# SPEL: Software tool for Porting E3SM Land Model with OpenACC in a Function Unit Test Framework

Peter Schwartz *
*Environmental Sciences Division*
*Oak Ridge National Laboratory*
Oak Ridge, TN 37831, USA
schwartzpd@ornl.gov

Dali Wang*
*Environmental Sciences Division*
*Oak Ridge National Laboratory*
Oak Ridge, TN 37831, USA
wangd@ornl.gov

Fengming Yuan, Peter Thornton
*Environmental Sciences Division*
*Oak Ridge National Laboratory*
Oak Ridge, TN 37831, USA
{yuanf, thorntonpe}@ornl.gov

***Abstract*—Most high-end computers adopt hybrid architecture, porting a large-scale scientific code onto accelerators is necessary. The paper presents a generic method for porting large-scale scientific code onto accelerators using compiler directives within a modularized function unit test platform. We have implemented the method and designed a software tool (SPEL) to port the E3SM Land Model (ELM) onto the GPUs in the Summit computer. SPEL automatically generates GPU-ready test modules for all ELM functions, such as CanopyFlux, SoilTemperature, and EcosystemDynamics. SPEL breaks the ELM into a collection of standalone unit test programs for easy code verification and further performance improvement. We further optimize several ELM test modules with advanced techniques, including memory reduction, reconstructed parallel loops, and asynchronous GPU kernel launch. We hope our study will inspire new toolkit developments that expedite large-scale scientific code porting with compiler directives.**

***Index Terms*—Compiler Directive, Function Unit Testing, Earth System Model, OpenACC**

## I. Introduction

In the past decades, hybrid architectures with accelerators have become a common trend in supercomputing. Porting large-scale scientific codes onto these hybrid architectures with accelerators is necessary and inevitable. Porting community-based scientific codes onto accelerators are continuous and iterative processes in which we may need to maintain scientific codes for both traditional CPU systems and hybrid systems with accelerators. Rewriting large-scale, actively evolved scientific codes in a new programming language for hybrid systems is not a practical approach, and many researchers have adopted compiler directives to port scientific code onto accelerators. However, most of them only focus on the performance tuning of a specific code.

This study presents a holistic method for porting a large-scale scientific code onto accelerators using compiler directives within a modularized unit testing framework. The method extends the function unit test (FUT) concepts [1], [2] originally developed for scientific function module generation and verification with captured data from reference simulations. The method contains several general steps for code porting, including 1) pre-porting software component analysis, 2) FUT framework for code porting, 3) standalone test module generation, and 4) advanced module optimization and performance tuning. We implemented the method and developed a Software tool to Port ELM (SPEL) onto GPUs using OpenACC.

This research was supported as part of the Energy Exascale Earth System Model (E3SM) project, funded by the U.S. Department of Energy, Office of Science, Office of Biological and Environmental Research. This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

* These authors contributed equally.

## II. Related Work

### A. Scientific Code Porting with Compiler Directives

Compiler directives are code statements that cause the compiler to take specific actions during compilation. In the context of code porting onto accelerators with compiler directives, we focus on OpenACC (Open Accelerators) and OpenMP (Open Multi-Processing). OpenACC is a programming standard for parallel computing on accelerators such as GPUs (mainly NVidia GPUs). OpenACC is designed to simplify GPU programming by inserting directives into the code to offload computation onto GPUs and parallelize the code at CUDA core level. OpenACC uses directives to tell the compiler where to parallelize loops and how to manage data between host and accelerator memories. OpenMP is an application programming interface (API) that supports multi-platform shared memory multiprocessing programming on many platforms. OpenMP is also based on directives to parallelize code, but it allows programmers to explicitly spread the execution of loops, code regions, and tasks across teams of threads. OpenMP started support of offloading data and computing to accelerators since OpenMP 4.0.

For many applications in quantum chemistry, material science, and bioinformatics, accelerator-ready codes delivered impressive performance improvements over the CPU-based code [3]–[6]. While porting efforts associated with large-scale community-based codes, such as fluid dynamics [7] and Earth system model components [8]–[10], generally delivered medium performance speedups (around 1-5 times faster).

### B. Function Unit Test (FUT) of Scientific Code

FUT was originally designed to create a direct linkage between observation measurements and mechanistic ecosystem modeling [1]. FUT has been further developed as a scientific

function test framework for modular environmental model development [2]. It provides innovative and convenient (piece-by-piece) ways for process-based multiscale model verification and validation, covering both model structure and scientific workflow. It also enables scientific model reconfiguration, reuse, and reassembly. Advanced technologies, such as compiler technology, have also been integrated to enhance FUT's analysis capability on scientific model structure, library dependency, and performance profiling [11].

A FUT framework contains four major components: a unit test driver, target scientific functions, data capture, and verification modules. Code segments are inserted into the source code to collect function parameters (input and output) of a target function. Then the instrumented code is recompiled and executed to save the input and output parameters of the target function into files. A Unit Test Driver (UTD) is created to initialize and load the input parameters from the input file, launch an individual function independently, and save output parameters into a file. In the end, FUT generates a collection of standalone, executable unit test programs, each using the UTD to launch a function of interest. Code verification within FUT is conducted via a bit-4-bit comparison between the two output results: one from the reference simulation and the other from the standalone test module.

## III. Code Porting with Compiler Directives within a FUT Framework

We present a method that extends the FUT concepts for scientific code porting with compiler directives. The method contains several steps: 1) pre-porting software analysis, 2) a general procedure of code porting,3) approaches to generate standalone modules, and 4) several advanced techniques for module optimization and performance tuning.

### A. Pre-porting software component analysis

We collect basic information on software system, such as the size, programming languages, and external libraries. Some parts of the code, originally designed on the CPUs, may not be suitable for accelerators, such as string handling, logging information, and IO operations. For HPC applications, the commonly used Message-passing Interfaces (MPI) and many math libraries may not be ready for accelerators. At this step, we mark software modules/subroutines excluded from the code porting on accelerators. We also mark code regions that can potentially be converted into data and computing regions on accelerators.

### B. Code porting within a FUT framework

*1) Workflow of code porting in a FUT framework:* Similar to the procedure mentioned in Section 2.2, we start with the data (Input and Output_R) collection of a target function from the original code, as illustrated on the left side of Fig. 1. The right side of Fig. 1 presents how to generate a standalone test module of the target function for code development on the host and accelerators with compiler directives.

FUT contains a Unit Test Driver (UTD) and a series of Test Modules (TM), each corresponding to a target function in the original scientific code. UTD provides a generic template for modularized unit testing with two basic functions: initialization and run. The initialization function prepares all necessary data structures for the target function. After that, UTD reads the *Input parameters* collected from the reference simulation (*Input*), runs the target function, then saves the output parameters (*Output_H*) into an *Output_TM* file. Within the FUT framework, we also generate test modules of the target functions on the accelerator using compiler directives (shown in the green panel inside the Test Module). The initialization function is the same, input parameters are copied into accelerators' data regions via DeepCopy capability. The kernels of the target function on the accelerators are generated using directives or constructs, and each kernel can contain multiple parallel and data regions. UTD also launches and coordinates the computing kernels on accelerators. The output parameters on the accelerators are copied back onto the host *(Output_D)* and then saved as another *Output_TM* file.

Bit-for-bit verification can be conducted between the reference output (*Output_R*) and the output from the test module on the host *(Output_H)*. Due to the differences in the computing architecture and compilers, user-defined tolerance values would be applied to compare two outputs from *Output_H* and *Output_D*.

*2) Test data collection, allocation, and tracking:* An essential function of a FUT is to capture and collect Input and Output parameters from a reference simulation. These parameters include both explicit function parameters and implicit function parameters (global variables) of the target functions.

The identification and collection of the parameters for a target function can be time-consuming. Certain software analysis and documentation tools, such as [12], [13], are useful. However, highly customized scripts are needed to parse and collect the function parameters of individual functions of a given scientific code. Many scientific codes use global variables extensively. Global variables cause tight code coupling and make the isolation and function unit test difficult. However, global variables, only declared once in a program, are accessible to all the modules. FUT defines and allocates functional parameters as global variables. FUT also tracks the access and modifications of functional parameters and analyzes their dependency throughout the code.

FUT collects the information on the memory consumption of test modules on accelerators, because 1) data movement between the host and accelerators is expensive, 2) accelerator memory is limited, and 3) memory allocation operations are expensive on accelerators [14].

### C. Standalone test module generation

FUT generates a standalone CPU-based unit test program by inserting a test module into the UTD. Then it uses compiler directives to offload data and computing on accelerators. FUT first creates a corresponding unit test program on accelerators

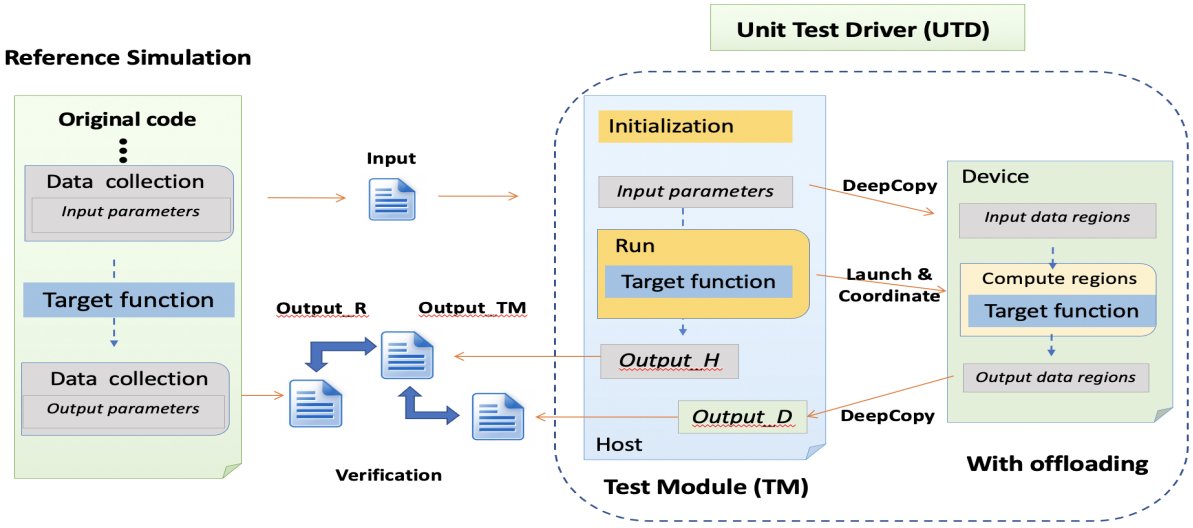


Fig. 1. Code porting within a FUT framework.

with minimum changes to the code first. For example, FUT iteratively instruments routine directives into the standalone test module and creates parallel loop regions in the source code. FUT tracks data dependency (function parameters) among test modules. FUT also tracks the dependency of global and local variables among iterative loops inside individual test modules.

### *D. Advanced module optimization and performance tuning*

We can further improve the performance of these standalone unit test programs by implementing several advanced techniques: 1) reduce the amount of data movements, 2) reduce the memory consumption of computing kernels, 3) improve the parallelism by flattening nested parallel loops, 4) reconstruct parallel loops for reduction operations, and 5) launch kernel asynchronously.

## IV. SPEL: A Software tool for Porting ELM on GPUs with OpenACC

The Exascale Energy Earth System Model (E3SM) is a fully coupled Earth system model supported by the US Department of Energy (DOE) to address the most critical scientific questions facing the nation, and the society [15]. The E3SM contains several community models such as atmosphere, ocean, land, sea ice, and glaciers. Within the E3SM framework, the E3SM Land Model (ELM) is designed to simulate how the changes in terrestrial land surfaces interact with other Earth system components and have been used to understand hydrologic cycles, biogeophysics, and ecosystem dynamics [16].

We are in the process of developing an ultra-high-resolution ELM (uELM) to support the high-fidelity land simulation (1km-resolution) over North America for the recent historical period (1980-2017). This land simulation would deliver high-resolution results describing the surface weather and climate, as well as the energy, water, carbon, and biogeochemistry processes over the continent. However, the landscape of North America (the entire computational domain) has around 24 million gridcells, 350 million columns, and 700 million vegetation patches and requires current high-end computers with GPUs.

We used several tools to improve the ELM software system and the computing performance [13], [17]. ELM adopts highly customized, globally accessible data structures to represent the heterogeneity of Earth's landscape and simulates the biophysical and biochemical processes over individual land surface units (called gridcell) independently. All the ELM functions ($>$ 1000 subroutines) are computationally non-intensive. We have also designed a hybrid computational model for the uELM simulation within the current E3SM framework [18]. We adopted a coupler-bypass tool that reads ELM input datasets directly from disk. We also keep most of the original ELM configuration within the E3SM Common Infrastructure for Modeling the Earth (CIME) infrastructure. We modify the original static domain decomposition algorithm to assign gridcells onto a collection of MPI processes. Each MPI process manages massive, independent ELM simulations on one single accelerator using OpenACC. In the study, we present a Software tool for Porting ELM (SPEL) using OpenACC within a FUT Framework. The computational platform used in the study is the Summit computer at Oak Ridge National Laboratory. Summit has 4,608 computing nodes, most of which contain two 22-core IBM POWER9 CPUs and six 16-GB NVIDIA Volta GPUs. The software environment used in our study includes NVIDIA HPC/NVFORTRAN 21.3 and several libraries: spectrum-mpi (10.4), NetCDF (4.8), pnetcdf(1.12), HDF (1.10), and CUDA (11.1).

SPEL is a collection of Python scripts designed to assist the ELM code porting onto GPUs with OpenACC. SPEL contains customized functions to handle the complicated ELM data structures (such as the large nested global variables in user-derived data types) and also customized the Unit Test Driver based on main ELM function (ELM_DRV). SPEL also

TABLE I
LIST OF NON-PORTABLE SUBROUTINES AND MODULES

| | |
|---|---|
| **Omit Modules** | **Modules for string handling, performance profiling, and IO activities** (i.e., abortutils, shr_log_mod, clm_time_manager, shr_infnan_mod, ncdio_pio, controlmod, getglobalvaluesmod) |
| **Omit Subroutines** | **Utility functions** (i.e., endrun, restartvar, hist_addfld1d, Prepare_Data_for_EM_PTM_Driver) |
| **Special Modules** | elmfatesinterfacemod elmfatesinterfacemod betrsimulationalm |

provides key functions for each step of the ELM code porting, including

- generate FORTRAN code to extract function parameters of individual subroutines based on information collected from static code analysis utilities/tools,
- automatically create standalone unit test modules by generating data regions and copying data in and out of the device with deepcopy and by inserting "routine seq" (first pass) or "parallel loop" directives into the ELM code, and
- reconstruct parallel do loops with predefined templates and check loops for potential reduction operations and print loop information for inspection if detected.

### *A. Pre-porting ELM system analysis*

We first listed modules or subroutines that are excluded for the GPU offloading and kernel generation. Good examples of these modules are string handling, IO, functions from external libraries, as well as some modules that we are not interested in porting. Table I lists the collection of modules and special subroutines for SPEL to comment out from the ELM source code before the porting effort.

### *B. Reference ELM simulation and code verification with the synthesized input dataset*

We create a reference simulation with the original ELM code (on CPU). We used the built-in Common Infrastructure for Modeling the Earth (CIME) to configure the ELM in a coupler-bypass mode with a static domain partition. The reference simulation adopts synthesized data using the observational data (including climate forcing, soil properties, surface properties, CNP, and other ELM parameters) from 42 AmeriFlux sites in the US.

We adopt a node-level performance comparison where a similar full workload is placed onto all the CPUs or GPUs within a single Summit node. The execution times of the original CPU code and GPU implementation at a single hourly simulation timestep are collected for performance evaluation. Specifically, we assign around 6000 gridcells (142 copies of the 42 AmeriFlux sites) on each of the six GPUs, and record the single timestep execution time of the ELM simulation. Then we assign a similar workload (around 36000 gridcells) to 42 CPU cores (using 42 MPIs, each handles 840 gridcells (20 copies of the 42 AmeriFlux sites) on the same Summit node (2 CPU cores are reserved for system service tasks).

### *C. SPEL functions on ELM function parameters operations and management*

SPEL provides essential functions to manage, track, and evaluate the ELM function parameters for the standalone ELM unit test. SPEL also tracks the data dependency among parallel loops inside each ELM test module for further performance tuning.

*1) ELM function parameters declaration and collection operations:* To make the ELM global variables more manageable, standardized templates are used in SPEL to allocate, deallocate and initialize all the global variables, such as state, flux, and water variables. To better address the tracking and debugging challenges caused by the extensive use of global variables in ELM, we design a read and a write module in the SPEL to collect the values of the global variables from both the reference simulation and unit test experiments. Specifically, we use the write module to save input variables from reference simulation, use the read module to load the input parameters for the test modules of ELM functions, and use a specific verification module to save only the output parameters from unit test experiments for analysis.

TABLE II
GLOBAL VARIABLES FOR SURFACE PROPERTIES

| | **Gridcell** | **Topo** | **Land** | **Column** | **Veg** |
|---|---|---|---|---|---|
| **Site info** | Variables including area, lat, daylength, max-daylength, etc. | Variable including area, lon, lat, wtgcell, aspect, slope, elevation, emissivity, surfalb | Variables including type, lake, urban, soil, glacier, canyon-hwr, wtroad-perv, z0town | Variables including type, active_flag, layers, topo-slope, topostd, nlevbed, snl,z , lakedepth, hydro-active | Variables including type, location, active_flag, isveg, isbare-ground, isfates |
| **Subgrid** | Grid_id and the size of subd-grids (topo, landunit, column, veg) | Topo_id, reference to the upper level grid components (gridcell) and the size of subgrids (landunit, column, and veg) | Land_id, reference to the upper level grid components (gridcell, topography) and the size of subgrids (column and veg) | Col_id, reference to the upper level grid components (gridcell, top, landunit) and the size of subgrids (veg) | Veg_id, reference to the upper level grid components (gridcell, top, landunit, column) |

*2) ELM function parameters management and handling:* SPEL divides the ELM function parameters (most are global variables/arrays) into two categories: surface properties of ELM gridcell and process variables associated with ELM functions. The surface property parameters represent the surface properties of ELM gridcells (Table II) and are needed by every ELM function unit test. The surface property parameters of the 42 gridcells in the reference simulation were duplicated

to make synthesized data set for the ELM unit test at any user-defined size.

The process variables are used by individual ELM function to model the system mechanisms. SPEL generates a complete list of these variables used by all ELM functions. For a given ELM function, we only activate a subset of parameters that are used by the target function. Table III illustrates the process variables used by LakeTemperature.

TABLE III
GENERATED INPUT/OUTPUT FOR LAKETEMPERATURE

| Type | Input | Output |
|---|---|---|
| **solar radiation vars** | fsds_nir_i_patch, fsds_nir_d_patch, fsr_nir_d_patch sabg_lyr_patch, sabg_patch fsr_nir_i_patch | |
| **column-level physical proprieties** | lakedepth, dz, z, zi, z_lake, snl, dz_lake | |
| **column-level energy state** | t_lake, t_soisno, t_grnd | t_lake, t_soisno, hc_soi, hc_soisno |
| **column-level energy flux** | | imelt, eflx_snomelt errsoi |
| **column-level water state** | h2osno, h2osoi_liq, snow_depth, h2osoi_ice, | frac_iceold, h2osno, h2osoi_liq, snow_depth, h2osoi_ice |
| **veg-level energy flux** | eflx_sh_tot, eflx_soil_grnd, eflx_gnet, eflx_sh_grnd | eflx_sh_tot, eflx_soil_grnd, eflx_gnet, eflx_sh_grnd |
| **lake state vars** | lake_raw_col, ws_col, ks_col, etal_col, lake_icethick_col, lake_icefrac_col | lakeresist_col, betaprime_col, savedtke1_col, lake_icethick_col, lake_icefrac_col |
| **Misc** | veg_pp%column | grnd_ch4_cond_col |

LakeTemperature calculates the temperature solution of the snow, lake body (water and/or ice), soil, and bedrock system. The governing equation is a partial differential equation of temperature over time and is associated with volumetric heat capacity, thermal conductivity, and solar radiation penetrating to the depth of a lake. The system is discretized into layers, including 10 lake layer, 5 snow layers, and 10 soil layers. LakeTemperature also models the lake phase change (PhaseChange_Lake) governed by the energy conservation equation that is associated with volumetric heat capacity, latent heat of fusion per unit volume, snow mass, and energy flux at each lake layer [19].

*3) ELM function parameters verification:* The SPEL's verification procedure contains two phases. The first phase to verify the output from the standalone test module on the host is the same as the output from the reference simulation. The second phase is to verify the two outputs from the standalone test modules (on CPU and GPU) are equivalent. Specifically, SPEL contains a data analysis module to verify the outputs from the test modules (both CPU and GPU-based) are the same as the output from the reference simulation. Most of the ELM functional parameters (arrays) are defined as double precision, and a tolerance value of $10^{-10}$ is used in the element-wise comparison of these function parameters (arrays) (such as the state and flux variables at different levels of landscape components).

*4) Data dependency among ELM function parameters:* SPEL also provides functions to track the function parameters' dependency inside ELM test modules so that we can track the parameters access mode (read-only (ro), write-only(wo), and read-write(rw) between parallel loops. The information is valuable for further parallel algorithm design with fewer synchronizations. For example, Fig. 2 shows access patterns of global variables within parallel loops in the LakeTemperature, which has seven explicit synchronizations (enforced by acc_wait and represented as vertical lines). Among these synchronizations, only two synchronizations (represented by two thick blue vertical lines) are mandatory, therefore, we can potentially reduce the number of synchronizations (from 7 to 2) and enable asynchronous kernel launch to achieve better performance. Furthermore, we can use the same tracking function to analyze the data (function parameters) dependency between different test modules.

### D. ELM function unit test via SPEL

SPEL modifies the main ELM function (elm_drv) to create the Unit Test Driver (UTD) for the ELM unit test. For each target ELM function, SPEL integrates the UTD, target ELM function, and associated function parameters to generate a standalone unit test program. The program can be executed over a user-defined scale (user-defined number of gridcells) with a given number of MPI processes.

SPEL also provides essential functions to automatically generate GPU-ready ELM test unit programs (specified by users) using OpenACC in three steps: 1) Deepcopy input function parameters into data regions on GPUs, 2) Generate GPU-ready module with routine and parallel loop directives, 3) Create subroutine to transfer the output function parameters from GPUs for comparison. We use LakeTemperature as an example to illustrate the procedure of standalone unit test program generation via SPEL (Fig. 3). The GPU-ready code generation has the following steps:

- Tracks all subroutines and creates a call tree, adding "routine seq" to all subroutines within LakeTemperature and commenting out the necessary functions in the non-portable list (Table I).
- Create unstructured data regions to manage device data with deepcopy. SPEL uses deepcopy to move the data in and out of the unstructured data regions with *!$acc enter data copyin* and *!$acc update self* over the input and output parameters shown in Table III.
- Insert "parallel loop" directive so that the Unit Test Driver can launch LakeTemperature test modules in parallel across gridcells. At this step, SPEL does not change the structure and algorithms of the LakeTemperature test module.
- If the performance of LakeTemperature unit test module is not satisfied, other SPEL scripts and predefined tem-

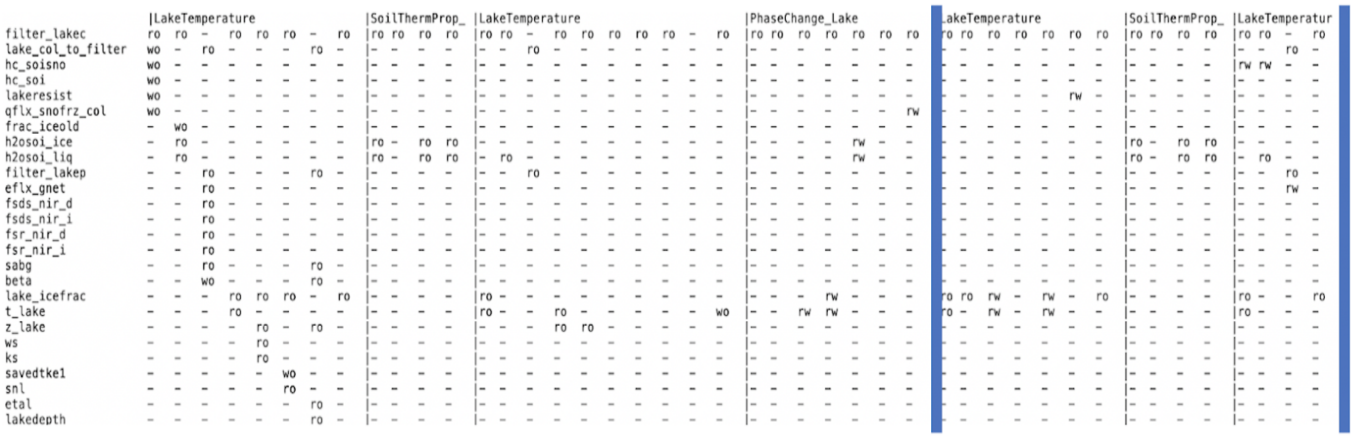

Fig. 2. Access patterns of function parameters in parallel loops of LakeTemperature. The first column is the variable name. The first row contains the ELM function names. Vertical lines represent explicit synchronization (acc_wait). Each fine column represents one parallel loop. Possible values for each table field are read-only (ro), write-only (wo), read-write (rw), and not-used (-).

plates are applied to improve its performance (Section IV-G).

It is also worth mention that we have developed SPEL scripts to clean up incompatibility syntax errors related to specific compiler or OpenACC implementation. For example, the FORTRAN compilers used do not support calling class methods as type-bound procedures on a device (i.e., call photosyns_vars%TimeStepInit(args*)), so we have to replace these function calls with direct calls (i.e., call TimeStepInit(args*)).

**Algorithm 1** LakeTemperature Unit Test

**Input:** gridcells, gridcells per process
INITALIZEMPI
INITIALIZEELM(data_file) ▷ Modified by SPEL
  initGlobalVars ▷ From Dependency Analysis
  readSurfaceProp(data_file)
  domainDecomp
READSIMDATA(data_file)
**acc enter data copyin**
LAKETEMPERATURE(args)
VERIFYLAKETEMP(output_file)

Fig. 3. A standalone LakeTemperature unit test program via SPEL.

.

### *E. End-to-end ELM simulation and performance profiling*

With SPEL, we have ported the entire ELM code (around 1000 subroutines, except for these listed in Table 1) onto GPUs using routine and parallel loops over gridcells. We generated a collection of ELM test modules (on both CPUs and GPUs), then we insert these test modules into ELM code to make the end-to-end simulation. SPEL tracks the execution time of the ELM functions on CPUs and GPUs. Table IV shows the execution time of ELM functions on the CPUs and GPUs of a single fully-loaded Summit node (with around 36000 gridcells). The timing data is collected in a single timestep (on January 1st) within the end-to-end ELM simulation.

Half of the ELM functions are with satisfactory performance after the development of OpenACC routine and parallelizing loops over gridcells. The execution times of 9 (out of 19) ELM functions (in white) are under 60 ms. The execution times also include the time used for moving the GPU kernels onto GPU memory at the first timestep.

TABLE IV
EXECUTION TIME OF ELM FUNCTIONS IN AN END-TO-END SIMULATION USING A SINGLE FULLY-LOADED SUMMIT NODE

| Function Group | GPU(ms) | CPU(ms) |
|---|---|---|
| **DecompVertProfiles** | 3.81 | 25.3 |
| **CNPInitSum** | 175 | 5.03 |
| **DynSubgrid** | 296 | 35.2 |
| **ColBalanceCheck** | 14.7 | 2.01 |
| **BiogeophysSetup** | 11.8 | 1.08 |
| **Radiation** | 275 | 1.07 |
| **CanopyTemp** | 0.946 | 1.36 |
| **BareGroundFlux** | 3.05E-02 | 16.1 |
| **CanopyFlux** | 5.52 | 142 |
| **UrbanFlux** | 3.08 | 4.18 |
| **LakeFlux** | 0.824 | 5.45 |
| **LakeTemperature** | 13.8 | 9.00 |
| **Dust** | 268 | 8.07 |
| **SoilFlux/p2c** | 58.8 | 3.42 |
| **Hydro-Aerosol** | 1750 | 18.5 |
| **SnowAgeGrain** | 0.015 | 20.264 |
| **EcosysDynNoLeaching** | 66.3 | 99.1 |
| **AnnualUpdate** | 7.66 | 0.015 |
| **HydroDrainage** | 37.75 | 1.23 |

By placing parallel loops over gridcells and launching ELM subroutine via routine seq, we achieve good performance on some ELM functions directly, but for subroutines with many internal loop structures and nested function calls, performance can be extremely slow. For example, the EcosystemDynNoLeaching and LakeTemperature took over 7600 and 1900 ms per timestep, respectively. Through a series of optimiza-

tions, we have successfully improved the execution times of these two ELM functions to 66 and 14 ms, respectively (see Table IV).

Currently, we have already fine-tuned 5 ELM functions (EcosystemDynNoLeaching, LakeTemperature, CanopyFLux, UrbanFlux, and LakeFlux) (in lightgray) (More information on performance optimization can be found in Section IV-G). There are another 5 ELM functions (in gray) that need to be optimized. It is important to note that the single timestep ELM simulation time varies dramatically and depends on many factors, such as the location of gridcells, vegetation types, and growth seasons. Therefore, comprehensive profiling over a longer period of time (months to years) is needed for definitive performance comparison between the CPU and GPU implementations.

### F. Memory consumption of ELM simulation

We have tracked the GPU memory consumption of the ELM simulations. Table V shows the available GPU memory after data copy and ELM kernel computing in two ELM simulations over 1000 and 6000 gridcells, respectively.

TABLE V
AVAILABLE GPU MEMORY DURING EXECUTION

| Phase | 1000 Gridcells | 6000 Gridcells |
|---|---|---|
| After Data Copy | 14.22GB | 4.72GB |
| After Kernels Compute | 12.16GB | 2.40GB |

From Table V, we can find the following conclusions: in average, the user data of every 1000-gridcell simulation requires around 1.9 GB of memory (i.e., approximately (14.22-4.72)/5) and the GPU kernels take around 2.06 GB and 2.3 GB of memory in the 1000 gridcell and 6000 gridcell cases, respectively (approximately available memory after data copy minus available memory after kernel compute). All the ELM kernels are residents in the memory during the ELM simulation, which was confirmed by the essentially unchanged amount of available GPU memory after the first timestep of simulation in both experiments with different numbers of gridcells.

### G. Advanced performance tuning of ELM unit test programs

Three techniques have been applied to improve the performance of the ELM test modules: 1) reduce memory footprint, 2) increase parallelism across ELM subgrid, and 3) accelerate internal loops using reduction or atomic clause. SPEL modifies the ELM source code to automatically implement the first two techniques. SPEL reduces the memory footprint with customized functionality being tailored to the ELM data structure and the filters. The SPEL functions insert parallel loop directives over loops inside subroutines, using the "collapse(n)" for tightly nested loops and defaulting to adding "!$acc loop seq" over non-nested inner loops. Our scripts perform a simple check for reduction operations and logs the loop location to be investigated for proper loop acceleration. Testing with the verification module is still necessary to resolve more subtle race-conditions.

*1) Reduce memory footprint:* Each NVIDIA V100 has 16 GB of memory to contain all the data and ELM kernels. Memory allocation operations on GPU are more expensive than those on the host, so Many methods have been deployed to reduce the size of local variables in test modules, such as converting arrays into scalars and compressing the sparse arrays into dense arrays [10]. We have reduced memory consumption of several ELM functions, such as EcosystemDynNoLeaching, CanopyFlux and LakeTemperature. For example, we have condensed sparse matrix in the LakeTemperature. These 42 Ameriflux sites only contain 32 lake columns, but the original code use arrays of 17*42 columns to represent the lake columns. Condensing arrays to use 32 columns results a 95% reduction in memory required for the local arrays.

*2) Reconstruct/flatten parallel loops:* For ELM functions with many internal loop structures and nested function calls, we need to restructure these routines to "flatten" their internal loops at column and patch levels and group them under different parallel loop constructs. The technique has also been applied to develop several GPU-ready modules of EcosystemDynNoLeaching[10], UrbanFlux, LakeFlux and LakeTemperature. As a specific example, we have reconstructed the parallel loop to columns (from gridcells) in LakeTemperature. This technique helped to achieve a performance of 13.8 ms, improved from 1930 ms with the automatically generated test module with routine and parallel loop over gridcells.

*3) Accelerate internal loop with reduction or atomic:* For subroutines that must have internal loops, we deploy reduction and atomic operations to improve the performance. The reduction operation is very efficient in many cases to efficiently utilize the GPU threads [10]. For the illustration purpose, we present two code segments (Fig. 4) in LakeTemperature that uses OpenACC atomic to set Lake Filters and uses reduction for energy calculation (such as heat capability and thermal conductivity).

## V. CONCLUSIONS AND DISCUSSION

The paper presented a generic method for porting large-scale scientific code onto accelerators using compiler directives within a FUT framework. This method contains four major steps: 1) pre-porting software component analysis, 2) FUT framework creation, 3) standalone unit test program generation, and 4) advanced optimization and performance tuning.

We have implemented the method and developed a software tool (SPEL), for porting the E3SM Land Model (ELM) onto GPUs in Summit. SPEL created a FUT framework for ELM, broke the ELM code into pieces, and generated a collection of GPU-ready test modules of ELM functions. SPEL also created standalone ELM test modules for easy code verification and further performance improvement. Several advanced techniques have been applied to improve the performance of several ELM test modules.

**Algorithm 2** Set Lake Filter (with atomic)

```
fn = 0
!$acc parallel loop gang vector &
!$acc private(fidx) present(filter(:)) copy(fn)
for each col do                          ▷ Go through each column
    l ← landunit_from_column
    if l has lake then
        !$acc atomic capture
        fn ← fn + 1
        fidx ← fn
        !$acc end atomic
        filter(fidx) ← col
```

**Algorithm 3** Energy calculation (with reduction)

```
!$acc parallel loop gang worker private(sum)
for each fc in lake columns do           ▷ Using lake filter
    sum ← 0.0
    !$acc loop vector reduction(+:sum)
    for each j in lake layers do
        sum ← sum + E(fc, j)             ▷ E: energy variables
    heat(fc) ← sum
```

Fig. 4. Code examples with atomic/reduction clauses in LakeTemperature.

We have ported the entire ELM code (over 1000 subroutines) onto GPUs. Half of the GPU-ready ELM functions come with satisfactory performance after deploying OpenACC routine and parallel loops over gridcells. With advanced optimization techniques, we have archived great performance in several ELM functions (EcosystsemDyn, LakeTemperature, CanopyFlux, LakeFlux) on GPU, while there are still several ELM functions that need to be further optimized.

SPEL plays a critical role in our ultra-high-resolution ELM development to support the high-fidelity land simulation (1km-resolution) over North America using Summit. We are developing SPEL functions and experimenting the ELM code porting on Summit and Crusher using OpenMP. At the automatic code generation phase, we may replace "!$acc routine seq" with "!$omp declare target" directive, and insert the "!$omp parallel do loop" over gridcells. In the advanced optimization phase, the SPEL scripts for memory footprint reduction and parallel do loop reconstruction also work with appropriate OpenMP syntax. However, whether deepcopy works well with OpenMP implementations and other compilers (besides the NVFORTRAN/PGI compiler) is unknown. Without deepcopy, we have to extend the scripts to generate manual deep copy for all the variables into the data regions explicitly and recursively. We are experimenting with OpenMP on Summit and Crusher now and will report our experience back to the community.

The study presented a generic method to port large-scale scientific code onto accelerators with compiler directives. When applying this tool to other codes, we will need to customize these SPEL functions to collect function parameters of individual subroutines using static code analysis utilities. Once we have the lists of function parameters, we can use the SPEL scripts to generate the code for data collection. We will need to create a Unit Test Driver based on the new code. Standalone unit test module generation and data region creation are straightforward with deepcopy functions. The deployment of "parallel do loop" and "routine seq" is also technically straightforward. In ELM, since all the simulations are executed over gridcell independently, good performances of parallel do loops of many ELM functions are expected. However, "parallel do loops" may not work well for some parts of the new code. The SPEL script can reconstruct parallel do loops and identify nested loops for reduction operations, and then users can decide how to deploy these operations case by case.

Deepcopy was extensively used in our study for data handling. Deepcopy caused issues in both standalone test module generation and end2end simulation (GPU). Depending on the data structures used in other codes and how compilers implement deepcopy, data copy functions may need to be verified in the new code porting process.

## VI. Code Availability

A copy of the SPEL and ELM source code is available at https://github.com/peterdschwartz/SPEL_OpenACC.

## References


[1] D. Wang, Y. Xu, P. Thornton, A. King, C. Steed, L. Gu, and J. Schuchart, "A functional test platform for the community land model," *Environmental Modelling & Software*, vol. 55, pp. 25–31, 2014.

[2] D. Wang, W. Wu, T. Janjusic, Y. Xu, C. Iversen, P. Thornton, and M. Krassovisk, "Scientific functional testing platform for environmental models: An application to community land model," in *International Workshop on Software Engineering for High Performance Computing in Science, 37th International Conference on Software Engineering*, 2015.

[3] P. Maris, C. Yang, D. Oryspayev, and B. Cook, "Accelerating an iterative eigensolver for nuclear structure configuration interaction calculations on gpus using openacc," *Journal of Computational Science*, vol. 59, p. 101554, 2022.

[4] A. Xu, L. Shi, and T. Zhao, "Accelerated lattice boltzmann simulation using gpu and openacc with data management," *International Journal of Heat and Mass Transfer*, vol. 109, pp. 577–588, 2017.

[5] R. Searles, S. Chandrasekaran, W. Joubert, and O. Hernandez, "Mpi+ openacc: Accelerating radiation transport mini-application, minisweep, on heterogeneous systems," *Computer Physics Communications*, vol. 236, pp. 176–187, 2019.

[6] E. Wright, M. H. Ferrato, A. J. Bryer, R. Searles, J. R. Perilla, and S. Chandrasekaran, "Accelerating prediction of chemical shift of protein structures on gpus: Using openacc," *PLoS computational biology*, vol. 16, no. 5, p. e1007877, 2020.

[7] J. Vincent, J. Gong, M. Karp, A. Peplinski, N. Jansson, A. Podobas, A. Jocksch, J. Yao, F. Hussain, S. Markidis *et al.*, "Strong scaling of openacc enabled nek5000 on several gpu based hpc systems," in *International Conference on High Performance Computing in Asia-Pacific Region*, 2022, pp. 94–102.

[8] S. Zhang, H. Fu, L. Wu, Y. Li, H. Wang, Y. Zeng, X. Duan, W. Wan, L. Wang, Y. Zhuang *et al.*, "Optimizing high-resolution community earth system model on a heterogeneous many-core supercomputing platform," *Geoscientific Model Development*, vol. 13, no. 10, pp. 4809–4829, 2020.

[9] J. Y. Kim, J.-S. Kang, and M. Joh, "Gpu acceleration of mpas microphysics wsm6 using openacc directives: Performance and verification," *Computers & Geosciences*, vol. 146, p. 104627, 2021.

[10] P. Schwartz, D. Wang, F. Yuan, and P. Thornton, "Developing an elm ecosystem dynamics model on gpu with openacc," in *International Conference on Computational Science*. Springer, 2022, pp. 291–303.

[11] D. Wang, Y. Pei, O. Hernandez, W. Wu, Z. Yao, Y. Kim, M. Wolfe, and R. Kitchen, "Compiler technologies for understanding legacy scientific code: A case study on an acme land module," *Procedia Computer Science*, vol. 108, pp. 2418–2422, 2017.

[12] D. Van Heesch, "Doxygen: Source code documentation generator tool," *URL: http://www. doxygen. org*, 2008.

[13] W. Zheng, D. Wang, and F. Song, "Xscan: an integrated tool for understanding open source community-based scientific code," in *International Conference on Computational Science*. Springer, 2019, pp. 226–237.

[14] M. Winter, M. Parger, D. Mlakar, and M. Steinberger, "Are dynamic memory managers on gpus slow? a survey and benchmarks," in *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2021, pp. 219–233.

[15] J.-C. Golaz, P. M. Caldwell, L. P. Van Roekel, M. R. Petersen, Q. Tang, J. D. Wolfe, G. Abeshu, V. Anantharaj, X. S. Asay-Davis, D. C. Bader *et al.*, "The doe e3sm coupled model version 1: Overview and evaluation at standard resolution," *Journal of Advances in Modeling Earth Systems*, vol. 11, no. 7, pp. 2089–2129, 2019.

[16] S. Burrows, M. Maltrud, X. Yang, Q. Zhu, N. Jeffery, X. Shi, D. Ricciuto, S. Wang, G. Bisht, J. Tang *et al.*, "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*, vol. 12, no. 9, p. e2019MS001766, 2020.

[17] Y. Xu, D. Wang, T. Janjusic, W. Wu, Y. Pei, and Z. Yao, "A web-based visual analytic framework for understanding large-scale environmental models: A use case for the community land model," *Procedia Computer Science*, vol. 108, pp. 1731–1740, 2017.

[18] D. Wang, P. Schwartz, F. Yuan, P. Thornton, and W. Zheng, "Toward ultra-high-resolution e3sm land modeling on exascale computers," *Computing in Science and Engineering*, vol. in revision, 2022.

[19] D. M. Lawrence, R. A. Fisher, C. D. Koven, K. W. Oleson, S. C. Swenson, G. Bonan, N. Collier, B. Ghimire, L. van Kampenhout, D. Kennedy *et al.*, "The community land model version 5: Description of new features, benchmarking, and impact of forcing uncertainty," *Journal of Advances in Modeling Earth Systems*, vol. 11, no. 12, pp. 4245–4287, 2019.