Developing an ELM Ecosystem Dynamics Model on GPU with OpenACC

Peter Schwartz, Dali Wang(✉), Fengming Yuan, and Peter Thornton

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

Abstract. Porting a complex scientific code, such as the E3SM land model (ELM), onto a new computing architecture is challenging. The paper presents design strategies and technical approaches to develop an ELM ecosystem dynamics model with compiler directives (OpenACC) on NVIDIA GPUs. The code has been refactored with advanced OpenACC features (such as deepcopy and routine directives) to reduce memory consumption and to increase the levels of parallelism through parallel loop reconstruction and new data structures. As a result, the optimized parallel implementation achieved more than a 140-time speedup (50 ms vs 7600 ms), compared to a naive implementation that uses OpenACC routine directive and parallelizes the code across existing loops on a single NVIDIA V100. On a fully loaded computing node with 44 CPUs and 6 GPUs, the code achieved over a 3.0-times speedup, compared to the original code on the CPU. Furthermore, the memory footprint of the optimized parallel implementation is 300 MB, which is around 15% of the 2.15 GB of memory consumed by a naive implementation. This study is the first effort to develop the ELM component on GPUs efficiently to support ultra-high-resolution land simulations at continental scales.

Keywords: Earth system models · Exascale energy earth system model · E3SM land model · OpenAcc · Functional unit testing · Ecosystem dynamics

1 Introduction

The Exascale Energy Earth System Model (E3SM) is a fully coupled Earth system model that uses code optimized for the Department of Energy (DOE)'s advanced computers to address the most critical scientific questions facing the

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.

US and the society [5]. The E3SM contains several community models to simulate major Earth system components: 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 has been used to understand hydrologic cycles, biogeophysics, and ecosystem dynamics [3]. The ELM software has several distinguishing computational features: 1) all the biophysical and biochemical processes are simulated on individual land surface units (i.e., gridcell) independently; 2) highly customized globally accessible, hierarchical data structures are used to represent the heterogeneity of Earth's landscape; 3) none of the subroutines are computationally intensive [11].

Most current high-end supercomputers use heterogeneous hardware with accelerators [2]. The E3SM, consisting of millions of lines of code developed for traditional homogeneous multicore processors, cannot automatically benefit from the advancement of these supercomputers. Refactoring and optimizing the E3SM models for new architectures with accelerators is challenging but inevitable.

Rewriting a large-scale legacy code in a new programming language (such as CUDA) is not practical, two general approaches (compiler directives and the use of GPU-ready libraries) have been adapted to develop E3SM code for computing systems with accelerators. For example, the Kokkos libraries have been used to increase the performance on the E3SM atmosphere model on NVIDIA GPUs in the Summit supercomputer [1] with a performance similar to that of the CPU code. The OpenACC and Athread have been used to develop the community atmosphere model and parallel ocean program on many-core 64-bit RISC processors in the Taihu Light supercomputer [10] with performance improvements of up-to 8 times. However, the performance improvement in this effort mainly came from the extensive programming using Athread. The OpenACC is mainly used as a pre-preparation for code porting onto the RISC system. The OpenACC has been used to accelerate the Model for Prediction Across Scales (MPAS) microphysics WSM6 (WRF Single Motion) model on a single NVIDIA GPU [6] with a performance improvement of up to 2.4 times.

With the availability of high-resolution atmospheric forcing (such as temperature, precipitation, shortwave radiation, and vapor pressure) [7] and land surface proprieties (such as vegetation and soil properties maps), it is desirable to conduct high-fidelity land simulations with ELM at 1km-resolution to deliver a "gold standard" set of results describing the surface weather and climate, as well as the energy, water, carbon, and biogeochemistry processes at a continental scale. The ultra-high-resolution ELM simulation over North America covers a landscape of 24 million gridcells, 350 million columns, and 700 million vegetation patches, is only feasible with highly efficient use of the accelerators within high-end supercomputers. Furthermore, for the reason that there are over one thousand subroutines in ELM, the majority of which are computationally non-intensive, using compiler directives is the appropriate approach to accelerate ELM onto GPU systems. The paper reports development of an ecosystem dynamics model within ELM on NVIDIA GPU using OpenACC.

1.1 Computational Platform and Software Environment

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 which contain two 22-core IBM POWER9 CPUs, six 16-GB NVIDIA Volta GPUs, and 512 GB of shared memory. Technically, this study uses 42 CPU cores (2 CPU cores are reserved for system functions) and all the non-tensor cores in GPUs. The software environment used in our study include NVIDIA HPC 20.11 and several libraries: OpenMPI (spectrum-mpi/10.4.0.3-20210112), NetCDF (netcdf-c/4.8.0, netcdf-fortran/4.4.5), pnetcdf(1.12.2), HDF (1.10.7), and CUDA (11.1).

2 Method

The ecosystem dynamics model simulates the biogeochemical cycles of the ecosystem, including carbon, nitrogen, and phosphorus. It contains many function groups, such as nitrogen deposition/fixation, maintenance and growth respiration, phosphorus deposition, and soil litter decomposition. The ecosystem dynamics model is the most sophisticated model within ELM that contains over 90 subroutines and accesses over 2000 globally accessible variables, many of them 3D arrays. All the ELM routines can access these hierarchical global data structures during the simulation.

We have developed a Functional Unit Testing (FUT) framework to generate standalone ELM models to accelerate code porting and performance tuning. The FUT is a python toolkit built upon the previous software system designed to facilitate scientific software testing [8,9]. We first generate a standalone ecosystem dynamics model for code porting and performance evaluation. To quickly assess the performance of ELM, we also create synthesized data using the observational data from AmeriFlux (ameriflux.lbl.gov) as the forcing data set for the code development. The forcing data and the global variables for 6000 gridcells take 11 GB GPU memory (approximate 1.8 MB data per gridcell). Each NVIDIA V100 contains 16 GB of memory, so 5 GB of shared GPU memory is available for other globally accessible variables, history data, external forcing data, and ELM kernels. The ecosystem dynamics model takes an hourly timestep. An ELM spin-up simulation[1] generally covers a period of 800 to 1000 years, and an ELM transit simulation[2] runs over a period of 100–200 years. Therefore, the ecosystem dynamics model usually is executed around 8–10 million times on each gridcell.

2.1 ELM Data Structure and Ecosystem Dynamics Model

The ELM uses highly customized, hierarchical data structures (Fig.Z 1) to represent the heterogeneity of Earth's landscape. Due to historical reasons, these

[1] The spin-up simulation is used for the ELM to reach a state of statistical equilibrium under the applied climatic forcing.

[2] The transit simulation is referred to the ELM simulation of post industry revolution period (1850-present) with rising CO2, greenhouse emission, dynamical vegetation, and land use and land change, including urbanization.

data structures are declared as globally accessible entities that can be referenced and modified by individual ELM functions during the simulation. Specifically, ELM contains 5 landscape datatypes (gridcell, topographic unit, land cover, soil column, and vegetation patch) representing several aspects of the land surface. Each gridcell represents a small region on the Earth's surface, and is a customized datatype that is derived from a standalone geospatially explicit data library and has dozens of global variables. Each gridcell contains 9 landcover units, 11 column units (each soil column can have up to 20 soil layers and 5 bedrock layers), and 24 vegetation units. Furthermore, The ELM has another 25+ customized model-related datatypes (such as the state and flux datatype for carbon, nitrogen, temperature, and soil). Altogether, these derived datatypes (around 90) contain over 2000 global variables, many of which are multidimensional arrays.

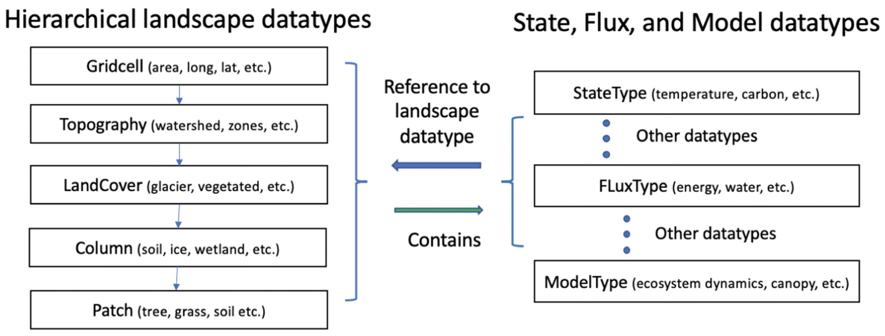
Hierarchical landscape datatypes
Gridcell (area, long, lat, etc.)
Topography (watershed, zones, etc.)
LandCover (glacier, vegetated, etc.)
Column (soil, ice, wetland, etc.)
Patch (tree, grass, soil etc.)
Reference to landscape datatype
Contains
State, Flux, and Model datatypes
StateType (temperature, carbon, etc.)
Other datatypes
FLuxType (energy, water, etc.)
Other datatypes
ModelType (ecosystem dynamics, canopy, etc.)

Fig. 1. Highly customized, hierarchical ELM data structure

The ecosystem dynamics model is the most complex submodel within ELM, containing four major subroutine groups (initialization, leaching, noleaching1, and noleaching2). In total, these subroutine groups contain over 90 subroutines and access around 2000 global variables. From the function perspective, the ecosystem dynamics model can be grouped into several modules, such as carbon, nitrogen, and phosphorus dynamics, phenology, soil litter, gap mortality, and fire (Fig. 2). For a better illustration, many secondary and supporting functions (such as IO, timing, and other utility functions) are not shown in the picture.

2.2 Performance Comparison and a Naive OpenACC Implementation

This study adopts a node-level performance comparison. A similar full workload is placed onto all the CPUs or GPUs within a single node. The execution times of the original CPU code and GPU implementation at a single hourly simulation timestep are collected for performance evaluation. Specifically, on a single Summit node, we assign 36000 gridcells on GPUs (6000 gridcells on each GPU) and 36036 gridcells on CPUs (858 gridcells on each CPU core). With the workload

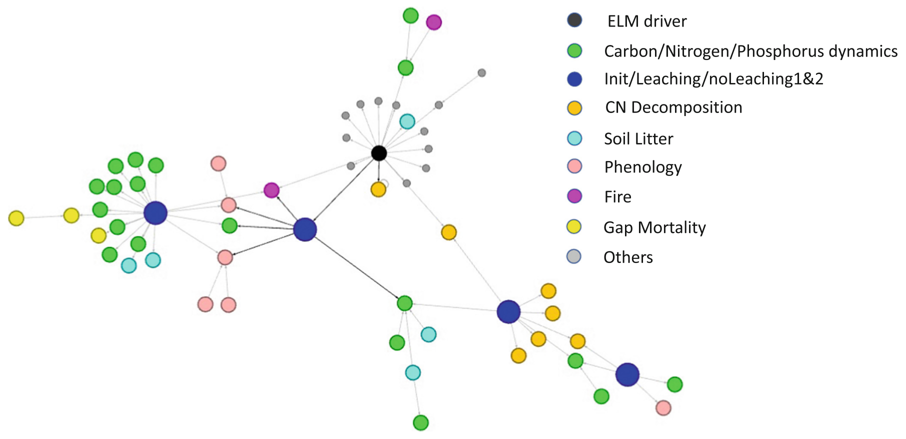

Fig. 2. The ecosystem dynamics model contains several function modules. There are 129 nodes, each represents a subroutine. The four blue node represent the entrance to subroutine groups: initialization, leaching, noleaching phase 1, noleaching phase 2.

of 858 gridcell, the original CPU-based ecosystem dynamics model takes 150 milliseconds (ms) to finish a single timestep (hourly) simulation. Note that the node-level comparison with a similar workload has also been adopted by other scientific code porting [1,6].

The original code organizes gridcells into clumps on each CPU core, and the ecosystem dynamics model runs over these clumps of gridcells. Therefore one of the most straightforward implementations is to instrument OpenACC directives into the original code and onto these existing loops. Specifically, this implementation contains four major steps: 1) use the OpenACC data directive to create a data region and copy all the input data into the data region once at the beginning of simulation; 2) use the routine directives to generate GPU kernels of the majority of ecosystem dynamics subroutines, including three subroutine groups, Leaching, NoLeaching Phase 1, and NoLeaching Phase 2, and many subroutines inside them; 3) place parallel loop constructs to loop over the gridcells, then 4) launch these GPU subroutine kernels in parallel (Fig. 3). This naive implementation does work but comes with abysmal performance (over 7 s) and consumes a large amount of memory (2.15 GB) for just the kernel.

2.3 Code Optimization

Several optimizations were developed to improve the code performance, such as reducing the memory consumption, restructuring parallel loops, deploying reduction clauses, and increasing parallelism over independent elements.

Algorithm 1 A Naive Implementation

```
if first step then
    acc enter data copyin                          ▷ Start Unstructured Data Region
acc parallel loop default(present)
for each gridcell do
    GET BOUNDS OF GRIDCELL                         ▷ Bounds holds subgrid info
    LEACHING
    NOLEACHING PHASE 1
    NOLEACHING PHASE 2
```

Fig. 3. All subroutines were ported using acc routine directive and parallelized across the existing gridcell loop.

Reducing the Memory Allocation of Local Variables: 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 we need to reduce the memory consumption of each kernel. In this study, we deployed many methods to reduce the size of these local variables, such as converting arrays into scalars and compressing the sparse arrays into dense arrays. For example: In SoilLittVertTransp, 9 local arrays would be allocated with a size of 238 doubles for each gridcell (that is 17136 (9*238*8) bytes in total). After refactorization, these arrays were replaced with a couple of 64-bit scalar and 4 dense arrays (less than 550 bytes in total). The memory reduction decreases the total execution time of the ecosystem dynamics model to 150 ms from 7600 ms. This memory saving is also necessary since we want to put up to 6000 gridcells on a single GPU.

Restructure Parallel Loops: The routine directive provides the ability to test functions on the GPU quickly, but for subroutines with internal loop structures and nested function calls, performance degradation is expected. In the ecosystem dynamics model, routines that loop over the same subgrid element (Column or Patch) tend to be clustered together. For example, Fig. 4 shows a group of functions labeled as SetValues, where the ELM subroutines compute over the Patch and Column arrays, respectively (Algorithm 2). A reliable optimization strategy is to refactor these routines to remove their internal loops and change the external gridcell based loops to the relevant subgrid element, which allows them to be grouped under different parallel loop constructs (Algorithm 3). In this case, each subroutine is actually completely independent of the others so that the SetValues group of functions can utilize asynchronous kernel launch. The loop re-factorization decreases the execution time from 30 ms (CPU code) to 16 ms (GPU code) and proves that the routine directive can provide a significant speedup over the CPU implementation.

Algorithm 2 Parallelize Gridcell

```
acc parallel loop
for each Gridcell do
    vegcfSetValues(array)
    vegnfSetValues(array)
    vegpfSetValues(array)
```

Algorithm 3 Parallelize Patch

```
acc parallel loop async
for each Patch do
    vegcfSetValues(Patch_index)
acc parallel loop async
for each Patch do
    vegnfSetValues(Patch_index)
acc parallel loop async
for each Patch do
    vegpfSetValues(Patch_index)
```

Fig. 4. The SetValues routines were refactored to remove the internal loops. In the original code (Algorithm 2), the SetValues routines take global Patch arrays as arguments and calculations are performed on the elements of these arrays. After the refactorization, the outer loop is over Patches with only the Patch index passed as an argument, and the multiple kernels can be launched asynchronously. (Algorithm 3).

Accelerate Internal Loops: For subroutines that must have their internal loops, we forego the routine directive and deploy different parallel techniques specific to each loop. One good example is the fire module that includes two large subroutines (FireFluxes and FireArea), each also contains many internal loops. When deploying the OpenACC routine directive within the fire module, a single step execution on GPU takes around 50 ms. After the internal loop acceleration, the execution time becomes 4.84 ms (Table 1).

Algorithm 4 CPU Code

```
for each active Patch do
    for each SoilLayer do
        Get: Column from Patch
        Sum: Patch vars to Column
```

Algorithm 5 GPU Code

```
acc parallel loop collapse gang worker
for each SoilLayer do
    for each active Column do
        Init: sums
        acc vector reduction(+:sums)
        for each Patch in Column do
            if Patch is active then
                Reduce: Patch vars to Column
```

Fig. 5. CPU Code(left) only uses two loops, the GPU code(right) required an additional loop to prevent race conditions during reduction ,but this requires looping over all Patches rather than only the active Patches. Despite the increase in loop size, the achieved speedup is over 2x for the entire FireMod section (Table 1)

The ecosystem dynamics model uses hierarchical data structures, and the lower-level variables (i.e. Patch variables) are aggregated into the corresponding higher-level variables (i.e. Column variables). Figure 5 shows an example of aggregating variables from Patch to Column in the FireFluxes subroutine. The code is optimized by using gangs and workers to parallelize the *Soil Layer* and *Column* (collapsed) loops with an inner vector loop performing the reduction. The reduction operation is very efficient in our case because there are a maximum of 33 patches in each column and each warp of the NVIDIA V100 has 32 threads. With the reduction, we can finish the operations in 0.3 ms. After the optimizations, the execution time of the entire GPU-based fire module is reduced to 4 ms, which includes 20 kernels in total.

Parallelize over Nutrients or Output Variables: To further improve the performance of the ecosystem dynamics model, we investigate the task parallelism of algorithms that don't distinguish among nutrients, such as carbon, nitrogen, and phosphorus (CNP), aside from the input and output variables (i.e. sources and sinks of CNP). A good example are the transport calculations inside the SoilLittVertTransp subroutine (Fig. 6), where we take advantage of asynchronous compute to pipeline the creation of local arrays for tridiagonal coefficients. A new array of derived types is created with each element pointing to a set of 3D arrays corresponding to a CNP nutrient input or output variable. This data structure is initialized and moved to the GPU only at the start of a run via deepcopy [4]. We can then either collapse the nutrient loop into the parallel loop construct or use the nutrient loop to asynchronously launch kernels to improve the code performance.

Algorithm 6 Init Pointer List

```
Loop over ntypes (C, N, P)
for each ntype do
    $list[ntype]\%conc \rightarrow ntype\%conc$
    $list[ntype]\%src \rightarrow ntype\%src$
    $list[ntype]\%trcr \rightarrow ntype\%trcr$
acc copyin(list)
```

Algorithm 7 Parallelize Over ntypes

```
for each ntype do
    acc parallel loop async(ntype)
    for each decomp source do
        for each soil levels do
            for each soil column do
                Set outputs using list
                Solve TRANSPORT
```

Fig. 6. (Left) An array of a derived type is created whose fields point to an output field of a CNP global variable and copied to device at start of run. (Right) The CPU nutrient type loop launches the transport kernels asynchronously so that independence of CNP provides a fourth level of parallelism to occupy the GPU.

The left panel of Fig. 6 illustrates the creation of the new derived type, and the right panel shows the new asynchronous loop structure of the transport algorithm inside the SoilLittVertTransp module. The vertical transport among soil levels in each soil column is calculated using a tridiagonal solver. The solving algorithm does not have enough dimensions (vertical soil layers) to fully utilize the GPU resources, so we take advantage of the independence between CNP inputs and outputs to occupy the device. The asynchronous launching of these kernels allows the whole SoilLittVertTransp module to finish in 5 ms, compared to over 40 ms using the routine directive after removing excess memory allocations. This method of using arrays of pointers is also used to reorganize the code structure of other models within ELM, so we can use OpenACC directive to explore the parallelism of these models efficiently. A good example is the ELM output model that generates a history buffer containing over 500 aggregated and averaged variables to be discussed in the future work on the full ELM simulation.

3 Results

3.1 Overall Performance Improvement

We gather the subroutines in the ecosystem dynamics model into many groups and collect the individual execution time sequentially (Table 1). The GPU-based ecosystem dynamics model achieves a 3.0 times speedup over the original code on the CPU when the Summit node is fully loaded. The timing data of the majority (10 out of 15) function groups show good speedups (ranging from 2.3 to 8.3 times). Especially, gap mortality (GapMortality) contains many global-variable operations (similar to SetValues) that have been refactored into parallel do loop and are launched asynchronously on GPU. Respiration contains two simple loops that have been refactored into parallel do loops with a reduction clause. The execution times of the other 5 groups are relatively short (less than 2 ms). The slow down of these groups is mainly because the overhead associated with GPU kernels overshadowed the small computational part of these subroutines. After optimization, the entire ecosystem dynamics model (GPU kernels with all the nested subroutines) requires around 300 MB of memory.

3.2 Profiling Details

To determine the limitations of OpenACC's routine directive and better understand performance improvements, we used NVIDIA's Nsight Compute[3] and Nsight Systems[4] to collect GPU metrics and traces for SoilLittVertTransp, FireMod, and SetValues subroutines at various stages of optimization. Table 2 shows the metrics of the kernels that illustrate different shortcomings of prior versions of the GPU kernels. Several metrics, such as wallclock time, kernel launch overhead, and total instructions issued, are measured to frame performance discussion in this work. The overheads listed in Table 2 are the percentage of wallclock time spent on launching the kernel, allocating memory, and creating device streams (if applicable). Nsight Systems drastically increases initial kernel launch times but agrees with Nsight Compute in reporting the kernel compute time, so to consistently calculate overhead, we subtract the kernel compute time from the total wallclock time of non-profiling runs and take the percentage.

[3] AnNVIDIAinteractivekernelprofiler, https://developer.nvidia.com/nsight-compute.
[4] An NVIDIA performance analysis tool, https://developer.nvidia.com/nsight-systems.

Table 1. Comparison of average execution time between GPU and CPU versions of the ecosystem dynamics model (single timestep) on a fully loaded single Summit node. Each node has the the same size grid divided between 6 GPUs or 42 CPUs

Function group	GPU(ms)	CPU(ms)	Speedup
SetValues	15.49	46.16	2.98
NDeposition/Fixation	0.09	0.02	0.24
Respiration/PDeposition	0.33	2.77	8.30
Decomp. rate	1.31	3.08	2.35
Vertical decomp.	2.74	7.77	2.84
Alloc Phase 1	1.51	0.93	0.62
SoilLittDecomp 1	6.12	23.30	3.81
SoilLittDecomp 2	1.18	1.89	1.60
Phenology	2.26	0.58	0.26
Growth and root respiration	1.03	0.43	0.42
StateUpdate 1	2.79	12.21	4.39
SoilLittVertTransp	5.94	17.35	2.92
GapMortality	2.56	17.54	6.85
StateUpdate 2	1.88	4.95	2.64
FireMod	4.84	11.19	2.31
Total	50.05	150.2	3.00

SoilLittVertTransp: The initial OpenACC implementation of SoilLittVertTransp had the slowest performance of the ecosystem dynamics model and even the full E3SM Land Model runs. The reliance of the CPU version on dynamically allocated arrays can be distilled to the over-inflated number of instructions reported on the GPU, which is nearly three orders of magnitude larger than that with the arrays refactored out (Table 2). The memory workload and warp state metrics further support that this memory is the bottleneck by showing very high L1 and L2 hit rates and related stalls (not listed in the paper).

After the initial refactoring (Mem Opt. in Table 2), the performance is still 2.6 times slower than a single CPU core given a similar workload. While the kernel only uses half the threads in a warp (on average), the profilers report the kernel compute time as around 10 ms. The overheads associated with launching the kernel contribute the most to the total time, and the subroutine is simply too big and complicated to be ported using OpenACC routine directives. The metrics for the final OpenACC implementation illustrate much more efficient utilization of the GPU's warps and pipelines due to smaller kernel size and techniques described in Sect. 2. Note that the massive over 35 billion instructions in the original code is caused by the excessive memory allocations.

FireFluxes: The initial naive implementation of OpenACC worked better for the FireFluxes subroutine, which consists exclusively of computations on global data types. However, the overheads are significant because many nested loop structures and the hundreds of global variables have to be passed as arguments by the kernel launcher. The final optimizations resulted in a great increase in the required instructions due to the changes shown in Fig. 5. With the asynchronous kernel launches and high-efficient utilization of the GPUs resources, the model performance on the GPU delivers a 2.3 speedup over the performance on the CPU.

Table 2. High-level metrics for three kernels showing differences between optimization methods. The routine directive can result in very high overheads, and poor memory allocations result in large excess in instructions issued. For FireFluxes, the excess instructions are due to extra loops but is alleviated by better saturation of GPU.

Kernel	Time(ms)	Overhead(%)	Instruction (millions)
SoilLittVertTransp			
Orig	10,000	49.8	34,500
Mem Opt	44.0	70.0	48.7
Final Opt	6.95	11.8	40.9
FireFluxes			
Orig.	45.0	92	9.83
Opt.	3.67	27.9	197
SetValues			
Orig.	29.4	70.2	15.3
Opt.	15.4	11.6	14.5

SetValues: The naive OpenACC implementation of the SetValues functional group had a speedup of around 1.5 times over the CPU code, but the total time taken was significant relative to the other modules. The SetValues subroutines involved initializing hundreds of arrays of three derived types at the Patch and Column level, which can all be done independently. While the OpenACC routine directive allows for multiple levels of parallelism, compiling as a sequential routine is the most reliable and free of compilation errors for our use case. The profiling data identified that kernel launch overhead was the bottleneck, and so the routines were simplified as described in Sect. 2.3. The overhead is greatly reduced and mostly due to CUDA API calls to allocate memory on the host and device and create streams with a final speedup of 3.0.

4 Conclusion and Future Work

The study reports design and optimization strategies for developing an ELM ecosystem dynamics model using compiler directives (OpenACC) on NVIDIA

GPUs. We have restructured the code to reduce the memory footprint and to increase the parallelism, so that the code can be programmed with OpenACC directives efficiently. The routine directive and deepcopy capabilities of OpenACC provided a robust method for accelerating very complex ELM modules, with certain functions receiving immediate speedup.

After code analyses and refactoring, the parallel GPU implementation (with a small memory footprint of 300 MB) achieved a 3.0-times speedup over the original CPU code on a fully loaded Summit computing node. Computationally, ELM doesn't have a single submodel that dominates run time but around a dozen complex components that contribute more or less equally. Accelerating the full code base requires a flexible approach that can handle different algorithms, and this work demonstrates methods for reworking algorithms and data structures in tandem with compiler directives to tackle exascale climate simulations. Although this study is not intended to improve the ELM performance on the CPUs, this study revealed several limitations of the original CPU code that was implemented without prudent considerations of memory usage and allocation. Some techniques mentioned in the study (such as reducing the memory footprint) also have benefits for the CPU-based ELM code. Future work will focus on the parallelization of other models within ELM and further integrated performance tuning. For the ultra-high resolution ELM simulation over North America, if we assign 36000 gridcells to each node, the 24 million gridcells of North America require 680 nodes, which is around 15% of the total capability of the 4608-node Summit. With an estimated 3.0 times overall speedup, a 100-year simulation over North America takes around 2 days.

References

1. Bertagna, L., et al.: 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. Bourzac, K.: Supercomputing poised for a massive speed boost. Nature **551**(7680), 554–557 (2017)
3. Burrows, S., et al.: The doe e3sm v1. 1 biogeochemistry configuration: Description and simulated ecosystem-climate responses to historical changes in forcing. J. Adv. Model. Earth Syst. **12**(9), e2019MS001766 (2020)
4. Ghane, M., Chandrasekaran, S., Cheung, M.S.: Assessing performance implications of deep copy operations via microbenchmarking. CoRR abs/1906.01128 (2019). http://arxiv.org/abs/1906.01128
5. Golaz, J.C., et al.: The doe e3sm coupled model version 1: overview and evaluation at standard resolution. J. Adv. Model. Earth Syst. **11**(7), 2089–2129 (2019)
6. Kim, J.Y., Kang, J.S., Joh, M.: Gpu acceleration of mpas microphysics wsm6 using openacc directives: performance and verification. Comput. Geosci. **146**, 104627 (2021)
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. Sci. Data **8**(1), 1–17, 104627 (2021)

8. Wang, D., et al.: 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)
9. Wang, D.: A functional test platform for the community land model. Environ. Model. Softw. **55**, 25–31 (2014)
10. Zhang, S.: Optimizing high-resolution community earth system model on a heterogeneous many-core supercomputing platform. Geosci. Model Dev. **13**(10), 4809–4829 (2020)
11. Zheng, W., Wang, D., Song, F.: XScan: an integrated tool for understanding open source community-based scientific code. In: Rodrigues, J.M.F., et al. (eds.) ICCS 2019. LNCS, vol. 11536, pp. 226–237. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-22734-0_17