Journal of Computer and Communications, 2025, 13(2), 28-36
https://www.scirp.org/journal/jcc
ISSN Online: 2327-5227
ISSN Print: 2327-5219

Portable Software Environment for Ultrahigh-Resolution ELM Development on GPUs

Dali Wang[1*], Peter Schwartz[1], Fengming Yuan[1], Franklin Eaglebarge[2], Danial Riccuito[1], Peter Thornton[1], Chris Layton[1], Qinglei Cao[3]

[1]Environmental Sciences Division, Oak Ridge National Laboratory, Oak Ridge, TN, USA
[2]Pellissippi State Community College, Knoxville, TN, USA
[3]Computer Science, Saint Louis University, St. Louis, MO, USA
Email: *wangd@ornl.gov

How to cite this paper: Wang, D.L., Schwartz, P., Yuan, F.M., Eaglebarge, F., Riccuito, D., Thornton, P., Layton, C. and Cao, Q.L. (2025) Portable Software Environment for Ultrahigh-Resolution ELM Development on GPUs. *Journal of Computer and Communications*, **13**, 28-36.
https://doi.org/10.4236/jcc.2025.132003

Received: January 16, 2025
Accepted: February 17, 2025
Published: February 20, 2025

Abstract

This paper presents our endeavors in developing the large-scale, ultra-high-resolution E3SM Land Model (uELM), specifically designed for exascale computers furnished with accelerators such as Nvidia GPUs. The uELM is a sophisticated code that substantially relies on High-Performance Computing (HPC) environments, necessitating particular machine and software configurations. To facilitate community-based uELM developments employing GPUs, we have created a portable, standalone software environment preconfigured with uELM input datasets, simulation cases, and source code. This environment, utilizing Docker, encompasses all essential code, libraries, and system software for uELM development on GPUs. It also features a functional unit test framework and an offline model testbed for comprehensive numerical experiments. From a technical perspective, the paper discusses GPU-ready container generations, uELM code management, and input data distribution across computational platforms. Lastly, the paper demonstrates the use of environment for functional unit testing, end-to-end simulation on CPUs and GPUs, and collaborative code development.

Keywords

E3SM Land Model, Ultrahigh-Resolution ELM, Portable Software Environment, GPU-Ready Environment

1. Introduction

Advanced Earth system models (ESM) are crucial for providing insights into climate variations and enhancing our comprehension of the interplay between natural and human systems and the Earth's climate. The Energy Exascale Earth System Model (E3SM) is a fully integrated ESM that employs code tailored for the US Department of Energy's (DOE) supercomputers to tackle the most pressing Earth system science issues [1]. Within the E3SM framework, the E3SM Land Model (ELM) models the interactions between terrestrial land surfaces and other Earth system components, contributing to our understanding of hydrologic cycles, biogeophysics, and terrestrial ecosystem dynamics [2].

The ELM software is a complex system of a half million lines of code that utilizes highly customized datatypes, approximately 2000 global arrays, and over 1000 subroutines [3] [4]. Significant progress has been made in the past couple of years in developing a large-scale, ultrahigh-resolution ELM (uELM) simulation utilizing Exascale computers for high-fidelity land simulation at continental and global levels. For example, a novel computational model and framework for uELM simulation on hybrid architectures in exascale computers have been developed [5] [6]. Several uELM porting strategies have been formulated and a functional unit test (FUT) framework has been established [7]. This FUT framework dissects the code, enabling swift code generation and verification, and has facilitated the completion of several individual ELM modules [8]. However, uELM is a complex code with a significant dependency on HPC environments, including machine architecture, compilers, and a range of external libraries. The development of uELM on Exascale computers, equipped with GPUs, still presents a considerable challenge, imposing much more stringent requirements on machine and software configurations, such as compiler and OpenACC implementation.

We have created a portable software environment to promote rapid uELM development and support community-based uELM development using GPU. It is the first effort to allow seamless uELM deployment on hybrid computer architectures with both CPUs and GPUs. The software environment includes all necessary HPC system packages, libraries, and software tools for uELM development. It is also preconfigured with input datasets, simulation cases, and source code.

This paper is structured as follows: it delves into the creation of GPU-ready containers, the management of uELM code, and the sharing of input data across various computational platforms. Lastly, the paper presents the use cases of the software environment for code development on GPUs (e.g., functional unit testing and end-to-end simulation), utilizing diverse computational resources.

2. Portable Software Environment for uELM Development

We describe a software environment that utilizes OS-level virtualization (Docker technologies) to contain all the code, libraries, and system software needed for uELM development over NVIDIA GPUs. Then we explain the code and container management, and data sharing among several computational platforms.

2.1. GPU-Ready uELM Container Creation

The process consists of five steps:

1) Identify a Docker image with Nvidia GPU support: To take advantage of the power of GPUs for our research, we need a base Docker system that supports GPU. After attempting to customize docker images for Nvidia GPU support without success, we decided to focus on identifying and evaluating preconfigured NVidia-docker images (nvcr.io) that already include compatible NVHPC programming environments and the necessary CUDA driver support. This allows us to efficiently run our simulations and computations on Nvidia GPUs across multiple computational resources.

2) Customize a Docker image for ELM simulation: On the top of the base Docker system, we integrate the ELM code, including the Offline ELM Model Testbed (OLMT) for automatic configuration and simulation of ELM at the site level. We accomplished this by leveraging our previous experience in developing customized docker images from https://ngee-arctic.ornl.govthe NGEE-Arctic project. Through a multi-step procedure for creating docker images, we successfully generated a new docker image for site-based ELM simulation on CPU.

3) Integrate a FUT tool for the standalone uELM module creation: We have developed a software tool (named SPEL [7] [8]) we developed to enable uELM code generation and optimization over GPUs using Compiler directives within a FUT framework [9]. We use SPEL functions to instrument code segment into the ELM source code to capture and save the input and output data streams from a target ELM functional unit from a reference ELM simulation on CPUs. Then we create standalone uELM functional unit modules, in which, SPEL constructs a unit test driver that handles the initialization and reading of input parameters and data streams, executes a target uELM functional unit module, and saves the output datastreams, and compares the output datastreams with the output streams from the reference solution to verify the correctness and robustness of the standalone uEML module.

4) Deploy SPEL for GPU-ready uELM code porting and optimization: SPEL also provides functions to facilitate uELM GPU code generation and optimization. Users have the flexibility to develop their own strategies for quickly porting the GPU code, efficiently utilizing GPU resources, and maximizing the performance of individual uELM functional units. Additionally, users can combine several individual uELM functional units to generate an aggregated functional unit for further testing via SPEL.

5) Conduct end-to-end uELM code development on GPUs: The unit test of uELM via SPEL creates a good foundation for end-to-end code development on GPUs with compiler directives. Here, we enumerate four prevalent issues encountered during the process using NVHPC packages: a) The function of passing the array index to subroutines is flawed. We need to restructure the routines associated with the index and now pass the array elements explicitly as arguments. b) The atomic operations (atomic directive) are inconsistent, even though they might function within the FUT environment. We need to make sure these atomic operations are over the primitive datatypes. c) The most common error encountered is a

fatal error, which complains that the variable (be it local or global) is only partially present on the device. Given that we lack direct access to the underlying algorithms and libraries, we are compelled to explore various methods to circumvent this issue. and s) The CUDA Debugger (within NVHPC) does not perform as efficiently as its CPU counterparts and demands excessive GPU memory. Consequently, we have automated the generation of variable checking and verification functions (in FORTRAN) using Python scripts.

We have achieved a threefold speedup on a fully loaded computing node within Summit [5]. The code is integrated into our standard container images.

2.2. Standard uELM Containers

We have developed Docker images for various computing resources (machine type, compiler, and library) with exemplary uELM configurations and simulation cases. These images are managed via Dockerhub, enabling developers to establish their Dockerhub accounts and pull these images into their local computing systems. Developers can initiate the container as a model user, construct new cases using the E3SM's Common Infrastructure for Modeling Earth (CIME) or OLMT, and initiate simulations effortlessly. They also have the option to modify the code to make it GPU-ready with NVHPC/OpenACC, using SPEL (e.g., verification functions). Users can choose to exit and halt the container or detach from it while allowing it to run. They can later reattach to the running container for further action. Users can also create new Docker images with code changes or uELM cases for subsequent developments.

The containers are designed for uELM development on CPU-only or hydrid CPU/GPU systems. Each container is incorporated with the complete E3SM source code, which includes CIME and a collection of external models. They also contain the necessary input data for exemplary small simulation cases, along with useful tools like OLTM and SPEL. Most importantly, the container is equipped with all the necessary software libraries and environments for uELM code development and simulation right out of the box.

Several important file directories of a standard uELM container are listed in **Table 1**, including the source code, input data and output directories, and tools that are used for uELM simulation case creation and code development.

Table 1. File structure of a standard uELM container.

E3SM	E3SM source code (with uELM)
inputdata	input data and domains for uELM
output/cime_case_dirs	uELM simulation cases directory
output/cime_run_dirs	uELM simulations output directories
tools	scripts/utilities used by SPEL and OLTM
app/SPEL_OpenACC	SPEL working directory (with a LakeTemperature example)

2.3. uELM Code Management and Data Sharing

The E3SM project employs GitHub as its version control system. Our container does not have a source control system. Users can establish an E3SM local repository on their computing resources initially, then map a local git folder to the uELM container with the "-volume" or "-mount" option. To circumvent potential folder privilege issues, it is advisable to initiate the uELM container as a "root" user. Following code alterations, users may disengage from the container and commit the modifications to the local E3SM repository. Alternatively, users can initiate a separate shell session and execute git commands outside the container while it remains running.

uELM simulations may require a substantial input dataset and generate voluminous outputs. It is impractical to incorporate large inputs and outputs into the container. Instead, users can utilize the "-volume" or "-mount" options to mount local data directory with the uELM container. In addition, if all computational resources are located within the same network, it is possible to share the uELM input and output data directory through Network File System (NFS). (Refer to Figure 1 or Section 2 for more details). Figure 1 illustrates the major components of the portable software environment and its deployments on computational resources within ORNLs open research domain.

The creation of the uELM software environment does not require powerful computing systems and can be implemented with a diverse range of computers, including laptops and PCs. Docker images can be deployed across a multitude of high-performance workstations or computational resources.

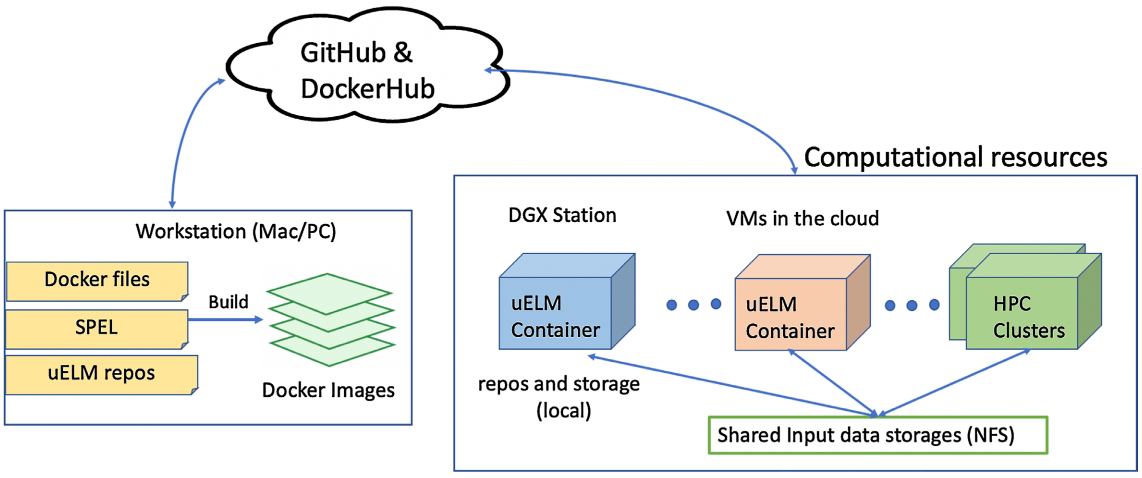

Figure 1. Portable Software Environment and its deployment in computational resources.

3. Case Studies and Demonstration

This section first presents federated computational resources for uELM development, and then we describe the procedure to generate reference solutions from an exemplary uELM simulation (FluxNet). After that, we illustrate how to use the SPEL to port and optimize an individual uELM module (LakeTemperature) for

GPUs. We present the process to conduct an end2end uELM code development using user repository and datasets. Finally, we showcase how to use the software environment as a collaborative platform for uELM code development.

3.1. Federated Computational Resources for uELM Development

We have established a federated uELM development environment at ORNL. It consists of several laptops and PCs (such as MacBook Pros and Dell PCs). We have also connected two powerful stations: a NVidia DGX-station (2 20-core Intel Xeons, 256 GB Memory, 4 16 GB Nvidia V100, and 2TB storage), one virtual machine (5-core Intel Xeon, 128 GB Memory, one 32 GB NVidia V100 GPU, 3 TB NFS storage) in ORNL open cloud managed with Red-hat OpenStack. The 3T NFS data space is also accessible via the DGX station and an HPC cluster in CADES at ORNL.

3.2. Reference Simulations: FluxNet Case on CPUs

The standard container encompasses an exemplary case that employs observational forcing data to drive the uELM simulations at 42 FluxNet sites globally (https://fluxnet.org). It incorporates all the necessary input datasets (e.g. domain, forcing, and surface properties), along with batch scripts, for configuration, construction, and automatic launch of the simulation. Additionally, a data duplication function is integrated, enabling users to replicate the 42 datasets to generate larger simulations as required for testing the parallel computing performance and scalability of the uELM code over HPC resources. Specifically, users can use the "OLMT_docker_42fluxnetsites_example.sh" script to create three simulation cases under the "/output/cime_case_dirs". They are ad-spinup for model initialization, transit run (from 1980 to present), and future projection. Each case contains files and tools for case setup, build, and submission scripts, as well as input namelists. These simulations are used as reference solutions for both SPEL FUT test (Section 3.3) and end2end uELM development on GPUs (Section 3.4).

Figure 2 illustrates the location of these 42 FluxNet Site (left) and sample

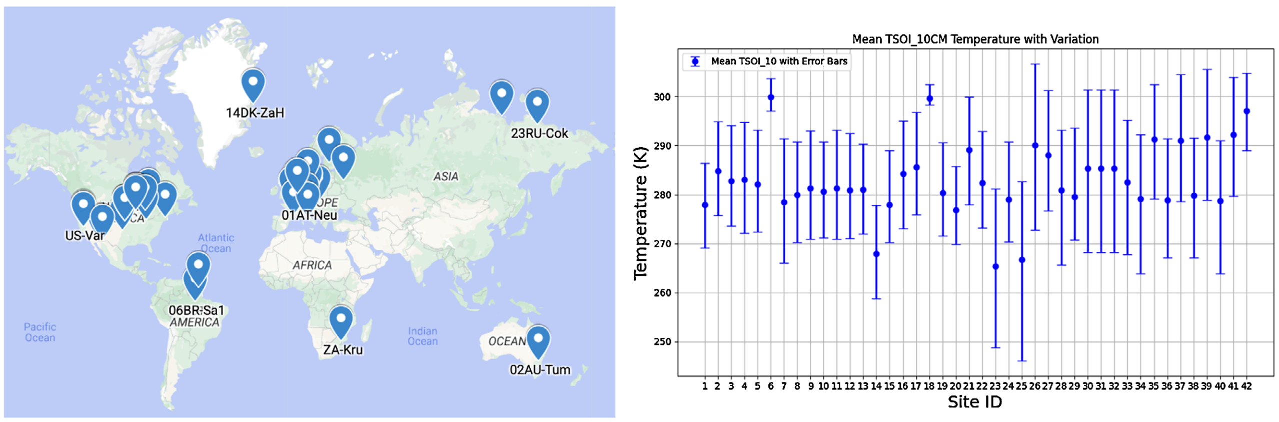

Figure 2. Location of 42 FluxNet Site (left) and sample TSOI_10CM monthly mean and variation using the uELM outputs from these sites.

output (soil temperature in top 10 cm of soil (TSOI_10CM)) of the uELM simulations.

3.3. FUT Demonstration: Lake Temperature Module

We have incorporated the SPEL toolkit in the standard uELM development container. The folder "/app/SPEL_OpenACC" houses Python utilities and FORTRAN templates, which are designed to generate standalone functional unit testing programs for individual uELM modules. These include read and write codes for the IO of uELM module testing, a verification code to compare outputs from unit testing modules, and a FUT driver. Additionally, it contains a folder named "SourceFiles" that includes the ELM Fortran source files and GPU-ready ELM test modules. Another folder, "scripts", comprises SPEL Python scripts and a few FORTRAN modules to generate ELM test modules. The generated unit test modules are stored in the "unit-tests" folder. Further information on the SPEL tool can be found in [7].

We have included a complete FUT case within the container to illustrate the process of developing a functional unit test for a specific uELM module (LakeTemperature). Technically, the user employs the UnitTestforELM.py to generate the LakeTemperature unit test code (FORTRAN), creates executables for both CPUs and GPUs, and subsequently uses a verification code to ensure that the GPU code yields identical bit4bit results as the CPU code.

3.4. End2End uELM Development: User Repo and Data

The standard container is utilized for end-to-end uELM development with users' development branches (e.g., the latest GPU-ready uELM code that has shown a threefold speed increase on a single Summit node). In this scenario, we check out the latest uELM development branch into a local repository and subsequently mount the local development branch into the standard uELM container by substituting the default uELM source code. We further create uELM cases, modify input data, and create new docker images (e.g., wangdl1108/uelm_dev:e2e uELM_dev in dockerhub).

We followed the E3SM development guide during the development and testing of uELM. Initially, we utilized the ELM master branch (comes with the container) to generate baseline solutions via a series of developer tests. Subsequently, we mounted local uELM development branch and the local "inputdata" into the docker (see subsection 2.3) and initiated the same developer tests on CPUs to establish new baselines (called uELM_ref). We then compare uELM_ref with the baselines derived from the master branch to ensure that the uELM code modifications were either "bit4bit" or "climate change-related". The next step involved repeating the uELM developer testing on GPUs and confirming that the newly generated simulation result was equivalent to the uELM_ref on CPUs.

3.5. uELM Software Environment: Collaborative Platform

The software environment offers a unique platform for interactive, collaborative uELM code development. We have established a Virtual Machine (VM) on the ORNL OpenCloud, equipped with a static IP address. We have also implemented a unified user ID, "cloud", and user access is managed by the Public key infrastructure (PKI) systems. This allows authorized users to access the VM from any approved computer using the "cloud" account. One user can initiate the uELM software environment (container), and others can connect (attach) to the running container. This enables everyone to interact and collaborate through their own computers to co-develop the uELM simultaneously. This setup proves highly beneficial for uELM development in a geographically distributed manner.

4. Conclusions

Recent advancements have led to the development of uELM simulations that utilize Exascale computers for high-fidelity land simulation at global levels. To support this, a portable software environment has been created for deployment on hybrid computer architectures.

The uELM software environment, inclusive of SPEL and representative simulation cases, is designed to aid users in uELM development on both CPUs and GPUs. This software environment is executable with a variety of devices including laptops, PCs, and HPC workstations. Furthermore, it can serve as an interactive, collaborative platform for uELM development.

Our current focus is on NVIDIA GPUs and OpenACC due to their compatibility with our target machine, the Summit supercomputer at ORNL. The insights leaned will be utilized to create equivalent portable software for uELM deployment using OpenMP on Frontier, the inaugural US Exascale computer, equipped with AMD CPUs and GPUs. Additionally, we are working on a multi-user uELM development environment in HPC clusters using Apptainer.

Software availability: This study refers to the E3SM code, inclusive of the uELM code, which can be found at uELM git repository, including uELM reference code and uELM GPU-ready code. The standard uELM development environment is hosted at dockerhub with a short user guide.

Acknowledgements

This research leverages previous efforts that were supported as part of the Energy Exascale Earth System Model (E3SM) project and the Next Generation Ecosystem Experiment-Arctic (NGEE-Arctic) project, funded by DOE's Office of Science, Office of Biological and Environmental Research. This research used resources of the Oak Ridge Leadership Computing Facility and Computing and Data Environment for Science (CADES) at the Oak Ridge National Laboratory, which are supported by the DOE's Office of Science under Contract No. DE-AC05-00OR22725. Mr. Franklin Eaglebarger is funded by the Community College Internships (CCI) program in the US Department of Energy (DOE) through the Oak Ridge Institute for Science and Education.

Conflicts of Interest

The authors declare no conflicts of interest regarding the publication of this paper.

References

[1] Golaz, J.-C., Caldwell, P.M., Van Roekel, L.P., Petersen, M.R., Tang, Q., Wolfe, J.D., Abeshu, G., Anantharaj, V., Asay-Davis, X.S., Bader, D.C., *et al.* (2019) The Doe e3sm Coupled Model Version 1: Overview and Evaluation at Standard Resolution. *Journal of Advances in Modeling Earth Systems*, **11**, 2089-2129.

[2] Burrows, S.M., Maltrud, M., Yang, X., Zhu, Q., Jeffery, N., Shi, X., *et al.* (2020) The DOE E3SM V1.1 Biogeochemistry Configuration: Description and Simulated Ecosystem-Climate Responses to Historical Changes in Forcing. *Journal of Advances in Modeling Earth Systems*, **12**, 9. https://doi.org/10.1029/2019ms001766

[3] Xu, Y., Wang, D., Janjusic, T., Wu, W., Pei, Y. and Yao, Z. (2017) A Web-Based Visual Analytic Framework for Understanding Large-Scale Environmental Models: A Use Case for the Community Land Model. *Procedia Computer Science*, **108**, 1731-1740. https://doi.org/10.1016/j.procs.2017.05.181

[4] Zheng, W., Wang, D. and Song, F. (2019) Xscan: An Integrated Tool for Understanding Open Source Community-Based Scientific Code. In: *Lecture Notes in Computer Science*, Springer International Publishing, Faro, 226-237. https://doi.org/10.1007/978-3-030-22734-0_17

[5] Wang, D., Schwartz, P., Yuan, F., Thornton, P. and Zheng, W. (2022) Toward Ultra-high-Resolution E3SM Land Modeling on Exascale Computers. *Computing in Science & Engineering*, **24**, 44-53. https://doi.org/10.1109/mcse.2022.3218990

[6] Yuan, F., Wang, D., Kao, S., Thornton, M., Ricciuto, D., Salmon, V., *et al.* (2023) An Ultrahigh-Resolution E3SM Land Model Simulation Framework and Its First Application to the Seward Peninsula in Alaska. *Journal of Computational Science*, **73**, Article ID: 102145. https://doi.org/10.1016/j.jocs.2023.102145

[7] Schwartz, P., Wang, D., Yuan, F. and Thornton, P. (2022) SPEL: Software Tool for Porting E3SM Land Model with Openacc in a Function Unit Test Framework. 2022 *Workshop on Accelerator Programming Using Directives* (*WACCPD*), Dallas, TX, 13-18 November 2022, 43-51. https://doi.org/10.1109/waccpd56842.2022.00010

[8] Schwartz, P., Wang, D., Yuan, F. and Thornton, P. (2022) Developing an Elm Ecosystem Dynamics Model on Gpu with Openacc. *Computational Science-ICCS* 2022: 22*nd International Conference*, London, UK, 21-23 June 2022, 291-303.

[9] Wang, D., Xu, Y., Thornton, P., King, A., Steed, C., Gu, L., *et al.* (2014) A Functional Test Platform for the Community Land Model. *Environmental Modelling & Software*, **55**, 25-31. https://doi.org/10.1016/j.envsoft.2014.01.015