Towards Portability at Scale: A Cross-Architecture Performance Evaluation of a GPU-enabled Shallow Water Solver

Johansell Villalobos* - jovillalobos@cenat.ac.cr
Daniel Caviedes-Voullième†‡ - d.caviedes.voullieme@fz-juelich.de
Silvio Rizzi§ - srizzi@anl.gov
Esteban Meneses*¶ - emeneses@cenat.ac.cr

*National High Technology Center, San José, Costa Rica
†Simulation and Data Lab. Terrestrial Systems, Jülich Supercomputing Center
‡Institute for Bio- and Geosciences: Agrosphere (IBG-3), Forschungzentrum Jülich, Germany
§Argonne National Laboratory, Lemont, Illinois, US
¶, Costa Rica Technological Institute, Cartago, Costa Rica

***Abstract*—Current climate change has posed a grand challenge in the field of numerical modeling due to its complex, multiscale dynamics. In hydrological modeling, the increasing demand for high-resolution, real-time simulations has led to the adoption of GPU-accelerated platforms and performance portable programming frameworks such as Kokkos. In this work, we present a comprehensive performance study of the SERGHEI-SWE solver, a shallow water equations code, across four state-of-the-art heterogeneous HPC systems: Frontier (AMD MI250X), JUWELS Booster (NVIDIA A100), JEDI (NVIDIA H100), and Aurora (Intel Max 1550). We assess strong scaling up to 1024 GPUs and weak scaling upwards of 2048 GPUs, demonstrating consistent scalability with a speedup of 32 and an efficiency upwards of 90% for most almost all the test range. Roofline analysis reveals that memory bandwidth is the dominant performance bottleneck, with key solver kernels residing in the memory-bound region. To evaluate performance portability, we apply both harmonic and arithmetic mean-based metrics while varying problem size. Results indicate that while SERGHEI-SWE achieves portability across devices with tuned problem sizes (<70%), there is room for kernel optimization within the solver with more granular control of the architecture specifically by using Kokkos teams and architecture specific tunable parameters. These findings position SERGHEI-SWE as a robust, scalable, and portable simulation tool for large-scale geophysical applications under evolving HPC architectures with potential to enhance its performance.**

***Keywords*—Performance evaluation, Graphical Processing Units, Shallow Water Equations**

I. Introduction

Flash flood forecasting has posed a grand challenge due to its phenomenological complexity, which arises from the combination of multiple factors, including highly variable precipitation patterns, complex terrain, land use changes, and nonlinear hydrological responses. Addressing this challenge requires Early Warning Systems (EWS) that are composed of high-resolution numerical models that can simulate water flow dynamics, integrate real-time sensor data and most importantly be computationally efficient to ensure appropriate lead times for decision making.

There have been efforts developed in this direction that are well established such as the European Flood Awareness System (EFAS), its global variant Global Flood Awareness System (GloFAS), and the Flooded Locations And Simulated Hydrographs (FLASH) Project. These systems integrate real-time meteorological and hydrological data with advanced numerical models to enhance flood forecasting capabilities. EFAS and GloFAS leverage large-scale deterministic atmospheric predictions and the LISFLOOD hydrological model operationally as well as the LISFLOOD-FP model, while FLASH utilizes high-resolution radar-based precipitation estimates to rapidly assess flash flood risks leveraging the CREST hydrological model.

Despite having already established workflows for early warning and having success with these models, these lack high resolution in their forecasts providing results only up to 1km of resolution. In the context of flash flooding this can suffice to provide a general overview of the response of a watershed. However, specific valleys and streams (usually ungauged) cannot be studied with the required detail. Indeed, at such 1km resolution, these valleys and streams are very poorly represented. Additionally, hydrological models are unable to represent wave propagation which is crucial to estimate time-to-flood. Instead, hydrodynamic models are used to properly forecast and achieve a better understanding of complex flooding events, as they consider the spatial and temporal variation of water depth and velocity. The accepted best choice for hydrodynamic models for this purpose are a system of 2D partial differential equations, better known as the Shallow Water Equations (SWE).

Not until recently have SWE solvers been applied at very fine resolution and large domain sizes, primarly because of its high computational complexity. Advancements in numerical methods, hardware acceleration, and scalable parallel algorithms have made it possible to efficiently solve the SWE. More recently, the uptake of high performance computing

(HPC) techniques, such as domain decomposition, GPU acceleration, and asynchronous communication strategies, has significantly reduced computational bottlenecks, enabling faster-than-realtime predictions at a high resolution and accuracy. In the era of exascale computing, an HPC enabled SWE model then becomes feasible and beneficial to implement in EWS given the information that we currently have and the finer predictions that it would provide.

EFAS has implemented this with the LISFLOOD-FP model that solves the SWE. However, their current implementation yields forecasts up to 100m in resolution. Although this is better than the 1km resolutions, flash floods usually act at the local level in which resolutions closer to $5 - 10\mathrm{m}$ are more advantageous.

There are several open-source SWE solvers that have been developed in the last decade that can provide results for resolutions around $1\mathrm{m}$. This is favorable for flash flood prediction, however these results may be useless if computational walltime is excessive. As timely simulations are fundamental to ensure enough forecast lead time, the acceleration of computational methods is paramount. This can be accomplished using software that leverages the heterogeneous nature of current supercomputers.

Several SWE solvers have been developed to take advantage of graphics processing units (GPUs) as well as distributed programming techniques to accelerate the solution process. Most of these rely on the CUDA-Message Passing Interface (MPI) backend combination [1, 2] which sets them at a disadvantage when considering the current state of supercomputing platforms.

Until June 2022, the TOP500 list predominantly featured supercomputers built around NVIDIA GPUs paired with CPUs. However, with the advent of systems like Frontier and Lumi, both leveraging AMD GPUs, or Aurora made with Intel GPUs, the high-performance computing landscape shifted. This transition further underscores the limitations of SWE solvers that depend on the CUDA-MPI backend, as they are not readily optimized for these emerging architectures.

Avoiding this vendor lock trap is crucial for the development of efficient, reliable, and transferable EWS, so that they can be deployed anywhere, regardless of the available hardware. This is, at the core, the issue of performance portability.

The capability of porting software to different computing architectures is particularly attractive in the context of projects like the exascale computing project (ECP) or the EuroHPC Joint Undertaking that invest in considerably large supercomputing resources to solve these specific problems [3, 4].

In the case of the EuroHPC JU, one of its goals is to maintain a federated, hyper-connected supercomputing system built on platforms with completely different architectures. This diversity leads to the idea of leveraging these vast resources for use in EWS. Portable software then becomes an indispensable tool, allowing for the seamless deployment of EWS across heterogeneous computing environments. By abstracting away hardware-specific details, it mitigates the constraints of vendor lock-in while enhancing both the resilience and scalability of

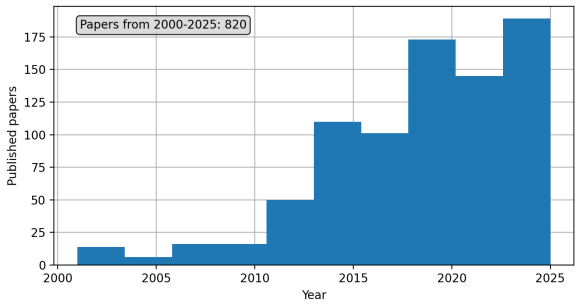


FIGURE 1: Distribution of published papers (SCOPUS) from 2000 to 2025 containing performance portability in their title or abstract. Only computer science publications were considered.

EWS. This flexibility ensures that these systems can effectively harness the immense and varied computational power provided by initiatives like the ECP and EuroHPC JU, ultimately leading to more robust and future-proof solutions.

Currently, the only fully performance portable, scalable SWE solver for flood simulation available is SERGHEI-SWE [5]. SERGHEI leverages the Kokkos performance portability abstraction layer [6, 7] to enable GPU acceleration in combination with distributed programming with MPI. SERGHEI, by using Kokkos, is CUDA, OpenMP, HIP, SYCL, and Pthreads-ready. While SERGHEI has this advantage of portability, performance and computational efficiency remains a crucial part in a SWE solver dedicated to flash flood modeling.

A. Related Work

Performance portability has been a topic of research since the 2000s, and it gained more interest among the computer science community since 2010, see Figure 1. The introduction of better processors and the general-purpose GPU enabled the construction of different HPC systems with the end goal to create exascale systems [8]. Performance portability libraries like Kokkos, RAJA, and OCCA have then been developed and studied extensively to evaluate their performance both in framework testing environments and large-scale scientific applications.

For example, in [9], the HARVEY massively parallel multi-physics code undergoes a comparative performance analysis of the CUDA, SYCL, HIP, Kokkos, and OpenMP programming paradigms in leadership systems, Frontier, Aurora, Polaris, that contain devices from the major GPU vendors NVIDIA, AMD, and Intel. Their main contribution is the quantitative comparison of all the performance portability frameworks and key insights into the current state of HPC programming models. An interesting finding is that the SYCL programming paradigm showed itself as the most performance portable programming model on both CPU and GPU. A key takeaway from their work is that HPC developers are no longer limited neither by vendor-specific solutions nor by performance portable paradigms when implementing their HPC applications.

Another recent effort was made in [10], by implementing an N-body problem in the following frameworks, OpenMP,

pSTL, Kokkos, CUDA, HIP, SYCL, and OpenACC. The implementations were tested across four different types of GPU architectures and evaluated using two different performance portability metrics based on application efficiency normalized to the best performance on each platform. The metrics used are based on arithmetic means, reflecting recent advancements in the field as established in [11, 12, 13, 14] which question the initial harmonic mean approach [15, 16]. Key outlooks from this work refer to Kokkos being a better suited platform for complex memory access pattern GPU algorithms and OpenMP being a close to perfect portability layer according to their results.

In the context of the SWE there have been efforts to design software to test different architectures and programming paradigms such as in [17], in which the SWE were solved using discontinuous Galerkin methods using the SYCL programming model for portability. They evaluated the code on CPU, GPU and FPGA which is a strength of this paradigm. Their performance portability evaluation did not implement any of the metrics used currently, however they reported architectural and application performance for which these metrics can be calculated. By using roofline models they assure their implementation lies close to the performance peaks of CPUs and GPUs. FPGA execution is assured to surpass CPU and GPU performance when data fits in their generated caches.

Within this same context, in [18] the SWE are solved over a sphere using the radial basis function finite difference method as a simplified atmospheric dynamics code. They implemented the method using OpenMP and OpenACC combined with MPI to test both portability and scalability using Intel CPUs and NVIDIA GPUs. This work lacks tests on other GPU and CPU architectures and performance portability metric evaluations, yet they deemed their code performance portable as it achieves near theoretical peak performance on all platforms.

B. Contributions

In this work we conduct a performance analysis to assess the SERGHEI-SWE solver [5], under different GPU architectures focusing on the most popular vendors NVIDIA, AMD and Intel. The goal of the study is to (i) provide a performance evaluation of the SERGHEI-SWE solver in terms of scalability, portability and GPU kernel execution; (ii) gain insights into the similarities and differences in performance and overall behaviour of the solver in different GPU architectures; (iii) provide evidence and practical information for users on what to expect when migrating from one architecture or system to another, in terms of runtime and scalability.

We consider our efforts and results both important and novel, as they contribute to the growing body of knowledge on performance portability in scientific applications, and represent one of the first, if not the first, studies to evaluate the performance portability of a production-ready, highly scalable shallow water solver.

This work will be organized as follows:

- Section II discusses the background of this work.
- Section III explains the methodology used to evaluate the SERGHEI-SWE solver.
- Section IV shows the experimental results obtained in this work.
- Section V refers to recommendations and future work for our study.

II. Background

A. Numerical Method

The SERGHEI code solves for the Cartesian 2D SWE specifically,

$$\frac{\partial \mathbf{U}}{\partial t} + \frac{\partial \mathbf{F}}{\partial x} + \frac{\partial \mathbf{G}}{\partial y} = \mathbf{S}_\mathrm{r} + \mathbf{S}_\mathrm{b} + \mathbf{S}_\mathrm{f},$$
$$\mathbf{U} = \begin{bmatrix} h \\ hu \\ hv \end{bmatrix} \quad \mathbf{F} = \begin{bmatrix} hu \\ hu^2 + \frac{1}{2}gh^2 \\ huv \end{bmatrix} \quad \mathbf{G} = \begin{bmatrix} hv \\ huv \\ hv^2 + \frac{1}{2}gh^2 \end{bmatrix}, \tag{1}$$
$$\mathbf{S}_\mathrm{r} = \begin{bmatrix} r_o - r_f \\ 0 \\ 0 \end{bmatrix} \quad \mathbf{S}_\mathrm{b} = \begin{bmatrix} 0 \\ -gh\frac{\partial z}{\partial x} \\ -gh\frac{\partial z}{\partial y} \end{bmatrix} \quad \mathbf{S}_\mathrm{f} = \begin{bmatrix} 0 \\ -\sigma_x \\ -\sigma_y \end{bmatrix},$$

in which h is the water height over the surface being simulated, u and v are the fluid velocity components in the x and y respectively, and g is gravity. The source terms $\mathbf{S}_\mathrm{r}$, $\mathbf{S}_\mathrm{b}$, and $\mathbf{S}_\mathrm{f}$ take into account infiltration and exfiltration, the slope of the domain, and the friction of the terrain over the fluid being modeled. SERGHEI's numerical method starts with the integral over a control volume Ω of the equations,

$$\frac{\partial}{\partial t}\int_\Omega \mathbf{U} d\Omega + \oint_{\partial\Omega} (\mathbf{E}\cdot\mathbf{n}) dl = \int_\Omega (\mathbf{S}_\mathrm{r} + \mathbf{S}_\mathrm{b} + \mathbf{S}_\mathrm{f}) d\Omega, \tag{2}$$

in which $\mathbf{E} = (\mathbf{F}, \mathbf{G})$ is the convective flux vector and $\mathbf{n} = (n_x, n_y)$ is the outward unit normal vector. The finite volume method equation is then derived by discretizing over the edges of the cell and considering a piecewise representation of the conserved variables,

$$\frac{\partial}{\partial t}\int_{\Omega_i} \mathbf{U} d\Omega + \sum_{k=1}^{4} (\mathbf{E}\cdot\mathbf{n})_k l_k = \int_{\Omega_i} \mathbf{S} d\Omega + \int_{\Omega_i} \mathbf{S}_\mathrm{r} d\Omega. \tag{3}$$

Here $\mathbf{S} = \mathbf{S}_\mathrm{b} + \mathbf{S}_\mathrm{f}$, and the source term $\mathbf{S}_\mathrm{r}$ is separate since SERGHEI uses a different numerical scheme for integration. The fluxes $\mathbf{E}\cdot\mathbf{n}$ can be found by applying the theory of approximate Riemann solvers [19, 20]. SERGHEI specifically implements an augmented solver such that the source terms provide a contribution to the fluxes at each interface, this ensures well-balancing in the solution.

To find these fluxes, equation (1) can be projected onto $\mathbf{n}$ such that a one dimensional Riemann problem can be defined on the rotated set of coordinates $(\hat{x}, \hat{y})$ that lie on the normal and tangential directions of each edge. An approximate solution $\widehat{\mathbf{U}}(\hat{x}, t)$ can be defined and its integral over a defined control volume must be equal to the integral of the exact solution $\mathbf{U}(\hat{x}, t)$ over the same control volume [19]. Following

this property, time linearization and projection of the source term onto the edge, equation (3) turns into

$$\frac{\partial}{\partial t}\int_{\Omega_i}\mathbf{U}d\Omega+\sum_{k=1}^{4}(\delta\mathbf{E}-\mathbf{T})_k\cdot\mathbf{n}_k l_k=\mathbf{0} \tag{4}$$

where $\mathbf{S}=\mathbf{T}_k\cdot\mathbf{n}_k$ for an appropriate source term matrix $\mathbf{T}_k$, and the fact that $\sum_{k=1}^{4}\mathbf{n}_k l_k=0$ helps define $\delta\mathbf{E}=\mathbf{E}_j-\mathbf{E}_i$ in the sum. This problem can be solved approximately with,

$$\begin{aligned}&\frac{\partial}{\partial t}\int_{\Omega_i}\widehat{\mathbf{U}}d\Omega+\sum_{k=1}^{4}\widehat{\mathbf{J}}_{\mathbf{n},k}\delta\widehat{\mathbf{U}}l_k=\mathbf{0}\\&\widehat{\mathbf{U}}(\hat{x},0)=\begin{cases}\mathbf{U}_i & \text{if } \hat{x}>0\\ \mathbf{U}_j & \text{if } \hat{x}<0,\end{cases}\end{aligned} \tag{5}$$

in which an approximation of the flux $(\delta\mathbf{E}-\mathbf{T})_k\cdot\mathbf{n}_k=\widehat{\mathbf{J}}_{\mathbf{n},k}\delta\hat{\mathbf{U}}_k$ was applied. The Jacobian $\widehat{\mathbf{J}}_{\mathbf{n},k}$ can be approximated locally by using $\widetilde{\mathbf{J}}_k=\partial\mathbf{F}(\hat{\mathbf{U}})/\partial\hat{\mathbf{U}}$ evaluated at the Roe states. The analysis of this eigenstructure is useful to define an basis onto which the wave and source terms can be projected, resulting in,

$$\widetilde{\mathbf{J}}_k=\widetilde{\mathbf{P}}_k\widetilde{\mathbf{\Lambda}}_k\widetilde{\mathbf{P}}_k^{-1};\quad \delta\widehat{\mathbf{U}}=\widetilde{\mathbf{P}}_k\mathbf{A}_k;\quad \mathbf{T}\cdot\mathbf{n}_k=\widetilde{\mathbf{P}}_k\mathbf{B}_k, \tag{6}$$

where $\mathbf{\Lambda}_k$ is the matrix of the eigenvalues of $\widetilde{\mathbf{J}}_k$ and $\widetilde{\mathbf{P}}_k=(\widetilde{\mathbf{e}}_1,\widetilde{\mathbf{e}}_2,\widetilde{\mathbf{e}}_3)$ the matrix made up of its eigenvectors. Some algebra leads to define the fluxes as,

$$\widetilde{\mathbf{J}}_k\delta\widehat{\mathbf{U}}_k-\mathbf{T}_k\cdot\mathbf{n}_k=\sum_{m=1}^{3}\left[(\tilde{\lambda}\alpha-\beta)\tilde{\mathbf{e}}\right]_{m,k} \tag{7}$$

where $\tilde{\lambda}_m$ are the elements of $\mathbf{\Lambda}_k$ and α_m,β_m the elements of $\mathbf{A}_k$ and $\mathbf{B}_k$ respectively. The approximate problem is integrated in time using a forward Euler method, substituting equation (7) into (5), assuming a cell averaged approximate solution, and performing upwinding in the discretization, the update rule for the conserved variables $\mathbf{U}$ is,

$$\mathbf{U}_i^{n+1}=\mathbf{U}_i^n-\frac{\Delta t}{\Delta x}\sum_{k=1}^{4}\sum_{m=1}^{3}\frac{\tilde{\lambda}^-}{\tilde{\lambda}}\left[(\tilde{\lambda}\alpha-\beta)\tilde{\mathbf{e}}\right]_{m,k}^{n}+[\mathbf{S}_\mathrm{r}]_i^n\Delta t \tag{8}$$

with a cell evaluated Courant-Friedrichs-Lewy (CFL) condition,

$$\Delta t=\mathrm{CFL}\min_i\left\{\frac{\Delta x}{|u_i|+\sqrt{gh_i},|v_i|+\sqrt{gh_i}}\right\} \tag{9}$$

B. Implementation

SERGHEI implements a distributed memory approach using MPI to divide the Cartesian computational domain across a Cartesian grid of processes. At initialization, the (N_x,N_y) domain grid is mapped onto a (P_x,P_y) process grid such that each process handles a local subgrid of size (n_x+2,n_y+2), including halo cells for communication. The process (P_i,P_j) in the grid is defined by $(P_i=R\%P_t,P_j=R/P_t)$ where R is the process rank number and $P_t=P_xP_y$ the total number of processes.

The fields in the local subgrid of size (n_x+2,n_y+2) are represented in SERGHEI as 1D arrays of size $N_\mathrm{arr}=(n_x+2)(n_y+2)$, with indexing $k=j(n_y+2)+i$. To fully exploit modern HPC resources, SERGHEI leverages the Kokkos library for shared memory and GPU parallelism using simple `Kokkos::parallel_x` concepts. These field arrays are implemented using the `Kokkos::View` data structure, ensuring efficient data placement close to where computations occur. By adopting a unified memory paradigm across all device backends, Kokkos abstracts explicit host-to-device memory transfers, allowing execution across heterogeneous architectures.

To ensure portability across GPU architectures, SERGHEI explicitly uses the `Kokkos::SharedSpace` and `Kokkos::DefaultExecutionSpace` concepts as these translate to the unified memory spaces and execution environments defined by the Kokkos configuration. For instance, on NVIDIA devices, they will map to `Kokkos::CudaUVMSpace` and `Kokkos::Cuda` respectively.

III. Methodology

This section details the methodology used to assess the computational performance of the SERGHEI code. Our analysis focuses on a key test: a circular dam break hydrological simulation. This case is perfect to characterize the performance of code in ideal conditions.

A. Test Case: circular dam break

The circular dam break is a generalisation from the one dimensional dam break to two dimensions by implementing a full rotation along a z axis. This case is particularly beneficial as it has regular, uniform dynamics such that load balancing is not an issue when performing scaling tests. Our circular dam break case is then set up as a square domain of side $N_\mathrm{side}\Delta x$ where N_side is the number of cells in that specific side and $\Delta x=0.5$. The initial conditions then can be set as,

$$h(x,y,t=0)=\begin{cases}4 & \text{if } r=\sqrt{x^2+y^2}\leq N_\mathrm{side}\Delta x/5\\ 1 & \text{otherwise.}\end{cases} \tag{10}$$

Figure 2 shows the initial conditions on the rz plane. The boundary conditions are defined to be reflective at the boundaries. Our implementation is dependent on the number of cells N_side, the number of processes in the x and y dimensions of the cartesian topology (P_x, P_y) the simulation length t_sim and the output frequency $t_\mathrm{I/O}$. We vary these parameters according to the different studies we carry out.

B. Systems and Configuration

Our performance analysis relies on four HPC supercomputing systems, each with a distinct accelerator architecture. This diversity enables the evaluation and execution of code with the MPI+(CUDA, HIP, SYCL) programming paradigms through the Kokkos portability layer. Table 1 and 2 list the compilers and systems used in our testing and data acquisition phase as well as the software stack employed in the experiments.

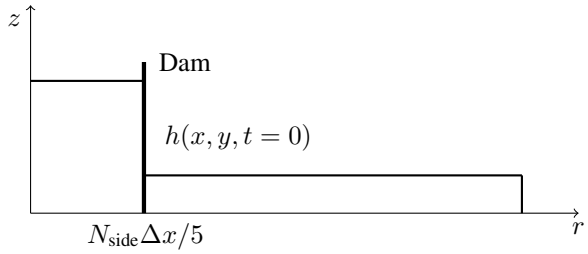


FIGURE 2: Diagram of the initial conditions of the implemented dam break test case in the rz plane.

TABLE 1: Compiler versions used for each code variant.

System	Compiler
AURORA	Intel(R) oneAPI DPC++/C++ Compiler 2025.0.0 (2025.x.0.20240629)
FRONTIER	AMD clang version 17.0.0 (roc-6.0.0 23483 7208e8d15fbf218deb74483ea8c549c67ca4985e)
JUWELS BOOSTER	g++ (GCC) 12.3.0
JEDI	g++ NVHPC/24.9-CUDA-12

This includes the versions of MPI libraries, GPU programming environments (CUDA, HIP, SYCL), compiler toolchains, and additional dependencies required for Kokkos-based execution.

C. Strong and weak scaling

We conducted strong scaling tests on each platform using a circular dam break scenario with $N_{\text{side}} = 36000$ cells, totaling $N_{\text{tot}} = 1.296 \times 10^9$ cells. The code was instrumented with the `Kokkos::timer` tool to measure execution times and `Kokkos::fence` to ensure correct GPU kernel measurements.

Only the simulation loop time was measured, as it is the computationally intensive part of the code (i.e., excluding initialization and I/O). For data analysis, we considered the number of physical GPUs per node rather than those detected by the resource schedulers, since some architectures feature multiple tiles or graphics compute dies (GCDs) per physical GPU. To reduce statistical variation, each execution was repeated five times. Parallel speedup was calculated as the ratio of the simulation time on a single node using four physical GPUs to the simulation time at larger scales, $s = t_{\text{node}}/t$. In the case of Intel GPUs we add several points at multiples of four GPUs instead of the six GPUs featured by an Aurora node to compare data points across all architectures.

Regarding weak scaling tests, we kept the number of cells in the circular dam break constant per physical GPU, specifically $N_{\text{weak}} = 1.28 \times 10^8$. The problem was scaled up to 1024 nodes where resources allowed. The cases where the physical GPU is divided into tiles or GCDs in the case of the Aurora or Frontier systems, this number is divided by the number of subchips in the GPU. Following this, we chose $N_{\text{weak}} = 6.4 \times 10^7$ for each subchip. Given that we used the dam break case with $N_{\text{weak}} = N_{\text{side}}^2$ it is evident that N_{side} is not necessarily an integer, in these cases we considered $N_{\text{weak}} = (N_{\text{side}} + 1)^2$. After measurement, the efficiency was quantified with respect to the ratio of simulation time at large scales to the simulation time on a single node using four physical GPUs, $e = t_{\text{node}}/t$. For the architectures that feature 6 physical GPUs per node this might imply that some efficiencies might surpass the ideal efficiency.

D. Roofline analysis

We focused on profiling key offloaded kernels, particularly those responsible for the most computationally intensive operations in our numerical method. Specifically, we analyzed the `computeDt` kernel, which is associated with equation (9) for timestep computation. We also measured `computeDeltaFluxXRoe` and `computeDeltaFluxYRoe`, which correspond to equation (7) for flux computations. Additionally, we profiled `computeNewState`, which updates the solution state following equation (8), and `computeTimeStepReduction`, which applies a timestep correction for wet-dry interfaces.

To perform this analysis, we employed NVIDIA Nsight Compute, AMD ROCm Profiler, and Intel Advisor. Following the methodology established in [21], we constructed roofline graphs for each GPU architecture tested. We then normalized the results for each roofline by peak floating operations and by an arithmetic intensity threshold to compare all architectures and their behavior in a single graph. The normalizations have this form,

$$p_{\text{norm}} = \frac{p_{\text{achieved}}}{p_{\text{peak}}}; \qquad a_{\text{norm}} = \frac{a_{\text{achieved}}}{a_{\text{thresh}}}; \qquad a_{\text{thresh}} = \frac{p_{\text{peak}}}{b_{\text{peak}}} \tag{11}$$

where p is processing speed measured in FLOP/s, a is arithmetic intensity measured in FLOP/Byte and b is memory bandwidth measured in Byte/s. The peak values were calculated empirically on NVIDIA and AMD architectures using the Empirical Roofline Tool (ERT) [22] and using Intel Advisor on Intel architectures. ERT is a toolkit that launches simple $d = ab + c$ operations to quantify real values for peak processing speed and peak bandwidth in specific hardware.

The circular dam break application was evaluated using two different configurations based on problem size: one with a fixed problem size ($N_x = 15000$) and another with the size that yielded the best performance. To determine the optimal configuration, we tested a range of problem sizes, varying N_x from 4000 to 20000.

E. Performance Portability Metrics

To quantify performance portability across architectures, we employed several Performance Portability (Φ) metrics as proposed in [11, 12, 15, 16]. The metrics we calculated take the following forms:

$$\Phi_1(\alpha, \varphi, \mathcal{H}) = \begin{cases} \dfrac{|\mathcal{H}|}{\sum_{i \in \mathcal{H}} \dfrac{1}{r_i(\alpha, \varphi)}} & \text{if} \quad \forall i \in \mathcal{H}, r_i \neq 0 \\ 0 \quad \text{otherwise,} \end{cases}$$

$$\Phi_2(\alpha, \varphi, \mathcal{H}) = \begin{cases} \dfrac{\sum_{i \in \mathcal{H}} r_i(\alpha, \varphi)}{|\mathcal{H}|} & \text{if} \quad |\mathcal{H}| \neq 0 \\ 0 \quad \text{otherwise,} \end{cases} \tag{12}$$

TABLE 2: HPC systems used to test the SERGHEI SWE solver.

System	Centre	CPU	GPU	Vendor	On-Node Accelerators	Nodes
AURORA	ANL	Intel Sapphire Rapids	Max 1550 Series	Intel	$6 \times$ (2 Tiles)	10624
FRONTIER	OLCF	AMD Optimized 3rd Gen EPYC	Instinct MI250X	AMD	$4 \times$ (2 GCD)	9408
JUWELS BOOSTER	JSC	AMD EPYC 7402	Ampere A100	NVIDIA	4	936
JEDI	JSC	NVIDIA Grace Arm Neoverse-V2	Hopper H100	NVIDIA	4	48

TABLE 3: HPC software stack used to compile and test the SERGHEI SWE solver. All systems implemented Kokkos 4.5.99.

System	MPI	Kokkos backend	PNetCDF	Linux Version
AURORA	MPICH/4.3.0rc3	OneAPI-2025.x.0.20240629	1.12.3	SUSE-5.14.21-150400.24.55-default
FRONTIER	CRAY-MPICH/8.1.29	ROCM-6.0.0	1.12.3	SUSE-6.4.0-150600.23.17_14.0.63-cray_shasta_c
JUWELS BOOSTER	ParastationMPI/5.9.2-1	CUDA-12.2.91	1.12.3	Rocky-5.14.0-503.23.1.el9_5.x86_64
JEDI	NVHPC-OpenMPI/4.1.5	CUDA-12.6.20	1.12.3	Red Hat-5.14.0-503.26.1.el9_5.aarch64+64k

where $r_i(\alpha, \varphi)$ corresponds to the relative performance on platform $i \in \mathcal{H}$ of a specific application α for a specific problem φ. It is calculated as,

$$r_i = \frac{p_{\text{achieved}}}{\min\left\{p_{\text{peak}}, b_{\text{peak}} a_{\text{achieved}}\right\}} \tag{13}$$

to account for both compute bound and memory bound kernels.

As we do not have low-level, highly optimized versions of our application for all architectures, we solely consider architectural efficiency as recommended in [12]. We computed these metrics to obtain a scalar value in the range $[0, 1]$, where a value of 1 indicated full portability with optimal efficiency across all platforms. We computed these metrics using the data collected for the roofline analysis both for uniform problem sizes and best performing problem sizes.

IV. Results and Discussion

A. Strong and Weak Scaling Analysis

Strong scaling results demonstrate efficient parallel performance across all tested architectures up to approximately 512 to 1024 GPUs, as shown in Figure 3a. Among the platforms, JEDI's NVIDIA GH200 system achieves the shortest execution times across all configurations, closely followed by Frontier's AMD MI250X. Interestingly, Aurora's Intel Max 1550 and JUWELS Booster's NVIDIA A100 exhibit comparable execution times, suggesting similar performance characteristics despite differences in underlying hardware and GPU architectures. These trends highlight the impact of platform-specific features, such as memory bandwidth, interconnect topology, and GPU architecture, on the scalability of large-scale simulations.

Speedup results, shown in Figure 3b, confirm that the SERGHEI-SWE code scales as expected on all platforms plateauing at about 32 times single node time. These results exhibit near-linear scaling behavior up to 32-64 GPUs. As the number of GPUs increases, deviations from ideal speedup emerge, which can be attributed to increasing communication overheads and reduced per-GPU workload. Nonetheless, all platforms maintain a good degree of parallel efficiency, particularly in the low-to-mid GPU range.

In practice, simulations with this solver generally do not exceed $N_{\text{tot}} = 1.296 \times 10^9$ cells. The circular dam break case serves as a well-defined analytical scenario, offering insight into the solver's behavior under ideal conditions. It provides a balanced framework for understanding how performance scales with increasing resources, establishing an upper bound for performance in the absence of load balancing issues.

On the other hand, our weak scaling analysis allowed us to test the limits of the SERGHEI-SWE code, examining its behavior for extremely large problem sizes. In this regard, Figure 3d presents the weak scaling efficiency for the circular dam break case across the four systems. All platforms maintain relatively high weak scaling efficiency up to 64 GPUs, with most above 95%. However, as the GPU count increases beyond this point, we observe a gradual decline in efficiency, more pronounced for the JUWELS BOOSTER system. This behavior may be explained by increased communication overhead and topology related issues as we reach the maximum number of nodes in the JUWELS BOOSTER system.

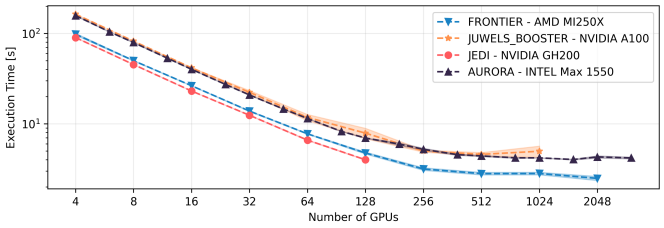
AURORA (Intel Max 1550) consistently delivers the highest efficiency across nearly the entire scaling range, particularly at large GPU counts (1024+), which highlights the strength of its interconnect architecture in sustaining load balance and minimizing communication bottlenecks. In contrast, JUWELS BOOSTER shows the steepest efficiency drop at higher scales, with performance falling below 90% at 4096 GPUs.

Even though the theoretical peak performance and memory bandwidth of JEDI's GH200 should favor weak scaling, its efficiency begins to trail off slightly earlier than AURORA and FRONTIER. This could reflect early-stage tuning, GPU underutilization, or network saturation issues (the reader should keep in mind that JEDI is the development instrument for the upcoming JUPITER system, and not yet the production machine).

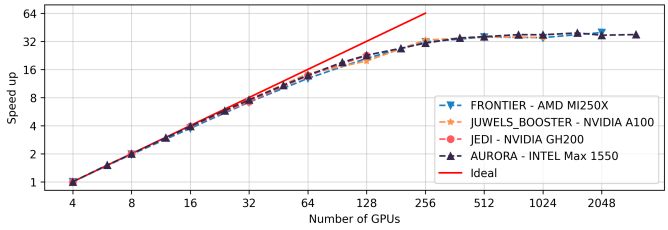
Overall, the results indicate that while all architectures demonstrate good weak scaling up to medium-large scales, maintaining high efficiency at extreme scales (2048+ GPUs) requires advanced interconnect performance and careful optimization of memory and communication patterns. These insights are valuable for informing future deployment strategies, targeting performance tuning efforts and optimization.

B. Roofline Analysis

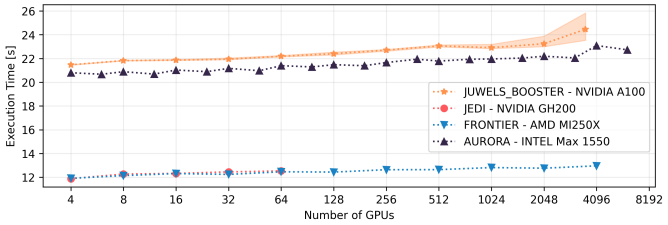
Normalized roofline graphs were constructed as means of qualitative comparison of performance behavior across all



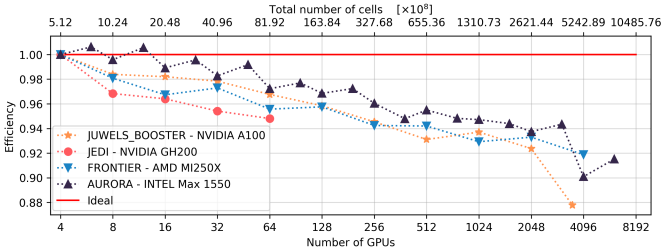
(A) Strong scaling execution time results for all HPC systems for the dam break case with $N_{\text{side}} = 36000$.



(B) Speedup results for all HPC systems for the dam break case with $N_{\text{side}} = 36000$.



(C) Weak scaling execution time results for all HPC systems along with total amount of computational cells.



(D) Weak scaling efficiency results for all HPC systems along with total amount of computational cells..

FIGURE 3: Strong and weak scaling results for all the tested architectures. Weak scaling results were collected up to 1024 computing nodes for the dam break case (in systems that supported this amount of nodes).

TABLE 4: Empirical peak performance results obtained with ERT and Intel Advisor for a single GPU/GCD/Tile.

System	Theoretical Peak (p_{peak} [GFLOP/s] | b_{peak} GB/s)	Empirical Peak (p_{peak} [GFLOP/s] | b_{peak} GB/s)
AURORA	(17000.00 | 3276.80) [23]	(11400.00 | 1203.43)
FRONTIER	(23900.00 | 1600.00) [24]	(20105.84 | 1229.92)
JEDI	(34000.00 | 4000.00) [25]	(31209.00 | 3032.59)
JUWELS BOOSTER	(9700.00 | 2039.00) [26]	(9494.71 | 1258.40)

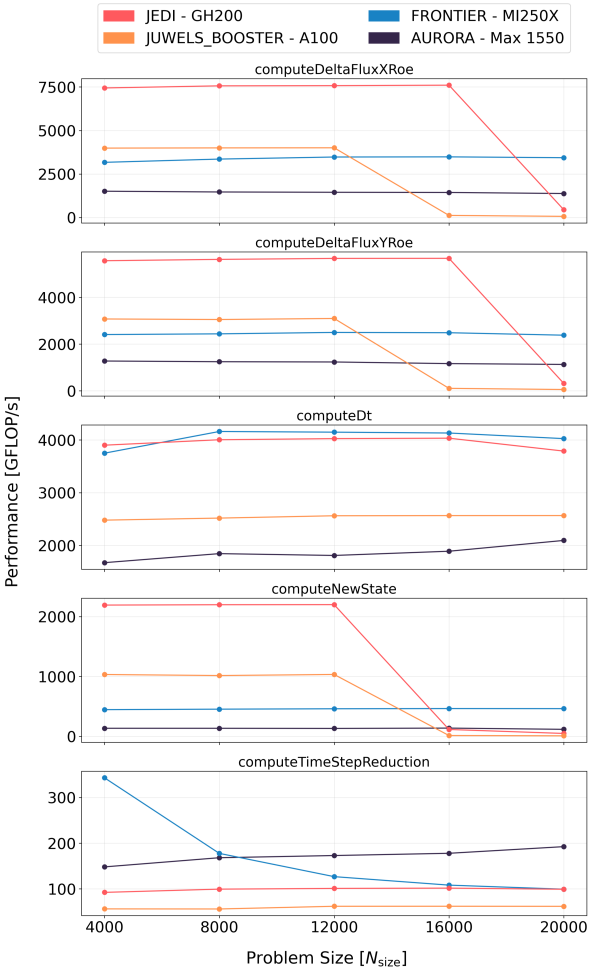
architectures. Table 4 summarizes the empirical peak values used to normalize both axes in the graphs.

Figure 4a illustrates the performance variation of all profiled kernels on a single GPU. The `computeDeltaFluxXRoe` and `computeDeltaFluxYRoe` are the most expensive kernels in SERGHEI-SWE, typically demanding around 40% of the simulation time, and thus we give them higher attention. As expected, performance increases with the number of cells across all architectures. However, performance plateaus around 10^7 to 10^8 cells. At larger problem sizes, the NVIDIA architectures exhibit an unusual sharp decline in performance.

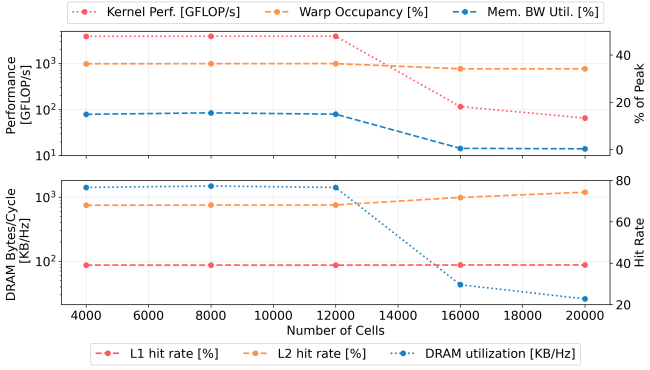
Figure 4b shines a light on this specific issue. In a dedicated profiling run, we measured the `computeDeltaFluxRoe`'s kernel performance and other metrics as a function of problem size to discard certain theories that may have explained the performance drop. Despite L1 and L2 cache hit rates remaining relatively stable (approximately 40% and 68–73%, respectively), FLOP/s collapses at larger problem sizes, indicating that cache inefficiency is not the primary bottleneck. Warp occupancy decreases only slightly from 39% to approximately 38%, confirming that sufficient warps remain active and that SM starvation is not the dominant factor. In contrast, effective DRAM information movement (Bytes/cycle) drops sharply to near zero, correlating closely with the observed FLOP/s degradation. These results suggest that the kernel is primarily limited by memory latency, rather than raw bandwidth, a limitation likely caused by coalescing inefficiencies arising from our flattened 1D array indexing. This behavior is likely to be propagating to all kernels since the code is based on this indexing scheme. It is important to note that this particular behavior does not happen on Intel and AMD architectures likely due to their different memory hierarchy and their ways of handling and optimizing low occupancy for kernels.

We can also see an abnormal behavior in the performance of the `computeTimeStepreduction` kernel, specifically for the FRONTIER system. Performance decreases with problem size. We hypothesize that the observed exponential performance decrease in the Frontier system for the `computeTimeStepReduction` kernel is attributed to a combination of architectural and algorithmic factors: (i) extremely low arithmetic intensity ($\sim$ 6 FLOPS per thread) leading to memory-bound behavior, (ii) low occupancy due to the lightweight nature of the kernel relative to the 64-thread wavefront width, (iii) non-coalesced 1D memory accesses across large arrays, which penalizes memory throughput on GPU architectures, and (iv) diverging warps which is caused by a conditional statement within the kernel [27, 28]. These factors compound as the problem size grows, explaining the kernel's performance degradation.

The plots in Figure 5 provide insights into the computational characteristics of the SERGHEI-SWE code's most intensive kernels during the circular dam break case. In both panels, we observe that the majority of the profiled kernels are situated within the memory-bound region of the normalized roofline, indicating that their performance is primarily limited by memory bandwidth rather than computational throughput. This



(A) Variation of performance with problem size for the analyzed GPU kernels. We hypothesize that drops may be caused by uncoalesced accesses in NVIDIA architectures.



(B) Variation of profiled variables with problem size for the `computeDeltaFluxXRoe` Kernel on the JUWELS Booster platform. We hypothesize that drops may be caused by uncoalesced accesses in NVIDIA architectures.

points to the current flat memory access patters implemented in the loops showing up as inefficiencies or costly memory operations. This also highlights the need of remapping data to fast scratch memory.

In Figure 5a, corresponding to a large fixed problem size ($N_{\text{side}} = 15000$), all architectures (JEDI, JUWELS BOOSTER, FRONTIER, and AURORA) exhibit a consistent pattern: key kernels such as `computeDeltaFluxXRoe`, `computeNewState`, and `computeDt` cluster along the sloped region below the bandwidth roofline. This suggests that despite differences in hardware architecture and peak compute capabilities, the kernels are similarly constrained by data movement. A significant shift toward compute-bound behavior is visible for the NVIDIA architectures, such as JEDI (GH200) JUWELS BOOSTER (A100). This is rather interesting given that most kernels are memory bound. This suggests a better handling of Kokkos kernels with the CUDA backend.

Figure 5b presents the best-performing configurations across a range of problem sizes (N_{side} from 4000 to 20000). Here, the arithmetic intensity of some kernels slightly increases compared to the fixed problem size case, particularly for JEDI and FRONTIER, moving a few kernels closer to the bandwidth roof. This shift suggests that larger problem sizes can marginally improve computational efficiency. Nonetheless, even in these best cases, the operations remain fundamentally memory-bound.

An interesting observation from both plots is that `computeTimeStepReduction` consistently exhibits the lowest arithmetic intensity across all platforms, positioning it furthest from the ideal roof. This behavior highlights it as a potential target for further optimization, perhaps through improved memory access patterns, kernel fusion, or algorithmic restructuring to increase arithmetic intensity. However, this kernel typically demands only around 5% of the simulation time.

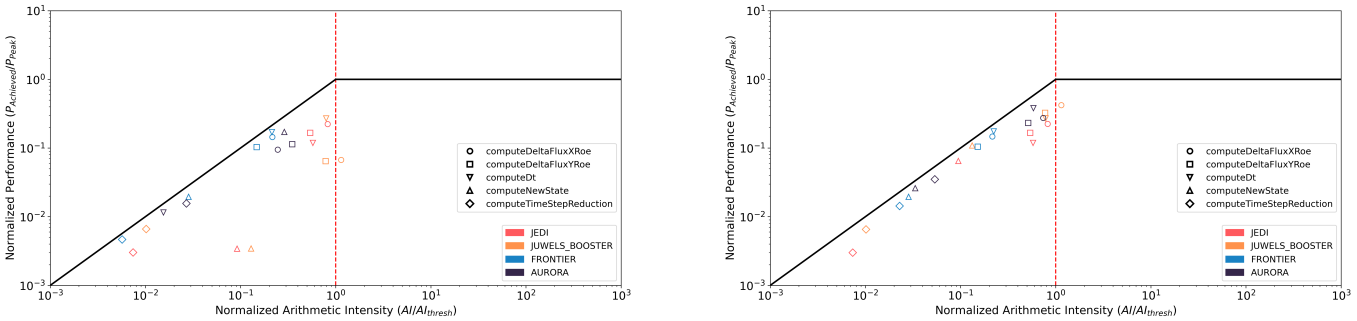
TABLE 5: Performance portability metric values for a fixed problem size at $N_{\text{side}} = 15000$

Kernel Name	Uniform Test	
	Φ_1	Φ_2
`computeDeltaFluxXRoe`	0.1779	0.3350
`computeDeltaFluxYRoe`	0.2090	0.3689
`computeDt`	0.3722	0.4847
`computeNewState`	0.0596	0.3736
`computeTimeStepReduction`	0.5731	0.6111

C. Performance Portability Analysis

Table 5 presents the performance portability metrics Φ_1 and Φ_2 for the five most intensive kernels under a fixed problem size. The `computeTimeStepReduction` kernel achieves the highest performance portability with $\Phi_1 = 0.5731$ and $\Phi_2 = 0.6111$, suggesting consistent and efficient performance across architectures even without fine-tuning. However, as we mentioned before this kernel may be fused with other kernels as it does not play a critical part of our simulation. In contrast, `computeNewState` shows the lowest portability ($\Phi_1 = 0.0596$), indicating Φ_1 presents a high sensitivity to relative performance differences. Figure 6 shines a light on the differences of each metric.

Comparing the two metrics, Φ_2 (arithmetic mean) values are consistently higher than Φ_1 (harmonic mean) across all kernels and tests. This is expected, as the harmonic mean is more sensitive to low-performing outliers, thus providing a



(A) Roofline graph for a circular dam break case with $N_{\text{side}} = 15000$ for all architectures.

(B) Roofline for the best performing configuration of all architectures among problem sizes N_{side} ranging from 4000 to 20000.

FIGURE 5: Roofline graphs for the five most intensive kernels in the circular dam break case.

more conservative estimate of portability. Even if the validity of Φ_1 has been heavily discussed with respect to the definition of performance portability [29] it might prove useful to assess portability from a more rigorous point of view, perhaps in a way to measure architecture-specific optimization for performance portable codes.

Regarding the SERGHEI-SWE code, we observe that its portability remains mostly under 70%, indicating potential areas for optimization. These optimizations can again be guided by this same performance portability analysis, focusing on bottlenecks such as memory access patterns, kernel launch overhead and low occupancy.

Establishing a fair experimental setup across GPU architectures proved difficult, underscoring the challenges of evaluating performance portability using architectural efficiency. Many hardware dependent variables such as memory bandwidth, compute throughput, and memory hierarchy strongly influence performance. Our Kokkos-based code, while portable, lacks tunable parameters to adapt to each architecture's strengths. For example, setting fixed problem sizes for all architectures leads to resource under-utilization in some cases. In other cases, setting problem sizes based on a uniform fraction of available GPU memory may lead to mismatched workloads: systems with more memory processed larger problems, distorting comparisons. This revealed a broader issue: there are few clear, singular points of comparison across heterogeneous platforms. These difficulties highlight the importance of careful benchmarking and the need for software that is not only portable, but fairly comparable across systems.

Performance portability analysis from an application efficiency point of view seems to avoid many of these issues, as it relies on comparing each architecture's performance against its own optimized baseline. Rather than enforcing a uniform problem size or relying on theoretical peak values, this approach evaluates how efficiently a portable application utilizes the available hardware. By comparing the performance of a portable version of the code to an architecture-specific optimized version, we can assess how much performance is retained without architecture-specific tuning. This method reduces the bias introduced by architectural asymmetries and avoids misleading conclusions drawn from raw runtime or peak-normalized metrics.

However, for codes developed using performance portable frameworks, such as the SERGHEI-SWE solver, this approach incurs a considerable amount of additional effort. Constructing and maintaining highly optimized, architecture-specific versions purely for benchmarking defeats the purpose of portability and diverts resources away from scientific or engineering goals. As a result, while application efficiency comparisons offer a rigorous perspective, they may not be practical or sustainable in real-world research and development workflows, especially when the goal is to evaluate how well a single implementation performs across platforms.

In this direction, developing a clear and fair methodology for evaluating performance portability across architectures could provide a consistent benchmark for the ever growing body of performance-portable software. Such an approach would help avoid diverting valuable resources from critical scientific efforts, such as developing more accurate flood simulation models and their optimization.

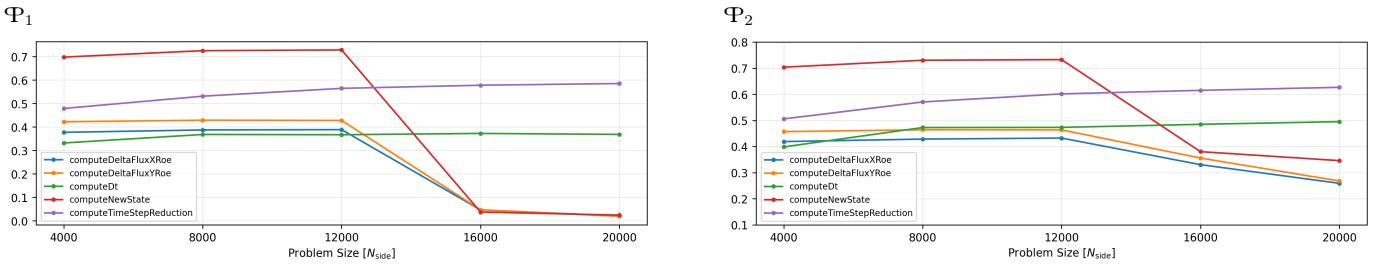
V. Final Remarks

A. Conclusions

We have conducted a performance analysis of the SERGHEI-SWE solver on four different HPC systems featuring four GPU architectures from the top vendors (NVIDIA, AMD, Intel) focusing on scaling performance, kernel performance and performance portability aspects of code.

Our strong scaling results show efficient parallel performance up to 512–1024 GPUs across all architectures, with JEDI's NVIDIA GH200 system achieving the fastest execution times, closely followed by Frontier's AMD MI250X platform. Aurora's Intel Max 1550 and JUWELS Booster's NVIDIA A100 showed comparable execution times, indicating similar levels of scalability and efficiency despite architectural differences. Weak scaling analysis further confirms the solver's robustness, maintaining high scalability (>90%) as the problem size grows proportionally to the computational resources.

The roofline analysis reveals that the most computationally intensive kernels are predominantly memory-bound across all



(A) Variation of $\mathcal{P}_1$ with respect to problem size.

(B) Variation of $\mathcal{P}_2$ with respect to problem size.

FIGURE 6: Variation of performance portability metrics with respect to problem size. We observe that for $\mathcal{P}_1$, the relative performance exhibits abrupt changes, similar to the raw performance shown in Figure 4a. In contrast, $\mathcal{P}_2$ does not penalize performance drops as severely.

platforms, highlighting memory bandwidth as a critical factor for achieving high performance. Additionally, variations in problem size show that the arithmetic intensity and performance of several kernels change in peculiar ways, particularly on NVIDIA architectures. We attribute this behavior to complex function calls within kernels, which may lead to register spilling and reduced occupancy on the device.

The performance portability study was based on both harmonic and arithmetic mean-based metrics demonstrated the portability of the SERGHEI-SWE code to be less than 70% for all kernels. This fact suggests that this solver can benefit from optimizations regarding memory layout and access patters specifically increasing granularity in memory management.

Performance portability metrics were based on architectural efficiency which directly measures performance consistency with respect to attainable peak hardware performance. This has the benefit of not needing a custom implementation for every architecture but the drawback of not taking into account attainable peak algorithm performance which is provided by highly optimized architecture specific implementations.

These insights can help users of the SERGHEI-SWE code characterize their runtime depending on the GPU architectures they implement for their research. It also gives key information about the current state of the solver that they can take into account when migrating from one family of GPUs to another.

Overall, while we see excellent behavior of the SERGHEI-SWE solver in terms of scalability. GPU kernel execution and performance portability analyses shine a light into device behavior which we can optimize. Doing this can further improve scalability and make the SERGHEI-SWE code an even better alternative for EWS and their applications.

B. Future Work

Future research with the SERGHEI-SWE needs to involve optimization of GPU kernels to tackle certain problems found in this study such as implementing more granular memory management using the Kokkos team concepts and the implementation of tunable architecture specific parameters such that parameter optimization can be achieved either manually or by implementing new tools such as Kokkos autotuning [30]. In the same direction, instrumentation using Kokkos tools might prove to be an advantage for Kokkos related debugging and testing.

As the SERGHEI-SWE solver is part of a bigger environment of SWE-related software, this work can be used as a methodology for benchmarking their performance and analyzing where optimization is needed. This is a critical task for EWS related software not only in hydrology but in numerical forecasting overall, hence it can also be an initial methodology to benchmark future performance portable numerical models.

Establishing a fair process for comparison using architectural efficiency is also within future research, as performance portability is becoming more relevant to leverage the current exascale era.

ACKNOWLEDGMENT

This research used resources of the Argonne Leadership Computing Facility, a U.S. Department of Energy (DOE) Office of Science user facility at Argonne National Laboratory and is based on research supported by the U.S. DOE Office of Science-Advanced Scientific Computing Research Program, under Contract No. DE-AC02-06CH11357.

This research also used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory under project GEO161: Advancing flood modeling with the SERGHEI code, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

We would like to acknowledge gratefully the Earth System Modelling Project (ESM) for supporting this work by providing computing time on the ESM partition of the JUWELS supercomputer at the Jülich Supercomputing Centre (JSC) through compute time projects RUGSHAS (project number 29000) and EHRTAS (project number 59866).

The authors would also like to acknowledge the JUPITER Research and Early Access Program (JUREAP) for providing access to JEDI, the JUPITER Exascale Development Instrument.

REPRODUCIBILITY

Scripts are provided for compiling the code, and reproducing the results presented in this study can be found in [31, 32].

References

[1] M. Morales-Hernández et al. "TRITON: A Multi-GPU open source 2D hydrodynamic flood model". In: *Environmental Modelling & Software* 141 (2021), p. 105034. ISSN: 1364-8152. DOI: https://doi.org/10.1016/j.envsoft.2021.105034. URL: https://www.sciencedirect.com/science/article/pii/S1364815221000773.

[2] Ayhan H. Saleem and Matthew R. Norman. "Accelerated numerical modeling of shallow water flows with MPI, OpenACC, and GPUs". English. In: *Environmental Modelling and Software* 180 (2024). Cited by: 0. ISSN: 13648152. DOI: 10.1016/j.envsoft.2024.106141. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85198519619&doi=10.1016%2fj.envsoft.2024.106141&partnerID=40&md5=200f0286de92ed485598b1afd9ae974c.

[3] Anshu Dubey et al. *Performance Portability in the Exascale Computing Project: Exploration Through a Panel Series*. Tech. rep. Accessed: 2025-04-30. U.S. Department of Energy, 2021. URL: https://www.osti.gov/servlets/purl/1823307.

[4] EuroHPC Joint Undertaking. *Multi-Annual Strategic Programme 2021–2027: Version 2024*. Accessed: 2025-04-30. Mar. 2024. URL: https://eurohpc-ju.europa.eu/document/download/2a5db80d-6bd3-4ea9-9418-4b49cadcf040%5C_en?filename=20240219%5C_For%5C_GB%5C_Decision%20Amendment%20MASP%202021-2027%20v220324%20Final%201%20%281%29.pdf.

[5] D. Caviedes-Voullième et al. "SERGHEI (SERGHEI-SWE) v1.0: a performance-portable high-performance parallel-computing shallow-water solver for hydrology and environmental hydraulics". In: *Geoscientific Model Development* 16.3 (2023), pp. 977–1008. DOI: 10.5194/gmd-16-977-2023. URL: https://gmd.copernicus.org/articles/16/977/2023/.

[6] Christian R. Trott et al. "Kokkos 3: Programming Model Extensions for the Exascale Era". In: *IEEE Transactions on Parallel and Distributed Systems* 33.4 (2022), pp. 805–817. DOI: 10.1109/TPDS.2021.3097283.

[7] H. Carter Edwards, Christian R. Trott, and Daniel Sunderland. "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns". In: *Journal of Parallel and Distributed Computing* 74.12 (2014). Domain-Specific Languages and High-Level Frameworks for High-Performance Computing, pp. 3202–3216. ISSN: 0743-7315. DOI: https://doi.org/10.1016/j.jpdc.2014.07.003. URL: http://www.sciencedirect.com/science/article/pii/S0743731514001257.

[8] Exascale Computing Project. *Exascale Computing Project*. https://www.exascaleproject.org/. Accessed: 2025-04-30.

[9] Aristotle Martin et al. "Performance Portability Evaluation of Fluid-Structure Interaction Simulations on Heterogeneous Platforms". In: *Proceedings of the International Supercomputing Conference*. Accepted for publication. 2025.

[10] Rodrigo A.C. Bartolomeu et al. "Effect of implementations of the N-body problem on the performance and portability across GPU vendors". In: *Future Generation Computer Systems* 169 (Aug. 2025), p. 107802. ISSN: 0167-739X. DOI: 10.1016/j.future.2025.107802. URL: http://dx.doi.org/10.1016/j.future.2025.107802.

[11] Ami Marowka. "Reformulation of the performance portability metric". In: *Software: Practice and Experience* 52.1 (June 2021), pp. 154–171. ISSN: 1097-024X. DOI: 10.1002/spe.3002. URL: http://dx.doi.org/10.1002/spe.3002.

[12] Ami Marowka. "On the Performance Portability of OpenACC, OpenMP, Kokkos and RAJA". In: *International Conference on High Performance Computing in Asia-Pacific Region*. HPC Asia2022. ACM, Jan. 2022, pp. 103–114. DOI: 10.1145/3492805.3492806. URL: http://dx.doi.org/10.1145/3492805.3492806.

[13] Ami Marowka. "On the Incorrect Use of Application Efficiency to Calculate Performance Portability". In: *Parallel Processing and Applied Mathematics*. Springer Nature Switzerland, 2025, pp. 105–118. ISBN: 9783031857003. DOI: 10.1007/978-3-031-85700-3_8. URL: http://dx.doi.org/10.1007/978-3-031-85700-3_8.

[14] Ami Marowka. "Portability efficiency approach for calculating performance portability". In: *Future Generation Computer Systems* 170 (Sept. 2025), p. 107826. ISSN: 0167-739X. DOI: 10.1016/j.future.2025.107826. URL: http://dx.doi.org/10.1016/j.future.2025.107826.

[15] S. J. Pennycook, J. D. Sewall, and V. W. Lee. *A Metric for Performance Portability*. Version Number: 1. 2016. DOI: 10.48550/ARXIV.1611.07409. URL: https://arxiv.org/abs/1611.07409 (visited on 03/03/2025).

[16] S. John Pennycook and Jason D. Sewall. "Revisiting a Metric for Performance Portability". In: *2021 International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. St. Louis, MO, USA: IEEE, Nov. 2021, pp. 1–9. ISBN: 978-1-6654-2439-4. DOI: 10.1109/P3HPC54578.2021.00004. URL: https://ieeexplore.ieee.org/document/9652845/ (visited on 04/09/2025).

[17] Markus Büttner et al. "Enabling Performance Portability for Shallow Water Equations on CPUs, GPUs, and FPGAs with SYCL". In: *Proceedings of the Platform for Advanced Scientific Computing Conference*. PASC '24. ACM, June 2024, pp. 1–12. DOI: 10.1145/3659914.3659925. URL: http://dx.doi.org/10.1145/3659914.3659925.

[18] Samuel Elliott et al. "Implementation of a scalable, performance portable shallow water equation solver using radial basis function-generated finite difference methods". In: *The International Journal of High Performance Computing Applications* 33.4 (Sept. 2018), pp. 619–631. ISSN: 1741-2846. DOI: 10.1177/

1094342018797170. URL: http://dx.doi.org/10.1177/1094342018797170.

[19] J. Murillo and P. García-Navarro. "Weak solutions for partial differential equations with source terms: Application to the shallow water equations". In: *Journal of Computational Physics* 229.11 (June 2010), pp. 4327–4368. ISSN: 0021-9991. DOI: 10.1016/j.jcp.2010.02.016. URL: http://dx.doi.org/10.1016/j.jcp.2010.02.016.

[20] J. Murillo and P. García-Navarro. "Augmented versions of the HLL and HLLC Riemann solvers including source terms in one and two dimensions for shallow flow applications". In: *Journal of Computational Physics* 231.20 (2012), pp. 6861–6906. ISSN: 0021-9991. DOI: https://doi.org/10.1016/j.jcp.2012.06.031. URL: https://www.sciencedirect.com/science/article/pii/S0021999112003464.

[21] Charlene Yang. *Hierarchical Roofline Analysis: How to Collect Data using Performance Tools on Intel CPUs and NVIDIA GPUs*. 2020. arXiv: 2009.02449 [cs.DC]. URL: https://arxiv.org/abs/2009.02449.

[22] Brian Van Straalen et al. *Empirical Roofline Tool (ERT) v1.1.0*. Feb. 2019. DOI: 10.11578/dc.20210423.2. URL: https://www.osti.gov/biblio/1779081.

[23] Argonne Leadership Computing Facility. *Aurora User Guide*. https://docs.alcf.anl.gov/aurora/. Accessed: 2025-04-22. 2025.

[24] Oak Ridge Leadership Computing Facility. *Frontier User Guide OLCF; User Documentation — docs.olcf.ornl.gov*. https://docs.olcf.ornl.gov/systems/frontier_user_guide.html#profiling-applications. [Accessed 22-04-2025].

[25] NVIDIA Corporation. *NVIDIA H100 Tensor Core GPU Datasheet*. https://resources.nvidia.com/en-us-tensor-core/nvidia-tensor-core-gpu-datasheet?ncid=no-ncid. Accessed: 2025-04-22. 2022.

[26] NVIDIA Corporation. *NVIDIA A100 Tensor Core GPU Datasheet*. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/a100/pdf/nvidia-a100-datasheet-nvidia-us-2188504-web.pdf. Accessed: 2025-04-22. 2020.

[27] AMD. *Performance Guidelines*. Accessed: 2025-09-18. 2024. URL: https://rocm.docs.amd.com/projects/HIP/en/latest/how-to/performance_guidelines.html.

[28] AMD. *Hardware Implementation*. Accessed: 2025-09-18. 2024. URL: https://rocm.docs.amd.com/projects/HIP/en/latest/understand/hardware_implementation.html.

[29] Ami Marowka. "Reformulation of the performance portability metric". In: *Software: Practice and Experience* 52.1 (June 2021), pp. 154–171. ISSN: 1097-024X. DOI: 10.1002/spe.3002. URL: http://dx.doi.org/10.1002/spe.3002.

[30] Kokkos Team. *Kokkos 4.5 Release Introduces New Auto-tuning Features*. Accessed: 2025-05-03. Dec. 2024. URL: https://kokkos.org/blog/blog-post-09/.

[31] CNCA-CeNAT. *SERGHEI Performance Portability Tests*. https://gitlab.com/CNCA_CeNAT/performance_portability/serghei_perfport_tests. Accessed: April 29, 2025. 2024.

[32] SERGHEI Workgroup. *SERGHEI: Simulation Environment for Geomorphology, Hydrodynamics and Ecohydrology in Integrated form*. https://gitlab.com/CNCA_CeNAT/performance_portability/serghei-amd. Accessed: April 29, 2025. 2024.