Developing Ultrahigh-resolution E3SM Land Model for GPU systems *

Peter Schwartz[1],**[0000-0002-0852-5528], Dali Wang[1],**[0000-0001-6806-5108], Fengming Yuan[1][0000-0003-0910-5231], and Peter Thornton[1][0000-0002-4759-5158]

Environmental Sciences Division, Oak Ridge National Laboratory, Oak Ridge TN 37830, USA
{schwartzpd,wangd,yuanf,thorntonpe}@ornl.gov

Abstract. Designing and refactoring complex scientific code, such as the E3SM land model (ELM), for new computing architectures is challenging. This paper presents design strategies and technical approaches to develop a data-oriented, GPU-ready ELM model using compiler directives (OpenACC/OpenMP). We first analyze the datatypes and processes in the original ELM code. Then we present design considerations for ultrahigh-resolution ELM (uELM) development for massive GPU systems. These techniques include the global data-oriented simulation workflow, domain partition, code porting and data copy, memory reduction, parallel loop restructure and flattening, and race condition detection. We implemented the first version of uELM using OpenACC targeting the NVidia GPUs in the Summit supercomputer at Oak Ridge National Laboratory. During the implementation, we developed a software tool (named SPEL) to facilitate code generation, verification, and performance tuning using these techniques. The first uELM implementation for Nvidia GPUs on Summit delivered promising results: 1) over 98% of the ELM code was automatically generated and tuned by scripts. Most ELM modules had better computational performances than the original ELM code for CPUs. The GPU-ready uELM is more scalable than the CPU code on fully-loaded Summit nodes. Example profiling results from several modules are also presented to illustrate the performance improvements and race condition detection. The lessons learned and toolkit developed in the study are also suitable for further uELM deployment using OpenMP on the first US exascale computer, Frontier, equipped with AMD CPUs and GPUs.

Keywords: Exascale Energy Earth System Model · E3SM Land Model · Ultrahigh-Resolution ELM · OpenACC · Compiler Directives

* 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 and Experimental Computing Laboratory at the Oak Ridge National Laboratory, which are supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

** These authors contributed equally.

1 Introduction

State-of-the-art Earth system models (ESM) provide critical information on climate changes and advance our understanding of the interactions among natural and human systems and the Earth's climate. Energy Exascale Earth System Model (E3SM) is a fully coupled ESM that uses code optimized for the US Department of Energy's (DOE) advanced computers to address the most critical Earth system science questions[3]. Inside the E3SM framework, the E3SM Land Model (ELM) simulates the interactions among terrestrial land surfaces and other Earth system components. ELM were used to understand hydrologic cycles, biogeophysics, and ecosystem dynamics of terrestrial ecosystems[2].

We are in the process of developing large-scale, ultrahigh-resolution ELM (uELM) simulation targeting the coming exascale computers[10] for high fidelity land simulation at continental and global scales. One major challenge is to develop an efficient ELM code suitable for the accelerators (e.g., GPUs) within the hybrid computing architecture of these Exascale computers. We have developed a function unit test framework that takes the code into pieces, and completed several individual ELM module development [6]. This study systematically presents the data structures and data flow of uELM, and summarize the technical experience gained in the uELM development to support unprecedented ultrahigh resolution simulations (1 km x 1 km) using GPU systems.

We first analyze the dataflow and computational characteristics of the ELM code, then present our strategies to develop a GPU-ready uELM, and demonstrate our first uELM implementation on a pre-exascale computer using a code porting toolkit and OpenACC. At last, we demonstrated several performance results of the first uELM implementation using a synthesized dataset.

2 Terrestrial Ecosystem Data-oriented ELM simulation

2.1 ELM datatypes and globally accessible variables

Highly-customized landscape datatypes (gridcell, topographic unit, land cover, soil column, and vegetation) are used to represent the heterogeneity of the Earth's surface and subsurface [15]. The gridcells are geospatially explicit datatypes, the subgrid components within gridcells (topographic unit, landunit, columns, and vegetation) were spatially implicit and configured with gridded surface properties dataset.

As shown in Table 1, eleven groups of landscape datatypes (over 2000 global arrays) are designed to store the state and flux variables at gridcells and their subgrid components. The physical properties datatypes contain association information among subgrid components so that the energy, water, and CNP variables can be tracked, aggregated, and distributed among gridcells and their subgrid components. Beside the landscape datatypes, customized ELM process datatypes are also used to represent the biogeophysical and biogeochemical processes in the terrestrial ecosystems. Examples of these ELM process datatypes are Aerosol_type, Canopystate_type, Lake_type, Photosynthesis_type, Soil_type,

Table 1: Major ELM landscape and their public (globally accessible) variables.

Name	Variables	Associated grid components
cnstate_type	125	patch, column
column_energy_state, column_water_state, column_carbon_state, column_nitrogen_state, column_phosphorus_state, column_energy_flux, column_water_flux, column_carbon_flux, column_nitrogen_flux, column_phosphorus_flux	850	column
column_physical_properties	27	column, and association with other subgrid components
gridcell_energy_state, gridcell_water_state, gridcell_carbon_state, gridcell_nitrogen_state, gridcell_phosphorus_state, gridcell_energy_flux, gridcell_water_flux, gridcell_carbon_flux, gridcell_nitrogen_flux, gridcell_phosphorus_flux	170	gridcell
gridcell_physical_properties_type	33	gridcell, and association with other subgrid components
landunit_energy_state, landunit_water_state, landunit_energy_flux	6	landunit
landunit_physical_properties	22	landunit, and association with other subgrid components
topounit_atmospheric_state, topounit_atmospheric_flux, topounit_energy_state	35	topounit
vegetation_energy_state, vegetation_water_state, vegetation_carbon_state, vegetation_nitrogen_state, vegetation_phosphorus_state, vegetation_energy_flux, vegetation_water_flux, vegetation_carbon_flux, vegetation_nitrogen_flux, vegetation_phosphorus_flux	930	Vegetation (patch)
vegetation_properties_type	120	vegetation
vegetation_physical_properties	16	vegetation, and association with other subgrid components

and Urban_type. In total, these ELM process datatypes contained more than 1000 global arrays associated with gridcells and their subgrid components. All the ELM datatypes (functions and variables) are initialized and allocated as static, globally accessible objects on each computing (e.g., MPI) process after domain partitioning.

2.2 Domain decomposition and gridcell aggregation

At the beginning of a simulation, ELM scans through the computational domain and assigned unique id to each land gridcell, then distributes these land gridcells to individual MPI processes using a round-robin scheme to achieve a balanced workload [4]. ELM uses special datatypes (Table 2) to define the computational domain on each MPI process. The gridcells on individual MPI processes are aggregated together as clumps. Each clump stores the procesor_id (MPI rank) and the total number of subgrid components with their starting and ending positions. Each MPI process allocates contiguous memory blocks (arrays) to hold the ELM variables across all the gridcells and associated subgrid components. All the ELM variables are allocated and initialized as globally accessible arrays. The maximum number of subgrid components are allocated within each gridcell, a group of filters are generated at each timestep to track the active subgrid components inside gridcells.

Table 2: Customized datatypes to define computational domain

int, npes	The number of MPI processes
int, clump_pproc	Max number of clumps per MPI process
int, nclumps	The number of clumps on individual process
clump_type	Owner, size of gridcells(subgrid components), begin and end indexes of these gridcell(subgrid components) in each clump
processor_type	nclumps, clump_id, size of gridcells (subgrid components), begin and end indexes of these gridcells(subgrid components) in each mpi_process
bounds_type*	Data_type to store the size and the begin and end indexes of gridcells (subgrid components) in clumps or processes

*Bounds_type is used to store the total number of gridcell components, as well as the start and end indexes of these gridcell components in either clumps or MPI processes.

2.3 Terrestrial ecosystem processes in ELM

ELM simulates key biogeochemical and biogeophysical processes in the terrestrial ecosystems, and their interactions with atmosphere. The general flow of an ELM simulation starts with the water and energy budget calculation and carbon-nitrogen balance check at each gridcell. ELM simulates phenomena of hydrology, radiation, lakes, soil, aerosols, temperature, ecosystem dynamics, dust, and albedo (Table 3). At each timestep, ELM also updates vegetation structure and checked the mass and energy balance. Over 1000 subroutines are developed to represent these biogeochemical and biogeophysical processes in the terrestrial ecosystems.

Table 3: ELM processes and execution sequence

Vertical decomposition, Dynamic Subgrid, CNP and Water Balance check	Determine decomposition vertical profile, update subgrid weights with dynamic landcover, and check mass balance
Canopy Hydrology and Temperature	Canopy Hydrology, and determine leaf temperature and surface fluxes based on ground temperature from previous time step
Surface and Urban Radiation	Surface Radiation Calculation
Flux Calculation (BareGround, Canopy, Urban, Lake)	Calculate energy fluxes in gridcell components (bareground, canopy, urban, and lake)
Dust Emission and DryDep	Dust mobilization and dry deposition
LakeTemperature and Hydrology	Lake temperature and hydrology
SoilTemperature and Fluxes	Set soil/snow temperatures including ground temperature and update surface fluxes for new ground temperature
HydrologyNoDrainage	Vertical (column) soil and surface hydrology
LakeHydrology	Lake hydrology
AerosolMasses	Calculate column-integrated aerosol masses
SoilErosion	Update sediment fluxes from land unit
EcosystemDynNoLeaching	Ecosystem dynamics: Uses CN, or static parameterizations
HydrologyDrainage	Calculate soil/snow hydrology with drainage (subsurface runoff)
EcosystemDynLeaching	Ecosystem dynamics: with leaching
SurfaceAlbedo and UrbanAlbedo	Determine albedos for next time step
subgridAve, Vegstructupdate, AnnualUpdate, Water and CNP balance check	Performance averaging among subgrid components, Update vegetation, WaterCPN balance check
Lnd2atm, lnd2grc	Interaction with atm and glacier
Hist_htapes_wrapup, restFile_write	HistoryRestart files (output)

2.4 ELM simulation with computational loops

In our study, ELM is configured for land-only mode (i.e., driven by atmospheric forcing derived from observed datasets to predict ecosystem responses under past climatic scenarios). Several datasets have been developed to drive the land-only ELM simulation [8, 5, 13, 7]. The land surface has been a critical interface through which climate change impacts humans and ecosystems and how humans and ecosystems can affect global environmental change. The land surface is configured with surface properties datasets. The ELM simulations have several phases: the first phase is spin-up simulation to find the equilibrium states of terrestrial systems (may take a long period), the second phase is to simulate terrestrial ecosystems' responses to the historical atmospheric forcing (e.g., 1850 - present), and the third phase is to predict the ecosystem responses to future climatic scenarios.

The ELM simulations take half-hourly or hourly timesteps. At each timestep, ELM loops over each clump (aggregated gridcells and their active subgrid components) to calculate the changes of terrestrial ecosystem states and fluxes (more than 3000 global arrays) through the ELM processes (Table 3). The majority of computation over gridcells, their subgrid components, as well as the nutrient elements (that is carbon, nitrogen, and phosphorus (CNP)) are independent, with exception of several functions inside ecosystem dynamics, such as carbon-nitrogen allocation and soil litter decomposition[12]. In these cases, the CNP functions are limited by the carbon-nitrogen ratio and carbon-phosphorus ratio. The computational complexity of ELM come from the accessing, tracking, calculating, updating, and conservation-law checking (mass and energy) of these large number of state and flux variables (global arrays) within the ELM datatypes across the entire computational domain. To ensure mass and energy conservation, water, energy, and CNP states are aggregated and thoroughly checked in each gridcell at every timestep. At the end, ELM generates rich simulation results: by default, each monthly history output contains more than 550 variables representing the terrestrial ecosystem's geophysical, and biogeochemical processes (such as water, energy, carbon, nitrogen, and phosphorus cycles) at every land gridcell.

In summary, ELM is a gridcell independent, data-centric terrestrial ecosystem simulation that contains massive computational loops over gridcells and their subgrid components (Figure 1). ELM has more than 1000 subroutines, none of which are computationally intensive.

3 GPU-ready uELM development on Summit

We first present design considerations for the GPU-ready uELM development, then we describe the first uELM implementation on a pre-exascale computer with Nvidia GPUs.

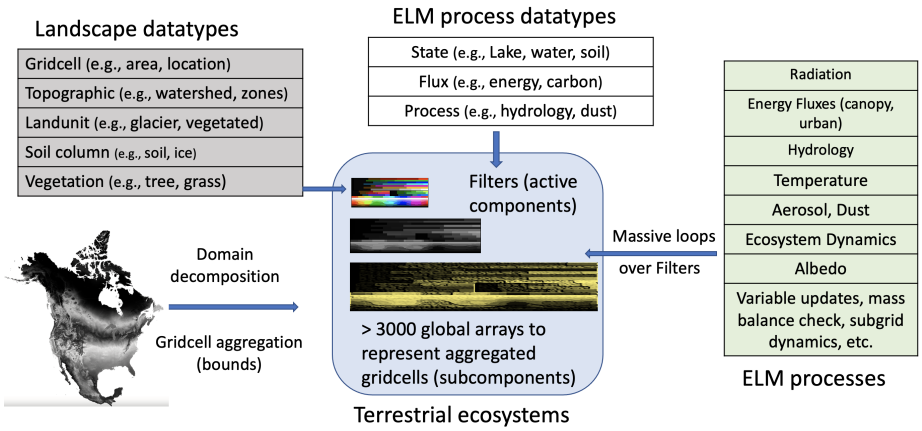

Fig. 1: Data-centric ELM simulation with numerous computational loops

3.1 uELM design considerations

ELM has been an integral part of E3SM simulations. We want uELM to maintain the maximum compatibility with the current E3SM framework and software engineering practices, including ELM datatypes and subroutines, data exchange with other E3SM components via a coupler, parallel IO, and model setup/configuration with the Common Infrastructure for Modeling the Earth.

Domain partition uELM is designed to support simulations over extremely large computational domains. For example, at a 1km by 1km resolution, the North America region contains around 22.5 million of land gridcells, that is approximately 230 times larger than the current global high resolution ELM simulation at a 0.5 by 0.5 degree resolution. Current implementation of uELM still support static domain decomposition with balanced workloads using a round-robin scheme. The size of subdomain (number of gridcells) are calculated to ensure all ELM inputs, datatypes (with global arrays), and code kernels present in GPU memory for many timesteps to achieve better performance. For example, each 16GB NVidia V100 GPU is used for uELM simulations over a subdomain of around 6000 gridcells. Technically, to increase the parallel execution performance on GPU, each clump in uELM contains 1 gridcell and each uELM subdomain contained around 6000 clumps. Approximately 700 computing nodes of Summit (4200 GPUs) will be needed for the uELM simulation over the entire North America region (22.5 millions of land gridcells).

Code porting and data copy As ELM contained more than 1000 computationally non-intensive subroutines, compiler directives (OpenACC or OpenMP) are selected for code porting[6], instead of the GPU-ready math libraries[1] or a new programming language[14]. To better handle these highly customized ELM

datatypes, we use unstructured data regions to store these global arrays in GPU shared memory, and applied deepcopy function extensively to expedite the data movements between CPUs and GPUs. All global variables are copied to device only at the beginning of simulation, but for each subroutine, the local variables are created and deleted every time they're called. Python scripts are developed to automatically generate the enter and exit data clauses for each subroutine.

Memory reduction To efficiently use GPU memory and save time on host-to-device data transfers, we use scripts to systematically reduce the size of local arrays based on active filters for most of the uELM subroutines as well as eliminating arrays where possible. Memory reduction also increases the performance of accessing and updating ELM global arrays.

Parallel loop reconstruction and flattening Considering that the majority of ELM computation over gridcells(subgrid components) and nutrient elements (carbon, nitrogen, and phosphorus (CNP)) are independent, we reconstruct and flatten parallel loops inside uELM to improve performance. We also assign a large number of clumps (each containing one gridcell) onto each GPU for efficient parallel loop execution. For ELM functions with many internal loop structures and nested function calls, we reconstruct 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 many GPU-ready modules, including EcosystemDynNoLeaching[6], UrbanFlux, LakeFlux and LakeTemperature.

Variable summation and race condition Race conditions in the parallel constructs need to be inspected in the ELM water, energy, and mass aggregation and balance checking process, as lower level (such as patch) variables are summed into higher-level (column or gridcell) variables. Reduction clauses are used to increase the model performance with the same results. However, these parallel loop reconstructions require changes in loop logic that could not easily be inspected automated. Technically, we scan every uELM parallel do loop and detect race conditions based on following criteria:

1. a variable (an array or scalar) found on both sides of an assignment
2. the gridcell component that the variable is associated with was NOT the same gridcell component that was being looped over.
3. the variable had less indices than the number of loop variables (e.g., a(c) = a(c) + b(c,j) where both c and j are looped over.)
4. a new value depends on a prior values (e.g., a(c,j) = a(c,j-1) + other_ops)

If the first and one of the other three criteria are met (true), a do loop is flagged for manual inspection of race conditions by outputting the subroutine name and line number of the outermost loop.

3.2 uELM code generation and performance tuning on Summit

The computational platform used in the study is the Summit leadership computing system at the Oak Ridge National Laboratory. Summit has 4,608 computing nodes, most of them contain two 22-core IBM POWER9 CPUs, six 16-GB NVIDIA Volta GPUs, and 512 GB of shared memory.

We have developed a python toolkit, called SPEL to port the all ELM module onto Nvidia GPUs using OpenACC. SPEL is used to generate 98% of the uELM code automatically. For the complete of the paper, the general workflow of code generation with SPEL is summarized in Figure 2.

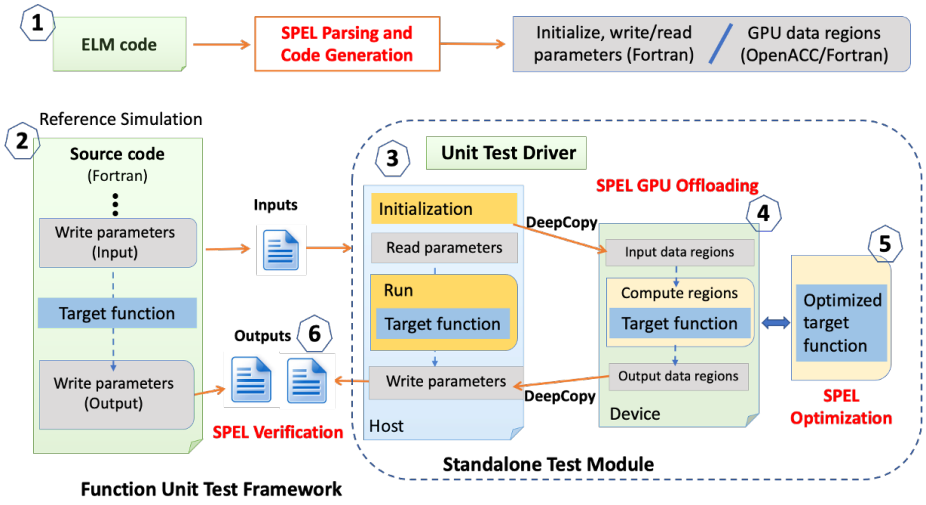

Fig. 2: uELM code porting workflow and SPEL functions

The SPEL workflow within a Functional Unit Testing framework [11, 9] contains six steps: 1) SPEL parses the ELM code and generates a complete list of ELM function parameters. For an individual ELM function, SPEL marks the active parameters generates Fortran modules to read and write, initialize, and offload parameters. 2) SPEL inserts the write modules before and after a target ELM function to collect the input and output parameters from a reference ELM simulation. 3) SPEL constructs a unit test driver for standalone ELM module test. The driver initializes and reads function parameters, executes the target ELM module, and saves the output. 4) SPEL generates GPU-ready ELM test modules with OpenACC directives. 5) SPEL optimizes the GPU-ready test module (e.g., memory reduction, parallel loop, and data clauses). And 6) SPEL verifies code correctness at multiple stages of the ELM module testing (CPU, GPU, and GPU-optimized).

After the standalone ELM models testing and performance tuning, we then conduct end2end code integration and overall performance tuning with SPEL's tuning function again.

4 Numerical experiments and performance evaluation

In this study, we use synthesized data from 42 AmeriFlux sites in the United States (https://ameriflux.lbl.gov/) to drive the uELM spin-up simulation for performance evaluation. On each Summit computing node, we launch 6 MPI processes, each managing one CPU core and one GPU. Around 6000 gridcells are assigned to each MPI process. The surface properties dataset are derived (downsampled) from a global 0.5x0.5 degree dataset. The GPU code validation is conducted by bit4bit comparison with the original CPU code at both individual module (function unit) level and the entire end2end simulation.

We present the overall execution time of the uELM simulation using the CPUs and GPUs of a single, fully-loaded Summit node (that is around 36000 gridcells are assigned to 6 GPUs or 42 CPU cores). The timing data is collected from a single timestep (on January 1st) within the end-to-end ELM simulation. After that we focus on technical details of performance improvements via memory reduction, loop restructure and race condition detection. At last, we present profiling results of several modules using Nvidia's Nsight Compute and Nsight Systems to reveal machine-level performance improvements.

4.1 Execution time

Table 4 shows the execution time of GPU implementation along with the original CPU version of ELM on a single fully-loaded Summit node. The domain for each GPU is 5954 gridcells (142 sets of these 42 AmeriFlux sites) and the domain for each CPU core is 840 gridcells (20 sets of these 42 AmerFlux sites). Better performance had been achieved with most of ELM functions, except soilFlux and ecosystemDynLeaching that still have more rooms for improvements.

4.2 Memory reduction

Table 5 lists examples of memory reduction within several individual uELM modules, On average, we achieved over 95% of reduction rate. The forcing data and uELM datatypes, including all the global variables, take 11GB GPU memory when 6000 gridcells (approximate 1.8MB data per gridcell) is assigned to each GPU. The total memory utilization after the memory reduction is around 14.4 GB, safely under the total 16 GB memory capability of the Nvidia V100.

Memory reduction improved code performance significantly as majority of ELM kernels are not compute bound. Herein, we present the timing results of major code sections inside an ELM module (LakeTemperature) to illustrate the speedup through memory reduction (Table 6). The most significant speedup occurred in the Initialization section where device copies of local variables are created and initialized to certain values. Memory reduction increases the percentage of computing time from 23% to 38%. Profiling individual kernels

Table 4: Execution time of ELM functions in an single timestep end-to-end simulation using a single fully-loaded Summit node

Function names	GPU (millisecond)	CPU (millisecond)
DecompVertProfiles	3.81	25.30
dynSubgrid	27.96	35.17
ColBalanceCheck	14.68	2.01
Biogeophys setup	11.81	1.08
Radiation	1.40	1.07
CanopyTemp	0.95	1.36
CanopyFluxes	5.52	142
UrbanFluxes	3.08	4.18
LakeFluxes	0.82	5.45
LakeTemps	13.82	9.00
Dust	2.87	8.07
SoilFluxp2c	58.78	3.42
Hydro-Aerosol	18.70	18.54
EcosystemDynNoLeaching	66.25	99.10
EcosystemDynLeaching	110.50	23.16

Table 5: Examples of memory reduction within individual uELM modules

Module	Before reduction	After reduction	Percentage
LakeTemperature	50 arrays * 704 cols	45 arrays * 32 cols	4%
SoilLittVertTransp	15 arrays * 704 cols	6 arrays * 42 cols	2%
UrbanFluxes	73 arrays * 200 landunits	43 arrays * 21 landunits	6%
HydrologyNoDrainage	63 arrays * 704 cols	63 arrays * 40 cols	5%
CanopyFluxes	63 arrays * 1376 pfts	59 arrays * (0 or 4) pfts	$< 1\%$
UrbanRadiation	21 array * 200 landunits	No arrays, all scalars	$< 1\%$

with Nsight™Compute confirm that the uncoalesced global accesses decreased or even disappeared, allowing better GPU utilization. For example, after memory reduction, kernels in the TriDiag Section require 4-8x less sectors and have less wasted cycles. These kernels also have a higher SM usage of approximately 25% (increased from 2% SM usage prior to memory reduction). Further performance improvements are also possible through fine-tuned pipelining and transpose of global and local arrays to accommodate new loop order.

Table 6: Timing results of major code sections in LakeTemperature

Code sections	Before reduction (millisecond)	Data reduction (millisecond)	Speedup
Initialize	20.8	6.82	3.05
Diffusion	0.36	0.20	1.76
SoilThermProp_Lake	1.23	0.48	2.57
EnergyCheck	0.14	0.13	1.12
Interface	0.32	0.13	2.43
TriDiag	0.58	0.17	3.45
PhaseChange_Lake	0.68	0.50	1.36
Mixing1stStage	1.12	1.11	1.00
MixingFinal	1.13	1.22	0.92
Diagnostic	0.77	0.22	3.53

4.3 Advanced performance improvements and race condition detection

Many ELM subroutines contain nested loop structures and nested function calls that yielded poor performance with the routine directive, and so we use SPEL to further reconstruct many of these subroutines. We remove the external gridcell loop so that the internal loops can be as large as possible and accelerated with OpenACC parallel constructs. ELM subroutines also contain many summation operations that aggregate states and fluxes from lower-level gridcell components into higher-level gridcell components. To achieve better performance, we reorder the loops and increased the number loops to ensure the outer loops are independent, then we use gangs and workers for the outer loops (collapsed) and vectors for the innermost loop with OpenACC reduction clause. Further parallelism is also enhanced by reconstructing the parallel loop over similar operations among CNP cycles and history buffer (output) calculation. Good examples of these loop reconstruction can be found in a previous paper[6].

Race condition cases are detected in many ELM modules by SPEL. For illustration purposes, we present the race condition detection in three ELM modules(Table 7). Among 144 parallel loops in three modules, 46 cases (approximately 30%) are flagged by SPEL for manual investigation. Only 2 of these 46 flagged cases (LakeTemperature:L386 and LakeTemperature:L666) are not confirmed as a race condition as they either involve a scalar variable (sabg_nir) that

Table 7: Examples of race conditions within several uELM modules

Modules	Loops*	Race condition flagged by SPEL ** (subroutine:line_number [variables involved])
LakeTemperature	14 (40)	LakeTemperature:L386 ['sabg_nir', 'sabg_nir'] LakeTemperature:L616 ['temp', 'temp', 'ocvts(fc)'] LakeTemperature:L666 ['zx(fc,j)'] phasechange_lake:L1693 ['qflx_snomelt(c)'] phasechange_lake:L1752 ['qflx_snofrz_col(c)'] LakeTemperature:L906 ['icesum'] LakeTemperature:L1156 ['temp', 'lakeresist(c)'] LakeTemperature:L1295 ['sum1']
Hydrology-NoDrainage	29 (90)	buildsnowfilter:L2891 ['snow_tot', 'nosnow_tot'] snowwater:L343 ['qout(fc)'] compute_effecrootfrac_and_verttransink:L1221 ['num_filterc'] soilwater_zengdecker2009:L452 ['smp1', 'vwc_zwt(fc)'] snowcompaction:L709 ['ddz1_fresh', 'ddz1', 'ddz3', 'ddz3', 'burden(fc)'] combinesnowlayers:L1130 ['sum1', 'sum2', 'sum3', 'sum4'] dividesnowlayers:L2001 ['dztot(fc)', 'snwicetot(fc)', 'snwliqtot(fc)'] HydrologyNoDrainage:L553 ['sum1', 'sum2', 'sum3']
UrbanFluxes	2 (14)	UrbanFluxes:L441['fwet_roof', 'fwet_road_imperv', 'taf_numer(fl)', 'taf_denom(fl)', 'qaf_numer(fl)', 'qaf_denom(fl)', 'eflx_wasteheat(l)', 'zeta', 'zeta', 'iter'] UrbanFluxes:L879['eflx_scale', 'qflx_scale']

* The first number showed the number of loops were flagged by SPEL for further investigation of race conditions. The second number showed the total number of loops in the module that were checked by SPEL

** For illustration purposes, we only listed up to 8 examples of race conditions in each module.

is made private or a special array index is used for a multi-dimensional array (e.g. zx).

5 Conclusions

The paper reviewed the ELM software, presented design strategies and technical approaches to develop a data-oriented, GPU-ready uELM using compiler directives. We also described a software tool (SPEL) to code porting and the first implementation of uELM using OpenACC on the Summit supercomputer. The lessons learned and toolkit (SPEL) developed in the study will be used for the uELM deployment using OpenMP on the first US exascale computer, Frontier, equipped with AMD CPUs and GPUs.

References

1. Bertagna, L., Guba, O., Taylor, M.A., Foucar, J.G., Larkin, J., Bradley, A.M., Rajamanickam, S., Salinger, A.G.: A performance-portable nonhydrostatic atmospheric dycore for the energy exascale earth system model running at cloud-resolving resolutions. In: SC20: International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 1–14. IEEE (2020)
2. Burrows, S., Maltrud, M., Yang, X., Zhu, Q., Jeffery, N., Shi, X., Ricciuto, D., Wang, S., Bisht, G., Tang, J., 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 **12**(9), e2019MS001766 (2020)
3. 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.: The doe e3sm coupled model version 1: Overview and evaluation at standard resolution. Journal of Advances in Modeling Earth Systems **11**(7), 2089–2129 (2019)
4. Hoffman, F.M., Vertenstein, M., Kitabata, H., White III, J.B.: Vectorizing the community land model. The International Journal of High Performance Computing Applications **19**(3), 247–260 (2005)
5. Qian, T., Dai, A., Trenberth, K.E., Oleson, K.W.: Simulation of global land surface conditions from 1948 to 2004. part i: Forcing data and evaluations. Journal of Hydrometeorology **7**(5), 953–975 (2006)
6. Schwartz, P., Wang, D., Yuan, F., Thornton, P.: Developing an elm ecosystem dynamics model on gpu with openacc. In: Computational Science–ICCS 2022: 22nd International Conference, London, UK, June 21–23, 2022, Proceedings, Part II. pp. 291–303. Springer (2022)
7. Thornton, P.E., Shrestha, R., Thornton, M., Kao, S.C., Wei, Y., Wilson, B.E.: Gridded daily weather data for north america with comprehensive uncertainty quantification. Scientific Data **8**(1), 1–17 (2021)
8. Viovy, N.: Cruncep version 7-atmospheric forcing data for the community land model. Research Data Archive at the National Center for Atmospheric Research, Computational and Information Systems Laboratory **10** (2018)
9. Wang, D., Wu, W., Janjusic, T., Xu, Y., Iversen, C., Thornton, P., Krassovisk, M.: 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)
10. Wang, D., Schwartz, P., Yuan, F., Thornton, P., Zheng, W.: Towards ultra-high-resolution e3sm land modeling on exascale computers. Computing in Science & Engineering (01), 1–14 (2022)
11. Wang, D., Xu, Y., Thornton, P., King, A., Steed, C., Gu, L., Schuchart, J.: A functional test platform for the community land model. Environmental Modelling & Software **55**, 25–31 (2014)
12. Xu, Y., Wang, D., Janjusic, T., Wu, W., Pei, Y., Yao, Z.: 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 (2017)
13. Yoshimura, K., Kanamitsu, M.: Incremental correction for the dynamical downscaling of ensemble mean atmospheric fields. Monthly weather review **141**(9), 3087–3101 (2013)

14. Zhang, S., Fu, H., Wu, L., Li, Y., Wang, H., Zeng, Y., Duan, X., Wan, W., Wang, L., Zhuang, Y., et al.: Optimizing high-resolution community earth system model on a heterogeneous many-core supercomputing platform. Geoscientific Model Development **13**(10), 4809–4829 (2020)
15. Zheng, W., Wang, D., Song, F.: Xscan: an integrated tool for understanding open source community-based scientific code. In: International Conference on Computational Science. pp. 226–237. Springer (2019)