



p-adaptive discontinuous Galerkin method for the shallow water equations on heterogeneous computing architectures

Sara Faghih-Naini^{1,2} · Vadym Aizinger¹  · Sebastian Kuckuk^{2,3} · Richard Angersbach² · Harald Köstler^{2,3}

Received: 15 November 2023 / Accepted: 13 March 2025 / Published online: 17 April 2025
© The Author(s) 2025

Abstract

Heterogeneous computing and exploiting integrated CPU–GPU architectures has become a clear current trend since the flattening of Moore’s Law. In this work, we propose a numerical and algorithmic re-design of a p-adaptive quadrature-free discontinuous Galerkin (DG) method for the shallow water equations. Our new approach separates the computations of the non-adaptive (lower-order) and adaptive (higher-order) parts of the discretization from each other. Thereby, we can overlap computations of the lower-order and the higher-order DG solution components. Furthermore, we investigate execution times of main computational kernels and use automatic code generation to optimize their distribution between the CPU and GPU. Several setups, including a prototype of a tsunami simulation in a tide-driven flow scenario, are investigated, and the results show that significant performance improvements can be achieved in suitable setups.

Keywords p-adaptivity · Heterogeneous architectures · GPU computing · System-on-a-chip (SoC) · Discontinuous Galerkin method · Quadrature-free integration · Shallow water equations

Mathematics Subject Classification 76-10 · 65Y10 · 65M50 · 65M60

1 Introduction

One of the key factors limiting the accuracy and the physical relevance of climate models is the computational performance of the hardware those models are executed on.

✉ Vadym Aizinger
vadym.aizinger@uni-bayreuth.de

¹ Chair of Scientific Computing, University of Bayreuth, 95440 Bayreuth, Germany

² Chair of Computer Science 10, Friedrich-Alexander Universität Erlangen-Nürnberg, 91058 Erlangen, Germany

³ Erlangen National High Performance Computing Center (NHR@FAU), Friedrich-Alexander Universität Erlangen-Nürnberg, 91058 Erlangen, Germany

The current computational paradigm for numerical models of ocean and atmosphere mostly relies on massively parallel and, in part, hybrid platforms. However, new ways are required in order to achieve improvements in computational performance in the time of failing Moore's Law and more heterogeneous landscape of relevant computing architectures.

There already exist some implementations of the shallow water equations (SWE) on heterogeneous hardware. In Echeverribar et al. (2020), for example, a coupled 1D-2D model for real flood cases is hybridized using a heterogeneous CPU-GPU architecture. A different approach is taken in Chaplygin et al. (2022) for a 2D shallow water model, where the domain is partitioned into subdomains distributed between CPUs and GPUs. Furthermore, in Fu et al. (2017), the global SWE are solved by dividing subblocks of patches into a CPU and an accelerator part. When considering the DG method for general types of PDEs, there exist several approaches that utilize heterogeneous CPU-GPU systems (Mu et al. 2013; Witherden et al. 2015; Bramas et al. 2020). However, all of these rely on distributing the work through domain partitioning. However, to the best of our knowledge, no works attempt to adapt the original algorithm's numerics to parallelize it between different architectures.

Among the most promising numerical methodologies aiming to optimize the computational performance of PDE discretizations are adaptive schemes. They are particularly attractive in combination with finite element and finite volume methods and rely on adaptive mesh refinement (h-adaptivity) or local adjustment of the polynomial order of the discretization (p-adaptivity). However, adaptive numerical schemes for time dependent problems have a critical limitation when used in combination with massively parallel (usually based on distributed memory parallelization) computing, namely the issue of obtaining a good load balance throughout the simulation. Several strategies of load-balancing have been proposed, e.g., (Hendrickson and Devine 2000; Biswas et al. 2000; Teresco et al. 2006; Baiges and Bayona 2017), but they increase the code complexity and result in additional computational overhead. This is one of the main reasons why dynamically adaptive numerical schemes were unable to gain a significant foothold in some communities such as numerical weather prediction or ocean modeling.

The current work attempts to take a new approach to the load balancing issue in connection with a p-adaptive DG method: by separating the numerical scheme and the solution algorithm into a lower-order non-adaptive (base computation) and a higher-order adaptive (correction computation) parts, we can execute the correction computation on a separate hardware without influencing the load balance of the base computation. This strategy of encapsulating the adaptive part of the numerical method in a separate kernel offers, in addition, a meaningful way to map time-dependent adaptive finite element schemes to task-based programming models particularly attractive for heterogeneous hardware architectures (see, e.g., (Bosilca et al. 2013; Garcia-Gasulla et al. 2019)).

The tight coupling between the base and correction computations favors hardware architectures minimizing the performance impact of communication between their parts (e.g., CPU and GPU). This motivates the focus of our current work on integrated CPU-GPU architectures represented by Systems-on-a-Chip (SoCs). They are well suited for heterogeneous computing environments, which is a clear current trend,

offer low-latency data transfer between the CPU and GPU, and tend to be energy efficient (Geveler et al. 2016). In order to fully exploit these systems, hardware-driven algorithm design, i.e., fundamental changes in the algorithm are necessary, and the multiple instruction multiple data (MIMD) level of parallelism in Flynn’s taxonomy (Flynn 1972) may become beneficial.

We begin in the next section by introducing the mathematical model and its discretization by a quadrature-free p-adaptive DG method. Section 3 details our new approach of separating the lower-order degrees of freedom from the remainder of the discrete solution. In Sect. 4, we explain the implementation of this approach within our code generation framework and discuss the necessary adaptations. Numerical results used for the evaluation of the computational performance of the proposed scheme are presented in Sect. 5. There we also detail individual kernel execution times for a realistic simulation setup. A short conclusions and outlook section wraps up the manuscript.

2 Quadrature-free DG formulation of the shallow water equations

The presentation in this section closely follows (Faghih-Naini and Aizinger 2022; Faghih-Naini et al. 2020, 2023) and is included here for completeness. The discontinuous Galerkin discretization of the 2D SWE used in this work was originally proposed in Aizinger and Dawson (2002). The 2D SWE in conservative form on some domain Ω are given by

$$\partial_t \xi + \nabla \cdot \mathbf{q} = 0, \tag{1}$$

$$\partial_t \mathbf{q} + \nabla \cdot \left(\mathbf{q} \mathbf{q}^T / H \right) + \tau_{\text{bf}} \mathbf{u} + \begin{pmatrix} 0 & -f_c \\ f_c & 0 \end{pmatrix} \mathbf{q} + g H \nabla \xi = \mathbf{F} \tag{2}$$

where ξ represents the surface elevation with respect to some datum (e.g., the mean sea level), $H = h_b + \xi$ is the total fluid depth (h_b denotes here the bathymetry with respect to the same datum), and $\mathbf{q} \equiv (U, V)^T$ is the depth integrated horizontal velocity. Further physical quantities are the bottom friction coefficient τ_{bf} , the Coriolis coefficient f_c , the gravitational acceleration g and the body Force \mathbf{F} . Their values used in the simulations are summarized in Sect. 5. For the quadrature-free formulation we also need the auxiliary equation $\mathbf{q} = \mathbf{u} H$ with depth averaged velocity $\mathbf{u} = (u, v)^T$. Using the notation $\mathbf{c} := (\xi, U, V)^T$, system (1)–(2) can be written in the following compact form (cf. Faghih-Naini et al. (2020)):

$$\partial_t \mathbf{c} + \nabla \cdot \mathbf{A} = \mathbf{r}, \tag{3}$$

$$\mathbf{u} H = \mathbf{q}, \tag{4}$$

where

$$\mathbf{A}(\mathbf{c}, \mathbf{u}) = \begin{pmatrix} U & V \\ Uu + \frac{g\xi(H+h_b)}{2} & Uv \\ Vu & Vv + \frac{g\xi(H+h_b)}{2} \end{pmatrix} \quad \text{and}$$

$$\mathbf{r}(\mathbf{c}, \mathbf{u}) = \begin{pmatrix} 0 \\ -\tau_{\text{bf}}u + f_c V + g\xi\partial_x h_b + F_x \\ -\tau_{\text{bf}}v - f_c U + g\xi\partial_y h_b + F_y \end{pmatrix}.$$

The examples in this work use the following types of boundary conditions for the SWE:

Land boundary: $\mathbf{q} \cdot \mathbf{n} = 0$.

Open-sea boundary: $\xi = \hat{\xi}$, where $\hat{\xi}$ is a specified water elevation.

Initial conditions for the elevation and velocity are provided as $\xi(\mathbf{x}, 0) = \xi_0(\mathbf{x})$ and $\mathbf{q}(\mathbf{x}, 0) = \mathbf{q}_0(\mathbf{x})$.

Given a triangulation $\Omega = \bigcup \Omega_e$, we obtain the local variational formulation of system (3)–(4) on an element Ω_e by multiplying with sufficiently smooth test functions ϕ and ψ , followed by the integration by parts. We use the notation $(\cdot, \cdot)_{\Omega_e}$ and $\langle \cdot, \cdot \rangle_{\partial\Omega_e}$ for the L^2 -scalar products on elements and edges, respectively, and denote by \mathbf{n}_e an exterior unit normal to $\partial\Omega_e$

$$(\partial_t \mathbf{c}, \phi)_{\Omega_e} - (\mathbf{A}, \nabla \phi)_{\Omega_e} + \langle \mathbf{A} \cdot \mathbf{n}_e, \phi \rangle_{\partial\Omega_e} = (\mathbf{r}, \phi)_{\Omega_e}, \tag{5}$$

$$(\mathbf{u}H, \psi)_{\Omega_e} = (\mathbf{q}, \psi)_{\Omega_e}. \tag{6}$$

Let $\mathbb{P}^p(\Omega_e)$ be the polynomial space of order (i.e., the highest polynomial degree) p on Ω_e . We derive the semi-discrete formulation from (5)–(6) by replacing \mathbf{c} and \mathbf{u} with the discrete solution $\mathbf{c}_\Delta, \mathbf{u}_\Delta$ and utilizing test functions $\phi_\Delta \in \mathbb{P}^p(\Omega_e)^3$, and $\psi_\Delta \in \mathbb{P}^p(\Omega_e)^2$

$$(\partial_t \mathbf{c}_\Delta, \phi_\Delta)_{\Omega_e} - (\mathbf{A}, \nabla \phi_\Delta)_{\Omega_e} + \langle \hat{\mathbf{A}}, \phi_\Delta \rangle_{\partial\Omega_e} = (\mathbf{r}, \phi_\Delta)_{\Omega_e}, \tag{7}$$

$$(\mathbf{u}_\Delta H_\Delta, \psi_\Delta)_{\Omega_e} = (\mathbf{q}_\Delta, \psi_\Delta)_{\Omega_e}. \tag{8}$$

The edge flux $\mathbf{A}(\mathbf{c}_\Delta, \mathbf{u}_\Delta) \cdot \mathbf{n}_e$ is approximated on $\partial\Omega_e$ by a numerical flux $\hat{\mathbf{A}}(\mathbf{c}_\Delta, \mathbf{u}_\Delta, \mathbf{c}_\Delta^+, \mathbf{u}_\Delta^+, \mathbf{n}_e)$ that depends on discontinuous values of the solution on element Ω_e (i.e., $\mathbf{c}_\Delta, \mathbf{u}_\Delta$) and its edge neighbor (i.e., $\mathbf{c}_\Delta^+, \mathbf{u}_\Delta^+$). On exterior domain boundaries, the specified boundary conditions for the elevation or velocity are utilized in the flux computation. In this work, we rely on the Lax–Friedrichs flux (Hajduk et al. 2018) modified (see Faghih-Naini et al. (2020)) for our quadrature-free integration scheme, that is

$$\hat{\mathbf{A}}(\mathbf{c}_\Delta, \mathbf{u}_\Delta, \mathbf{c}_\Delta^+, \mathbf{u}_\Delta^+, \mathbf{n}_e) = \frac{1}{2} \left((\mathbf{A}(\mathbf{c}_\Delta, \mathbf{u}_\Delta) + \mathbf{A}(\mathbf{c}_\Delta^+, \mathbf{u}_\Delta^+)) \cdot \mathbf{n}_e + \lambda(\mathbf{c}_\Delta - \mathbf{c}_\Delta^+) \right),$$

with the following approximation of $\lambda|_E$ for each edge E of Ω_e :

$$\lambda|_E := \max_{\Omega_e: \mathbf{x}_E \in \partial\Omega_e} |\mathbf{u}_\Delta|_{\Omega_e}(\mathbf{x}_E) \cdot \mathbf{n}_e + \max_{\Omega_e: \mathbf{x}_E \in \partial\Omega_e} \sqrt{gH_\Delta|_{\Omega_e}(\mathbf{x}_E)}, \tag{9}$$

where \mathbf{x}_E denotes the midpoint of edge E . The main computational kernels of the above discretizations are the evaluations of the element and edge integrals in (7) and

(8). As in Faghih-Naini et al. (2020), given a basis $\varphi_{ei}(\mathbf{x})$, $i = 1, \dots, K(p)$ of $\mathbb{P}^p(\Omega_e)$, \mathbf{c}_Δ and \mathbf{u}_Δ can be represented as

$$\mathbf{c}_\Delta(t, \mathbf{x})|_{\Omega_e} = (\xi_\Delta, U_\Delta, V_\Delta)^T(t, \mathbf{x}) = \sum_{j=1}^3 \sum_{i=1}^{K(p)} c_{ei}^j \varphi_{ei}(\mathbf{x}) \mathbf{e}_j, \tag{10}$$

$$\mathbf{u}_\Delta(t, \mathbf{x})|_{\Omega_e} = (u_\Delta, v_\Delta)^T(t, \mathbf{x}) = \sum_{j=1}^2 \sum_{i=1}^{K(p)} u_{ei}^j \varphi_{ei}(\mathbf{x}) \mathbf{e}_j \tag{11}$$

with \mathbf{e}_j denoting the j -th unit vector in \mathbb{R}^3 in (10) or \mathbb{R}^2 in (11).

Our implementation employs triangles. The number of basis functions $K(p)$ depends on the chosen polynomial approximation space; it has the following values in \mathbb{R}^2 : $K(0) = 1$, $K(1) = 3$, $K(2) = 6$, and $K(3) = 10$. The basis functions employed in our implementation are hierarchical and orthonormal with respect to the L^2 -scalar product on Ω_e .

The temporal discretization of system (7)–(8) is performed using a strong stability preserving (SSP) Runge–Kutta method (Gottlieb and Shu 1998). In the test cases presented in Sect. 5, we use a two-stage SSP Runge–Kutta method, known as Heun’s scheme, given by

$$\begin{aligned} \mathbf{c}_\Delta^{(0)} &:= \mathbf{c}_\Delta(t_n, \cdot), \\ \mathbf{c}_\Delta^{(1)} &:= \mathbf{c}_\Delta^{(0)} + \Delta_n t \mathbf{L}(t_n, \mathbf{c}_\Delta^{(0)}), \\ \mathbf{c}_\Delta^{(2)} &:= \frac{1}{2} \mathbf{c}_\Delta^{(0)} + \frac{1}{2} \left\{ \mathbf{c}_\Delta^{(1)} + \Delta_n t \mathbf{L}(t_n + \Delta_n t, \mathbf{c}_\Delta^{(1)}) \right\}, \\ \mathbf{c}_\Delta(t_{n+1}, \cdot) &:= \mathbf{c}_\Delta^{(2)}, \end{aligned}$$

where $0 = t_1 < t_2 < \dots < t_{\text{end}}$ is a possibly non-equidistant decomposition of the time interval and $\Delta_n t := t_{n+1} - t_n$ represents the size of the n -th time step. \mathbf{L} denotes the spatial discretization operator specified by (7)–(8).

Our scheme relies on a dynamic p-adaptive algorithm which adjusts and limits, if necessary, the local approximation order using an adaptivity indicator. For DG discretizations which rely on hierarchical bases, p-adaptive (as opposed to h- and hp-adaptive) schemes are particularly attractive due to simplicity of implementation. Numerical results in the literature and our previous studies for p-adaptive schemes, together with the performance measurements in Sect. 5 show savings of computational time compared to non-adaptive schemes of similar accuracy (Kubatko et al. 2009; Faghih-Naini and Aizinger 2022). All our dynamically p-adaptive numerical runs use the Jump-Reconstruction-Limiting (JRL) indicator from Faghih-Naini and Aizinger (2022) with a minor modification—namely in the left branch (p0) in Fig. 1 of Faghih-Naini and Aizinger (2022), the reconstructed limited p1 solution is accepted if its jump is smaller than 100% (instead of smaller than 90%) of the jump of the piecewise constant solution part.

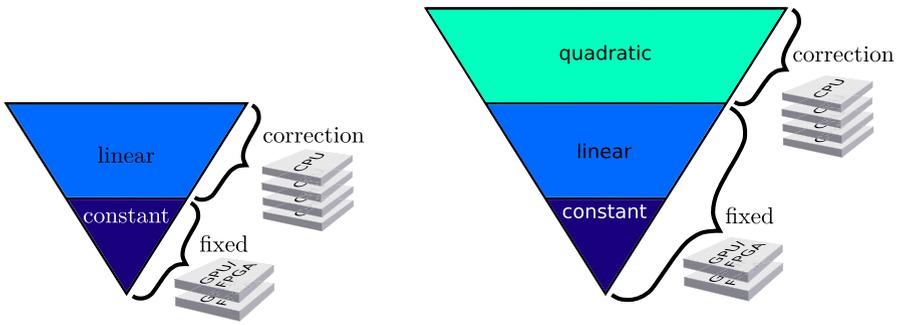


Fig. 1 Schematic idea of separation approach: The solution for the non-adaptive part (left: piecewise constant, right: piecewise linear) is computed for all elements on one hardware, e.g a GPU. An adaptive correction (left: linear, right: quadratic contributions) is then applied to some elements. The latter computation can use a different hardware, e.g., a CPU

3 Algorithmic adaptation: order separation

The quadrature-free formulation utilized in our solver—when combined with hierarchical bases—naturally separates the discrete equations for different polynomial orders. The main idea is to evaluate the update for the non-adaptive degrees of freedom of the DG approximation independently from the adaptive higher-order part of the solution. Since the quadrature-free formulation only contains product-type nonlinearities, a p -adaptive scheme for such a discretization boils down to adding and removing terms without affecting the rest of the DG approximation. The non-adaptive lower-order DG solution is computed for all elements, and a higher-order correction is applied where necessary. When using the quadrature-free formulation, it is technically possible to perform the splitting also for schemes performing quadrature-based integration; however, depending on the quadrature rules, this may not yield any performance benefits.

The setups evaluated in Sect. 5 include a constant non-adaptive part with a linear correction and a linear non-adaptive part with a quadratic correction as shown in Fig. 1. In the latter setup, it is technically possible to fix only the constant part and compute both the linear and quadratic contributions as corrections. However, since the constant part has a very small runtime impact, there is little performance benefit in doing so. Therefore, we focus on fixing the entire linear part. This approach is naturally extendable to any higher order DG discretization.

Our presentation of the separation methodology needs the algebraic representation of element integrals, thus we repeat equation (20) from Faghieh-Naini et al. (2020) (adapted to the notation in this work) and we furthermore follow Faghieh-Naini (2024). Inserting the basis representations (10) and (11) into (7) and testing the first equation with $\phi_\Delta = \varphi_{eq} e_1$ we obtain for $q \in 1, \dots, K(p)$:

$$(\mathbf{A}(\mathbf{c}_\Delta, \mathbf{u}_\Delta), \nabla(\varphi_{eq} e_1))_{\Omega_e} = \sum_{i=1}^{K(p)} \left[c_{ei}^2 \int_{\Omega_e} \frac{\partial \varphi_{eq}}{\partial x} \varphi_{ei} \, d\mathbf{x} + c_{ei}^3 \int_{\Omega_e} \frac{\partial \varphi_{eq}}{\partial y} \varphi_{ei} \, d\mathbf{x} \right]. \tag{12}$$

We assume that up to order b , the computation is done for all elements, and the correction for order $b + 1$ is only applied to some elements. Then the base computation would involve the following:

$$(\mathbf{A}(\mathbf{c}_\Delta, \mathbf{u}_\Delta), \nabla(\varphi_{eq}\mathbf{e}_1))_{\Omega_e}^{base} = \sum_{i=1}^{K(b)} \left[c_{ei}^2 \int_{\Omega_e} \frac{\partial \varphi_{eq}}{\partial x} \varphi_{ei} \mathbf{d}\mathbf{x} + c_{ei}^3 \int_{\Omega_e} \frac{\partial \varphi_{eq}}{\partial y} \varphi_{ei} \mathbf{d}\mathbf{x} \right]$$

for $q \in 1, \dots, K(b)$.

The correction consists of

$$\begin{aligned} & (\mathbf{A}(\mathbf{c}_\Delta, \mathbf{u}_\Delta), \nabla(\varphi_{eq}\mathbf{e}_1))_{\Omega_e}^{correction} \\ &= \sum_{i=K(b)+1}^{K(p)} \left[c_{ei}^2 \int_{\Omega_e} \frac{\partial \varphi_{eq}}{\partial x} \varphi_{ei} \mathbf{d}\mathbf{x} + c_{ei}^3 \int_{\Omega_e} \frac{\partial \varphi_{eq}}{\partial y} \varphi_{ei} \mathbf{d}\mathbf{x} \right] \end{aligned}$$

for $q \in 1, \dots, K(b)$ and

$$\begin{aligned} & (\mathbf{A}(\mathbf{c}_\Delta, \mathbf{u}_\Delta), \nabla(\varphi_{eq}\mathbf{e}_1))_{\Omega_e}^{correction} \\ &= \sum_{i=1}^{K(p)} \left[c_{ei}^2 \int_{\Omega_e} \frac{\partial \varphi_{eq}}{\partial x} \varphi_{ei} \mathbf{d}\mathbf{x} + c_{ei}^3 \int_{\Omega_e} \frac{\partial \varphi_{eq}}{\partial y} \varphi_{ei} \mathbf{d}\mathbf{x} \right] \end{aligned}$$

for $q \in K(b) + 1, \dots, K(p)$.

The edge integrals are separated in a similar manner, of course taking into account the local approximation order of the current element. Note that, when computing the coefficient λ , i.e., (9), in front of the penalty term for the Lax-Friedrichs flux, regardless of the approximation order, exclusively the constant part of the solution is used for evaluation of the velocities and the surface elevation fields. In a similar manner, the nonlinear bottom friction on the right-hand side uses the piecewise constant solution part, instead of using the full-order approximation for computing the velocity magnitude. This approach showed no loss in solution quality and is used in our implementation to avoid special treatment when applying a correction.

The solution $\mathbf{u}_\Delta(t, \mathbf{x})|_{\Omega_e} =: \mathbf{u}_e$ of (8) involves solving an element-local linear system which looks differently for different approximation orders, i.e., the higher-order degrees of freedom can actually affect the lower-order ones. However, using a setup with an analytical solution, we conducted convergence studies to investigate the solution quality when omitting higher-order contributions in the lower-order computations. The studies confirmed that the convergence order and L^2 -differences remain consistent. Furthermore, for the setup utilized in Sect. 5.1, we compare the differences in the surface elevation $\xi - \xi_{orig}$ at the final time in Fig. 2, where ξ_{orig} represents the original scheme. The integrated scaled absolute differences, are 7.03×10^{-5} for p0-1 and 2.97×10^{-5} for p1-2, i.e., very small, as anticipated.

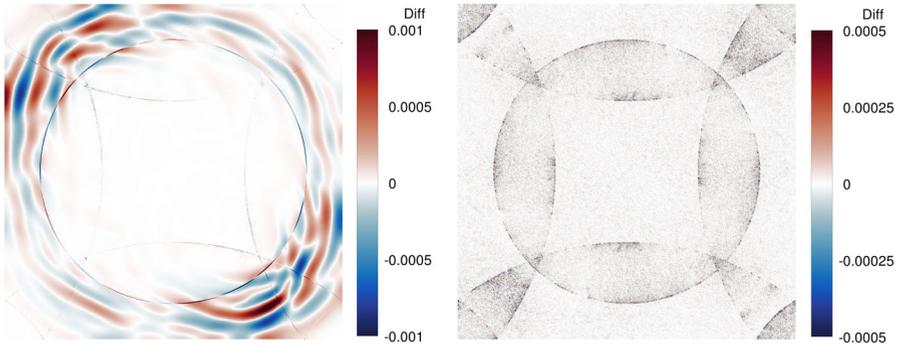


Fig. 2 Radial dam break: difference $\xi - \xi_{orig}$ at $t = 2.5$ s for approximation orders p0-1 (left) and p1-2 (right)

As this approach shows no loss in solution quality, in order to compute \mathbf{u}_e , using an LU-factorization, we are now generally solving

$$(H_{i,j})_{i,j \in \{1, \dots, K(p)\}} (u_j^l)_{j \in \{1, \dots, K(p)\}} = (c_i^{l+1})_{i \in \{1, \dots, K(p)\}} \quad l = 1, 2$$

with

$$\begin{aligned} (H_{i,j})_{i,j \in \{1, \dots, K(p)\}} &:= \int_{\Omega_e} \left(\sum_{n=1}^{K(p,i,j)} c_{en}^1 \varphi_{en} + h_b \right) \varphi_{ej} \varphi_{ei}, \\ (u_j^l)_{j \in \{1, \dots, K(p)\}} &:= u_{ej}^l, \\ (c_i^{l+1})_{i \in \{1, \dots, K(p)\}} &:= \int_{\Omega_e} \sum_{n=1}^{K(p)} c_{en}^{l+1} \varphi_{en} \varphi_{ei}, \quad l = 1, 2 \\ K(p, i, j) &= \begin{cases} 1, & \text{if } i = j = 1 \\ 3, & \text{if } p \leq 1 \wedge (i = j = 1) \\ 6, & \text{if } p \leq 2 \wedge (i \leq 3 \wedge j \leq 3) \\ 10, & \text{if } p \leq 3 \wedge (i \leq 6 \wedge j \leq 6) \end{cases}. \end{aligned}$$

Since the lower-order terms in $(c_i^{l+1})_{i \in \{1, \dots, K(p)\}}$ are not affected by the higher-order ones due to orthonormality property, the lower-order system does not have to be re-assembled when applying the higher-order correction.

4 Code generation

The new separation approach was implemented in GHODDESS¹ (Generation of Higher-Order Discretizations Deployed as ExaSlang Specifications), a Python front-end to the ExaStencils code generation framework (Faghih-Naini et al. 2020; Lengauer et al. 2020; Alt et al. 2023). GHODDESS uses SymPy² (Meurer et al. 2017) to perform symbolic differentiation and integration. It implements the complete program specification, including optimizations such as buffering geometric information.

ExaStencils is a highly advanced framework for generating C++ stencil codes on block-structured grids. The parallelization and communication routines of the code are automatically generated using OpenMP, MPI, and CUDA as backends. For automatic optimizations, ExaStencils supplies code transformations such as common sub-expression elimination, polyhedral loop transformations (Kronawitter and Lengauer 2018), explicit single instruction multiple data (SIMD) vectorization, and address pre-calculation. ExaStencils provides its external domain specific language (DSL) ExaSlang (Lengauer et al. 2020) consisting of four language layers with different abstraction levels. Each layer is designed to provide a tailored language for the different aspects of a problem and its corresponding solution methods. ExaSlang 4 is the most comprehensive layer of our DSL, as it can hold the whole program specification and makes concepts such as parallelization, domain partitioning, and data I/O available to users. For this reason, it was chosen as an intermediate target for the mapping from the symbolic description in GHODDESS. ExaStencils' source-to-source compiler, written in Scala, is then responsible for parsing the ExaSlang input, applying code transformations, and emitting the target C++ code.

For the hardware-driven algorithm design, we extend our code generation pipeline on different abstraction layers. The algorithmic changes to incorporate the separated kernels described in Sect. 3 are done in GHODDESS. The symbolic program description also contains the control flow for distributing individual kernels to specific architecture components. Executing kernels in this heterogeneous manner needs data synchronization and bookkeeping concepts. Automating these rather technical steps can be highly beneficial for productivity making them an ideal target for code generation with ExaStencils.

By default, ExaStencils supplies a standard data migration method for architectures with discrete memory locations for the host and device. Explicit memory transfer statements between the discrete memory locations and data structures for their version tracking are generated. These transfer operations, however, incur large latencies and can impact the execution time significantly when performed at a high frequency. Therefore integrated architectures such as the NVIDIA Jetson systems appear particularly promising for our heterogeneous approach and excel, in addition, in the energy-to-solution metric (Geveler et al. 2016). Since the CPU and the GPU share the same die and the system memory, no distinct memory locations and transfer operations are needed – thus allowing for a low-latency communication between the CPU and the GPU. Supporting the NVIDIA Jetson systems requires specialized extensions within

¹ <https://i10git.cs.fau.de/ocean/ghodde-release>.

² <https://www.sympy.org>.

ExaStencils. We implemented them as distinct building blocks so that all future users can benefit from the new code generation capabilities. The current Jetson systems employ ARM-based CPUs, for which we replenished an existing building block for an automatic SIMD vectorization with NEON intrinsics (Lengauer et al. 2020). The other building blocks are memory management techniques of the CUDA platform, namely pinned memory, unified memory access, and zero-copy memory explained in the following.

- **Pageable** memory is referred to as memory which can be automatically swapped (paged) by the operating system between the primary (usually RAM) and secondary (e.g., external drive) storage. Since GPUs cannot directly access data from pageable host memory, data transfers to and from GPU often incur overhead from internal copy operations to page-locked or pinned host buffers issued by the CUDA runtime. For this purpose, CUDA offers the (de-)allocation of **pinned** host memory to avoid the additional copy and increase the transfer bandwidth.
- **Unified** memory bundles the previously separate host and device memory allocations into a single allocation. It also obviates explicit memory transfers with an automatic on-demand migration determined by the CUDA runtime via a page-fault mechanism. While this model significantly simplifies the development of heterogeneous codes, it often performs poorly due to the overhead from the fault handling. Explicit prefetching of data can mitigate this performance penalty and allows for fine-grained overlapping with kernel executions at the cost of additional code complexity. Still, the complexity can be overcome by generating automated prefetching routines.
- **Zero-copy** memory allows GPU threads to access host memory directly. Users are provided with a shared virtual memory space for host and device data given by mapping the allocated host memory to the CUDA address space. This method is especially beneficial for systems with integrated GPUs such as the NVIDIA Jetson architectures. While this approach does not need explicit migration requests, synchronization between CPU and GPU execution is necessary for critical regions. The required bookkeeping for this purpose is automatically generated by the ExaStencils compiler.

5 Performance results

The performance measurements were carried out on two test platforms. The first one is an NVIDIA Jetson AGX Xavier SoC (called in the following ARM-AGX), which is a part of the ICARUS³ cluster at TU Dortmund, containing an NVIDIA Carmel Armv8.2 CPU with eight cores and an NVIDIA Volta GPU. The CPU's frequency was fixed at 2100 MHz in our test runs. The second test platform is a server with two AMD Epyc 7742 processors with 64 cores each and one NVIDIA Quadro RTX 6000 GPU (called in the following AMD-RTX). There, the CPU frequency was fixed at 2250 MHz. We used OpenMP for the CPU parallelization and chose the number of threads to get similar execution times between the pure CPU and pure GPU versions of

³ <http://www.mathematik.tu-dortmund.de/sites/icarus-green-hpc>.

our code: three threads on the ARM-AGX and 64 threads on the AMD-RTX turned out to produce the best match. For the measurements presented in the following sections, based on our exhaustive testing, we used the fastest memory management techniques for each specific setup. On the ARM-AGX, the pure GPU code was narrowly fastest with pageable memory, the heterogeneous one clearly with zero-copy memory. On the AMD-RTX, pinned memory turned out to be fastest for all code variants.

The main computational kernels in our SWE code are as follows (cf. Figures 7, 8): edge computation (cf. (7) in Sect. 2), element and right-hand-side (RHS) computation (cf. (7) in Sect. 2), auxiliary computation $\mathbf{u}H = \mathbf{q}$ (cf. (8) in Sect. 2), minimum depth control to avoid negative depths, boundary condition (BC) evaluation, the Runge–Kutta step update, and, in dynamically p-adaptive runs, the adaptivity indicator.

We investigate the computational performance using two simulation scenarios. The first one, a radial dam break on a randomly perturbed uniform mesh, was chosen because of the domain simplicity and easy problem customizability. The goal of the second test setup, a tide-driven flow in a realistic domain with a block-structured grid (Faghih-Naini et al. 2023) consisting of several blocks, is to demonstrate the applicability of the new approach for more complex problems. The first test problem is used to evaluate the performance of main code kernels on ARM-AGX, to quantify the effect of separation on the total execution time, and to evaluate which parts of the algorithm are better suited for which types of hardware. In addition, we designed a range of statically adaptive setups with varying fractions of higher-order elements to compare the overhead of using a p-adaptive scheme vs. a higher-order scheme without adaptivity. The same test—carried out on the AMD-RTX—is also employed to illustrate the latency effect of a discrete GPU on the execution time. Finally, the computational performance of a dynamic p-adaptive simulation is evaluated in detail using various configurations (separated, i.e. adaptive and non-adaptive part computed separately, unseparated, i.e. the standard approach, only CPU, only GPU, hybrid CPU and GPU). Here we also specifically study the overhead caused by our separation approach, which mostly boils down to transferring the solution parts between non-adaptive and adaptive kernels. Based on the insights originating from this detailed performance evaluation, we employ the second simulation scenario with the optimal settings to run a more complex test problem on a realistic domain.

Since the main focus of this performance evaluation lies on the novel algorithm, for an in-depth technical analysis of the utilization and memory throughput, which to a large extent evaluates the performance of the ExaStencils framework rather than that of our scheme, we refer to results in the literature. ExaStencils has been successfully employed to accelerate codes from a wide range of applications, including image processing, computational fluid dynamics, and molecular dynamics (Lengauer et al. 2020). In Kuckuk and Köstler (2018), a finite volume SWE scheme on Cartesian structured grids implemented for GPUs was analyzed with a Roofline model showing that the generated kernels achieved approximately 60% of the theoretical and 80% of the measured peak performance. A colored Gauss-Seidel smoother on CPUs was investigated in Kronawitter et al. (2018) demonstrating results exceeding 84% of the Roofline performance based on measured memory bandwidth. Furthermore, in Angersbach et al. (2024), the performance of smoothers within multigrid solvers

Table 1 Overview of physical quantities and their values in simulation setups

Variable name	Unit	Meaning	Value in setup Sect. 5.1	Value in setup Sect. 5.2
τ_{bf}	$\frac{m}{s}$	Bottom friction coefficient	$0.0001 \cdot H$	$0.009 \cdot u $
f_c	$\frac{1}{s}$	Coriolis coefficient	$1.0 \cdot 10^{-5}$	$3.19 \cdot 10^{-5}$
g	$\frac{m}{s^2}$	Gravitational acceleration	1	9.81
F	$\frac{m^2}{s^2}$	Body force (variable atmospheric pressure, tidal potential)	–	–

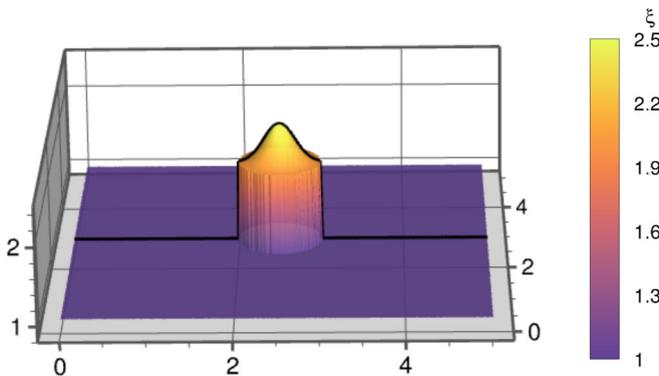


Fig. 3 Radial dam break: Analytic initial condition for surface elevation with slice at $y = 2.5$

was studied on CPUs with the generated kernels, including communication overhead, consistently achieving at least 73% of the maximum measured peak performance.

The values of the physical parameters used in the simulations in this section are listed in Table 1. In Sect. 5.1, the test problem uses a linear friction law, whereas the setup in Sect. 5.2 uses the quadratic one.

5.1 Radial dam break

Here we consider a slightly modified dam break example from Faghih-Naini and Aizinger (2022), which was based in turn on LeVeque (2002); Hajduk (2021). We set $\Omega = [0, 5] \times [0, 5]$ and $g = 1$; however, contrary to Faghih-Naini and Aizinger (2022), a constant bathymetry $h_b = 0.5$ is used. On the exterior boundaries, we impose land boundary conditions and the initial condition (see also Fig. 3) is given by

$$\xi(x, y, t) = \begin{cases} 2 + 0.5 e^{-15((x-2.5)^2+(y-2.5)^2)}, & \text{if } (x - 2.5)^2 + (y - 2.5)^2 < 0.25, \\ 1, & \text{otherwise,} \end{cases}$$

$$U(x, y, t) = 0, \quad V(x, y, t) = 0.$$

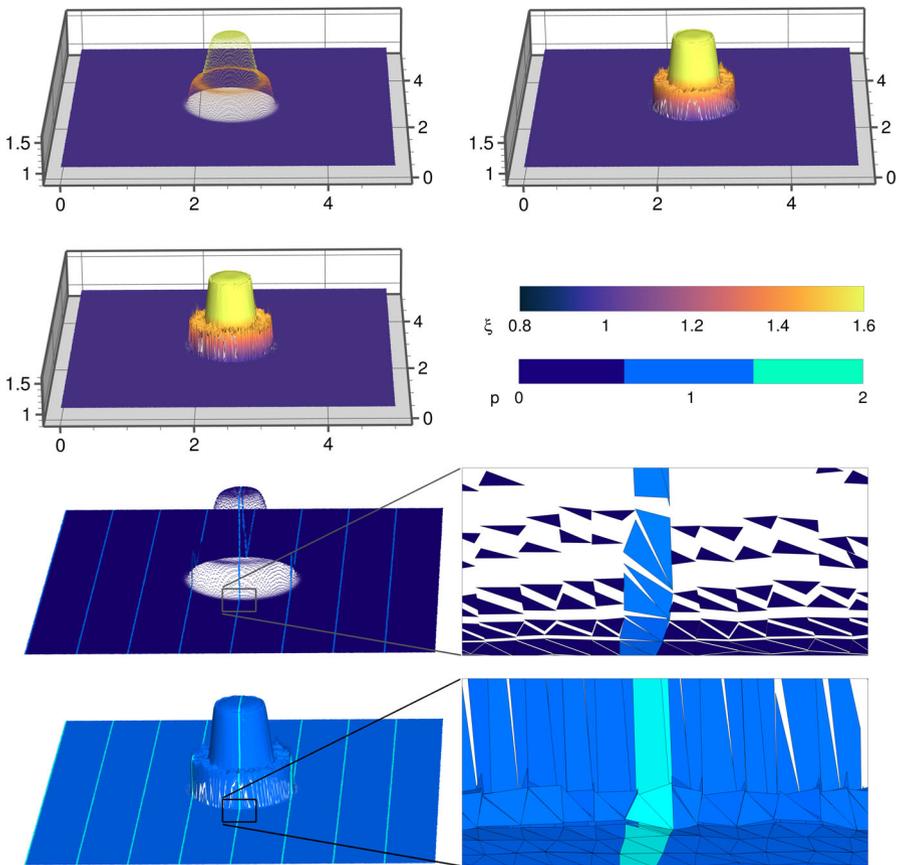


Fig. 4 Radial dam break: Surface elevation at $t = 0.1$ s. Constant (p_0 , top left), linear (p_1 , top right) and quadratic solution (p_2 , second row left). Statically adaptive solution with every 32^{nd} element using the higher approximation order: constant-linear (p_0-1 , third row) and linear-quadratic (p_1-2 , last row), color-coding shows the local approximation order

For all performance measurements, we use a randomly perturbed uniform mesh with 2097 152 triangles. For illustration purposes, Fig. 4 (top two rows) shows the surface elevation at $t = 0.1$ s on a mesh with 131 072 triangles for different approximation orders and Fig. 4 (bottom two rows) the local approximation order for the statically adaptive setup, in which every 32^{nd} element is forced to use a higher order. This particular test was selected because it is challenging for both CPU and GPU to achieve efficient vectorization and memory accesses. In the static setups, we compute 100 time steps with $\Delta t = 0.00001$ s and two substeps (Runge–Kutta stages) each. The execution time is then averaged over the 200 substeps.

The surface elevation and the local approximation order for the dynamic p-adaptive cases are shown in Figs. 5 and 6 at $t = 0.1$ s, $t = 1.0$ s and $t = 2.5$ s. These simulations were run for the total of 12 500 time steps with $\Delta t = 0.0002$ s. Kernel timings were

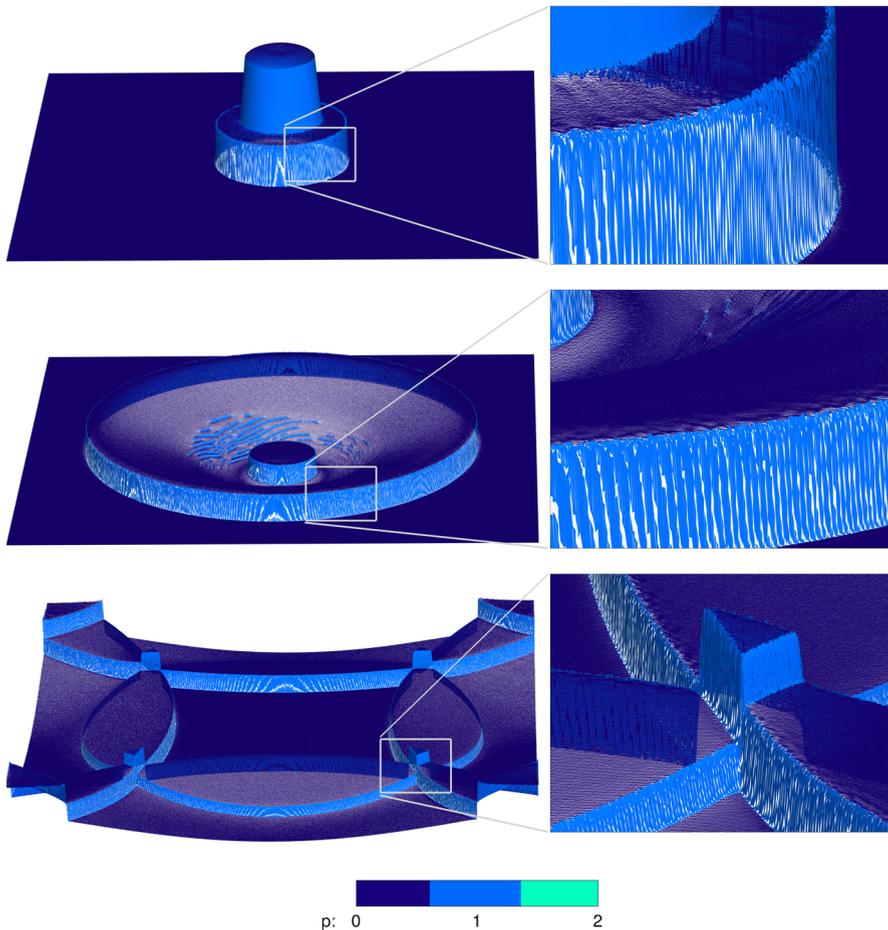


Fig. 5 Radial dam break: dynamic p-adaptive test. Surface elevation at $t = 0.1$ s (top), $t = 1.0$ s (middle) and $t = 2.5$ s (bottom). p0-1 solution with zoom-ins: p1-2. Color-coding shows the local approximation order

averaged over all substeps to capture the variations in the adaptive part of the solution algorithm.

In Fig. 7, we compare the unseparated setups for the piecewise constant, linear, and quadratic solutions without adaptivity to the statically adaptive ones with $1/32$ of the elements fixed at the higher order and to the dynamically p-adaptive results. We clearly see that, for the adaptive setups, the total execution times are much lower than those of the non-adaptive higher-order version.

Next, we turn on our new separation approach and consider the execution times of all kernels on the CPU and GPU (see Fig. 8 "hom." rows). To exploit further parallelism, we utilize these results to distribute the kernels of the separated algorithm between the GPU and CPU (see Fig. 8 "het." rows). The resulting optimal distribution assigns the correction computation to the CPU, whereas the base computation and the

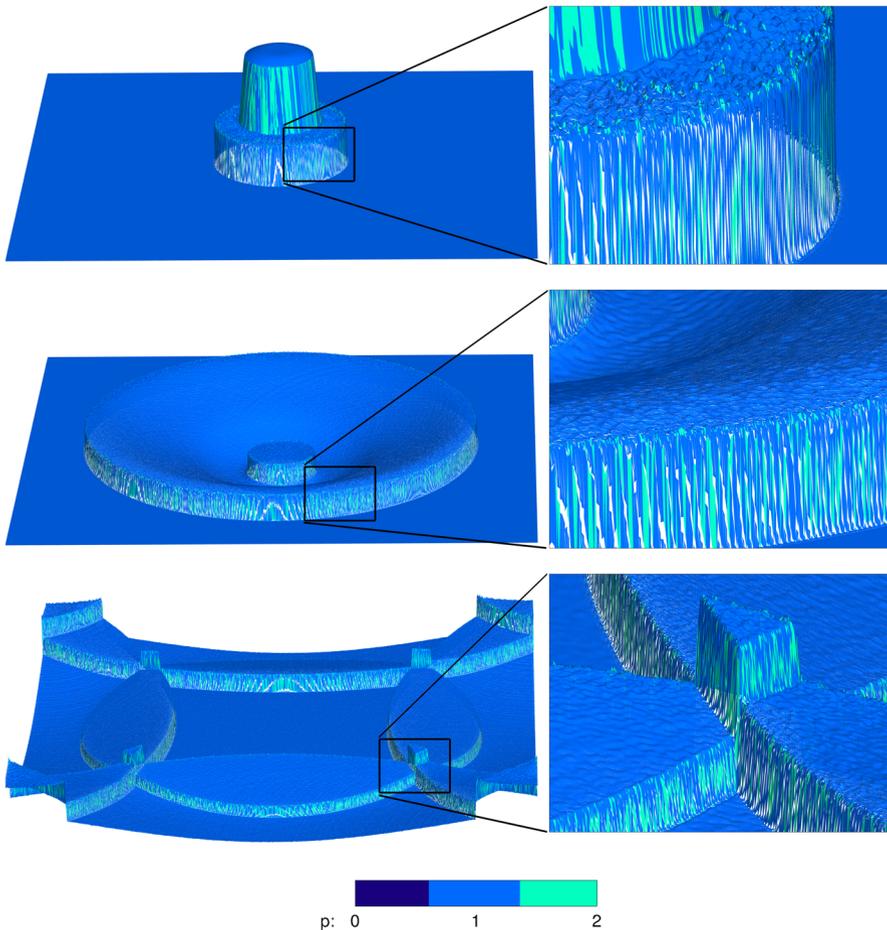


Fig. 6 Radial dam break: dynamic p-adaptive test. Surface elevation at $t = 0.1$ s (top), $t = 1.0$ s (middle) and $t = 2.5$ s (bottom). p1-2 solution with zoom-ins. Color-coding shows the local approximation order

remaining kernels, except for the BC computation, are run on the GPU. This outcome is expected because GPUs generally excel at performing uniform computations across the entire domain, such as the fixed part of the discretization, while CPUs are better suited for non-uniform tasks, such as higher-order corrections. This setup leads to ca. 22% speedup compared to the fastest pure (on either CPU or GPU) separated computation and is ca. 11% faster than the fastest unseparated one.

In Fig. 9, we show the substep execution times for statically adaptive scenarios with different ratios of higher-order elements, which are fixed during the run. Here, we compare the unseparated and separated schemes run on the CPU, the GPU, or with the optimal heterogeneous distribution of kernels between the CPU and GPU. First, the overhead caused by separation is now easy to quantify in each configuration. Furthermore, one can conclude that the CPU clearly outperforms the GPU if ca. 1/32 or more elements use the higher-order approximation, whereas the GPU is faster

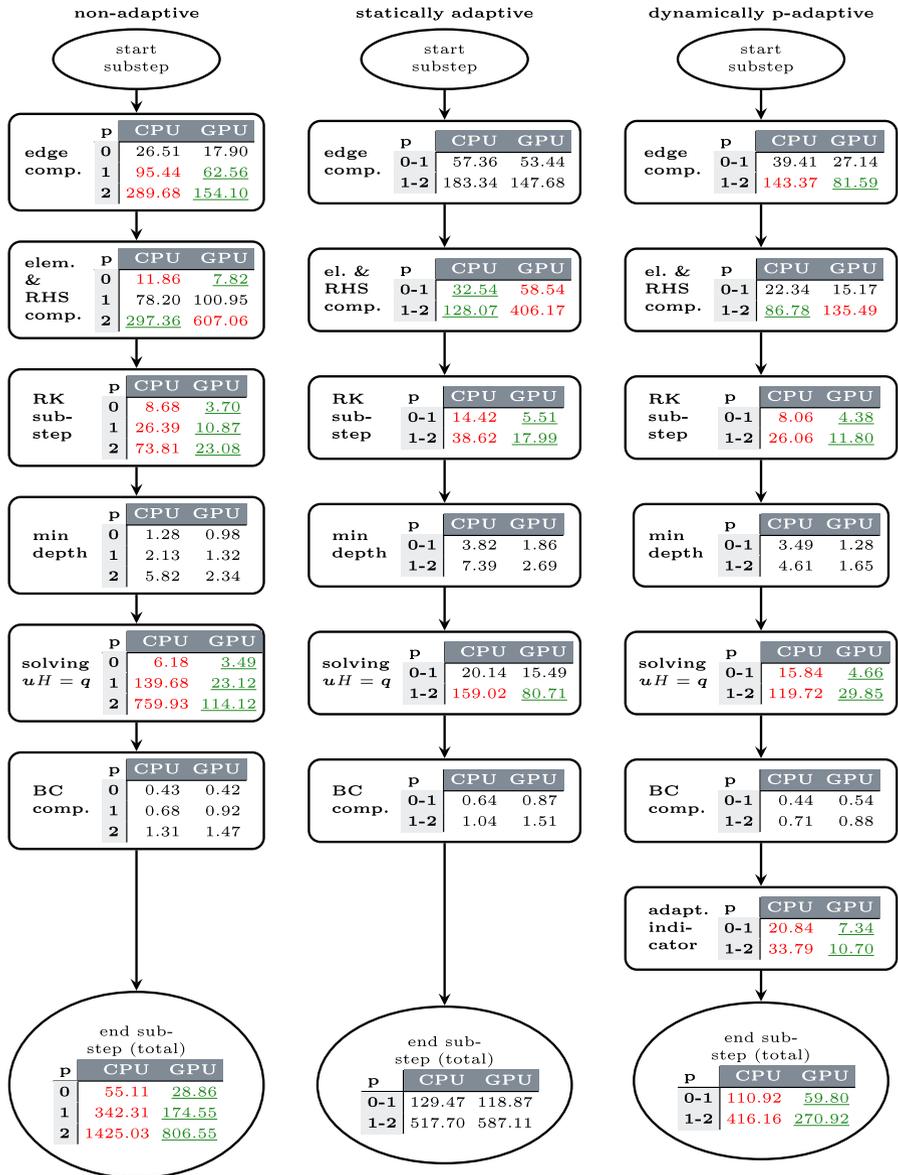


Fig. 7 Radial dam break: Data flow and kernel execution times (in ms) on the ARM-AGX platform for the unseparated setup. Piecewise constant, linear, and quadratic solutions (left), statically adaptive p0-1 and p1-2 solutions (middle), dynamically p-adaptive p0-1 and p1-2 solutions (right). We highlight significantly faster execution times (green, underlined) with a difference of more than 1/3 with respect to the slower ones (red) (color figure online)

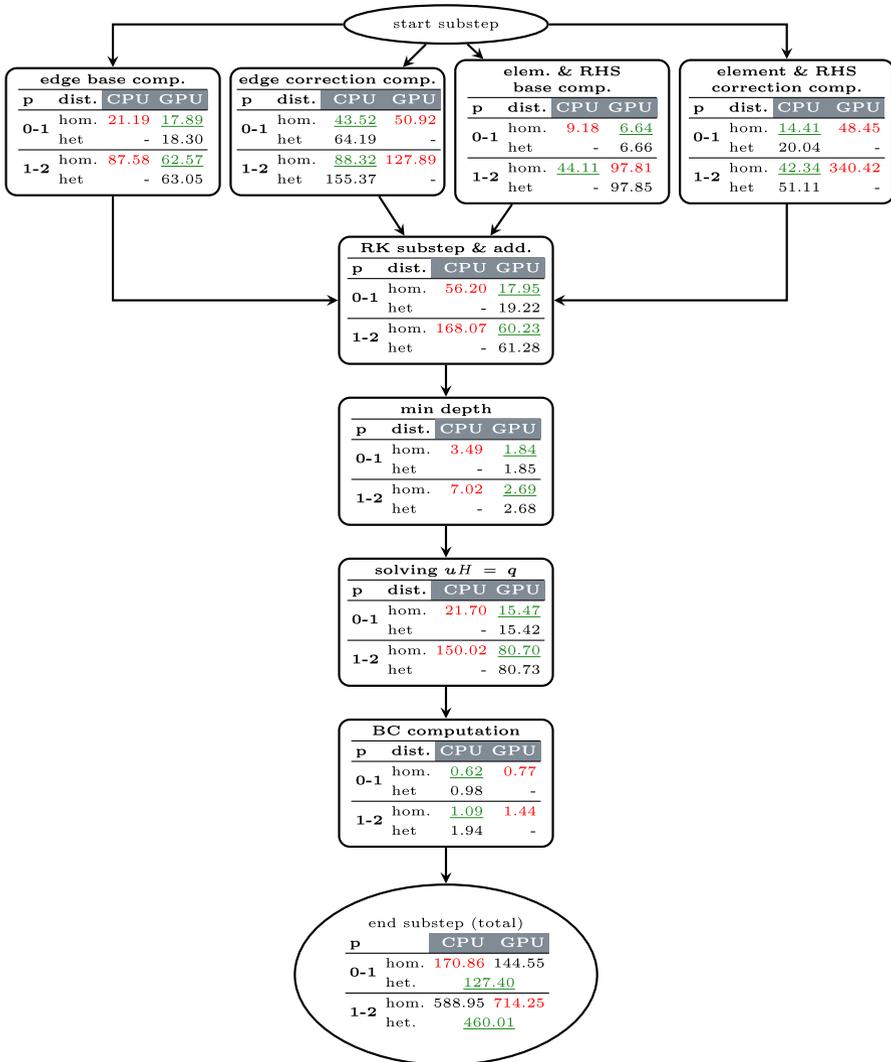


Fig. 8 Radial dam break: Data flow and kernel execution times (in ms) on the ARM-AGX platform for the separated statically adaptive p0-1 and p1-2 solution. The faster and slower execution times are highlighted in green (underlined) and red, respectively, to substantiate the decision on the heterogeneous kernel distribution

if the fraction of higher-order elements is small. Additionally, we observe that all adaptive computations are faster than the corresponding non-adaptive higher-order computations on the CPU. For the non-adaptive GPU execution times, this is also the case as long as 1/32 or fewer elements use the higher order. In the heterogeneous case, for p1-2 for some fractions of higher-order elements (i.e., 1/32 and 1/64), we achieve more than 10% speedup compared to the fastest homogeneous version.

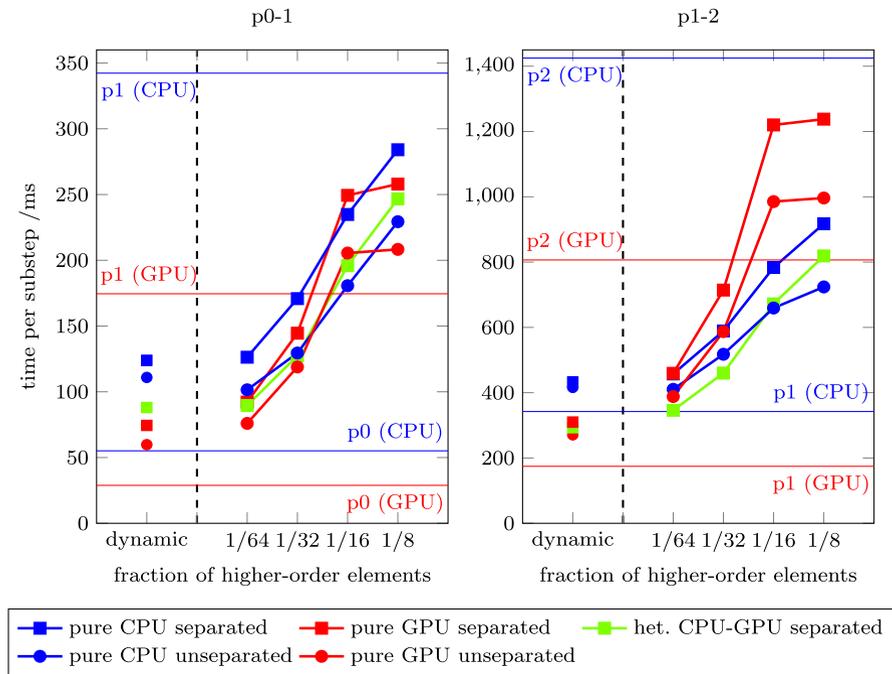


Fig. 9 Radial dam break: ARM-AGX total execution time for non-adaptive, statically adaptive with different fractions of higher-order elements, and dynamically p-adaptive setups. For p0-1, in the latter case, on average (over the whole simulation) ca. 1/482 of the elements use the higher order and, for p1-2, the average fraction of higher-order elements is 1/172. The horizontal lines mark the non-adaptive (p0, p1, and p2) execution times. Constant-linear (left) and linear-quadratic (right) approximation

The values left of the vertical dashed line in Fig. 9 show the substep execution times for the dynamically p-adaptive case (cf., Figs. 5 and 6) with the solution accuracy enforced to be similar to that of the full higher-order solution, cf. Faghih-Naini and Aizinger (2022). For p0-1, on average about 1/482 of the elements use the higher order and for p1-2, the portion is 1/172. Since the dynamically p-adaptive runs are more than twice as fast as the higher-order runs (p1 in Fig. 9 (left) and p2 in Fig. 9 (right)), these measurements confirm the advantages of p-adaptivity in general. The heterogeneous version for p1-2 is faster than the separated homogeneous ones but not faster than the unseparated GPU version. When comparing the p0-1 to the p1-2 versions, one must note that the performance difference between constant and linear computations is smaller than between linear and quadratic computations. Additionally, the overhead caused by separating the element and edge computations is, in some cases, so significant that it cannot be compensated by distributing the kernels and doing the computations in parallel. Here, flexible code generation can be used to the best advantage by easily generating configurations which provide the best performance depending on the problem setup and the hardware configuration.

Achieving efficient heterogeneous kernel distribution is made possible due to the CPU and the GPU sharing the memory on our ARM-AGX SoC architecture. For

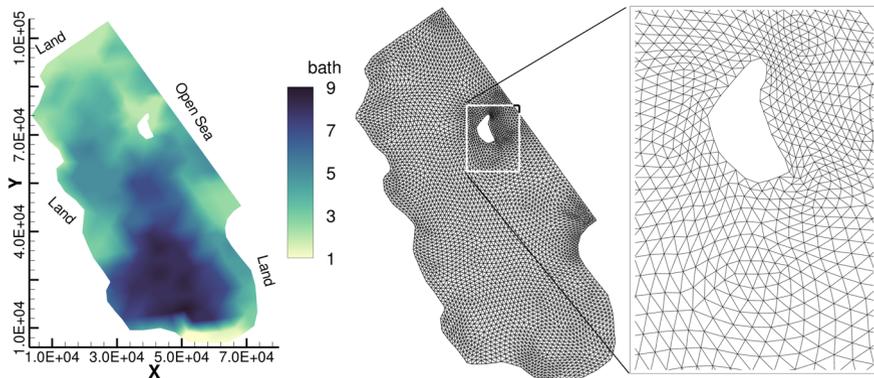


Fig. 10 Tidal flow at Bahamas: Bathymetry (left) and block-structured grid with 256 blocks of 32 elements each. The grid used for the computations was uniformly refined four times, i.e., contains 8192 elements per block. All units are meters

the conventional hardware with a discrete GPU (AMD-RTX in this case), memory transfers between the CPU and GPU are a significant bottleneck difficult to amortize by any performance benefits arising from heterogeneous kernel parallelism. For the separated versions and p0-1, we get reasonable substep execution times of 24.9 ms on the CPU and 16.1 ms on the GPU but 101.0 ms for the heterogeneous setup. This is similar for the p1-2 case where we get 68.1 ms, 72.4 ms, and 222.2 ms, respectively. For detailed execution times on the AMD-RTX architecture, we refer the reader to the last column of Table 2 in Appendix A.

5.2 Tidal flow at Bahamas with water hump

Next, we consider a tide-driven flow scenario in the Bight of Abaco (Bahamas). The simulations were started from the lake-at-rest initial conditions with an added water column of 2 meters height—a prototypical tsunami simulation without wetting and drying—and run for 50 min driven by the tidal surface elevation at the open sea boundary. We imposed no normal flow boundary conditions at the land boundaries (see Faghih-Naini et al. (2020) for more details on the tidal problem setup). Figure 10 shows the bathymetry (left) and the block-structured grid (BSG). The displayed BSG contains 256 blocks with only 32 elements each for better visualization. In the computations, a four times uniformly refined (via bisecting each edge) BSG with 8192 elements per block was used, that is, the total number of elements was the same as in the uniform dam break examples. The simulations were run for 12000 time steps with $\Delta t = 0.25$ s.

Figure 11 illustrates the surface elevation at different times for the constant-linear approximation (top) along with the corresponding local approximation orders for the constant-linear (middle) and the linear-quadratic (bottom) discretization. In the p0-1 case, about 24.5% of the elements use order 1, and, in the p1-2 case, about 12.1% of the elements use order 2.

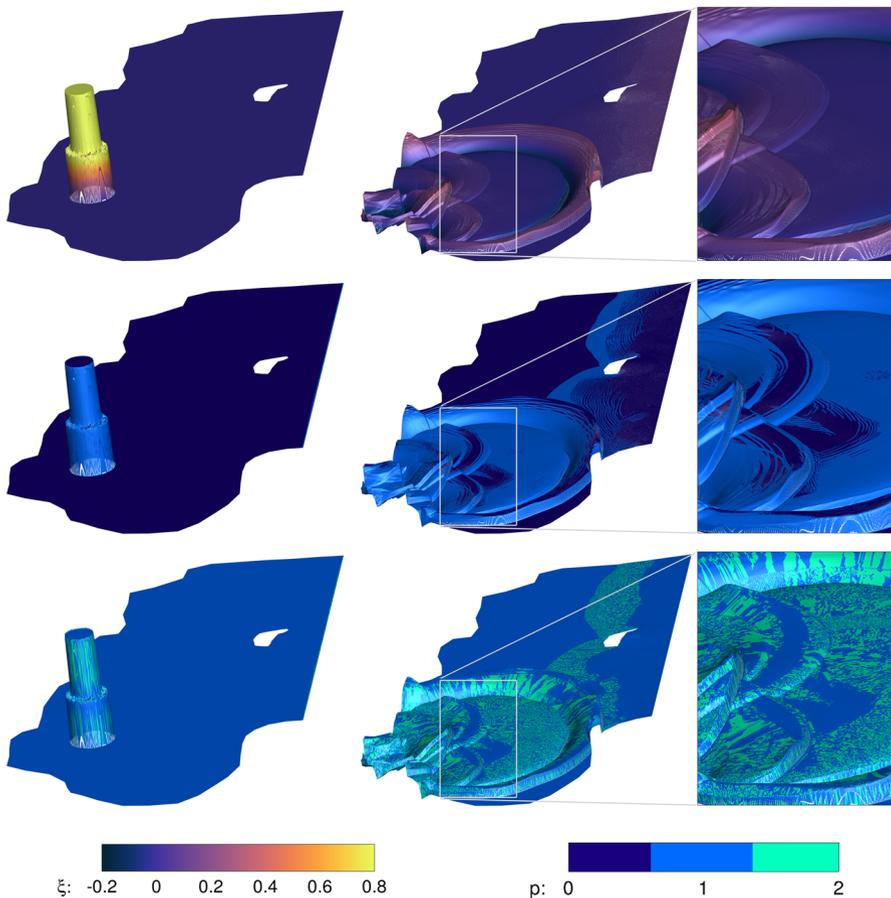


Fig. 11 Tidal flow at Bahamas: surface elevation (top row) and local approximation orders for p0-1 (middle row) and p1-2 (bottom row) at $t = 1$ s (left) and $t = 50$ s (right with zoom-in). The z-axis is scaled up by factor 10000

Figure 12 details the kernel execution times for the unseparated and separated setups on the CPU, the GPU, as well as for the heterogeneous kernel distribution. In the heterogeneous case, the CPU-kernels are on the left and the GPU ones are on the right of the corresponding bar. The total execution time is larger than the individual ones because not all kernels can be overlapped due to data dependencies. We use the heterogeneous kernel distribution derived in the previous section, i.e., the correction computations are performed on the CPU, whereas the base computations and the remaining kernels, except for the BC computation, are done on the GPU. Therefore,

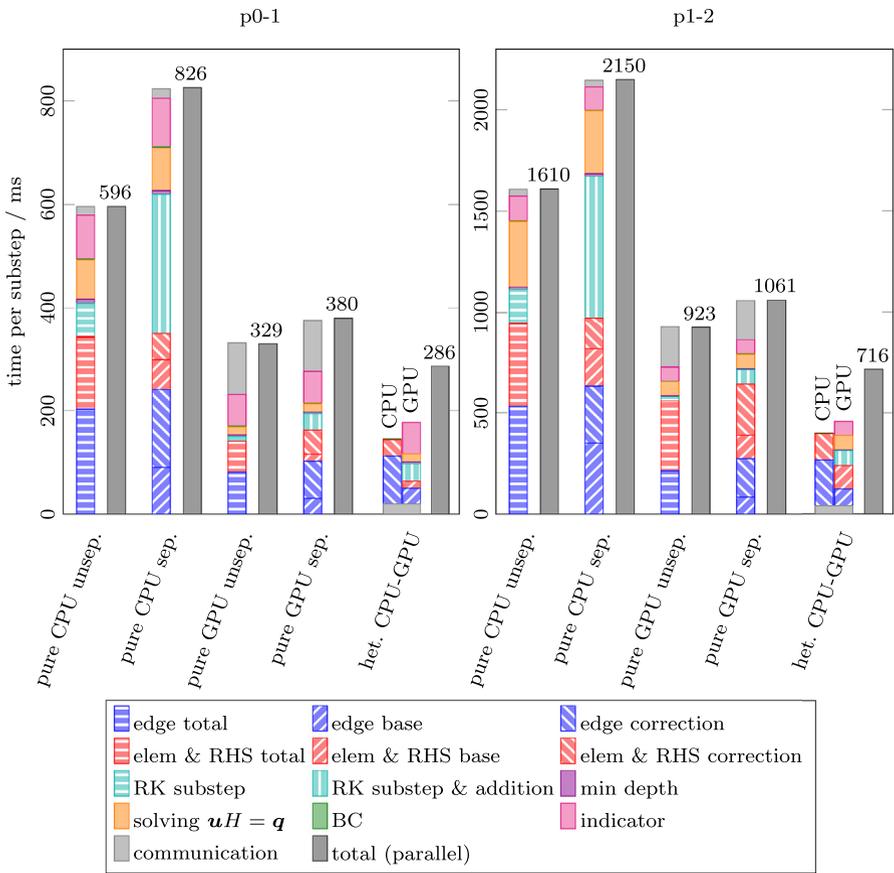


Fig. 12 Tidal flow at Bahamas: detailed kernel and total execution times for dynamically p-adaptive simulation with unseparated and separated setup on the CPU and the GPU as well as with the optimal heterogeneous distribution

the heterogeneous bar (the rightmost bar of the corresponding subplots of Fig. 12) mostly consists of faster kernels (i.e., smaller blocks) out of separated CPU and GPU execution times plotted in the corresponding bars. When using the CPU and the GPU in parallel, we obtain ca. 13% speedup for the constant-linear approximation and ca. 22% speedup for the linear-quadratic one. These results show that our approach also works well in realistic simulation scenarios.

6 Conclusions and outlook

In this work, we proposed and tested a specially re-designed quadrature-free p-adaptive discontinuous Galerkin scheme for the shallow water equations. Using a hierarchical modal basis, our approach separates the lower-order degrees of freedom computations from the rest of the discretization. Furthermore, by exploiting automatic code gener-

ation, we distribute the computational kernels between the CPU and the GPU based on kernel performance evaluation for specific hardware. Performance measurements demonstrate that this approach can lead to significant performance improvements for certain simulation scenarios if used on a hardware where the CPU and the GPU share memory.

Since integrated architectures such as the SoC used as a test platform in our work are often also particularly energy efficient, this is a promising approach for the future HPC applications.

In future work, developing a detailed performance model could prove to be a worthwhile endeavor. While measuring the runtime contributions of individual kernels for different discretization spaces provides valuable insights for practical application tuning, a performance model would help to guide this optimization process. Setting up such a performance model is challenging due to the complexity of kernels, particularly for higher orders or intricate operations like indicator evaluation. This complexity also limits the effectiveness of automated tools. After establishing an execution model for each kernel, these models must then be mapped to hardware models for both CPU and GPU, including their interconnects and, if relevant, shared memory spaces. Ideally, the models should also encompass other factors such as CPU and GPU caching, extending beyond the scope of individual kernel models. Existing research on CPU execution modeling is substantial, but literature on GPU and combined CPU–GPU models is sparse. Additionally, it is crucial to account for the impact of variations that occur at execution time, such as changes in the adapted elements, by quantifying these effects and adjusting the tuning approach accordingly.

A further improvement of our p-adaptive approach may include an online performance measurement system which, e.g., could evaluate the kernel execution times at certain time points during the simulation run and automatically re-distribute the kernels as needed. Also porting our implementation to other types of SoC (e.g., integrated Intel GPUs or the NVIDIA Grace Hopper Superchip) and comparing its performance in the energy-to-solution metric to traditional CPU and GPU realizations of the same numerical scheme could generate interesting insights.

Appendix A Detailed performance results

The kernel execution times for all presented adaptive measurements are detailed in Table 2.

Table 2 Detailed kernel execution times (in ms) for different scenarios. The partial execution times were measured without overlap, i.e., with synchronization after the kernel calls and therefore their sums do not not always match the total execution times

test scenario		dam break static, 1/8 ARM-AGX		dam break static, 1/16 ARM-AGX		dam break static, 1/32 ARM-AGX		dam break static, 1/64 ARM-AGX		dam break dynamic ARM-AGX		Bahamas dynamic ARM-AGX		dam break static, 1/32 AMD-RTX		
kernel	adaptivity strategy hardware platform	CPU	GPU	CPU	GPU	CPU	GPU	CPU	GPU	CPU	GPU	CPU	GPU	CPU	GPU	
edge base	0-1	homog.	21.6	17.6	21.6	17.9	21.2	17.9	21.4	17.6	20.7	17.9	90.4	30.6	4.1	1.7
		heterog.	—	18.3	—	18.3	—	18.3	—	18.3	—	18.3	—	29.8	—	8.4
	1-2	homog.	89.1	62.6	85.5	62.6	87.6	62.6	90.9	62.2	83.7	61.7	351.3	83.8	14.7	5.8
		heterog.	—	63.1	—	63.1	—	63.1	—	63.1	—	63.1	—	81.5	—	23.1
edge correction	0-1	homog.	86.0	99.2	69.9	98.7	43.5	50.9	24.6	26.7	16.2	13.9	150.1	71.6	4.3	4.8
		heterog.	140.6	—	109.7	—	64.2	—	37.3	—	32.5	—	92.1	—	9.5	—
	1-2	homog.	196.0	254.0	161.4	251.9	88.3	127.9	48.7	65.5	20.5	23.5	279.9	189.5	12.3	12.0
		heterog.	411.0	—	281.7	—	155.4	—	83.4	—	38.3	—	225.4	—	22.9	—
elem & RHS base	0-1	homog.	9.2	6.6	9.2	6.6	9.2	6.6	9.3	6.6	8.1	6.6	57.6	13.6	2.0	0.8
		heterog.	—	6.7	—	6.7	—	6.7	—	6.7	—	6.7	—	13.8	—	0.8
	1-2	homog.	43.7	97.8	43.9	97.8	44.1	97.8	44.0	97.8	42.1	97.8	185.5	117.5	5.0	9.0
		heterog.	—	97.9	—	97.9	—	97.9	—	97.9	—	98.0	—	117.1	—	9.1
elem & RHS correction	0-1	homog.	26.3	96.2	21.9	96.2	14.4	48.5	8.3	24.5	6.0	7.2	51.4	46.6	1.8	4.5
		heterog.	42.0	—	30.0	—	20.0	—	13.7	—	11.5	—	31.7	—	1.8	—
	1-2	homog.	105.3	680.5	75.7	679.7	42.3	340.4	22.5	170.7	8.8	36.9	155.8	251.8	3.7	31.5
		heterog.	127.5	—	92.8	—	51.1	—	29.3	—	12.5	—	131.8	—	3.9	—
RK substep & addition	0-1	homog.	97.7	32.1	76.6	25.3	56.2	18.0	42.3	14.1	31.7	16.3	270.4	32.2	9.9	4.3
		heterog.	—	32.8	—	26.2	—	19.2	—	15.5	—	17.7	—	34.1	—	59.4
	1-2	homog.	259.7	96.5	211.3	85.6	168.1	60.2	139.3	45.0	123.8	48.6	701.8	71.2	23.1	11.6
		heterog.	—	95.9	—	86.1	—	61.3	—	46.4	—	50.2	—	74.2	—	118.1
min depth	0-1	homog.	3.6	2.5	3.7	2.5	3.5	1.8	3.5	1.5	2.9	1.3	7.1	3.1	0.5	0.2
		heterog.	—	2.5	—	2.5	—	1.9	—	1.5	—	1.3	—	3.1	—	0.2
	1-2	homog.	8.9	3.8	8.1	3.8	7.0	2.7	5.6	2.1	4.7	1.7	11.0	4.0	0.9	0.3
		heterog.	—	3.8	—	3.8	—	2.7	—	2.1	—	1.7	—	4.0	—	0.3
solving $uH = q$	0-1	homog.	38.6	26.8	30.9	26.9	21.7	15.5	15.6	9.7	16.4	4.7	82.5	15.7	1.0	1.9
		heterog.	—	26.8	—	26.9	—	15.4	—	9.7	—	4.8	—	15.6	—	4.7
	1-2	homog.	213.8	143.6	196.4	138.1	150.0	80.7	105.0	52.3	112.5	29.9	311.4	71.3	5.8	11.7
		heterog.	—	143.2	—	138.3	—	80.7	—	52.5	—	30.0	—	71.3	—	14.5
BC	0-1	homog.	0.6	0.4	0.6	0.8	0.6	0.8	0.6	0.4	0.4	0.5	1.6	1.5	0.9	0.1
		heterog.	1.0	—	1.0	—	1.0	—	1.0	—	0.8	—	1.8	—	12.6	—
	1-2	homog.	1.2	1.5	1.2	1.5	1.1	1.4	1.1	0.8	0.7	0.9	2.3	1.8	1.6	0.2
		heterog.	2.2	—	2.1	—	1.9	—	1.9	—	1.2	—	2.5	—	25.6	—
indicator	0-1	homog.	—	—	—	—	—	—	—	—	—	21.1	7.4	93.9	61.2	—
		heterog.	—	—	—	—	—	—	—	—	—	—	8.1	—	60.5	—
	1-2	homog.	—	—	—	—	—	—	—	—	—	34.7	10.7	113.9	69.2	—
		heterog.	—	—	—	—	—	—	—	—	—	—	11.3	—	68.9	—
total (parallel)	0-1	homog.	284.1	258.0	234.8	249.4	170.9	144.6	126.3	92.2	123.9	74.4	826.2	379.7	24.9	16.1
		heterog.	—	246.7	—	195.9	—	127.4	—	89.6	—	88.1	—	286.5	—	101.0
	1-2	homog.	917.5	1237.8	783.5	1220.2	589.0	714.3	457.5	459.3	432.6	310.2	2149.5	1061.0	68.1	72.4
		heterog.	—	818.8	—	672.2	—	460.1	—	346.1	—	292.7	—	716.1	—	222.2

Acknowledgements The authors gratefully acknowledge the scientific support and HPC resources provided by the Erlangen National High Performance Computing Center (NHR@FAU) of the Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), in particular support from Dominik Ernst and Jan Laukemann. The hardware is funded by the German Research Foundation (DFG). The authors are very grateful to Prof. Stefan Turek for granting access to the ICARUS cluster at the TU Dortmund and to Markus Geveler and Dominik Mütter for technical support on this cluster. ICARUS hardware is financed by MIWF NRW under the lead of MERCUR. We also thank Dinesh Parthasarathy for the initial setup of the performance measurement script. The work in this paper was supported in part by the DFG through grant AI 117/6-1 ‘Performance optimized software strategies for unstructured-mesh applications in ocean modeling’.

Funding Open Access funding enabled and organized by Projekt DEAL.

Data Availability Data used in this work can be made available on request.

Declarations

Conflict of interest The authors declare that they have no Conflict of interest.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give

appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit <http://creativecommons.org/licenses/by/4.0/>.

References

- Aizinger, V., Dawson, C.: A discontinuous Galerkin method for two-dimensional flow and transport in shallow water. *Adv. Water Resour.* **25**(1), 67–84 (2002). [https://doi.org/10.1016/S0309-1708\(01\)00019-7](https://doi.org/10.1016/S0309-1708(01)00019-7)
- Alt, C., Kenter, T., Faghih-Naini, S., Faj, J., Opdenhövel, J.O., Plessl, C., Aizinger, V., Höning, J., Köstler, H.: Shallow water DG simulations on FPGAs: design and comparison of a novel code generation pipeline. In: Bhatele, A., Hammond, J., Baboulin, M., Kruse, C. (eds.) *High Performance Computing*, pp. 86–105. Springer Nature Switzerland, Cham (2023)
- Angersbach, R., Kuckuk, S., Köstler, H.: Code generation for octree-based multigrid solvers with fused higher-order interpolation and communication. In: Carretero, J., Shende, S., Garcia-Blas, J., Brandic, I., Olcoz, K., Schreiber, M. (eds.) *Euro-Par 2024: Parallel Processing*, pp. 240–254. Springer Nature Switzerland, Cham (2024)
- Baiges, J., Bayona, C.: Refficientlib: an efficient load-rebalanced adaptive mesh refinement algorithm for high-performance computational physics meshes. *SIAM J. Sci. Comput.* **39**(2), C65–C95 (2017). <https://doi.org/10.1137/15M105330X>
- Biswas, R., Das, S.K., Harvey, D., Olikar, L.: Parallel dynamic load balancing strategies for adaptive irregular applications. *Appl. Math. Model.* **25**(2), 109–122 (2000). [https://doi.org/10.1016/S0307-904X\(00\)00040-8](https://doi.org/10.1016/S0307-904X(00)00040-8)
- Bosilca, G., Bouteiller, A., Danalis, A., Faverge, M., Hault, T., Dongarra, J.: PaRSEC: exploiting heterogeneity to enhance scalability. *Comput. Sci. Eng.* **15**(6), 36–45 (2013). <https://doi.org/10.1109/MCSE.2013.98>
- Bramas, B., Helluy, P., Mendoza, L., Weber, B.: Optimization of a discontinuous Galerkin solver with OpenCL and StarPU. *Int. J. Finite Vol.* **15**(1), 1–19 (2020). <https://doi.org/10.1007/s11589-013-0047-7>
- Chaplygin, A., Gusev, A., Diansky, N.: High-performance shallow water model for use on massively parallel and heterogeneous computing systems. *Supercomput. Front. Innov.* (2022). <https://doi.org/10.14529/jsfi210407>
- Echeverribar, I., Morales-Hernández, M., Brufau, P., García-Navarro, P.: Analysis of the performance of a hybrid CPU/GPU 1D2D coupled model for real flood cases. *J. Hydroinform.* **22**(5), 1198–1216 (2020). <https://doi.org/10.2166/hydro.2020.032>
- Faghih-Naini, S.: Discontinuous Galerkin methods and high performance computing approaches for ocean simulations (2024). https://doi.org/10.15495/EPub_UBT_00007728
- Faghih-Naini, S., Aizinger, V.: p-adaptive discontinuous Galerkin method for the shallow water equations with a parameter-free error indicator. *Int. J. Geomath.* (2022). <https://doi.org/10.1007/s13137-022-00208-3>
- Faghih-Naini, S., Kuckuk, S., Aizinger, V., Zint, D., Grosso, R., Köstler, H.: Quadrature-free discontinuous Galerkin method with code generation features for shallow water equations on automatically generated block-structured meshes. *Adv. Water Resour.* **138**, 103552 (2020). <https://doi.org/10.1016/j.advwatres.2020.103552>
- Faghih-Naini, S., Kuckuk, S., Zint, D., Kemmler, S., Köstler, H., Aizinger, V.: Discontinuous Galerkin method for the shallow water equations on complex domains using masked block-structured grids. *Adv. Water Resour.* **182**, 104584 (2023). <https://doi.org/10.1016/j.advwatres.2023.104584>
- Flynn, M.: Some computer organizations and their effectiveness. *IEEE Trans. Comput.* **21**, 948–960 (1972). <https://doi.org/10.1109/TC.1972.5009071>
- Fu, H., Gan, L., Yang, C., Xue, W., Wang, L., Wang, X., Huang, X., Yang, G.: Solving global shallow water equations on heterogeneous supercomputers. *PLOS ONE* **12**, e0172583 (2017). <https://doi.org/10.1371/journal.pone.0172583>

- Garcia-Gasulla, M., Houzeaux, G., Ferrer, R., Artigues, A., López, V., Labarta, J., Vázquez, M.: MPI+X: task-based parallelisation and dynamic load balance of finite element assembly. *Int. J. Comput. Fluid Dyn.* **33**(3), 115–136 (2019). <https://doi.org/10.1080/10618562.2019.1617856>
- Geveler, M., Reuter, B., Aizinger, V., Göddeke, D., Turek, S.: Energy efficiency of the simulation of three-dimensional coastal ocean circulation on modern commodity and mobile processors. *Comput. Sci. Res. Dev.* **31**(4), 225–234 (2016). <https://doi.org/10.1007/s00450-016-0324-5>
- Gottlieb, S., Shu, C.W.: Strong stability-preserving high-order time discretization methods. *Math. Comput.* **67**(221), 73–85 (1998). <https://doi.org/10.1090/S0025-5718-98-00913-2>
- Hajduk, H.: Monolithic convex limiting in discontinuous Galerkin discretizations of hyperbolic conservation laws. *Compu. Math. Appl.* **87**, 120–138 (2021). <https://doi.org/10.1016/j.camwa.2021.02.012>
- Hajduk, H., Hodges, B.R., Aizinger, V., Reuter, B.: Locally filtered transport for computational efficiency in multi-component advection-reaction models. *Environ. Model. Softw.* **102**, 185–198 (2018). <https://doi.org/10.1016/j.envsoft.2018.01.003>
- Hendrickson, B., Devine, K.: Dynamic load balancing in computational mechanics. *Comput. Methods Appl. Mech. Eng.* **184**(2), 485–500 (2000). [https://doi.org/10.1016/S0045-7825\(99\)00241-8](https://doi.org/10.1016/S0045-7825(99)00241-8)
- Kronawitter, S., Kuckuk, S., Köstler, H., Lengauer, C.: Automatic data layout transformations in the ExaStencils code generator. *Parallel Process. Lett.* **28**(03), 1850009 (2018). <https://doi.org/10.1142/S0129626418500093>
- Kronawitter, S., Lengauer, C.: Polyhedral search space exploration in the ExaStencils code generator. *ACM Trans. Archit. Code Optim.* (2018). <https://doi.org/10.1145/3274653>
- Kubatko, E.J., Bunya, S., Dawson, C., Westerink, J.J.: Dynamic p-adaptive Runge-Kutta discontinuous Galerkin methods for the shallow water equations. *Computer Methods in Applied Mechanics and Engineering* **198**(21), 1766–1774 (2009). *Advances in Simulation-Based Engineering Sciences - Honoring J. Tinsley Oden.* <https://doi.org/10.1016/j.cma.2009.01.007>
- Kuckuk, S., Köstler, H.: Whole program generation of massively parallel shallow water equation solvers. In: 2018 IEEE International Conference on Cluster Computing (CLUSTER), pp. 78–87. (2018). <https://doi.org/10.1109/CLUSTER.2018.00020>
- Lengauer, C., Apel, S., Bolten, M., Chiba, S., Rüde, U., Teich, J., Größlinger, A., Hannig, F., Köstler, H., Claus, L., Grebhahn, A., Groth, S., Kronawitter, S., Kuckuk, S., Rittich, H., Schmitt, C., Schmitt, J.: Exastencils: advanced multigrid solver generation. In: Bungartz, H.J., Reiz, S., Uekermann, B., Neumann, P., Nagel, W.E. (eds.) *Software for Exascale Computing - SPPEXA 2016–2019*, pp. 405–452. Springer International Publishing, Cham (2020)
- LeVeque, R.J.: *Finite Volume Methods for Hyperbolic Problems*; Cambridge Texts in Applied Mathematics. Cambridge University Press, Cambridge (2002)
- Meurer, A., Smith, C.P., Paprocki, M., Čertík, O., Kirpichev, S.B., Rocklin, M., Kumar, A., Ivanov, S., Moore, J.K., Singh, S., Rathnayake, T., Vig, S., Granger, B.E., Muller, R.P., Bonazzi, F., Gupta, H., Vats, S., Johansson, F., Pedregosa, F., Curry, M.J., Terrel, A.R., Roučka, v., Saboo, A., Fernando, I., Kulal, S., Cimrman, R., Scopatz, A.: SymPy: symbolic computing in Python. *PeerJ Computer Science*, **3**, e103 (2017). <https://doi.org/10.7717/peerj-cs.103>
- Mu, D., Chen, P., Wang, L.: Accelerating the discontinuous Galerkin method for seismic wave propagation simulations using multiple GPUs with CUDA and MPI. *Earthq. Sci.* **26**(6), 377–393 (2013). <https://doi.org/10.1007/s11589-013-0047-7>
- Teresco, J.D., Devine, K.D., Flaherty, J.E.: Partitioning and dynamic load balancing for the numerical solution of partial differential equations. In: Bruaset, A.M., Tveito, A. (eds.) *Numerical Solution of Partial Differential Equations on Parallel Computers*, pp. 55–88. Springer, Berlin Heidelberg, Berlin, Heidelberg (2006)
- Witherden, F., Vermeire, B., Vincent, P.: Heterogeneous computing on mixed unstructured grids with PyFR. *Comput. Fluids* **120**, 173–186 (2015). <https://doi.org/10.1016/j.compfluid.2015.07.016>