Efficient implementation of modern entropy stable and kinetic energy preserving discontinuous Galerkin methods for conservation laws

Hendrik Ranocha[*1], Michael Schlottke-Lakemper[†2], Jesse Chan[‡3], Andrés M. Rueda-Ramírez[§4], Andrew R. Winters[¶5], Florian Hindenlang[6], and Gregor J. Gassner[**7]

[1]Applied Mathematics, University of Hamburg, Germany

[2]High-Performance Computing Center Stuttgart, University of Stuttgart, Germany

[3]Computational and Applied Mathematics, Rice University, USA

[4]Department of Mathematics and Computer Science, University of Cologne, Germany

[5]Computational Mathematics, Division of Applied Mathematics, Linköping University, Sweden

[6]Max Planck Institute for Plasma Physics, NMPP division, Garching, Germany

[7]Department of Mathematics and Computer Science, Center for Data and Simulation Science, University of Cologne, Germany

April 4, 2022

Many modern discontinuous Galerkin (DG) methods for conservation laws make use of summation by parts operators and flux differencing to achieve kinetic energy preservation or entropy stability. While these techniques increase the robustness of DG methods significantly, they are also computationally more demanding than standard weak form nodal DG methods. We present several implementation techniques to improve the efficiency of flux differencing DG methods that use tensor product quadrilateral or hexahedral elements, in 2D or 3D respectively. Focus is mostly given to CPUs and DG methods for the compressible Euler equations, although these techniques are generally also useful for other physical systems including the compressible Navier-Stokes and magnetohydrodynamics equations. We present results using two open source codes, Trixi.jl written in Julia and FLUXO written in Fortran, to demonstrate that our proposed implementation techniques are applicable to different code bases and programming languages.

Key words. flux differencing, entropy stability, conservation laws, summation by parts, discontinuous Galerkin

[*]ORCID: 0000-0002-3456-2277
[†]ORCID: 0000-0002-3195-2536
[‡]ORCID: 0000-0003-2077-3636
[§]ORCID: 0000-0001-6557-9162
[¶]ORCID: 0000-0002-5902-1522
ORCID: 0000-0002-0439-249X
[**]ORCID: 0000-0002-1752-1158

1 Introduction

Stability and robustness of numerical methods are key to efficient and reliable simulations. However, these properties are not trivial to obtain, in particular for high-order methods. Usually, techniques to improve the stability and (numerical) robustness also increase the computational complexity and costs. Thus, special care must be taken when implementing these methods to avoid losing (too much) efficiency in practice [25, 41].

In this article, we focus on so-called flux differencing schemes. These methods have their origins in the second-order accurate entropy-conserving methods of Tadmor [68, 69] and have successfully been extended to high-order methods in periodic [40] and bounded domains [10, 15, 46]. Flux differencing methods are not only useful for constructing entropy-conservative or -dissipative (EC/ED) methods but also to recover split forms used, for example, in finite difference methods [24]. While flux differencing methods are central-type schemes without explicit dissipation, they have been demonstrated to increase the robustness of numerical methods significantly [3, 19, 34, 59, 64, 65, 76]. Thus, these schemes can be used as baseline methods to which dissipation can be added in a controlled manner [16, 17].

Flux differencing methods require two key ingredients: a discrete derivative operator satisfying the summation by parts (SBP) property and a two-point numerical flux. The SBP property guarantees a discrete equivalent of integration by parts and allows to transfer properties of the numerical flux to high-order methods, e.g., entropy conservation, preservation of the kinetic energy [9, 20, 32, 47] or important (quasi-) steady states including pressure equilibria [52] and lake-at-rest-type states for the shallow water equations [49, 75, 77].

SBP operators originate in finite difference methods [39]. Due to their attractive properties, they have recently received a lot of interest. Many common numerical schemes can be formulated in terms of SBP operators, including finite difference methods [39, 66], finite volume methods [42, 43], continuous Galerkin methods [1, 28, 29], discontinuous Galerkin (DG) methods [5, 6, 21], and flux reconstruction methods [30, 54]. Thus, they can be used to analyze a broad range of numerical methods in a unified fashion [50, 53]. For background information on SBP operators and further references, we recommend the review articles [14, 67].

This article presents our experience with efficient implementation techniques for flux differencing methods based on our open source projects Trixi.jl [56, 61] (written in Julia [4]) and FLUXO[1] (written in Fortran), e.g., [60]. These two codes are written in different languages and use different approaches in many aspects. In this respect, we posit that the techniques discussed in this article are general and can be applied successfully to other (open or closed source) DG codes such as FLEXI [38] and SSDC [44]. Additionally, many of the techniques can also be applied directly to other SBP schemes including finite difference methods.

We begin our discussion with a brief introduction to SBP methods in Section 2. There, we also describe broad techniques for the efficient implementation of flux differencing schemes applicable to general conservation laws. Next, we focus on specific aspects related to the compressible Euler equations in Section 3. These apply directly to the advective components of related models such as the compressible Navier-Stokes equations and magnetohydrodynamics. Having established a solid basis of efficient implementation techniques, we present some baseline performance benchmarks in Section 4. In Section 5, we compare flux differencing to overintegration, another common technique to increase the robustness of DG methods. Next, we discuss extensions of the efficient implementation for entropy-based flux differencing DG methods to general nodal distributions in Section 6. Thereafter, we discuss more invasive optimizations specifically tuned for the compressible Euler equations and related models in Section 7. This discussion includes technical optimizations whose complexity depends on the programming language. In Section 8, we investigate SIMD (single instruction, multiple data) optimizations and present related performance improvements. Finally, we summarize and discuss our results in Section **??**.

[1] `https://gitlab.com/project-fluxo/fluxo`

The code and instructions to reproduce all numerical results shown in this article are available in our reproducibility repository [55]. We use explicit time integration methods from the Julia library OrdinaryDiffEq.jl [45] for Trixi.jl.

2 Summation by parts operators and flux differencing

Consider a hyperbolic conservation law

$$\partial_t u(t,x) + \sum_{j=1}^{d} \partial_j f^j(u(t,x)) = 0, \quad t \in (0,T), x \in \Omega, \tag{2.1}$$

where the conserved variables are $u\colon [0,T] \times \Omega \to \Upsilon \subset \mathbb{R}^c$. The conservation law (2.1) must be supplemented by initial and boundary conditions. Since our techniques for an efficient implementation will concentrate on volume terms and internal interfaces, we do not present details of boundary conditions here. Moreover, we will suppress the dependency on time $t \in [0,T]$ and space coordinates $x \in \Omega \subset \mathbb{R}^d$ in the following.

Entropy estimates for conservation laws are based on the chain rule and symmetry properties of the differential operator with respect to the L^2 scalar product. Thus, these two ingredients are mimicked discretely by two-point numerical fluxes and appropriate SBP derivative operators. We consider the method of lines and first discretize (2.1) in space. The spatial semidiscretization uses a division of the domain Ω into non-overlapping elements Ω_l. The numerical solution is represented by a vector in a finite dimensional space on each element. For example, DG methods typically use polynomial spaces [27, 35]. On each element, SBP operators are applied.

Definition 2.1. A p-th order accurate *derivative matrix* D_j satisfies $\forall k \in \{0,\dots,p\}\colon D_j \boldsymbol{x}_j^k = k\boldsymbol{x}_j^{k-1}$, with the convention $\boldsymbol{x}^0 = \mathbf{1}$. We say D_j is consistent if $p \geq 0$. ◁

Definition 2.2. A (first-derivative) *SBP operator* on a d-dimensional element Ω_l consists of consistent first-derivative matrices D_j, $j \in \{1,\dots,d\}$, a symmetric and positive-definite matrix M approximating the scalar product on $L^2(\Omega_l)$, a restriction operator R approximating the restriction of functions on the volume Ω_l to the boundary $\partial\Omega_l$, a symmetric and positive-definite matrix B approximating the scalar product on $L^2(\partial\Omega_l)$, and multiplication operators N_j representing the multiplication of functions on $\partial\Omega_l$ by the j-th component of the outer unit normal n such that

$$MD_j + D_j^T M = R^T B N_j R. \tag{2.2}$$

We refer to M as a mass matrix or norm matrix[2]. ◁

Example 2.3. We mainly focus on tensor product elements using nodal Legendre-Gauss-Lobatto (LGL) bases since they are SBP operators with diagonal mass matrix M including the boundary nodes [21]. In particular, the boundary operators $R^T B N_j R$ are diagonal. The resulting method is often called discontinuous Galerkin spectral element method (LGL-DGSEM). ◁

Interface coupling between elements is usually performed via numerical fluxes in DG methods. We write normal vectors as $n \in \mathbb{S}^{d-1}$, where $\mathbb{S}^{d-1}$ is the unit sphere in $\mathbb{R}^d$.

Definition 2.4. A (Cartesian) *numerical flux* in the j-th coordinate direction is a Lipschitz continuous mapping $f^{\text{num,j}}\colon \Upsilon^2 \to \mathbb{R}^c$ satisfying $\forall u \in \Upsilon\colon f^{\text{num,j}}(u,u) = f^j(u)$. It is *symmetric* if $\forall u_1, u_2 \in \Upsilon\colon f^{\text{num,j}}(u_1,u_2) = f_j^{\text{num}}(u_2,u_1)$. A directional numerical flux is a Lipschitz continuous mapping $f^{\text{num}}\colon \Upsilon^2 \times \mathbb{S}^{d-1} \to \mathbb{R}^c$ satisfying $\forall u \in \Upsilon, n \in \mathbb{S}^{d-1}\colon f^{\text{num}}(u,u,n) = \sum_{j=1}^d n_j f^j(u)$ and $\forall u_{\text{in}}, u_{\text{out}} \in \Upsilon, n \in \mathbb{S}^{d-1}\colon f^{\text{num}}(u_{\text{in}}, u_{\text{out}}, n) = -f^{\text{num}}(u_{\text{out}}, u_{\text{in}}, -n)$. It is *symmetric* if $\forall u_1, u_2 \in \Upsilon, n \in \mathbb{S}^{d-1}\colon f^{\text{num}}(u_1,u_2,n) = f^{\text{num}}(u_2,u_1,n)$. ◁

[2] The term "mass matrix" is common for finite element methods. In the finite difference SBP community, the name "norm matrix" is more common.

We assume that the conservation law (2.1) is equipped with an entropy function U and corresponding entropy fluxes F^j satisfying $\partial_u U \cdot \partial_u f^j = \partial_u F^j$. We denote the *entropy variables* as $w = \partial_u U$ and the *flux potentials* as $\psi^j = w \cdot f^j - F^j$. Then, EC/ED fluxes are given as follows [69].

Definition 2.5. A Cartesian numerical flux is EC if $\forall u_-, u_+ \in \Upsilon\colon (w(u_+) - w(u_-)) \cdot f^{\mathrm{num,j}}(u_-, u_+) - (\psi^j(u_+) - \psi^j(u_-)) = 0$ and ED if $\leq$ holds instead of equality. A directional numerical flux is EC if $\forall u_{\mathrm{in}}, u_{\mathrm{out}} \in \Upsilon, n \in \mathbb{S}^{d-1}\colon (w(u_{\mathrm{out}}) - w(u_{\mathrm{in}})) \cdot f^{\mathrm{num}}(u_{\mathrm{in}}, u_{\mathrm{out}}, n) - \sum_{j=1}^d n_j(\psi^j(u_{\mathrm{out}}) - \psi^j(u_{\mathrm{in}})) = 0$ and ED if $\leq$ holds instead of equality. ◁

Collecting numerical fluxes at the interface between a given element and its neighbors in $\boldsymbol{f}^{\mathrm{num}}$, a typical strong form DG formulation on one element can be written as

$$\partial_t \boldsymbol{u} + \sum_{j=1}^d D_j \boldsymbol{f}^j + M^{-1} R^T B \left(\boldsymbol{f}^{\mathrm{num}} - \sum_{j=1}^d N_j R \boldsymbol{f}^j \right) = \boldsymbol{0}. \tag{2.3}$$

This mimicks, for all polynomials v of degree p, the variational form of (2.1)

$$\int_{\Omega_l} v \cdot \partial_t u + \int_{\Omega_l} v \cdot \partial_j f^j + \int_{\partial\Omega_l} \left(f^{\mathrm{num}} - \sum_{j=1}^d n_j f^j \right) = 0. \tag{2.4}$$

Flux differencing methods replace the volume term $\sum_{j=1}^d D_j \boldsymbol{f}^j$ by another volume term $\mathbf{VOL}$ using so-called volume fluxes f^{vol} [24], which are *symmetric* numerical fluxes, resulting in

$$\partial_t \boldsymbol{u} + \mathbf{VOL} + M^{-1} R^T B \left(\boldsymbol{f}^{\mathrm{num}} - \sum_{j=1}^d N_j R \boldsymbol{f}^j \right) = \boldsymbol{0}, \quad \mathbf{VOL}_i = \sum_{j=1}^d \sum_k 2 (D_j)_{i,k} f^{\mathrm{vol,j}}(\boldsymbol{u}_i, \boldsymbol{u}_k). \tag{2.5}$$

Here, the sum $\sum_k$ is performed over all degrees of freedom on the given element. In the following, we assume that the mass matrix M and the boundary operators $R^T B N_j R$ are diagonal. Then, this flux differencing form can be rewritten in locally conservative form; it is EC/ED if the volume fluxes are EC and the surface fluxes are EC/ED [5, 24]. Moreover, it is of the same order of accuracy as the derivative operator for general symmetric volume fluxes [11, 46].

On Legendre-Gauss-Lobatto tensor product elements with polynomials of degree p, the volume terms of the flux differencing form (2.5) require asymptotically $(p+1)$-times more flux evaluations than the classical strong form (2.3), see Table 1. However, the asymptotic number of floating point operations without computing fluxes is the same for both volume terms (basically one matrix vector product along each line in a tensor product layout per spatial dimension).

Table 1: Computational complexity of strong form and flux differencing volume terms on Legendre-Gauss-Lobatto tensor product elements in d spatial dimensions using polynomials of degree p. Pointwise fluxes are used in the strong form, while two-point fluxes are used in the flux differencing formulation.

	Strong form (2.3)	Flux differencing (2.5)
Flux evaluations	$d(p+1)^d$	$O\left(d(p+1)^{d+1}\right)$
Floating point operations (without fluxes)	$O\left(d(p+1)^{d+1}\right)$	$O\left(d(p+1)^{d+1}\right)$

Having introduced the basic form of flux difference semidiscretizations (2.5), we next present techniques for their efficient implementation. Most of these implementation aspects are designed to reduce the total number of operations, based on the hypothesis that the evaluation of fluxes f^j and volume fluxes f^{vol} is expensive, which holds for the compressible Euler equations and related models.

2.1 Separation of volume and surface terms

Using the SBP property (2.2), the strong form (2.3) is equivalent to the classical weak form semidiscretization [37]

$$\partial_t \boldsymbol{u} - \sum_{j=1}^{d} M^{-1} D_j^T M \boldsymbol{f}^j + M^{-1} R^T B \boldsymbol{f}^{\text{num}} = \boldsymbol{0}. \tag{2.6}$$

The weak form (2.6) is slightly more computationally attractive as no additional pointwise flux evaluations are required for the surface terms. The same efficiency optimization can be achieved for flux differencing discretizations by rewriting the volume terms in (2.5) to use the *flux differencing operator*

$$\widetilde{D}_j = 2D_j - M^{-1} R^T B N_j R, \tag{2.7}$$

which is skew-symmetric with respect to M, see Section 2.2 below. Indeed, since we assume that the mass matrix and the boundary operators are diagonal, $M^{-1} R^T B N_j R$ is also diagonal. Thus,

$$\sum_k (\widetilde{D}_j)_{i,k} f^{\text{vol,j}}(\boldsymbol{u}_i, \boldsymbol{u}_k) = \sum_k 2(D_j)_{i,k} f^{\text{vol,j}}(\boldsymbol{u}_i, \boldsymbol{u}_k) - (M^{-1} R^T B N_j R)_{i,i} f^{\text{vol,j}}(\boldsymbol{u}_i, \boldsymbol{u}_i). \tag{2.8}$$

From the consistency of the volume flux $f^{\text{vol,j}}$, the last term in (2.8) simplifies to be

$$\left(M^{-1} R^T B N_j R \boldsymbol{f}^j \right)_i, \tag{2.9}$$

which is exactly the surface flux term in (2.5). Hence, an optimized version of (2.5) that uses the flux differencing operators $\widetilde{D}_j$ (2.7) takes the form

$$\partial_t \boldsymbol{u} + \mathbf{VOL} + M^{-1} R^T B \boldsymbol{f}^{\text{num}} = \boldsymbol{0}, \quad \mathbf{VOL}_i = \sum_{j=1}^{d} \sum_k (\widetilde{D}_j)_{i,k} f^{\text{vol,j}}(\boldsymbol{u}_i, \boldsymbol{u}_k). \tag{2.10}$$

2.2 Symmetry properties of numerical fluxes and SBP operators

The flux differencing matrix $\widetilde{D}_j$ (2.7) is skew-symmetric with respect to M. Indeed, the SBP property (2.2) yields

$$M\widetilde{D}_j = 2MD_j - R^T B N_j R = -2D_j^T M + R^T N_j^T B R = -(M\widetilde{D}_j)^T. \tag{2.11}$$

In particular, the diagonal entries $(\widetilde{D}_j)_{i,i}$ are zero. Thus, exploiting the symmetry of the volume fluxes, it suffices to compute two-point volume fluxes $f^{\text{vol,j}}(\boldsymbol{u}_i, \boldsymbol{u}_k)$ only for combinations with, say, $i < k$. This saves more than half of the total number of volume flux evaluations.

2.3 Sparsity structure of tensor product operators and curvilinear coordinates

Given one-dimensional SBP operators $D_{j,1D}$, two-dimensional SBP operators on the tensor product domain can be constructed as $D_1 = D_{1,1D} \otimes \mathrm{I}$ and $D_2 = \mathrm{I} \otimes D_{2,1D}$. Thus, they are naturally sparse. Exploiting this sparsity structure is a fundamental step in an efficient implementation. This is easily possible on a Cartesian mesh using the flux differencing form (2.5) directly with Cartesian volume fluxes. Here, we describe how to transfer the same efficiency to curvilinear meshes.

We restrict this section to two spatial dimensions to make the presentation simpler. Assume we are given a Cartesian reference quadrilateral with coordinates ξ^i and a mapped curved version with coordinates x^i. The discrete Jacobian in 2D can be calculated directly as

$$J = (\boldsymbol{Ja})_2^2 (\boldsymbol{Ja})_1^1 - (\boldsymbol{Ja})_2^1 (\boldsymbol{Ja})_1^2, \tag{2.12}$$

where

$$(\boldsymbol{Ja})_1^1 = \operatorname{diag}\left(D_{2,\xi}\boldsymbol{x^2}\right), \quad (\boldsymbol{Ja})_2^1 = -\operatorname{diag}\left(D_{2,\xi}\boldsymbol{x^1}\right), \quad (\boldsymbol{Ja})_1^2 = -\operatorname{diag}\left(D_{1,\xi}\boldsymbol{x^2}\right), \quad (\boldsymbol{Ja})_2^2 = \operatorname{diag}\left(D_{1,\xi}\boldsymbol{x^1}\right), \tag{2.13}$$

are the components of scaled contravariant basis vectors $(Ja)_j^n$ in 2D [35, Chapter 6], $\boldsymbol{x^i}$ is the vector containing the nodal values of the curvilinear coordinates of an element, and $D_{j,\xi}$ are tensor product SBP operators on the Cartesian reference coordinates ξ^j. Using these ingredients, SBP operators on the curved element can be constructed as [2]

$$D_{j,x} = \frac{1}{2}\boldsymbol{J}^{-1}\sum_{n=1}^{d}\left((\boldsymbol{Ja})_j^n D_{n,\xi} + D_{n,\xi}(\boldsymbol{Ja})_j^n\right), \qquad 1 \le j \le d. \tag{2.14}$$

The curvilinear SBP operators (2.14) in the volume terms of the flux differencing form (2.5) are linear combinations of the underlying Cartesian SBP operators. Specifically, the entry corresponding to nodes i and k is

$$(D_j)_{i,k} = \frac{1}{2J_i}\sum_{n=1}^{d}\left(\sum_l \left((\boldsymbol{Ja})_j^n\right)_{i,l}\left(D_{n,\xi}\right)_{l,k} + \sum_l \left(D_{n,\xi}\right)_{i,l}\left((\boldsymbol{Ja})_j^n\right)_{l,k}\right), \tag{2.15}$$

where we used that $\boldsymbol{J}$ is diagonal. Since $(\boldsymbol{Ja})_j^n$ is diagonal, too,

$$(D_j)_{i,k} = \frac{1}{2J_i}\sum_{n=1}^{d}\left(\left((\boldsymbol{Ja})_j^n\right)_{i,i}\left(D_{n,\xi}\right)_{i,k} + \left(D_{n,\xi}\right)_{i,k}\left((\boldsymbol{Ja})_j^n\right)_{k,k}\right). \tag{2.16}$$

This can be abbreviated as

$$(D_j)_{i,k} = \frac{1}{J_i}\sum_{n=1}^{d}\alpha_{i,k}^{j,n}(D_{n,\xi})_{i,k}, \qquad 1 \le j \le d, \tag{2.17}$$

where $\alpha_{i,k}^{j,n} = \frac{1}{2}\left(((Ja)_j^n)_i + ((Ja)_j^n)_k\right)$, which is the arithmetic average of the scaled contravariant basis vectors at nodes i and k. Thus, the volume terms are recast to be

$$\sum_{j=1}^{d}\sum_k 2(D_j)_{i,k} f^{\mathrm{vol,j}}(\boldsymbol{u}_i, \boldsymbol{u}_k) = \frac{1}{J_i}\sum_{j=1}^{d}\sum_k 2\left(\sum_{n=1}^{d}\alpha_{i,k}^{j,n}(D_{n,\xi})_{i,k}\right) f^{\mathrm{vol,j}}(\boldsymbol{u}_i, \boldsymbol{u}_k). \tag{2.18}$$

If implemented in this form, we need only compute Cartesian volume fluxes instead of directional ones, which is usually slightly more efficient. However, the form given in (2.18) loses the sparsity of the individual tensor product operators. Thus, it is advantageous to rearrange terms as

$$\sum_{j=1}^{d}\sum_k 2\left(\sum_{n=1}^{d}\alpha_{i,k}^{j,n}(D_{n,\xi})_{i,k}\right) f^{\mathrm{vol,j}}(\boldsymbol{u}_i, \boldsymbol{u}_k) = \sum_{n=1}^{d}\sum_k 2(D_{n,\xi})_{i,k}\left(\sum_{j=1}^{d}\alpha_{i,k}^{j,n} f^{\mathrm{vol,j}}(\boldsymbol{u}_i, \boldsymbol{u}_k)\right). \tag{2.19}$$

The sum over the Cartesian volume fluxes is effectively a directional volume flux $f^{\mathrm{vol}}(\boldsymbol{u}_i, \boldsymbol{u}_k, \alpha_{i,k}^{\cdot,n})$ in the direction of the arithmetic average of the contravariant basis vectors at nodes i and k.

In this form, the sparsity structure of the underlying Cartesian tensor product SBP operators is completely retained. Since the evaluation of volume fluxes is usually relatively expensive and the evaluation of directional volume fluxes is only marginally more expensive than the computation of their Cartesian equivalents, it is advantageous to use the latter form (2.19). Of course, the usual care must be taken to ensure free-stream preservation etc. in multiple space dimensions when computing the contravariant basis vectors [36, 71]. Moreover, the techniques discussed in the previous Sections 2.1 and 2.2 can still be applied.

2.4 Discussion

The impact of the optimizations discussed in this section depend on the complexity of the numerical fluxes. For relatively expensive volume fluxes, such as EC fluxes for the compressible Euler equations, taking advantage of the symmetry (Section 2.2) and sparsity (Section 2.3) properties are most important. These optimizations should always be applied before focusing on problem-specific optimizations like the ones discussed in the following section. On top of these optimizations, common efficient implementation techniques for discontinuous Galerkin methods still apply. For example, it is usually efficient to compute the numerical fluxes at surfaces once per surface (instead of once per element) and use them to update the right-hand side on CPUs.

3 Compressible Euler equations

To discuss further efficient implementation strategies we consider the compressible Euler equations

$$\partial_t \begin{pmatrix} \varrho \\ \varrho v_i \\ \varrho e \end{pmatrix} + \sum_{j=1}^{d} \partial_j \begin{pmatrix} \varrho v_j \\ \varrho v_j v_i + p\delta_{ij} \\ (\varrho e + p) v_j \end{pmatrix} = 0, \qquad 1 \le i \le d, \tag{3.1}$$

as an example system of conservation laws. Here, ϱ is the fluid density, v the velocity, e the specific total energy, and p the pressure. We assume a perfect gas law with ratio of specific heats γ, i.e.,

$$p = (\gamma - 1)\left(\varrho e - \frac{1}{2}\varrho |v|^2\right). \tag{3.2}$$

3.1 Logarithmic mean

Since the seminal work of Ismail and Roe [31] on affordable EC fluxes for the compressible Euler equations, the logarithmic mean

$$\{\!\{a\}\!\}_{\log} = [\![a]\!] / [\![\log(a)]\!] \tag{3.3}$$

has played a crucial role. Indeed, it is even necessary if some desirable additional properties are to be satisfied [52]. Here, we have used the common jump notation

$$[\![a]\!] = a_+ - a_-. \tag{3.4}$$

Since a naive implementation of the logarithmic mean (3.3) is subject to floating point accuracy issues, special care must be taken when $a_+ \approx a_-$. Here, we present an efficient version based on Algorithm 1 presented in [31].

Since divisions are more expensive (in terms of latency and inverse throughput) than multiplications on modern (CPU) hardware [18], this algorithm can be improved by re-writing divisions in terms of cheaper multiplications, even if more additions are required on top; these can be combined into fused multiply-add (FMA) instructions. The resulting optimized implementation of the logarithmic mean is given in Algorithm 2.

Another variant of Algorithm 2 replaces the division in the "if" branch by a multiplication using a polynomial approximation of $1/\log(\cdot)$. For example, the term

$$(a_- + a_+) / \big(2 + u \cdot (2/3 + u \cdot (2/5 + u \cdot 2/7))\big) \tag{3.5}$$

can be replaced by

$$(a_- + a_+) \cdot \big(1/2 + u \cdot ((-1/6) + u \cdot ((-2/45) + u \cdot (-22/945)))\big) \tag{3.6}$$

without changing other parts of Algorithm 2. Our benchmarks indicate that this does not result in significant performance differences on the hardware currently available to us.

Algorithm 1 Computation of the logarithmic mean as described by Ismail and Roe [31].

Require: Input $a_-, a_+ > 0$
Ensure: Stable approximation of the logarithmic mean $\{\!\{a\}\!\}_{\log}$
  $\xi = a_-/a_+$
  $f = (\xi - 1)/(\xi + 1)$
  $u = f \cdot f$
  $\varepsilon = 10^{-4}$ ▷ for 64 bit floating point numbers
  if $u < \varepsilon$ **then**
    $F = 1 + u/3 + u \cdot u/5 + u \cdot u \cdot u/7$
  else
    $F = (\log(\xi)/2)/f$
  end if
  return $(a_- + a_+)/(2F)$

Algorithm 2 Optimized computation of the logarithmic mean.

Require: Input $a_-, a_+ > 0$
Ensure: Stable approximation of the logarithmic mean $\{\!\{a\}\!\}_{\log}$
  $u = (a_- \cdot (a_- - 2a_+) + a_+ \cdot a_+)/(a_- \cdot (a_- + 2a_+) + a_+ \cdot a_+)$ ▷ equivalent to f^2, $f = \frac{\xi-1}{\xi+1}$, $\xi = \frac{a_-}{a_+}$
  $\varepsilon = 10^{-4}$
  if $u < \varepsilon$ **then**
    return $(a_- + a_+)/\big(2 + u \cdot (2/3 + u \cdot (2/5 + u \cdot 2/7))\big)$ ▷ use Horner's rule
  else
    return $(a_+ - a_-)/\log(a_+/a_-)$
  end if

Some EC numerical fluxes for the compressible Euler equations presented in the following sections contain also the inverse logarithmic mean, i.e., factors of the form $1/\{\!\{a\}\!\}_{\log}$. Since the computation of the logarithmic mean in Algorithm 2 already contains a division in the last step, it is advantageous to avoid the additional division and also implement Algorithm 3 for computing the inverse logarithmic mean.

Algorithm 3 Optimized computation of the inverse logarithmic mean.

Require: Input $a_-, a_+ > 0$
Ensure: Stable approximation of the inverse logarithmic mean $1/\{\!\{a\}\!\}_{\log}$
  $u = (a_- \cdot (a_- - 2a_+) + a_+ \cdot a_+)/(a_- \cdot (a_- + 2a_+) + a_+ \cdot a_+)$ ▷ equivalent to f^2, $f = \frac{\xi-1}{\xi+1}$, $\xi = \frac{a_-}{a_+}$
  $\varepsilon = 10^{-4}$
  if $u < \varepsilon$ **then**
    return $\big(2 + u \cdot (2/3 + u \cdot (2/5 + u \cdot 2/7))\big)/(a_- + a_+)$ ▷ use Horner's rule
  else
    return $\log(a_+/a_-)/(a_+ - a_-)$
  end if

3.2 Numerical fluxes for the compressible Euler equations

As described in Section 2.3, it is advantageous to use directional volume fluxes for flux differencing on non-Cartesian meshes. First, we present such a directional form of the non-EC flux of Shima

et al. [62], which is kinetic energy and pressure equilibrium preserving.

$$\begin{aligned} f_\varrho &= \{\!\{\varrho\}\!\}\{\!\{v \cdot n\}\!\}, \\ f_{\varrho v} &= f_\varrho \{\!\{v\}\!\} + \{\!\{p\}\!\} n, \\ f_{\varrho e} &= f_\varrho \frac{(\!(v\,;v)\!)}{2} + \{\!\{p\}\!\}\{\!\{v \cdot n\}\!\} \frac{1}{\gamma - 1} + (\!(p\,; v \cdot n)\!), \end{aligned} \tag{3.7}$$

where we introduced the arithmetic mean

$$\{\!\{a\}\!\} = \frac{a_+ + a_-}{2} \tag{3.8}$$

and the product mean

$$(\!(a\,;b)\!) = \frac{a_+ \cdot b_- + a_- \cdot b_+}{2} = 2\{\!\{a\}\!\} \cdot \{\!\{b\}\!\} - \{\!\{a \cdot b\}\!\}. \tag{3.9}$$

The version (3.7) already clarifies how common subexpressions can be used efficiently. In the spirit of Section 3.1, modern hardware will benefit from avoiding divisions by storing $1/(\gamma - 1)$ and turning the division by $\gamma - 1$ into a multiplication.

Next, we present a directional version of the EC flux of Ranocha [46–48], which is also kinetic energy and pressure equilibrium preserving.

$$\begin{aligned} f_\varrho &= \{\!\{\varrho\}\!\}_{\log}\{\!\{v \cdot n\}\!\}, \\ f_{\varrho v} &= f_\varrho \{\!\{v\}\!\} + \{\!\{p\}\!\} n, \\ f_{\varrho e} &= f_\varrho \frac{(\!(v\,;v)\!)}{2} + f_\varrho \frac{1}{\{\!\{\varrho/p\}\!\}_{\log}} \frac{1}{\gamma - 1} + (\!(p\,; v \cdot n)\!). \end{aligned} \tag{3.10}$$

The direct and inverse logarithmic means should use Algorithms 2 and 3, respectively. Moreover, the inverse logarithmic mean in (3.10) can also be rewritten as

$$\frac{1}{\{\!\{\varrho/p\}\!\}_{\log}} = \frac{\log\big((\varrho_+/p_+)/(\varrho_-/p_-)\big)}{\varrho_+/p_+ - \varrho_-/p_-} = p_+ p_- \frac{\log\big((\varrho_+ p_-)/(\varrho_- p_+)\big)}{\varrho_+ p_- - \varrho_- p_+} = p_+ p_- \frac{1}{\{\!\{\varrho_+ p_-, \varrho_- p_+\}\!\}_{\log}} \tag{3.11}$$

to further avoid divisions, resulting in a speed-up on modern hardware. Nevertheless, EC fluxes such as (3.10) involving the logarithmic mean are computationally more demanding than fluxes using only the arithmetic mean such as (3.7). Thus, we will sometimes refer to them as "cheap" (flux of Shima et al.) and "expensive" (flux of Ranocha) fluxes. The same optimizations described here can also be applied successfully to other entropy-conserving two-point fluxes, e.g., the ones of Ismail and Roe [31] or Chandrashekar [9].

4 Numerical experiments

Here, we present numerical experiments and benchmarks ranging from microbenchmarks of single numerical flux evaluations to full simulation runs. The code and instructions to reproduce all numerical results shown in this article are available in our reproducibility repository [55]. The benchmark results shown in this paper were obtained on a dual socket compute node with two 2.5 GHz Intel® Xeon® Gold 6248 20-core processors and 384 GiB RAM, except where noted otherwise. We only report serial performance using one core; neither shared memory parallelization (multiple threads, e.g., `OpenMP`) nor distributed memory parallelization (e.g., `MPI`) is used. The Trixi.jl code was executed using version v1.7.0 of Julia [4] with bound checking disabled and otherwise default options of the official binaries. FLUXO was compiled in `Release` mode with the Intel® Fortran Compiler 19.1.3, i.e., using the flags `-O3`, `-xHost`, `-shared-intel`, `-inline-max-size=1500`, `-no-inline-max-total-size`, and `-no-prec-div`, which generates code optimized for the current

processor architecture. Note that we are at an optimization level where the particular instruction set used can make a noticeable difference in the performance. For example, restricting the Fortran compiler to AVX2 reduces the performance index described below by up to 19 %. Since the preferred vector width for the LLVM version shipped with Julia v1.7 is 256 bits, all numerical experiments conducted with Trixi.jl only use the AVX2 instruction set.

4.1 Baseline performance results on Cartesian and curved meshes

Meshes and numerical algorithms for PDEs support different geometric features such as curved coordinates, nonconforming interfaces, and an unstructured connectivity. Sacrificing some of these features can increase the performance and reduce code complexity. Here, we compare different mesh types available in Trixi.jl and the standard mesh type of FLUXO to provide guidance as to whether practitioners might choose to drop certain geometric features for performance if the problem setup allows it. Specifically, we compare the serial performance on

- the `TreeMesh`, a Cartesian, h-nonconforming, tree-structured mesh of Trixi.jl,
- the `StructuredMesh`, a curved, conforming, structured mesh of Trixi.jl,
- the `P4estMesh`, a curved, h-nonconforming, unstructured mesh of Trixi.jl,
- the three-dimensional, curved, h-nonconforming, unstructured mesh of FLUXO.

Trixi.jl and FLUXO implement all efficient implementation techniques discussed in Sections 2 and 3. On top of that, for the compressible Euler equations FLUXO precomputes primitive variables, $1/\varrho$, and $|v|^2$ at all nodes before computing the volume flux terms. Such an invasive optimization improves the performance further, as discussed in Section 7.1. Furthermore, both Trixi.jl and FLUXO use explicit inlining of volume fluxes (and setting the polynomial degree at compile time for FLUXO), as discussed in Section 7.3. Since FLUXO is natively a 3D code, 2D results will only be shown for Trixi.jl.

For these performance benchmarks, we use the isentropic vortex setup, a widely used benchmark problem [63] with an analytical solution. We use slightly different parameters than the original setup to make sure that all possible paths in the logarithmic mean are followed. In particular, we strongly increase the strength of the vortex to $\varepsilon = 20$. Hence, the initial conditions read as

$$\begin{aligned} T &= T_0 - \frac{(\gamma-1)\varepsilon^2}{8\gamma\pi^2}\exp\left(1-r^2\right), \quad \varrho = \varrho_0 (T/T_0)^{1/(\gamma-1)}, \\ v &= v_0 + \frac{\varepsilon}{2\pi}\exp\left((1-r^2)/2\right)(-x_2, x_1, 0)^T, \end{aligned} \tag{4.1}$$

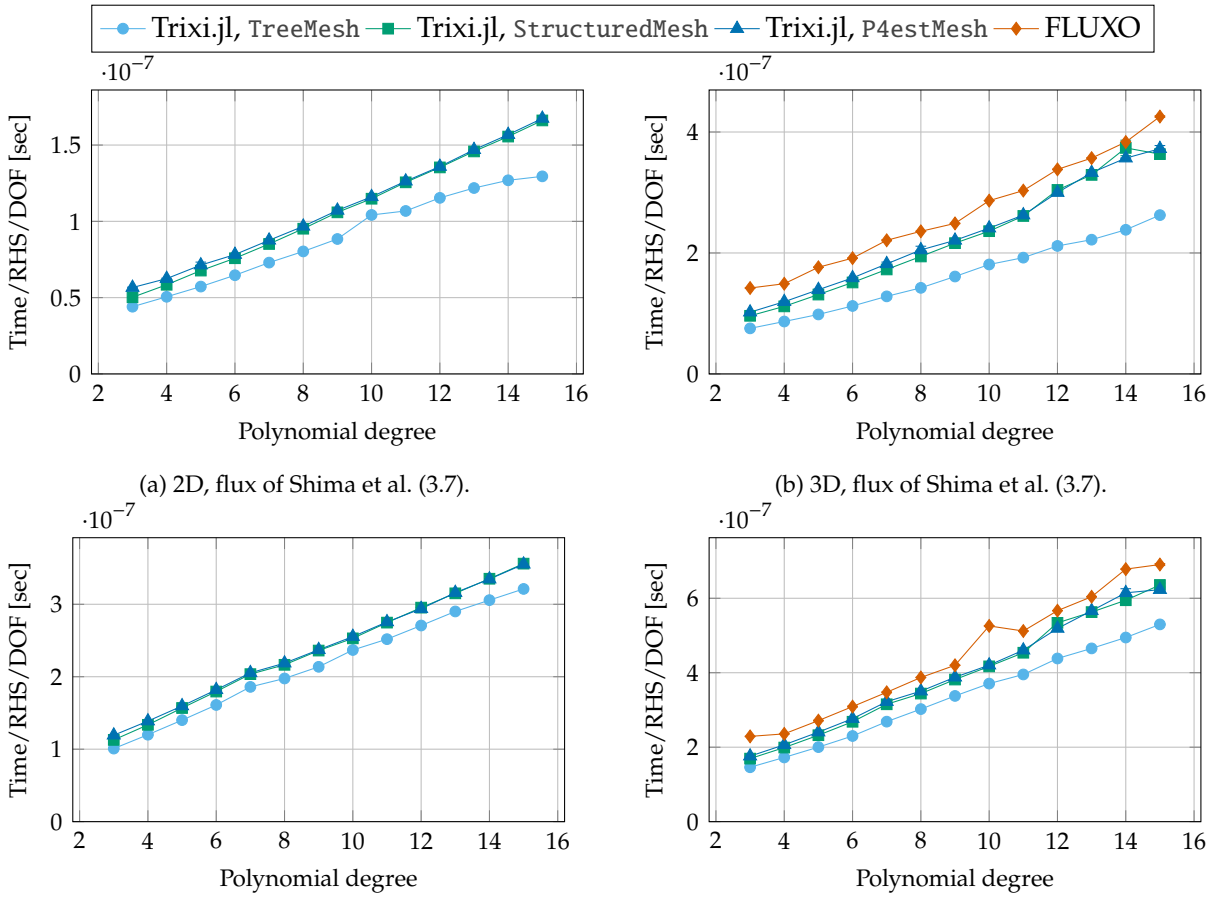
in 3D and its 2D analog, where r is the distance from the origin, $T = p/\varrho$ the temperature, $\varrho_0 = 1$ the background density, $v_0 = (1,1,0)^T$ the background velocity, $p_0 = 10$ the background pressure, $\gamma = 1.4$, and $T_0 = p_0/\varrho_0$ the background temperature. The domain $[-5,5]^d$ is equipped with periodic boundary conditions.

While all meshes allow more features, we restrict them to a simple Cartesian, conforming, structured setup here. We use eight elements per coordinate direction. The LGL-DGSEM discretizations use flux differencing with the same numerical flux in the volume and at the interfaces. The spatial semidiscretization is integrated in Trixi.jl for 50 time steps using the nine-stage, fourth-order FSAL Runge-Kutta method of [51] with error-based step size control using a tolerance of 10^{-8}. In FLUXO, we integrate in time for 90 time steps using the five-stage, fourth-order Runge-Kutta method of [33] with CFL-based step size control. This ensures that both codes use the same number of right-hand side (RHS) evaluations.

For baseline performance benchmarks, we use the performance index PID as the measure of runtime efficiency.

Definition 4.1. The *performance index* PID is defined as the runtime of one evaluation of the spatial semidiscretization divided by the total number of degrees of freedom #DOF. Here and in the following, each DG volume node counts as a single DOF, i.e., #DOF is given by $(p+1)^d$ times the number of elements. ◁

That is, a *lower* PID means a *faster* execution of the code. For all PID benchmarks, we average the wall clock time per RHS evaluation over a complete simulation and take the mean value (and standard deviation) of five runs.



(a) 2D, flux of Shima et al. (3.7).

(b) 3D, flux of Shima et al. (3.7).

(c) 2D, flux of Ranocha (3.10).

(d) 3D, flux of Ranocha (3.10).

Figure 1: Runtime per right-hand side evaluation and degree of freedom for different mesh types and fluxes, using the LGL-DGSEM discretization with flux differencing of the compressible Euler equations.

The baseline PID results are visualized in Figure 1. As expected based on the number of operations, the PID increases linearly with the polynomial degree p (see Table 1; the computational complexity $O\left(d(p+1)^{d+1}\right)$ needs to be divided by the number of DOFs per element, $(p+1)^d$, to get the scaling of the PID). The current handling of unstructured meshes in the `P4estMesh` of Trixi.jl compared to the `StructuredMesh` has no visible impact in 2D and only a minor impact of approx. 5 % in 3D. In contrast, the Cartesian `TreeMesh` can improve the performance significantly by up to 33 % for cheap volume fluxes and up to 20 % for expensive volume fluxes involving logarithmic mean values; the impact in 2D is reduced by approx. ten percentage points.

Moreover, these benchmarks show that the serial performance results of FLUXO (written in Fortran) and Trixi.jl (written in Julia) are similar. These comparisons are based on compiling FLUXO with all available performance tuning options, including explicit inlining of the volume fluxes and setting the polynomial degree (and node type) as constant at compile time. Thus, these numerical results demonstrate that Julia can be used for performance-critical scientific computing. Therefore, we will present most microbenchmarks using only Julia code in the following. This

also simplifies the presentation in the accompanying repository [55] since it is easier to work with a high-level language (Julia) and a library-based approach (Trixi.jl).

4.2 Different versions of numerical fluxes

We perform microbenchmarks comparing different versions of numerical fluxes for the compressible Euler equations. In particular, we benchmark the optimized directional fluxes presented in Section 3.2 and their corresponding Cartesian versions. Since the compressible Euler equations are rotationally invariant, a common approach to compute numerical fluxes in arbitrary directions is to rotate the states into the first coordinate direction, compute the standard Cartesian flux there, and rotate the resulting flux back, see [72, Section 16.7.3]. We also benchmark this approach including the on-the-fly computation of an appropriate rotation matrix (tangent vectors) from a given normal direction/vector, see Algorithm 4. Additionally, we benchmark an optimized alternative thereof with precomputed rotation matrix.

Algorithm 4 Computation of rotated numerical fluxes.

Require: Input states $u_{\text{in}}, u_{\text{out}}$, normal direction $\tilde{n}$, Cartesian numerical flux f^{num}
Ensure: Numerical flux $f^{\text{num}}(u_{\text{in}}, u_{\text{out}}; \tilde{n})$
  Normalize the normal direction $n = \tilde{n}/\|\tilde{n}\|$
  Compute $d-1$ orthonormal tangent vectors $t_i \perp n$ to get the rotation matrix $(n, t_i)^T$
  Rotate states $u_{\text{in}}, u_{\text{out}}$ to first coordinate direction, resulting in rotated states u_-, u_+
  Compute the Cartesian numerical flux $f^{\text{num},1}(u_-, u_+)$
  Rotate the flux $f^{\text{num},1}(u_-, u_+)$ back from first coordinate direction to obtain $f^{\text{num}}(u_{\text{in}}, u_{\text{out}}; n)$
  return $f^{\text{num}}(u_{\text{in}}, u_{\text{out}}; \tilde{n}) = f^{\text{num}}(u_{\text{in}}, u_{\text{out}}; n)\|\tilde{n}\|$

Table 2: Microbenchmarks of different versions of numerical fluxes for the compressible Euler equations.

	Cartesian	Directional	Rotated	
			(on the fly)	(precomputed)
Flux of Shima et al. (3.7), 2D	7.7 ± 0.1 ns	9.1 ± 0.2 ns	18.0 ± 0.3 ns	11.8 ± 0.2 ns
Flux of Shima et al. (3.7), 3D	9.5 ± 0.2 ns	12.1 ± 0.2 ns	53.4 ± 0.4 ns	18.7 ± 0.3 ns
Flux of Ranocha (3.10), 2D	33.2 ± 0.3 ns	34.8 ± 0.3 ns	42.1 ± 0.3 ns	37.4 ± 0.3 ns
Flux of Ranocha (3.10), 3D	35.4 ± 0.3 ns	39.7 ± 0.3 ns	82.1 ± 0.4 ns	46.0 ± 0.3 ns
LLF flux, 2D	18.3 ± 0.3 ns	19.9 ± 0.2 ns	28.3 ± 0.2 ns	21.2 ± 0.2 ns
LLF flux, 3D	19.7 ± 0.2 ns	21.5 ± 0.2 ns	68.2 ± 0.4 ns	27.7 ± 0.4 ns
HLL flux, 2D	21.1 ± 0.2 ns	21.4 ± 0.2 ns	31.6 ± 0.3 ns	25.4 ± 0.3 ns
HLL flux, 3D	23.3 ± 0.2 ns	22.9 ± 0.3 ns	79.0 ± 0.4 ns	34.6 ± 0.3 ns

The results of these microbenchmarks are shown in Table 2. The symmetric numerical fluxes used for the volume terms behave as follows. The Cartesian fluxes are usually the most efficient versions, followed directly by the directional approach without rotation. Depending on the computational complexity of the numerical flux, the Cartesian version is between 5 % and 20 % more efficient than the directional version. The rotated version with precomputed terms is significantly more expensive than the directional version, usually between 10 % (2D, expensive flux) and 2× (3D, cheap flux). Finally, the on-the-fly version without precomputed rotation matrix is significantly more expensive than the rotated version with precomputed terms, approximately between 10 % (2D, expensive flux) and 3× (3D, cheap flux).

For these comparisons we also benchmarked simple versions of the local Lax-Friedrichs/Rusanov flux (LLF) and the HLL flux [26]. Such numerical fluxes are usually used at interfaces to introduce additional dissipation. The runtimes of these fluxes is between the cheap flux of Shima et al. (3.7)

and the expensive EC flux of Ranocha (3.10). Thus, we continue to focus on the volume terms using flux differencing in this article.

The key messages of these microbenchmarks are as follows. First, the improved performance of the Cartesian mesh reported in Section 4.1 is not only caused by the different versions of numerical fluxes but also by the reduced amount of operations necessary to deal with general curvilinear coordinates. Second, a direct implementation of directional numerical fluxes is often preferable compared to an implementation that uses rotations; if the latter should nevertheless be used, the necessary rotation terms should be computed in advance, particularly for 3D implementations.

5 Comparison to overintegration

Another common strategy to increase the robustness of LGL-DGSEM discretizations is overintegration, i.e., interpolating the numerical solution to a higher polynomial degree, using standard weak form volume terms there, and projecting orthogonally on the given polynomial degree. A comparison of overintegration and flux differencing based DG methods for under-resolved turbulence is presented in [76].

Here, we perform microbenchmarks of the volume terms using flux differencing LGL-DGSEM and overintegration with different polynomial degrees. For the overintegration, we follow the procedure presented in, e.g., [22]. The interpolation and projection steps use sum factorization and efficient multiplication kernels using tools from LoopVectorization.jl[3], which is on par with (and sometimes faster than) optimized BLAS libraries such as Intel MKL for matrix multiplications at these sizes [12]. The numerical solution is initialized on a Cartesian `TreeMesh` with a single element for the isentropic vortex initial condition described in Section 4.1.

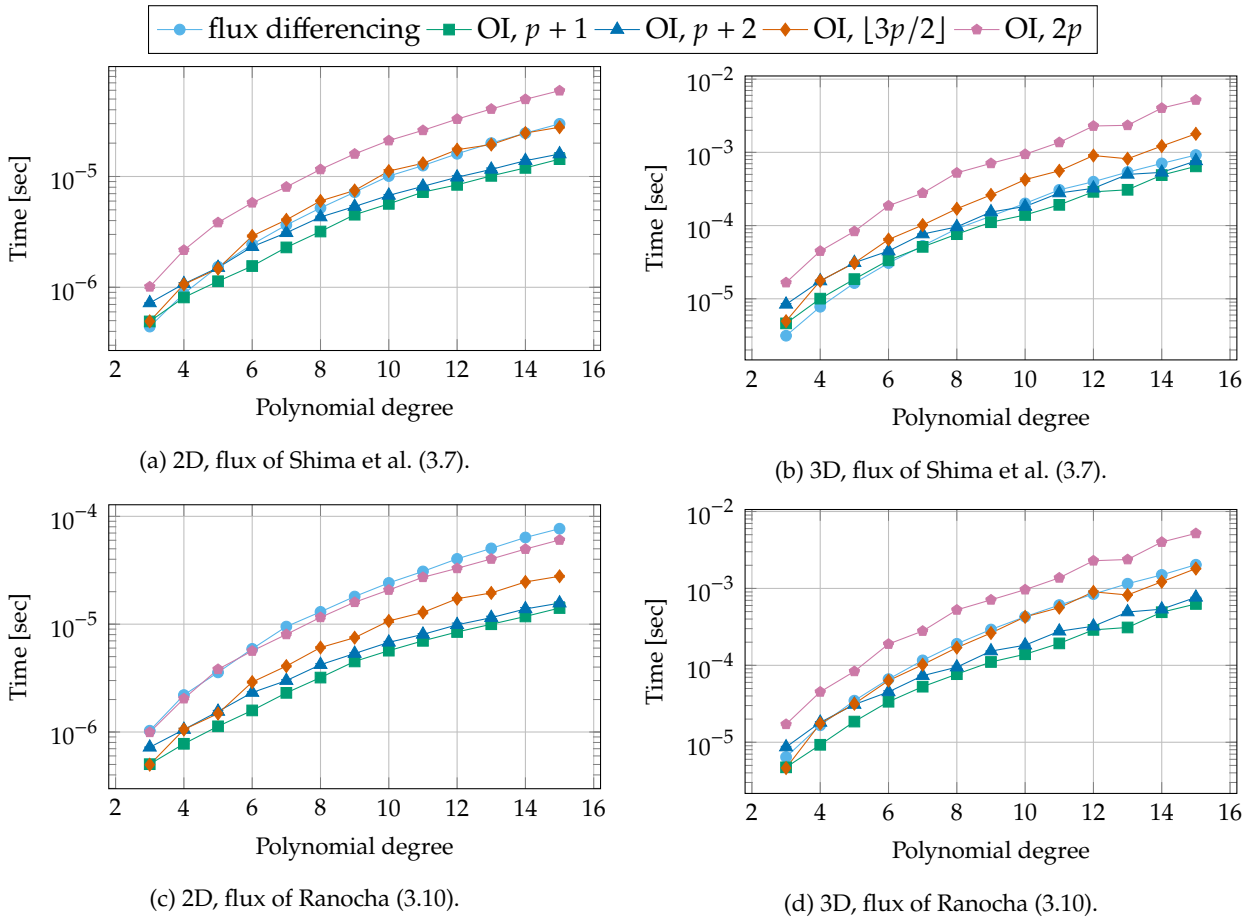
The results are visualized in Figure 2. The three-dimensional case is most relevant in practice. There, flux differencing with an inexpensive volume flux, such as the one of Shima et al. (3.7), is cheaper than any overintegration for small polynomial degrees $3 \leq p \leq 5$. For $5 < p \leq 7$, flux differencing is between overintegration with one or two additional nodes per coordinate direction. For an expensive volume flux involving logarithmic mean values, flux differencing is still between overintegration with one or two additional nodes per coordinate direction for $p \in \{3, 4\}$. For $p > 5$, flux differencing is between overintegration with internal polynomial degrees of $\lfloor 3p/2 \rfloor$ and $2p$. In 2D, flux differencing is relatively more expensive than in 3D. Nevertheless, cheap volume fluxes make it still faster than overintegration with $2p$.

The key message of these benchmarks is that flux differencing is competetive with overintegration, in particular in 3D, and it has stability guarantees that the overintegration strategy typically does not have. For the most practically relevant case for computational fluid dynamics (3D, polynomial degree ≤ 3), flux differencing can be faster than overintegration of the volume terms with a single additional degree of freedom per coordinate direction. Moreover, there exist several cases where overintegration fails to provide appropriate robustness while flux differencing schemes are stable [76]. Additionally, analogous to the comparison of Legendre-Gauss and Legendre-Gauss-Lobatto quadrature rules in discontinuous Galerkin spectral element methods, overintegration comes at the cost of increased stiffness of the semidiscretization, resulting roughly in a factor of two in time step restrictions of explicit time integration methods [23]. Moreover, many overintegration variants apply a similar procedure also to surface terms, making them more expensive than the standard surface terms used for flux differencing methods.

6 Gauss collocation methods and entropy projections

In addition to entropy stable schemes based on (2.5), it is possible to construct entropy stable schemes based on generalized SBP operators [13]. These include, for example, collocation schemes

[3] `https://github.com/JuliaSIMD/LoopVectorization.jl`



(a) 2D, flux of Shima et al. (3.7).

(b) 3D, flux of Shima et al. (3.7).

(c) 2D, flux of Ranocha (3.10).

(d) 3D, flux of Ranocha (3.10).

Figure 2: Microbenchmarks of overintegration (OI) vs. flux differencing volume terms of DG discretizations with polynomials of degree p for the the compressible Euler equations.

constructed on Legendre-Gauss nodes [8]. These schemes can be written in the form [7]

$$\partial_t \boldsymbol{u} + \widetilde{\mathbf{VOL}} + M^{-1} R^T B \boldsymbol{f}^{\text{num}} = \mathbf{0}. \tag{6.1}$$

The operators M, R, D_j are defined as in Section 2. However, one key difference between (2.5) and (6.1) is that it is no longer assumed that volume nodes include nodes on the boundary. Instead, the boundary restriction operator R now maps from interior nodes to boundary nodes, resulting in a fully dense matrix. As a result, the volume terms must be modified to retain both high order accuracy and entropy stability. Recall that u denotes the mapping from entropy variables to conservative variables and w denotes the mapping from conservative variables to entropy variables. Then, the volume terms $\widetilde{\mathbf{VOL}}$ are computed via

$$\begin{aligned} \widetilde{\mathbf{VOL}} &= M^{-1} \begin{bmatrix} \mathrm{I} \\ R \end{bmatrix}^T \boldsymbol{f}^{\text{hybrid}}, \qquad \boldsymbol{f}_i^{\text{hybrid}} = \sum_{j=1}^{d} \sum_{k} 2(Q_{h,j})_{i,k} f^{\text{vol,j}}(\tilde{\boldsymbol{u}}_i, \tilde{\boldsymbol{u}}_k), \\ Q_{h,j} &= \frac{1}{2} \begin{bmatrix} MD_j - (MD_j)^T & R^T B N_j \\ -B N_j R & 0 \end{bmatrix}, \qquad \tilde{\boldsymbol{u}} = \begin{bmatrix} \boldsymbol{u} \\ u\big(Rw(\boldsymbol{u})\big) \end{bmatrix}, \end{aligned} \tag{6.2}$$

where k sums over the combined set of both volume and surface quadrature points. Here, $\tilde{\boldsymbol{u}}$ is the "entropy projection", and $Q_{h,j}$ denotes the *hybridized* SBP operator with respect to the jth coordinate direction [6]. Entropy stable schemes constructed on the Legendre-Gauss nodes can be alternatively formulated in terms of "correction" terms on the surface, allowing an implementation

which keeps the volume term **VOL** from (2.5) untouched:

$$\begin{aligned} \partial_t \boldsymbol{u} + \mathbf{VOL} + M^{-1}\left(R^T B \boldsymbol{f}^{\text{num}} + \begin{bmatrix} \mathrm{I} \\ R \end{bmatrix}^T \boldsymbol{f}^{\text{corr}}\right) &= \mathbf{0}, \\ \boldsymbol{f}_i^{\text{corr}} = \sum_{j=1}^{d} \sum_{k} (B_{h,j})_{i,k} f^{\text{vol,j}}(\tilde{\boldsymbol{u}}_i, \tilde{\boldsymbol{u}}_k), \qquad B_{h,j} &= \begin{bmatrix} 0 & R^T B N_j \\ -B N_j R & 0 \end{bmatrix}, \end{aligned} \tag{6.3}$$

where k sums again over the combined set of both volume and surface quadrature points and $B_{h,j}$ denotes a *hybridized* boundary matrix.

The efficient implementation of (6.1) and (6.3) needs special care to avoid unnecessary flux evaluations and multiplications by zero. This can be achieved by using sparse matrix formats and the tensor product structure within both Legendre-Gauss and Legendre-Gauss-Lobatto bases on tensor product elements. We note that Trixi.jl uses (6.1) for the implementation, while FLUXO uses (6.3). In the results of this section, we only use the entropy-conservative scheme, using the flux of Ranocha (3.10) in the flux differencing volume terms and for the surface fluxes.

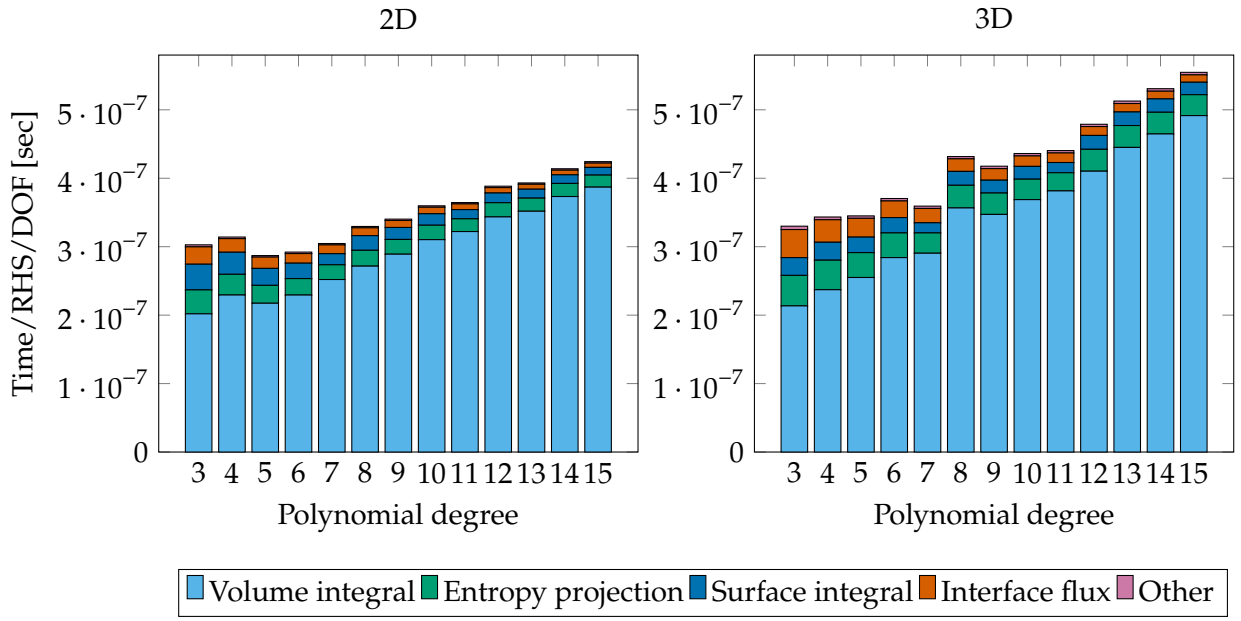
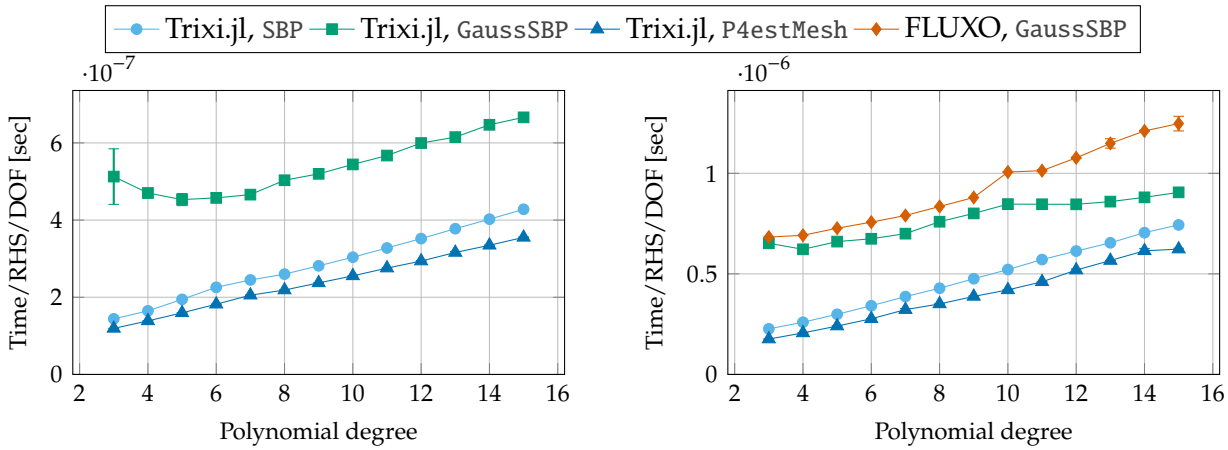


Figure 3: PID breakdowns for two and three dimensional benchmarks using a Legendre-Gauss collocation solver.

Figure 3 shows PID values (broken down by subroutines) for 2D and 3D Legendre-Gauss collocation solvers of degree $3, \ldots, 15$ in Trixi.jl. For all polynomial degrees studied here, we observe that the volume integral strongly dominates computational runtime. This step is dominated by flux evaluations, but also includes a "lifting" of face contributions to volume nodes when computing (6.2) or (6.3). The next most expensive step is the entropy projection, which makes up a smaller share of the overall runtime as p increases. For example, for $p > 10$ in both two and three dimensions, the cost of the entropy projection is close to the combined cost of all subroutines aside from flux differencing volume terms.

The relative costs (as percentage of PID) of flux differencing and the entropy projection depend strongly on the polynomial degree p; a higher polynomial degree increases the relative cost of flux differencing compared to the entropy projection. In 2D, these relative costs range from approx. 66 % (flux differencing) and 11 % (entropy projection) for $p = 3$ to approx. 91 % (flux differencing) and 4 % (entropy projection) for $p = 15$. In 3D, these relative costs range from approx. 65 % (flux differencing) and 14 % (entropy projection) for $p = 3$ to approx. 89 % (flux differencing) and 6 % (entropy projection) for $p = 15$.



(a) 2D.

(b) 3D.

Figure 4: Runtime per right-hand side evaluation and degree of freedom for different mesh/node types and entropy-conservative DG discretizations using the flux (3.10) for the compressible Euler equations.

Figure 4 compares PID results for several different implementations of entropy-conservative LGL-DGSEM (Trixi.jl, `SBP`, which uses affine mappings, and `P4estMesh`, which uses curvilinear mappings) and Legendre-Gauss collocation methods (Trixi.jl, `GaussSBP`, which uses affine mappings, and FLUXO, `GaussSBP`, which uses curvilinear mappings) in both two and three dimensions. We observe that the Legendre-Gauss collocation schemes are between $1.5\times$ and $5\times$ more expensive than LGL-DGSEM, with the gap in performance closing as the order of approximation increases in three dimensions. We note that these results imply that the estimate of the cost of Legendre-Gauss collocation in [8] was optimistic[4], as it did not take into account additional steps such as the entropy projection, interpolation of solution values to face nodes, and lifting of face contributions to volume nodes.

Finally, we note that the implementation of entropy-conservative Legendre-Gauss collocation schemes in Trixi.jl are slightly less optimized compared to the implementation of LGL-DGSEM schemes. First, for compatibility with analysis and visualization routines, Gauss schemes in Trixi.jl store the solution at Legendre-Gauss-Lobatto nodes and interpolate to Legendre-Gauss nodes prior to each right hand side evaluation. This interpolation can be performed in an efficient tensor product fashion, though this does still result in some overhead. The second difference is that Legendre-Gauss schemes compute surface fluxes locally on each element, as done in [27], which results in fluxes being computed twice per face. In contrast, the implementation of LGL-DGSEM in Trixi.jl computes surface fluxes only once per face then passes the computed fluxes to adjacent elements. Based on the PID results reported in Figure 3, however, we do not expect these implementational differences to drastically change the overall runtime of Legendre-Gauss collocation methods.

7 More invasive optimizations

The optimizations discussed hitherto are noninvasive in the sense that they can be applied to any flux differencing implementations. Sometimes, it can be beneficial to specialize the implementations even further for specific conservation laws — often at the cost of increased code complexity. We discuss precomputing primitive variables and certain logarithms needed in EC fluxes for

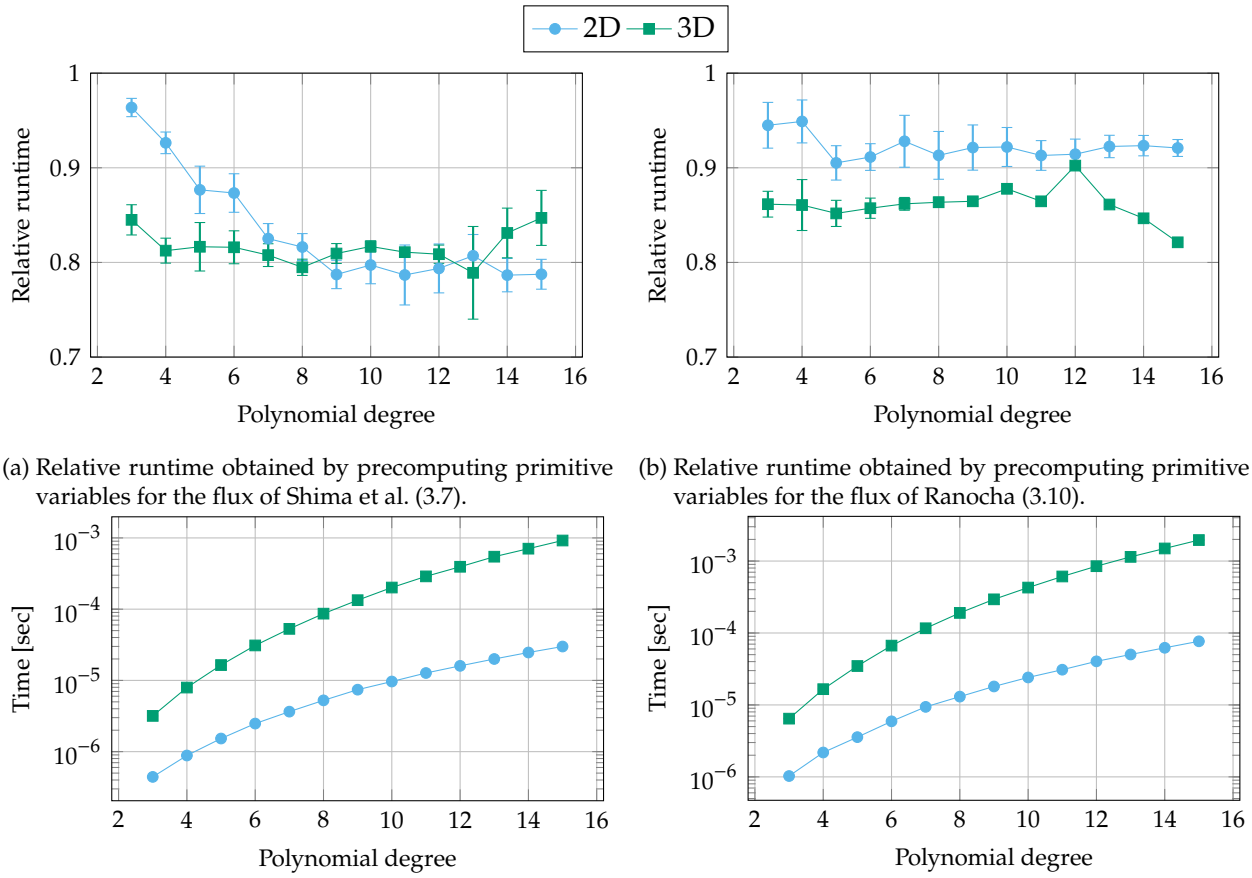
[4] In [8], it was shown that the number of flux evaluations for a degree p entropy-conservative Legendre-Gauss scheme is slightly less than the number of flux evaluations required for a degree $(p+1)$ entropy-conservative LGL-DGSEM method. Since the cost for entropy-conservative flux differencing schemes is typically dominated by flux evaluations, it was argued that a degree p entropy-conservative Legendre-Gauss scheme will be roughly the cost of a degree $(p+1)$ entropy-conservative LGL-DGSEM method.

the compressible Euler equations. Moreover, we present some technical optimization techniques such as inlining volume fluxes explicitly. All numerical experiments in this section are based on LGL-DGSEM implementations in Trixi.jl and FLUXO.

7.1 Precomputing primitive variables for the compressible Euler equations

Here, we benchmark the performance gains of volume terms by precomputing the primitive variables (ϱ, v, p) for the compressible Euler equations (3.1). All implementation techniques discussed so far can of course be used to improve the efficiency of flux differencing algorithms using the conservative variables $u = (\varrho, \varrho v, \varrho e)$ directly. However, several divisions and other arithmetic operations necessary to compute the primitive variables from the conservative variables can be saved by precomputing them. For this task, we use efficient specialized implementations based on LoopVectorization.jl.

The benchmark setup is the same as in Section 5, i.e., we benchmark the total runtime of the volume term computation for a single element on the Cartesian `TreeMesh` of Trixi.jl initialized with the isentropic vortex initial condition (4.1).



(a) Relative runtime obtained by precomputing primitive variables for the flux of Shima et al. (3.7).

(b) Relative runtime obtained by precomputing primitive variables for the flux of Ranocha (3.10).

(c) Absolute timings using the flux of Shima et al. (3.7) with conserved variables.

(d) Absolute timings using the flux of Ranocha (3.10) with conserved variables.

Figure 5: Microbenchmarks of precomputing the primitive variables compared to using the conserved variables (baseline) for flux differencing volume terms of LGL-DGSEM discretizations with polynomials of degree p for the compressible Euler equations.

The results of this comparison are shown in Figure 5. Clearly, precomputing the primitive variables improves the efficiency of the volume term computations significantly, approximately between 5 % and 25 % (on this computer system). Usually, 3D computations benefit more from this invasive optimization than 2D computations. Moreover, relatively cheap numerical fluxes such as the one of Shima et al. (3.7) benefit more than relatively expensive EC fluxes.

These runtime improvements come at the cost of additional memory requirements. Naively computing primitive variables at all volume nodes on a Cartesian mesh in 3D requires additional memory $5 \cdot \#\text{DOF}$. Since any time integration method will also require at least the same amount of temporary storage, this would at most increase the memory requirements by one half. However, a curved mesh requires storing the curvilinear coordinate information (contravariant basis vectors), which requires $9 \cdot \#\text{DOF}$ memory. Thus, the additional storage requirement is at most approx. one quarter. This can be further reduced by computing and storing the primitive variables only for a single element before computing the corresponding volume terms.

7.2 Precomputing logarithms for the compressible Euler equations

In addition to precomputing the primitive variables, one may also be interested in computing other auxiliary quantities used in the evaluation of numerical fluxes. For example, the evaluation of EC fluxes for the compressible Euler equations requires computing the logarithmic mean of density and pressure, which in turn requires computing the natural logarithm of density and pressure at two sets of solution states. Computing logarithms (and other special functions) is significantly more expensive than performing basic arithmetic operations. For example, on a 2019 Macbook Pro laptop (with a 2.3 GHz Intel® Core™ i9 processor), computing $\log(x) + \log(y)$ takes between 11.5 ± 3.8 ns, while computing a simpler arithmetic operation such as $2x + 3y$ takes 1.7 ± 0.4 ns.

For a degree p approximation on a single tensor product element in d dimensions, computing entropy-conservative two-point fluxes for the compressible Euler equations requires $O\left(d(p+1)^{d+1}\right)$ logarithm evaluations. Precomputing logarithms for density and pressure reduces this to $O\left((p+1)^d\right)$ evaluations, as logarithms are computed once for each solution node instead of once for each pair of nodes. We perform microbenchmarks for the entropy-conservative flux (3.10) similar to those in Section 7.1. Specifically, we precompute logarithms of density and pressure in addition to precomputing the primitive variables.

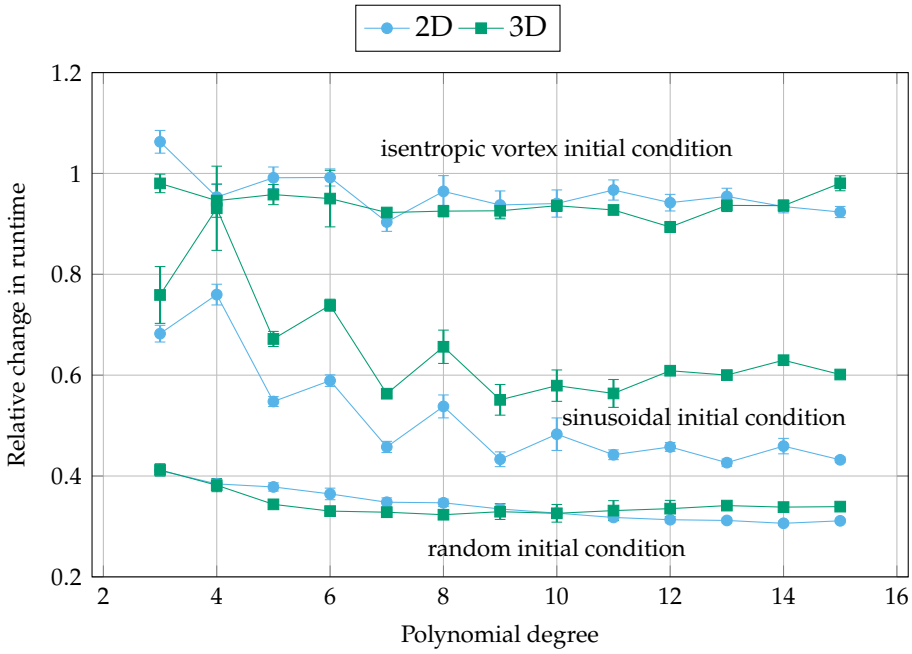


Figure 6: Relative runtime when precomputing both logarithms and primitive variables compared to precomputing only primitive variables (baseline) for flux differencing volume terms of LGL-DGSEM discretizations using the flux of Ranocha (3.10) with polynomials of degree p for the compressible Euler equations.

Figure 6 shows relative runtimes when precomputing both primitive variables and logarithms (compared with precomputing only the primitive variables). We tested three initial conditions:

the isentropic vortex (4.1), a random initial condition, and a sinusoidal initial condition

$$\begin{aligned} \varrho(x, y, 0) &= 2 + \sin(\pi x/5)\sin(\pi y/5), && \text{(in 2D)} \\ \varrho(x, y, z, 0) &= 2 + \sin(\pi x/5)\sin(\pi y/5)\sin(\pi z/5), && \text{(in 3D)} \\ p &= \varrho^{\gamma}. \end{aligned} \tag{7.1}$$

For the isentropic vortex, we observe relatively modest speedup of roughly 10 %. In contrast, we observe more significant speedups for the sinusoidal initial condition (between 25 % and 55 %) and for the random initial condition (between 60 % and 75 %). This is because Algorithms 2 and 3 for computing logarithmic means do not evaluate logarithms between two solution states if the density or pressure are close to each other (instead, a high order Taylor approximation is evaluated). Thus, precomputing logarithms does not yield any speedup (and in fact is slightly slower) when the solution is near-constant.

For the isentropic vortex initial condition, the solution is compactly supported and thus constant in a large percentage of the domain (especially in 3D). As a result, we observe very modest gains in the overall runtime. For the sinusoidal and random initial conditions, the solution varies globally over the entire domain, and we observe more significant speedup when precomputing logarithms.

7.3 Defining options at compile time

The techniques discussed so far are at the level of the mathematical description of the algorithms. While we will mostly stay at this high level in this article, we would like to point out that an efficient implementation will also require appropriate programming techniques. The effort required to achieve these goals depends on the specific programming language. Here, we briefly investigate the use of inlining of the volume fluxes and setting the polynomial degree at compile time in FLUXO.

In traditional scientific computing languages such as Fortran, C, and C++, the code is compiled prior to execution. To avoid users having to modify and/or compile the source code multiple times to change simulation options (such as the polynomial degree or the volume flux), a common practice is to read all simulation parameters from a text file and assign them at runtime.

The definition of functions used many times throughout the simulation at runtime implies the repeated evaluation of switch statements during the computation or the use of procedure pointers or type polymorphism. The repeated evaluation of switch statements might increase code complexity and deteriorate performance. Moreover, even though procedure pointers and polymorphic types can be used to simplify code, many compilers and their branch predictors fail at inlining functions that use them, which also affects performance.

FLUXO allows the user to specify the polynomial degree and the volume flux at runtime for flexibility, but also provides the possibility to define these and other options at compile time to improve performance of long runs. Of course, the coexistence of compile time and runtime specifications increases code complexity.

Figure 7 shows computed PID values obtained with LGL-DGSEM of FLUXO for the isentropic vortex setup introduced in Section 4.1. Setting the volume flux at compile time enables more compiler optimizations such as inlining and thus increases the performance by approx. 5 %. Setting also the polynomial degree at compile time options allows more optimizations, resulting in a speedup of up to 15 %.

Julia [4] uses a "just ahead of time" (or "just in time") compiler approach. In particular, user defined functions can be inlined into library code without additional programming effort. Currently, the Julia compiler uses a heuristic to determine whether a function should be inlined. Programmers can hint the compiler to inline a function by prepending the function definition by the macro `@inline`.

Figure 8 shows the effect of inlining volume fluxes in Trixi.jl. As expected, the relative improvement of this optimization is better for cheap volume fluxes such as the one of Shima et al. (3.7).

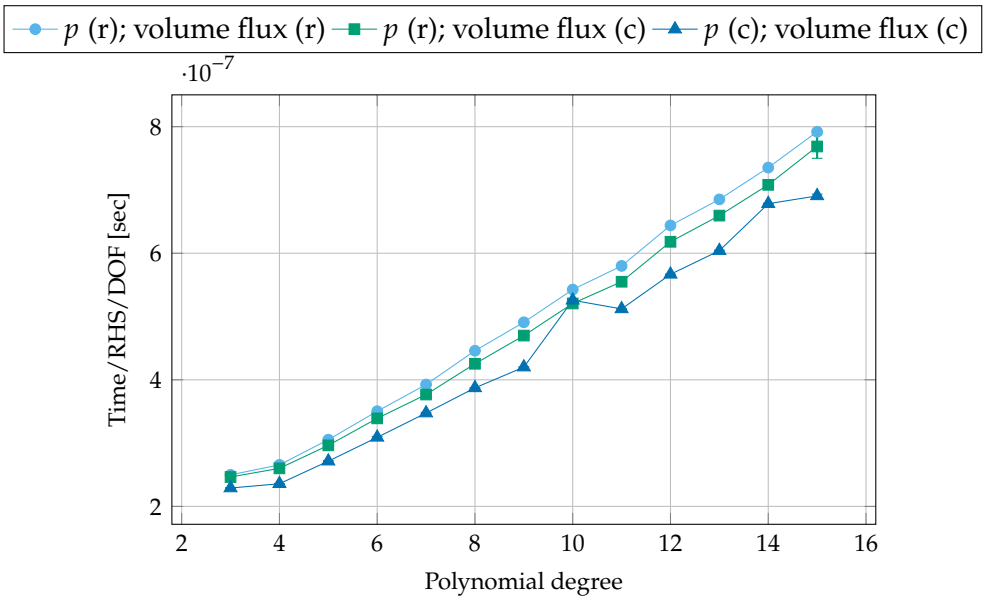
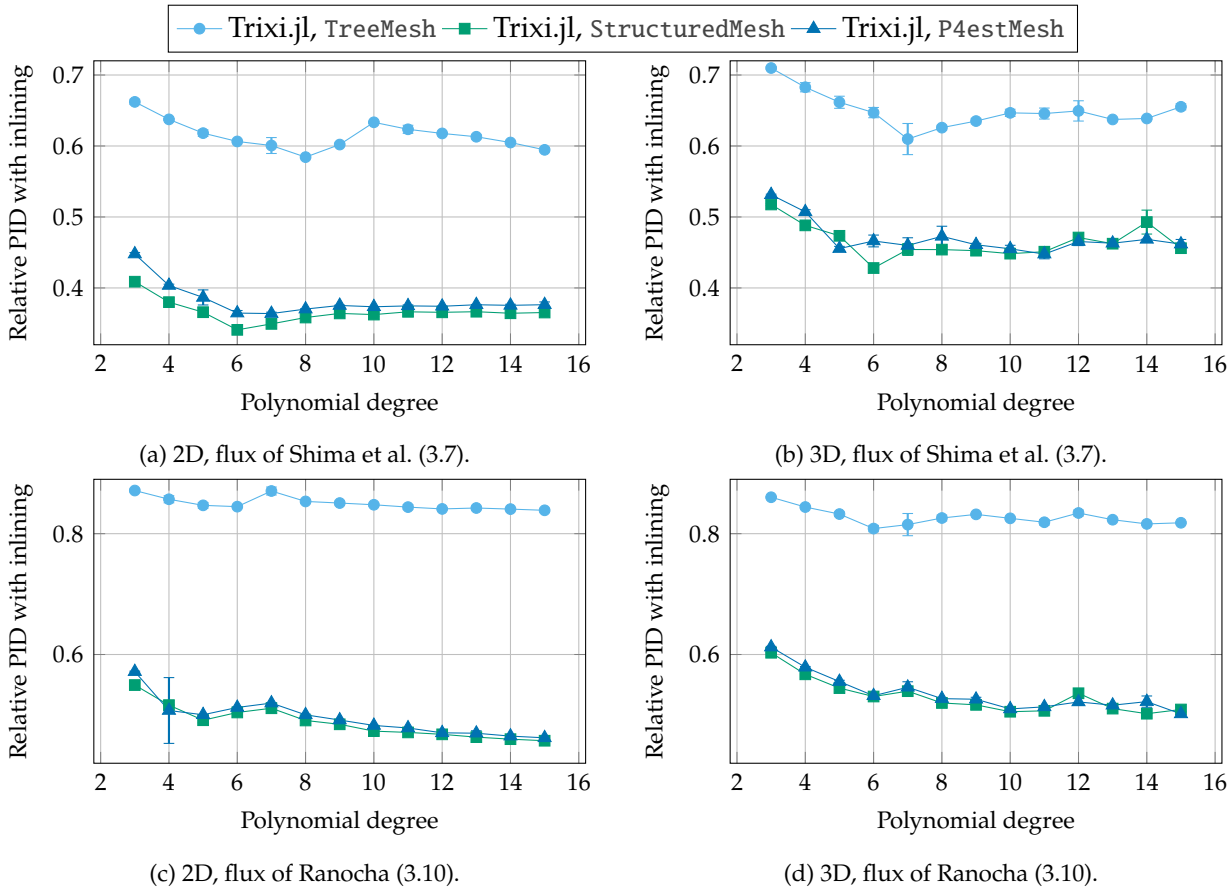


Figure 7: Influence of setting the volume flux and the polynomial degree p at compile time (c) or at runtime (r) on the PID of FLUXO for entropy-conservative LGL-DGSEM discretizations using the flux of Ranocha (3.10) for the compressible Euler equations.



(a) 2D, flux of Shima et al. (3.7).

(b) 3D, flux of Shima et al. (3.7).

(c) 2D, flux of Ranocha (3.10).

(d) 3D, flux of Ranocha (3.10).

Figure 8: Relative performance obtained by inlining the volume fluxes for flux differencing volume terms of LGL-DGSEM discretizations with polynomials of degree p for the compressible Euler equations. The absolute PID with inlining is visualized in Figure 1.

Additionally, inlining is more important on the curved meshes of Trixi.jl. There, it can reduce the PID by up to approx. 2 × (compared to approx. 25 % on the Cartesian `TreeMesh`).

8 Explicit SIMD optimizations

The operational intensity of flux differencing volume terms is relatively high. Thus, we expect to be in the compute bound regime in a standard roofline model [74] of standard CPUs. This is the reason why we described several optimizations that reduce the amount of work and increase runtime performance. To further increase the (serial) performance, instruction level parallelization is required. To make optimal use of modern CPUs, utilizing SIMD (single instruction, multiple data) instructions is key. However, current compiler and code generation techniques are often not advanced enough to handle all expressions efficiently. Thus, some manual intervention is needed to write SIMD-friendly code. Here, we focus on implementations based on LoopVectorization.jl in Trixi.jl. The same kind of optimizations are also effective for auto-vectorization by decent Fortran compilers in FLUXO.

Trixi.jl is written as a research code accessible to students and newcomers. In particular, some design choices are based on the goal to make setting up new physical models easy. Thus, physics including (numerical) fluxes is handled pointwise by small (inlined) functions and the global solution uses an array of structures (AoS) memory layout. To enable efficient SIMD optimizations at the element level, we permute array dimensions effectively to an array of structures of arrays (AoSoA), i.e., we use a temporary structure of arrays for a single element. Loops in a tensor product ansatz are structured to keep the triangular part at the outermost loop (making use of the skew-symmetry of the flux differencing operator (2.7)) and plain loops over all nodes in the inner part for SIMD vectorization. Moreover, the memory is rearranged to ensure the first dimension is one of the dimensions to which SIMD instructions can be applied (since Julia uses a Fortran-style column-major memory layout by default). Such a rearrangement of memory is also crucial for Fortran compilers to optimize similar parts in FLUXO [57].

Moreover, we precompute primitive variables before computing the flux differencing volume terms (cf. Section 7.1). For the EC flux requiring logarithmic means, we also precompute logarithms of the density ϱ and the pressure p. In contrast to the results reported in Section 7.2, this step becomes more important even for the isentropic vortex initial condition. Scalar logarithm implementations are often based on algorithms including table lookups and conditional branches, e.g., [70]. In contrast, logarithm implementations optimized for SIMD instructions are usually more demanding and cannot use fast paths. We observed up to ca. $2\times$ speed-up by precomputing logarithms on an Intel® Core™ i7-8700K (CPU from 2017 with AVX2) for the isentropic vortex initial condition.

All of these specializations are available for the fluxes of Shima et al. (3.7) and Ranocha (3.10) on 2D and 3D meshes in recent versions of Trixi.jl. The benchmarks reported thus far do not use these SIMD optimizations.

Table 3: Performance metrics of 3D flux differencing volume terms in Trixi.jl using SIMD optimizations on an Intel® Core™ i7-8700K (CPU from 2017 with AVX2).

	`TreeMesh`	`StructuredMesh`	`P4estMesh`
Flux of Shima et al. (3.7)			
Vectorization ratio in %	98.36	98.85	98.85
Absolute performance in Gflops/s	21.44	18.51	18.44
Relative to peak performance in %	29.09	25.12	25.02
Flux of Ranocha (3.10)			
Vectorization ratio in %	99.09	99.26	99.26
Absolute performance in Gflops/s	18.80	15.28	15.25
Relative to peak performance in %	25.51	20.74	20.69

We used LIKWID [73] via its Julia interface LIKWID.jl to measure some performance metrics of the new volume terms optimized for SIMD instructions. For this, we used the same setup as in

Section 5 with polynomials of degree $p = 3$. We used a single batch computing the volume terms $5 \cdot 10^3$ times on an Intel® Core™ i7-8700K (CPU from 2017 with AVX2). The results are shown in Table 3. For all numerical fluxes and meshes, we achieved vectorization ratios of more than 98 %. This includes all required memory rearrangements described above.

We used the same setup with LIKWID to estimate the percentage of peak performance of floating point operations per second. We used the LIKWID benchmark tool to estimate the peak performance for fused multiply-add (FMA) instructions and measured the floating point performance of the flux differencing volume terms in Trixi.jl. The results are also shown in Table 3. Given that the estimated peak performance uses only FMAs and is rather optimistic while the actual algorithm involves other operations including divisions and logarithms for the EC flux, the obtained performance results are quite good. The volume terms on the curved meshes require additional memory rearrangements of the contravariant basis vectors (due to the choice of AoS style memory layouts described above). Together with the increased memory loads, this explains the reduced peak performance of flux differencing on curved meshes.

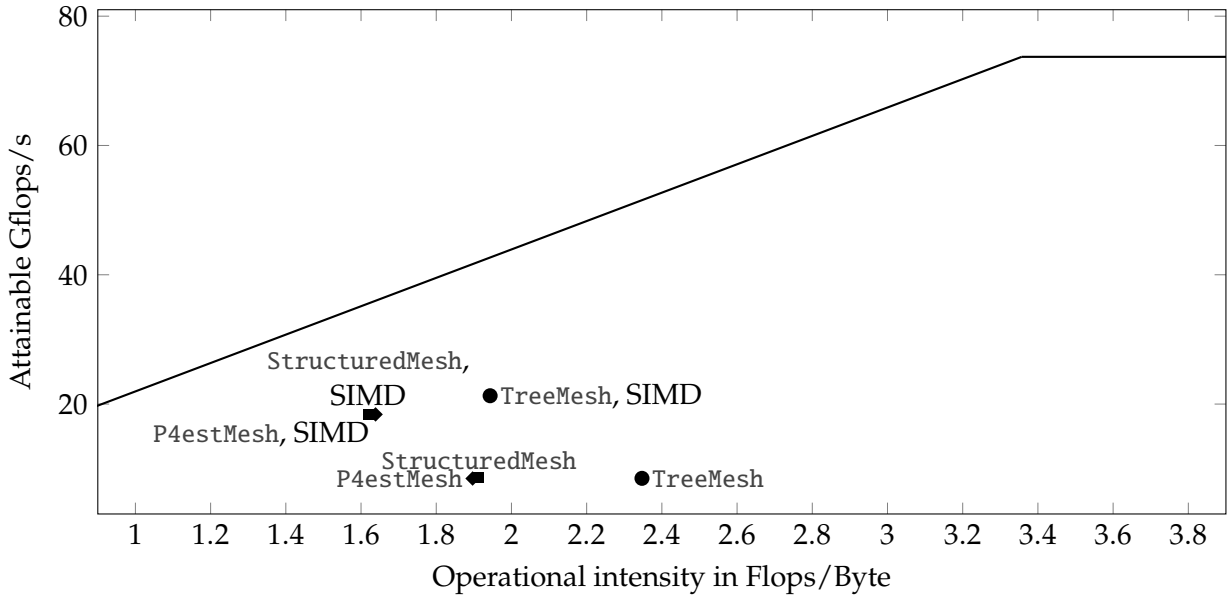
The results with and without SIMD optimizations described in this section are embedded into an empirical roofline model shown in Figure **??**. Again, the measurements have been performed using LIKWID, following their tutorial on creating an empirical roofline model. The peak performance has been estimated using FMA instructions and the maximum bandwidth has been estimated by load instructions (using LIKWID benchmark tools for both). First, note that the volume terms with SIMD optimizations have a reduced operational intensity compared to the plain versions. This is due to the fact that we precompute primitive variables and logarithms. Second, the SIMD-optimized versions have a higher performance. On the system we used for this benchmark, many data points (in particular for the cheaper non-EC flux) are under the slant of the roof, i.e., the regime that is usually characterized as limited by memory bandwidth. However, note that the peak performance is estimated by pure FMA instructions while the numerical fluxes involve more expensive operations such as divisions. Finally, these measurements are only valid when considering a single process operating on the CPU. In case of a parallel simulation, all processes on a single node need to share the same memory bandwidth. This can affect the serial performance of each process and skew the results further towards the memory-bound regime.

The new PID measurements including precomputing terms and SIMD optimizations are shown in Figure **??**. In contrast to the baseline results shown in Figure 1, the PID does not scale linearly with the polynomial degree. In 2D, the PID is approximately linearly *decreasing* for $p \in \{3, \dots, 7\}$. After the local minimum at $p = 7$, the PID plateaus again at a higher level. In 3D, a rough linear trend of the PID can still be observed. However, the PID plateaus for small polynomial degrees up to $p = 6$ or $p = 7$, depending on the mesh type and numerical flux.

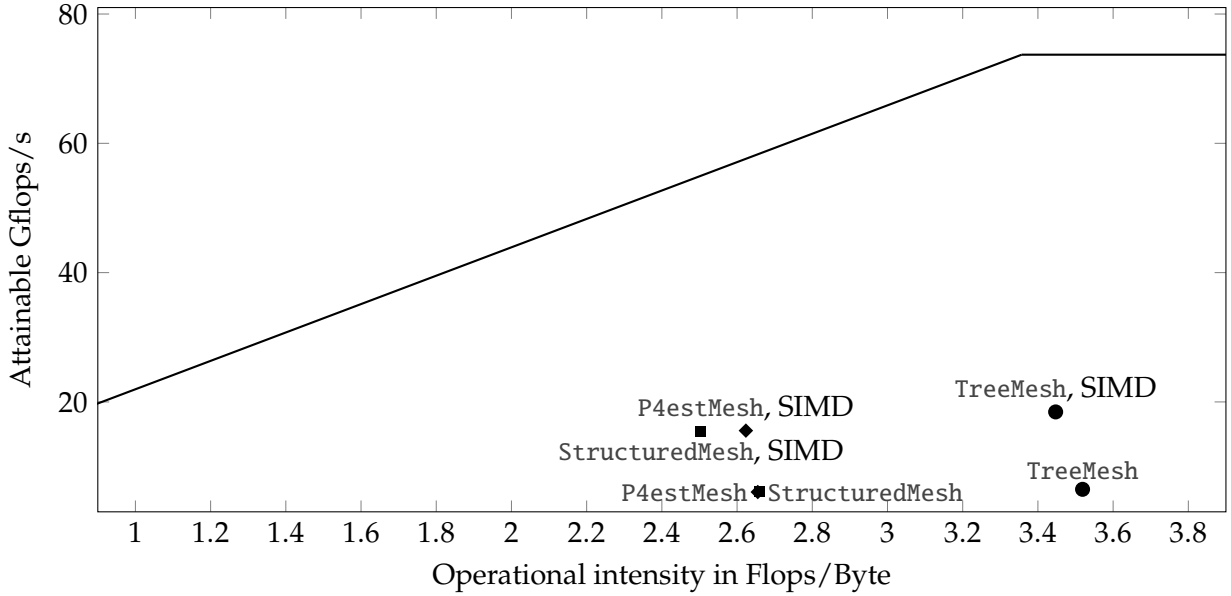
Figure **??** shows the relative PID improvements obtained by SIMD optimizations of the flux differencing volume terms, i.e., the ratio of the PID from Figures **??** and 1. The trends described above are even more visible here. In particular, the speedup obtained by all optimizations (precomputing terms and using SIMD optimizations) improves for low polynomial degrees. In 2D, the speedup shows a local optimum at $p = 7$.

Note that the PID measures also the time needed for other parts of the right-hand side computations. In particular, the cost of surface terms increases relatively as the volume terms become cheaper. Some SIMD optimizations could also be applied to surface terms to further increase the total speedup. However, we restrict our attention to the flux differencing volume terms in this article.

In total, precomputing variables and using SIMD optimizations improves the PID in 2D between 20 % (low polynomial degree, cheap volume flux) and 3 × ($p = 7$, EC volume flux). In 3D, the speedup obtained by these optimizations is between 30 % (low polynomial degree, cheap volume flux) and 3 × (high polynomial degree, EC flux).



(a) Flux of Shima et al. (3.7).



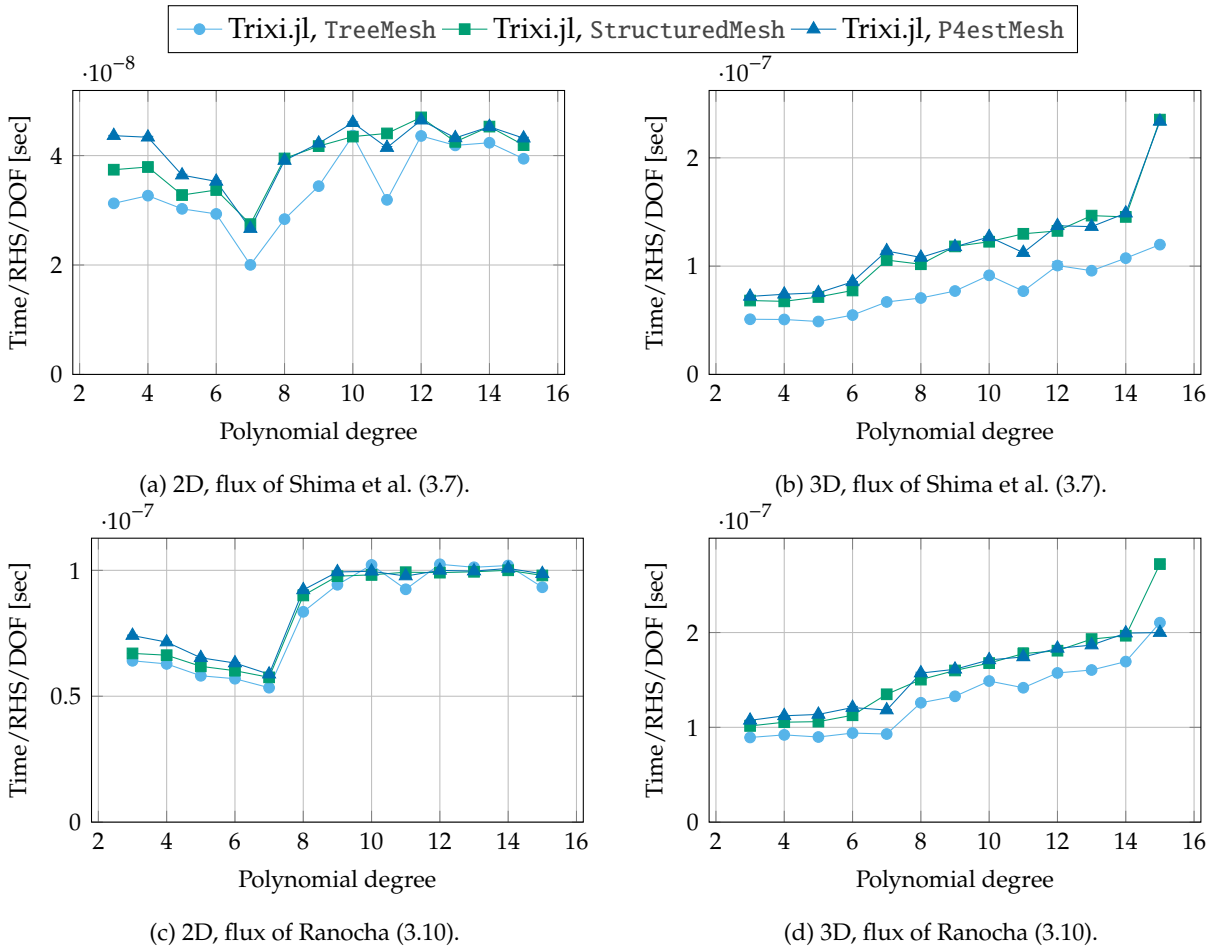
(b) Flux of Ranocha (3.10).

Figure 9: Empirical roofline model [74] measured using LIKWID [73] for the flux differencing volume terms of the 3D compressible Euler equations using polynomials of degree $p = 3$ and 8 elements per coordinate direction on an Intel® Core™ i7-8700K (CPU from 2017 with AVX2). Results are shown on the `TreeMesh` (circles), `StructuredMesh` (squares), and the `P4estMesh` (diamonds) of Trixi.jl.

9 Summary and conclusions

We discussed techniques for the efficient implementation of flux differencing schemes, focusing on the compressible Euler equations and discontinuous Galerkin methods. Starting with a high-level description of the algorithms, we presented general modifications of the equations typically presented in research articles as first step towards an efficient implementation. All of these techniques are freely available in our open source codes Trixi.jl and FLUXO as well as our reproducibility repository [55].

We concentrated on the serial performance of flux differencing for the compressible Euler equations in 2D and 3D. Most of the techniques presented in this article are agnostic to the code base and



(a) 2D, flux of Shima et al. (3.7).

(b) 3D, flux of Shima et al. (3.7).

(c) 2D, flux of Ranocha (3.10).

(d) 3D, flux of Ranocha (3.10).

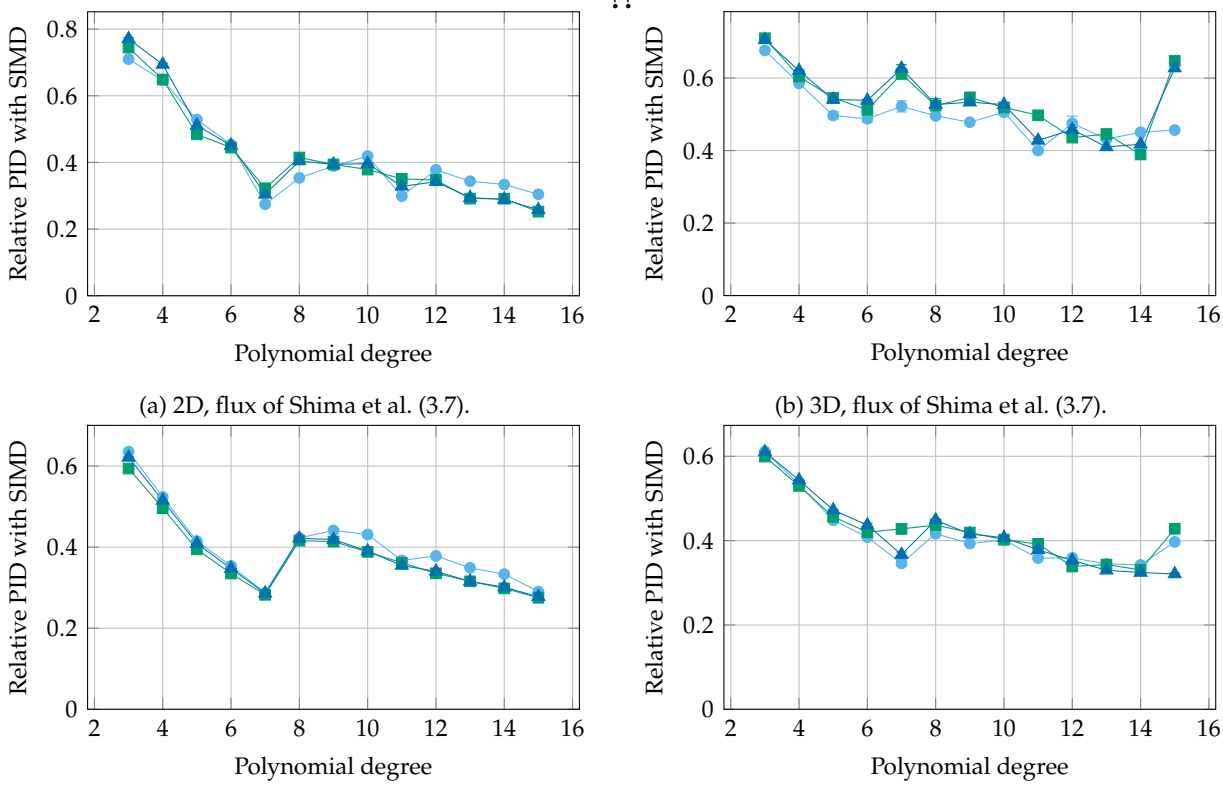
Figure 10: Runtime per right-hand side evaluation and degree of freedom for different mesh types and LGL-DGSEM discretizations with SIMD optimizations of the volume terms for the compressible Euler equations.

programming language, as demonstrated by results obtained with Julia and Fortran. Extensions to non-conservative terms as well as MPI parallelization will be discussed elsewhere, since these questions are largely orthogonal to the issues discussed here. In the MPI parallel case, one would usually expect a sublinear scaling up to a full single node due to memory bandwidth and cache competition followed by a (close to) ideal scaling when adding more nodes, as shown in [58].

From these general performance optimizations, we compared flux differencing to a simple version of overintegration. In general, flux differencing is quite competitive in terms of runtime performance. In addition, it comes with less strict explicit time step restrictions and is robust for many setups where overintegration fails [76]. For practically relevant parameters for computational fluid dynamics (3D, polynomials of degree $p = 3$), flux differencing is even faster than overintegration with a single additional node per coordinate direction.

We also discussed more invasive optimizations including memory layout adaptation for SIMD techniques. While these are rather code-specific, they can provide great benefits on modern hardware. Using a compromise of legacy code layout and SIMD optimizations, we could achieve the following performance indices PID (time per right-hand side evaluation and degree of freedom) for flux differencing discretizations of the compressible Euler equation in 3D with polynomials of degree $p = 3$ in Trixi.jl on an Intel® Core™ i7-8700K (CPU from 2017 with AVX2): 2.6×10^{-8} s on a Cartesian mesh and 4.3×10^{-8} s on an unstructured, curved mesh for standard split form discretizations using the flux of Shima et al. [62]; 4.2×10^{-8} s on a Cartesian mesh and 6.5×10^{-8} s on an unstructured, curved mesh for entropy-based discretizations using the flux of Ranocha [46–48].

??



(a) 2D, flux of Shima et al. (3.7).

(b) 3D, flux of Shima et al. (3.7).

(c) 2D, flux of Ranocha (3.10).

(d) 3D, flux of Ranocha (3.10).

Figure 11: Relative runtime improvements obtained by SIMD optimizations of the volume terms for different mesh types and LGL-DGSEM discretizations for the compressible Euler equations.

Acknowledgments

Funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy EXC 2044-390685587, Mathematics Münster: Dynamics-Geometry-Structure, and through the DFG research unit "SNuBIC" (FOR 5409; project number 463312734).

Hendrik Ranocha was supported by the Daimler und Benz Stiftung (Daimler and Benz foundation, project number 32-10/22).

This work has received funding from the European Research Council through the ERC Starting Grant "An Exascale aware and Un-crashable Space-Time-Adaptive Discontinuous Spectral Element Solver for Non-Linear Conservation Laws" (Extreme), ERC grant agreement no. 714487 (Gregor J. Gassner, Michael Schlottke-Lakemper, and Andrés Rueda-Ramírez).

Andrew R. Winters was supported through Vetenskapsrådet, Sweden grant agreement 2020-03642 VR.

Jesse Chan was supported through the United States National Science Foundation under awards DMS-1719818 and DMS-1943186.

The authors gratefully acknowledge the computing time provided on the supercomputer NEC Vulcan by the High-Performance Computing Center Stuttgart (HLRS) of the University of Stuttgart, Germany.

References

[1] R. Abgrall, J. Nordström, P. Öffner, and S. Tokareva. "Analysis of the SBP-SAT Stabilization for Finite Element Methods Part I: Linear problems." In: *Journal of Scientific Computing* 85.2 (2020), pp. 1–29. DOI: 10.1007/s10915-020-01349-z. arXiv: 1912.08108 [math.NA].

[2] O. Ålund and J. Nordström. "Encapsulated high order difference operators on curvilinear non-conforming grids." In: *Journal of Computational Physics* 385 (2019), pp. 209–224. DOI: 10.1016/j.jcp.2019.02.007.

[3] M Bergmann, C Morsbach, and G Ashcroft. "Assessment of Split Form Nodal Discontinuous Galerkin Schemes for the LES of a Low Pressure Turbine Profile." In: *Direct and Large Eddy Simulation XII*. Vol. 27. ERCOFTACSeries. Cham: Springer Nature, 2020, pp. 365–371. DOI: 10.1007/978-3-030-42822-8_48.

[4] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah. "Julia: A Fresh Approach to Numerical Computing." In: *SIAM Review* 59.1 (2017), pp. 65–98. DOI: 10.1137/141000671. arXiv: 1411.1607 [cs.MS].

[5] M. H. Carpenter, T. C. Fisher, E. J. Nielsen, and S. H. Frankel. "Entropy Stable Spectral Collocation Schemes for the Navier-Stokes Equations: Discontinuous Interfaces." In: *SIAM Journal on Scientific Computing* 36.5 (2014), B835–B867. DOI: 10.1137/130932193.

[6] J. Chan. "On discretely entropy conservative and entropy stable discontinuous Galerkin methods." In: *Journal of Computational Physics* 362 (2018), pp. 346–374. DOI: 10.1016/j.jcp.2018.02.033.

[7] J. Chan. "Skew-symmetric entropy stable modal discontinuous Galerkin formulations." In: *Journal of Scientific Computing* 81.1 (2019), pp. 459–485. DOI: 10.1007/s10915-019-01026-w.

[8] J. Chan, D. C. D. R. Fernández, and M. H. Carpenter. "Efficient entropy stable Gauss collocation methods." In: *SIAM Journal on Scientific Computing* 41.5 (2019), A2938–A2966. DOI: 10.1137/18M1209234.

[9] P. Chandrashekar. "Kinetic Energy Preserving and Entropy Stable Finite Volume Schemes for Compressible Euler and Navier-Stokes Equations." In: *Communications in Computational Physics* 14.5 (2013), pp. 1252–1286. DOI: 10.4208/cicp.170712.010313a.

[10] T. Chen and C.-W. Shu. "Entropy stable high order discontinuous Galerkin methods with suitable quadrature rules for hyperbolic conservation laws." In: *Journal of Computational Physics* 345 (2017), pp. 427–461. DOI: 10.1016/j.jcp.2017.05.025.

[11] J. Crean, J. E. Hicken, D. C. D. R. Fernández, D. W. Zingg, and M. H. Carpenter. "Entropy-stable summation-by-parts discretization of the Euler equations on general curved elements." In: *Journal of Computational Physics* 356 (2018), pp. 410–438. DOI: 10.1016/j.jcp.2017.12.015.

[12] C. Elrod. *Roadmap to Julia BLAS and Linear Algebra*. https://www.youtube.com/watch?v=KQ8nvlURX4M. Talk presented at JuliaCon 2021. virtual, 2021.

[13] D. C. D. R. Fernández, P. D. Boom, and D. W. Zingg. "A generalized framework for nodal first derivative summation-by-parts operators." In: *Journal of Computational Physics* 266 (2014), pp. 214–239. DOI: 10.1016/j.jcp.2014.01.038.

[14] D. C. D. R. Fernández, J. E. Hicken, and D. W. Zingg. "Review of summation-by-parts operators with simultaneous approximation terms for the numerical solution of partial differential equations." In: *Computers & Fluids* 95 (2014), pp. 171–196. DOI: 10.1016/j.compfluid.2014.02.016.

[15] T. C. Fisher, M. H. Carpenter, J. Nordström, N. K. Yamaleev, and C. Swanson. "Discretely conservative finite-difference formulations for nonlinear conservation laws in split form: Theory and boundary conditions." In: *Journal of Computational Physics* 234 (2013), pp. 353–375. DOI: 10.1016/j.jcp.2012.09.026.

[16] U. S. Fjordholm, S. Mishra, and E. Tadmor. "Arbitrarily High-Order Accurate Entropy Stable Essentially Nonoscillatory Schemes for Systems of Conservation Laws." In: *SIAM Journal on Numerical Analysis* 50.2 (2012), pp. 544–573. DOI: 10.1137/110836961.

[17] D. Flad and G. Gassner. "On the use of kinetic energy preserving DG-schemes for large eddy simulation." In: *Journal of Computational Physics* 350 (2017), pp. 782–795. DOI: 10.1016/j.jcp.2017.09.004.

[18] A. Fog. *Instruction tables. Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD, and VIA CPUs*. Version 2021-08-17, accessed 2021-10-19. Aug. 2021. URL: https://www.agner.org/optimize/instruction_tables.pdf.

[19] G. J. Gassner and A. R. Winters. "A Novel Robust Strategy for Discontinuous Galerkin Methods in Computational Fluid Mechanics: Why? When? What? Where?" In: *Frontiers in Physics* 8 (2021), p. 612. DOI: 10.3389/fphy.2020.500690.

[20] G. J. Gassner. "A kinetic energy preserving nodal discontinuous Galerkin spectral element method." In: *International Journal for Numerical Methods in Fluids* 76.1 (2014), pp. 28–50. DOI: 10.1002/fld.3923.

[21] G. J. Gassner. "A Skew-Symmetric Discontinuous Galerkin Spectral Element Discretization and Its Relation to SBP-SAT Finite Difference Methods." In: *SIAM Journal on Scientific Computing* 35.3 (2013), A1233–A1253. DOI: 10.1137/120890144.

[22] G. J. Gassner and A. D. Beck. "On the accuracy of high-order discretizations for underresolved turbulence simulations." In: *Theoretical and Computational Fluid Dynamics* 27.3-4 (2013), pp. 221–237. DOI: 10.1007/s00162-011-0253-7.

[23] G. J. Gassner and D. A. Kopriva. "A Comparison of the Dispersion and Dissipation Errors of Gauss and Gauss-Lobatto Discontinuous Galerkin Spectral Element Methods." In: *SIAM Journal on Scientific Computing* 33.5 (2011), pp. 2560–2579. DOI: 10.1137/100807211.

[24] G. J. Gassner, A. R. Winters, and D. A. Kopriva. "Split Form Nodal Discontinuous Galerkin Schemes with Summation-By-Parts Property for the Compressible Euler Equations." In: *Journal of Computational Physics* 327 (2016), pp. 39–66. DOI: 10.1016/j.jcp.2016.09.013.

[25] J.-L. Guermond, M. Kronbichler, M. Maier, B. Popov, and I. Tomas. "On the implementation of a robust and efficient finite element-based parallel solver for the compressible Navier-Stokes equations." In: *Computer Methods in Applied Mechanics and Engineering* 389 (Feb. 2022), p. 114250. DOI: 10.1016/j.cma.2021.114250. arXiv: 2106.02159 [math.NA].

[26] A. Harten, P. D. Lax, and B. van Leer. "On Upstream Differencing and Godunov-Type Schemes for Hyperbolic Conservation Laws." In: *SIAM Review* 25.1 (1983), pp. 35–61. DOI: 10.1137/1025002.

[27] J. S. Hesthaven and T. Warburton. *Nodal Discontinuous Galerkin Methods: Algorithms, Analysis, and Applications*. Vol. 54. Texts in Applied Mathematics. New York: Springer Science & Business Media, 2007. DOI: 10.1007/978-0-387-72067-8.

[28] J. E. Hicken. "Entropy-stable, high-order summation-by-parts discretizations without interface penalties." In: *Journal of Scientific Computing* 82.2 (2020), p. 50. DOI: 10.1007/s10915-020-01154-8.

[29] J. E. Hicken, D. C. D. R. Fernández, and D. W. Zingg. "Multidimensional Summation-By-Parts Operators: General Theory and Application to Simplex Elements." In: *SIAM Journal on Scientific Computing* 38.4 (2016), A1935–A1958. DOI: 10.1137/15M1038360.

[30] H. T. Huynh. "A Flux Reconstruction Approach to High-Order Schemes Including Discontinuous Galerkin Methods." In: *18th AIAA Computational Fluid Dynamics Conference*. American Institute of Aeronautics and Astronautics. 2007. DOI: 10.2514/6.2007-4079.

[31] F. Ismail and P. L. Roe. "Affordable, entropy-consistent Euler flux functions II: Entropy production at shocks." In: *Journal of Computational Physics* 228.15 (2009), pp. 5410–5436. DOI: 10.1016/j.jcp.2009.04.021.

[32] A. Jameson. "Formulation of Kinetic Energy Preserving Conservative Schemes for Gas Dynamics and Direct Numerical Simulation of One-Dimensional Viscous Compressible Flow in a Shock Tube Using Entropy and Kinetic Energy Preserving Schemes." In: *Journal of Scientific Computing* 34.2 (2008), pp. 188–208. DOI: 10.1007/s10915-007-9172-6.

[33] C. A. Kennedy and M. H. Carpenter. *Fourth Order 2N-Storage Runge-Kutta Schemes*. Technical Memorandum NASA-TM-109112. NASA Langley Research Center, Hampton VA 23681-0001, United States: NASA, June 1994.

[34] B. F. Klose, G. B. Jacobs, and D. A. Kopriva. "Assessing standard and kinetic energy conserving volume fluxes in discontinuous Galerkin formulations for marginally resolved Navier-Stokes flows." In: *Computers & Fluids* (2020), p. 104557. DOI: 10.1016/j.compfluid.2020.104557.

[35] D. A. Kopriva. *Implementing Spectral Methods for Partial Differential Equations: Algorithms for Scientists and Engineers*. New York: Springer Science & Business Media, 2009. DOI: 10.1007/978-90-481-2261-5.

[36] D. A. Kopriva. "Metric identities and the discontinuous spectral element method on curvilinear meshes." In: *Journal of Scientific Computing* 26.3 (2006), pp. 301–327. DOI: 10.1007/s10915-005-9070-8.

[37] D. A. Kopriva and G. J. Gassner. "On the Quadrature and Weak Form Choices in Collocation Type Discontinuous Galerkin Spectral Element Methods." In: *Journal of Scientific Computing* 44.2 (2010), pp. 136–155. DOI: 10.1007/s10915-010-9372-3.

[38] N. Krais, A. Beck, T. Bolemann, H. Frank, D. Flad, G. Gassner, F. Hindenlang, M. Hoffmann, T. Kuhn, M. Sonntag, and C.-D. Munz. "FLEXI: A high order discontinuous Galerkin framework for hyperbolic-parabolic conservation laws." In: *Computers & Mathematics with Applications* 81 (2021), pp. 186–219. DOI: 10.1016/j.camwa.2020.05.004.

[39] H.-O. Kreiss and G. Scherer. "Finite Element and Finite Difference Methods for Hyperbolic Partial Differential Equations." In: *Mathematical Aspects of Finite Elements in Partial Differential Equations*. Ed. by C. de Boor. New York: Academic Press, 1974, pp. 195–212.

[40] P. G. LeFloch, J.-M. Mercier, and C. Rohde. "Fully Discrete, Entropy Conservative Schemes of Arbitrary Order." In: *SIAM Journal on Numerical Analysis* 40.5 (2002), pp. 1968–1992. DOI: 10.1137/S003614290240069X.

[41] M. Maier and M. Kronbichler. "Efficient parallel 3D computation of the compressible Euler equations with an invariant-domain preserving second-order finite-element scheme." In: *ACM Transactions on Parallel Computing* 8.3 (2021), pp. 1–30. DOI: 10.1145/3470637.

[42] J. Nordström and M. Björck. "Finite volume approximations and strict stability for hyperbolic problems." In: *Applied Numerical Mathematics* 38.3 (2001), pp. 237–255. DOI: 10.1016/S0168-9274(01)00027-7.

[43] J. Nordström, K. Forsberg, C. Adamsson, and P. Eliasson. "Finite volume methods, unstructured meshes and strict stability for hyperbolic problems." In: *Applied Numerical Mathematics* 45.4 (2003), pp. 453–473. DOI: 10.1016/S0168-9274(02)00239-8.

[44] M. Parsani, R. Boukharfane, I. R. Nolasco, D. C. D. R. Fernández, S. Zampini, B. Hadri, and L. Dalcin. "High-order accurate entropy-stable discontinuous collocated Galerkin methods with the summation-by-parts property for compressible CFD frameworks: Scalable SSDC algorithms and flow solver." In: *Journal of Computational Physics* 424 (2021), p. 109844. DOI: 10.1016/j.jcp.2020.109844.

[45] C. Rackauckas and Q. Nie. "DifferentialEquations.jl – A Performant and Feature-Rich Ecosystem for Solving Differential Equations in Julia." In: *Journal of Open Research Software* 5.1 (2017), p. 15. DOI: 10.5334/jors.151.

[46] H. Ranocha. "Comparison of Some Entropy Conservative Numerical Fluxes for the Euler Equations." In: *Journal of Scientific Computing* 76.1 (July 2018), pp. 216–242. DOI: 10.1007/s10915-017-0618-1. arXiv: 1701.02264 [math.NA].

[47] H. Ranocha. "Entropy Conserving and Kinetic Energy Preserving Numerical Methods for the Euler Equations Using Summation-by-Parts Operators." In: *Spectral and High Order Methods for Partial Differential Equations ICOSAHOM 2018*. Ed. by S. J. Sherwin, D. Moxey, J. Peiró, P. E. Vincent, and C. Schwab. Vol. 134. Lecture Notes in Computational Science and Engineering. Cham: Springer, Aug. 2020, pp. 525–535. DOI: 10.1007/978-3-030-39647-3_42.

[48] H. Ranocha. "Generalised Summation-by-Parts Operators and Entropy Stability of Numerical Methods for Hyperbolic Balance Laws." PhD thesis. TU Braunschweig, Feb. 2018.

[49] H. Ranocha. "Shallow water equations: Split-form, entropy stable, well-balanced, and positivity preserving numerical methods." In: *GEM – International Journal on Geomathematics* 8.1 (Apr. 2017), pp. 85–133. DOI: 10.1007/s13137-016-0089-9. arXiv: 1609.08029 [math.NA].

[50] H. Ranocha. "SummationByPartsOperators.jl: A Julia library of provably stable semidiscretization techniques with mimetic properties." In: *Journal of Open Source Software* 6.64 (Aug. 2021), p. 3454. DOI: 10.21105/joss.03454. URL: https://github.com/ranocha/SummationByPartsOperators.jl.

[51] H. Ranocha, L. Dalcin, M. Parsani, and D. I. Ketcheson. "Optimized Runge-Kutta Methods with Automatic Step Size Control for Compressible Computational Fluid Dynamics." In: *Communications on Applied Mathematics and Computation* (Nov. 2021). DOI: 10.1007/s42967-021-00159-w. arXiv: 2104.06836 [math.NA].

[52] H. Ranocha and G. J. Gassner. "Preventing pressure oscillations does not fix local linear stability issues of entropy-based split-form high-order schemes." In: *Communications on Applied Mathematics and Computation* (Aug. 2021). DOI: 10.1007/s42967-021-00148-z. arXiv: 2009.13139 [math.NA].

[53] H. Ranocha, D. Mitsotakis, and D. I. Ketcheson. "A Broad Class of Conservative Numerical Methods for Dispersive Wave Equations." In: *Communications in Computational Physics* 29.4 (Feb. 2021), pp. 979–1029. DOI: 10.4208/cicp.OA-2020-0119. arXiv: 2006.14802 [math.NA].

[54] H. Ranocha, P. Öffner, and T. Sonar. "Summation-by-parts operators for correction procedure via reconstruction." In: *Journal of Computational Physics* 311 (Apr. 2016), pp. 299–328. DOI: 10.1016/j.jcp.2016.02.009. arXiv: 1511.02052 [math.NA].

[55] H. Ranocha, M. Schlottke-Lakemper, J. Chan, A. M. Rueda-Ramírez, A. R. Winters, F. Hindenlang, and G. J. Gassner. *Reproducibility repository for Efficient implementation of modern entropy stable and kinetic energy preserving discontinuous Galerkin methods for conservation laws*. https://github.com/trixi-framework/paper-2021-EC_performance. Dec. 2021. DOI: 10.5281/zenodo.5792576.

[56] H. Ranocha, M. Schlottke-Lakemper, A. R. Winters, E. Faulhaber, J. Chan, and G. J. Gassner. "Adaptive numerical simulations with Trixi.jl: A case study of Julia for scientific computing." In: *Proceedings of the JuliaCon Conferences* 1.1 (Jan. 2022), p. 77. DOI: 10.21105/jcon.00077. arXiv: 2108.06476 [cs.MS].

[57] T. T. Ribeiro. *Final report on HLST project OPT-DG2*. Final Report. Boltzmannstraße 2, 85748 Garching, Germany: Max-Planck-Institut für Plasmaphysik, Oct. 2020.

[58] M. Rogowski, L. Dalcin, M. Parsani, and D. E. Keyes. "Performance analysis of relaxation Runge-Kutta methods." In: *The International Journal of High Performance Computing Applications* (May 2022). DOI: 10.1177/10943420221085947.

[59] D. Rojas, R. Boukharfane, L. Dalcin, D. C. D. R. Fernández, H. Ranocha, D. E. Keyes, and M. Parsani. "On the robustness and performance of entropy stable discontinuous collocation methods." In: *Journal of Computational Physics* 426 (Feb. 2021), p. 109891. DOI: 10.1016/j.jcp.2020.109891. arXiv: 1911.10966 [math.NA].

[60] A. M. Rueda-Ramírez, S. Hennemann, F. J. Hindenlang, A. R. Winters, and G. J. Gassner. "An entropy stable nodal discontinuous Galerkin method for the resistive MHD equations. Part II: Subcell finite volume shock capturing." In: *Journal of Computational Physics* 444 (2021), p. 110580. DOI: 10.1016/j.jcp.2021.110580.

[61] M. Schlottke-Lakemper, A. R. Winters, H. Ranocha, and G. J. Gassner. "A purely hyperbolic discontinuous Galerkin approach for self-gravitating gas dynamics." In: *Journal of Computational Physics* 442 (June 2021), p. 110467. DOI: 10.1016/j.jcp.2021.110467. arXiv: 2008.10593 [math.NA].

[62] N. Shima, Y. Kuya, Y. Tamaki, and S. Kawai. "Preventing spurious pressure oscillations in split convective form discretization for compressible flows." In: *Journal of Computational Physics* (2020), p. 110060. DOI: 10.1016/j.jcp.2020.110060.

[63] C.-W. Shu. *Essentially Non-Oscillatory and Weighted Essentially Non-Oscillatory Schemes for Hyperbolic Conservation Laws*. Final Report NASA/CR-97-206253. Institute for Computer Applications in Science and Engineering, NASA Langley Research Center, Hampton VA United States: NASA, Nov. 1997.

[64] B. Sjögreen and H. Yee. "High order entropy conservative central schemes for wide ranges of compressible gas dynamics and MHD flows." In: *Journal of Computational Physics* 364 (2018), pp. 153–185. DOI: 10.1016/j.jcp.2018.02.003.

[65] B. Sjögreen, H. C. Yee, and D. Kotov. "Skew-symmetric splitting and stability of high order central schemes." In: *Journal of Physics: Conference Series*. Vol. 837. 1. IOP Publishing. 2017, p. 012019. DOI: 10.1088/1742-6596/837/1/012019.

[66] B. Strand. "Summation by Parts for Finite Difference Approximations for d/dx." In: *Journal of Computational Physics* 110.1 (1994), pp. 47–67. DOI: 10.1006/jcph.1994.1005.

[67] M. Svärd and J. Nordström. "Review of summation-by-parts schemes for initial-boundary-value problems." In: *Journal of Computational Physics* 268 (2014), pp. 17–38. DOI: 10.1016/j.jcp.2014.02.031.

[68] E. Tadmor. "Entropy stability theory for difference approximations of nonlinear conservation laws and related time-dependent problems." In: *Acta Numerica* 12 (2003), pp. 451–512. DOI: 10.1017/S0962492902000156.

[69] E. Tadmor. "The numerical viscosity of entropy stable schemes for systems of conservation laws. I." In: *Mathematics of Computation* 49.179 (1987), pp. 91–103. DOI: 10.1090/S0025-5718-1987-0890255-3.

[70] P.-T. P. Tang. "Table-driven implementation of the logarithm function in IEEE floating-point arithmetic." In: *ACM Transactions on Mathematical Software (TOMS)* 16.4 (1990), pp. 378–400. DOI: 10.1145/98267.98294.

[71] P. Thomas and C. Lombard. "Geometric conservation law and its application to flow computations on moving grids." In: *AIAA journal* 17.10 (1979), pp. 1030–1037. DOI: 10.2514/3.61273.

[72] E. F. Toro. *Riemann Solvers and Numerical Methods for Fluid Dynamics: A Practical Introduction*. Berlin Heidelberg: Springer, 2009. DOI: 10.1007/b79761.

[73] J. Treibig, G. Hager, and G. Wellein. "LIKWID: A lightweight performance-oriented tool suite for x86 multicore environments." In: *2010 39th International Conference on Parallel Processing Workshops*. IEEE. 2010, pp. 207–216. DOI: 10.1109/ICPPW.2010.38.

[74] S. Williams, A. Waterman, and D. Patterson. "Roofline: an insightful visual performance model for multicore architectures." In: *Communications of the ACM* 52.4 (2009), pp. 65–76. DOI: 10.1145/1498765.1498785.

[75] N. Wintermeyer, A. R. Winters, G. J. Gassner, and D. A. Kopriva. "An entropy stable nodal discontinuous Galerkin method for the two dimensional shallow water equations on unstructured curvilinear meshes with discontinuous bathymetry." In: *Journal of Computational Physics* 340 (2017), pp. 200–242. DOI: 10.1016/j.jcp.2017.03.036.

[76] A. R. Winters, R. C. Moura, G. Mengaldo, G. J. Gassner, S. Walch, J. Peiro, and S. J. Sherwin. "A comparative study on polynomial dealiasing and split form discontinuous Galerkin schemes for under-resolved turbulence computations." In: *Journal of Computational Physics* 372 (2018), pp. 1–21. DOI: 10.1016/j.jcp.2018.06.016.

[77] A. R. Winters and G. J. Gassner. "A comparison of two entropy stable discontinuous Galerkin spectral element approximations for the shallow water equations with non-constant topography." In: *Journal of Computational Physics* 301 (2015), pp. 357–376. DOI: 10.1016/j.jcp.2015.08.034.