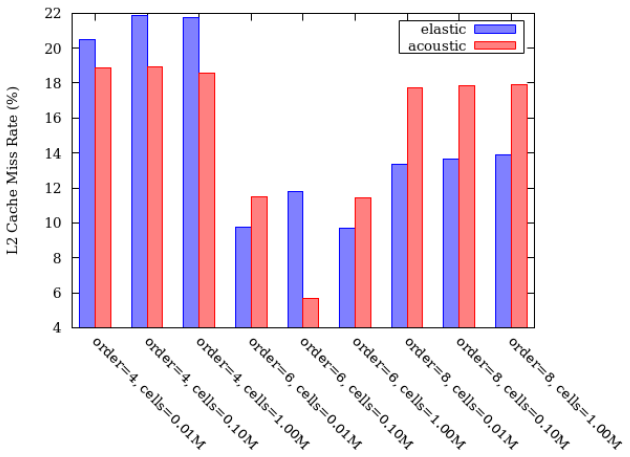

Fig. 8. L2 cache miss rates of SeisSol-proxy based on the elastic ($\mu = 0$) and acoustic implementations. A higher `order` indicates greater numerical accuracy and a larger computational workload. `cells` refers to the number of cells that need to be processed in parallel for each time step. Number of time steps: 100; kernel: `all`; number of threads: 28 (SMT disabled).

feature is that the cache bandwidth from L2 to L3 is negligible compared to that from L3 to L2 (note that a logarithmic scale is used on the vertical axis for better visualization). The former is due to the L2 cache replacement under the write-back policy, while the latter results from L3 cache hits. Additionally, both may be influenced by cache coherence maintenance and cross-core communication. For orders 6 and 8, the cell count does not significantly affect the bandwidth between L3 and L2 caches. However, the acoustic implementation increases the L3 to L2 cache bandwidth and decreases the bandwidth in the reverse direction. For order 4, in cases where the cell count is 0.10M and 1.00M, the changes in cache bandwidth between L3 and L2 induced by the acoustic implementation are completely opposite.

When a processor accesses data or instructions, the L1 cache is the first level of cache it interacts with, making its miss rate and bandwidth crucial performance metrics. However, due to the lack of hardware counter support, we did not conduct related experiments.

False sharing is a common and significant performance issue in multi-core systems. The root cause lies in the fact that the unit of cache coherence across cores is a cache line. When multiple cores attempt to access different variables (at least one write) located in the same cache line, the processor constantly exchanges and synchronizes the entire cache line, leading to performance degradation. False sharing occurring at the LLC has a particularly severe impact on performance due to the need for memory synchronization, especially in multi-socket nodes with remote memory access. Using the `FALSE_SHARE` event group in `likwid-perfctr`, we measured both local and remote false sharing occurrences at the LLC and the amount of data updated as a result. For both implementations, the values were negligible, so we do not present the results.

Typically, when optimizing cache usage, strategies such as adjusting data structures and memory access patterns or configuring caches at the hardware level are considered. However, the process is often complex and targeted. Even for the same application, measured results can vary greatly under different test conditions and on different machines, due to factors such as varying cache architectures, memory hierarchies, and workload characteristics. Moreover, optimizations based on a specific scenario may even result in negative effects for other scenarios.

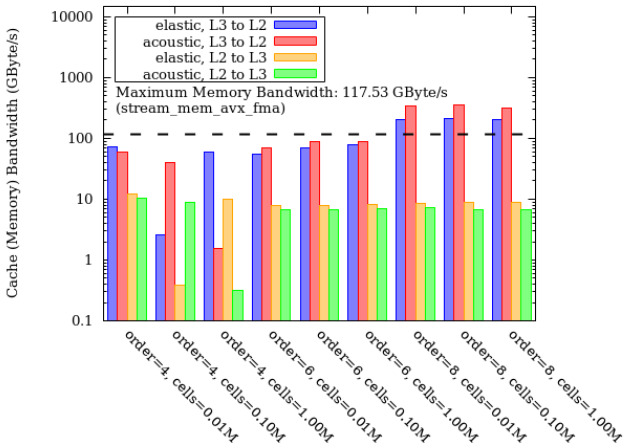

Fig. 9. Bandwidth between the L2 and L3 caches of SeisSol-proxy based on the elastic ($\mu = 0$) and acoustic implementations. The horizontal dashed line represents the maximum memory bandwidth. A higher `order` indicates greater numerical accuracy and a larger computational workload. `cells` refers to the number of cells that need to be processed in parallel for each time step. Number of time steps: 100; kernel: `all`; number of threads: 28 (SMT disabled).

IV. Conclusion and Future Work

In this paper, we efficiently implement the 4-DOF acoustic model, a special case of the 9-DOF elastic model, in SeisSol-proxy by tuning tensor parameters within the existing elastic framework. The roofline models built on the LRZ's CoolMUC-2 system show that the performance bottlenecks of this standalone implementation remain largely unchanged, as they are governed by inherent limitations in the computational kernels. They also provide valuable insights to identify the architectural boundaries of SeisSol-proxy's performance and highlight potential optimization directions on the test system. Additionally, under various typical test conditions, the runtime and memory data volume of the implementation are reduced by about 50%, while the cache miss rate fluctuates, with a maximum increase of approximately 17% and a maximum decrease of about 6%.

Looking ahead, several avenues for future work have emerged. First, the current work focuses on the proxy application designed specifically for performance tuning within the SeisSol framework. While this mini application demonstrates the feasibility of implementing the acoustic model, it serves as a preliminary step towards a more comprehensive implementation. Future research will aim to extend this mini application into the full acoustic model in SeisSol. This involves not only enhancing the model's capabilities to handle complex seismic wave phenomena but also rigorously evaluating its performance against a broader set of benchmarks to understand its behavior in real-world scenarios. This future work will ultimately contribute to a more robust and efficient tool for simulating seismic events.

Second, due to the characteristics of the roofline model, all experiments were conducted on a single node. While this approach provides valuable insights within a constrained environment, it is crucial to consider scalability in future work, especially for applications running on multi-node clusters. This includes evaluating the model's efficiency when scaling across multiple nodes and understanding how factors such as communication overhead and load balancing affect overall performance. By conducting these scalability tests, we aim to ensure that the full implementation can effectively leverage the capabilities of modern cluster architectures, ultimately facilitating more realistic and large-scale seismic simulations in real-world applications. Additionally, it is also worth exploring the construction of a three-dimensional roofline model that accounts for computational power, memory bandwidth, and network bandwidth in distributed memory systems.

Finally, while this study focuses on multi-threaded CPU performance, future work could extend the roofline analysis to GPU-based executions of SeisSol. Given the increasing importance of GPU acceleration in HPC, analyzing how the acoustic model performs on GPU architectures will be crucial for pushing the limits of seismic simulations. This would also involve investigating the interplay between GPU memory bandwidth and computational throughput, as well as exploring hybrid CPU-GPU implementations.

Acknowledgment

I would like to express my sincere gratitude to Professor Michael Bader for giving me the opportunity to be involved in this project. His support and trust have been instrumental in the success of this work. I am also deeply thankful to my advisors, Vikas Kurapati and Sebastian Wolf, who consistently provided me with detailed and insightful feedback in a timely manner. Their expertise and dedication greatly contributed to my research experience, and I am truly grateful for their invaluable guidance and support.

References

[1] L. Krenz, C. Uphoff, T. Ulrich, A.-A. Gabriel, L. S. Abrahams, E. M. Dunham, and M. Bader, "3D acoustic-elastic coupling with gravity: the dynamics of the 2018 Palu, Sulawesi earthquake and tsunami," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '21, (New York, NY, USA), pp. 1–14, Association for Computing Machinery, Nov. 2021.

[2] J. S. Hesthaven and T. Warburton, *Nodal Discontinuous Galerkin Methods*, vol. 54 of *Texts in Applied Mathematics*. New York, NY: Springer, 2008.

[3] V. A. Titarev and E. F. Toro, "ADER: Arbitrary High Order Godunov Approach," *Journal of Scientific Computing*, vol. 17, pp. 609–618, Dec. 2002.

[4] M. Dumbser and M. Käser, "An arbitrary high-order discontinuous Galerkin method for elastic waves on unstructured meshes — II. The three-dimensional isotropic case," *Geophysical Journal International*, vol. 167, pp. 319–336, Oct. 2006.

[5] J. De la Puente, M. Dumbser, and H. Igel, "Discontinuous Galerkin Methods for Wave Propagation in Poroelastic Media," *Geophysics*, vol. 73, Sept. 2008.

[6] M. Käser, M. Dumbser, J. De La Puente, and H. Igel, "An arbitrary high-order Discontinuous Galerkin method for elastic waves on unstructured meshes — III. Viscoelastic attenuation," *Geophysical Journal International*, vol. 168, pp. 224–242, Jan. 2007.

[7] S. Wollherr, A.-A. Gabriel, and C. Uphoff, "Off-fault plasticity in three-dimensional dynamic rupture simulations using a modal Discontinuous Galerkin method on unstructured meshes: implementation, verification and application," *Geophysical Journal International*, vol. 214, pp. 1556–1584, Sept. 2018.

[8] L. S. Abrahams, L. Krenz, E. M. Dunham, A.-A. Gabriel, and T. Saito, "Comparison of methods for coupled earthquake and tsunami modelling," *Geophysical Journal International*, vol. 234, pp. 404–426, July 2023.

[9] S. Williams, A. Waterman, and D. Patterson, "Roofline: An Insightful Visual Performance Model for Floating-Point Programs and Multicore Architectures," Tech. Rep. 1407078, Sept. 2009.

[10] L. D. S. Krenz, *A Fully Coupled Model for Petascale Earthquake-Tsunami and Earthquake-Sound Simulations*. PhD thesis, Technische Universität München, 2024.

[11] R. J. LeVeque, *Finite Volume Methods for Hyperbolic Problems*. Cambridge Texts in Applied Mathematics, Cambridge: Cambridge University Press, 2002.

[12] P. M. Shearer, "Introduction to Seismology," May 2019. ISBN: 9781316877111 Publisher: Cambridge University Press.

[13] K. Aki and P. Richards, *Quantitative Seismology, 2nd edition*. Mill Valley, California New York: University Science Books,U.S., 2nd edition ed., Apr. 2009.

[14] L. D. Landau, L. P. Pitaevskii, A. M. Kosevich, and E. M. Lifshitz, *Theory of Elasticity: Volume 7*. Amsterdam Heidelberg: Butterworth-Heinemann, 3rd edition ed., Jan. 1986.

[15] G. C. Lotto and E. M. Dunham, "High-order finite difference modeling of tsunami generation in a compressible ocean from offshore earthquakes," *Computational Geosciences*, vol. 19, pp. 327–340, Apr. 2015.

[16] W. A. Strauss, *Partial Differential Equations: An Introduction*. New York: John Wiley & Sons Inc, 2nd edition ed., Dec. 2007.

[17] P. K. Kundu, I. M. Cohen, and D. R. Dowling, *Fluid Mechanics*. Academic Press, 6th edition ed., June 2015.