Refactoring and Optimizing the Community Atmosphere Model (CAM) on the Sunway TaihuLight Supercomputer

Haohuan Fu[1,2], Junfeng Liao[1,2,3], Wei Xue[1,2,3], Lanning Wang[4], Dexun Chen[3], Long Gu[5], Jinxiu Xu[5], Nan Ding[1,3], Xinliang Wang[3], Conghui He[1,2,3], Shizhen Xu[1,3], Yishuang Liang[4], Jiarui Fang[1,2,3], Yuanchao Xu[3], Weijie Zheng[1,2,3], Jingheng Xu[1,2,3], Zhen Zheng[3], Wanjing Wei[1,2], Xu Ji[1,3], He Zhang[1,3], Bingwei Chen[1,2], Kaiwei Li[3], Xiaomeng Huang[1,2], Wenguang Chen[3], and Guangwen Yang[1,2,3]

[1]*Ministry of Education Key Laboratory for Earth System Modeling, and Center for Earth System Science, Tsinghua University, Beijing, China*
[2]*National Supercomputing Center in Wuxi, Wuxi, China*
[3]*Department of Computer Science and Technology, Tsinghua University, Beijing, China*
[4]*College of Global Change and Earth System Science, Beijing Normal University, Beijing, China*
[5]*National Research Center Of Parallel Computer Engineering and Technology, Beijing, China*

***Abstract*—**

This paper reports our efforts on refactoring and optimizing the Community Atmosphere Model (CAM) on the Sunway TaihuLight supercomputer, which uses a many-core processor that consists of management processing elements (MPEs) and clusters of computing processing elements (CPEs). To map the large code base of CAM to the millions of cores on the Sunway system, we take OpenACC-based refactoring as the major approach, and apply source-to-source translator tools to exploit the most suitable parallelism for the CPE cluster, and to fit the intermediate variable into the limited on-chip fast buffer. For individual kernels, when comparing the original ported version using only MPEs and the refactored version using both the MPE and CPE clusters, we achieve up to 22x speedup for the compute-intensive kernels. For the 25km resolution CAM global model, we manage to scale to 24,000 MPEs, and 1,536,000 CPEs, and achieve a simulation speed of 2.81 model years per day.

***Keywords*-atmospheric modeling, many-core, optimization, tool, OpenACC**

I. Introduction

Ever since the first generation of supercomputer systems (CDC 6600, Cray-I, etc.), the atmospheric models have been among the major users of computing resources [1], and evolved with the development of supercomputer systems. In the early years, the vector machines, such as the Cray systems (Cray 1, Cray X-MP, Cray Y-MP, etc.), and the famous Japanese earth simulator [2], have been the major computation platform for weather and climate modelers. Then, from the year of 2000 or so, Intel, IBM, and SGI clusters emerged as the replacements of the traditional vector machines, and also made the transition of the atmospheric modeling programs to the style of MPI programs. Along this transition process, with the increase of cores within each processor, we see the introduction of the hybrid parallelization scheme that combines MPI and OpenMP.

In the recent decade, again, we see the transition of supercomputers from homogeneous systems with only multi-core CPU processors to heterogeneous systems with both CPUs and many-core accelerators [3], [4]. This architectural transition, again, brings significant changes to existing high-performance computing software in various application domains, such as geophysics exploration, sky simulation, and phase-field simulation.

Unlike the above application domains that have made a quick adaptation to the many-core accelerators, the transition of the weather and climate models has been relatively slow. One big reason is the millions lines of legacy code that have been written for multi-core CPUs rather than many-core accelerators. As a result, most existing efforts either focus on standalone physics schemes ([5], [6]), or focus on the dynamic core part ([7], [8]).

In contrast, complete porting projects of entire models onto heterogeneous supercomputers are still few to be seen. Typical examples include the GPU-based acceleration of a next-generation high resolution meso-scale atmospheric model being developed by the Japan Meteorological Agency (JMA) [9], and complete porting of the Princeton Ocean Model (POM) onto GPU devices [10], both of which take a manual rewriting of the code into CUDA. Only in newly-developed climate or weather models, such as NIM [11] and COSMO [12], we see careful considerations for heterogeneous systems, and support for multiple architectures including CPU, GPU, and MIC.

In general, while the weather and climate models are calling for more computing power to support higher resolution and more complex physics [13], there is still a gap between the increasing demand and the increasing supply in the form of many-core accelerators. To fill the gap between the demand and the supply, in our work, we perform an

SC16; Salt Lake City, Utah, USA; November 2016
978-1-4673-8815-3/16/$31.00

extensive refactor and optimization of the CAM atmospheric model, for the Sunway TaihuLight Supercomputer, equipped with many-core processors that consists of Management Processing Elements (MPEs) and clusters of Computing Processing Elements (CPEs). We pick CAM [14] as the our target application, as it is one of the most widely used advanced atmospheric model in the world. Note that, to achieve a high scalability over the Sunway system with millions of cores, we select to use the SE dynamic core of CAM [15], and the other dynamic core options are not considered in this work.

We use the Sunway OpenACC compiler (a customized version that expands from the OpenACC 2.0 standard) as the major tool to achieve a suitable mapping of CAM onto the new Sunway heterogeneous many-core processors. Due to large code base developed over the last few decades, and the general demand from climate scientists to maintain a same source, we try to minimize the manual refactoring efforts (to only the dynamic core part), and mainly rely on the source-to-source translation tools to achieve an automated and efficient porting. Besides, compared with GPU and other many-core accelerators, both the on-chip fast buffer and the available memory bandwidth of the Sunway processor are relatively limited (detailed in Section III), which make our porting significantly more challenging. A large part of our tools and optimization strategies would focus on minimizing the memory footprints.

Our major contributions are as follows:

- Through a careful refactor of the SE dynamic core, we manage to combine the distributed loops in the major computational functions into aggregated multi-level loops, and expose a suitable level of both parallelism and variable storage space for the CPE cluster architecture.
- For the physics parts, which includes numerous modules with different code styles by different scientists, we design a loop transformation tool to identify and expose the most suitable level of loop body for the parallelization on the CPE cluster. In addition, we also design a memory footprint analysis and reduction tool, and a number of customized Sunway OpenACC features, to fit the frequently-accessed variables into the local fast buffer of the CPE.
- Comparing the refactored hybrid version using both MPE and the CPE cluster against the ported version using only MPE, we can achieve up to 22x speedup for compute-intensive kernels, and 2x to 7x speedup for kernels involving both computation and memory operations. For the entire CAM model, we achieve a performance improvement of around 2x.
- We manage to scale the refactored CAM model to 24,000 MPEs, and 1,536,000 CPEs, and achieve a simulation speed of 2.81 modeling years per day for the 25km resolution.

While the speedup is not significant, it provides an important base for us to continue tuning the performance of large and complicated scientific programs, such as CAM.

II. Related Work

When compared with other HPC application domains, the porting of the climate models onto many-core architectures has been relatively slow. Most early-stage efforts focused on the standalone physics modules. The Weather Research and Forecast (WRF) model, which is one of the most widely used numerical weather prediction (NWP) program in the world, was among one of the earliest weather/climate models that integrate GPU-accelerated microphysics schemes [5]. Other typical examples include GPU-based accelerations for the chemical kinetics modules in WRF-Chem [16], and the shortwave radiation parameterization in CAM [6]. As these physics modules are usually compute-intensive and do not involve communications, GPU-based acceleration can generally achieve a speedup of one order of magnitude.

In recent years, we start to see projects that accelerate the dynamic cores (or the major computation parts of the dynamics cores) on GPU devices, such as the efforts on GRAPES [7], CAM-SE [8]. Compared with the physics modules, the dynamic parts generally require communication across different grids, and are more difficult to achieve good parallel performance. The speedup is usually in the range of 3 to 5 times when comparing GPU solutions against parallel CPU solutions.

Complete porting projects that move complete atmospheric or ocean models onto many-core accelerators are still few to see. One example is the GPU-based acceleration of ASUCA, a next-generation high resolution meso-scale atmospheric model being developed by the Japan Meteorological Agency (JMA) [9], with an acceleration of 80-fold when compared against a single CPU core, and an excellent scalability for up to a few thousand nodes. Another example is the complete porting of the Princeton Ocean Model (POM) onto GPU devices [10], which performs a manual porting and optimization of POM onto a hybrid server with 4 GPUs, and achieves an equivalent performance to 408 CPU cores. While the two projects have managed to take advantage of GPU accelerators for complete models, these are relatively less complicated models that involve only tens of thousands of lines of code. Moreover, both projects take the approach of manual rewriting of the program in CUDA, which makes it difficult to keep a same source base and not possible to migrate to other computing architectures.

For newly developed weather or climate models, such as NIM [11] and COSMO [12], we see the emphasis on supporting multiple accelerator architectures. For example, by using OpenMP, OpenACC, and F2C-ACC directives, NIM managed to maintain a single source for application scientists, and the portability over CPU, GPU, and MIC

architectures [11]. Similarly, the COSMO model also maintains both OpenMP and OpenACC directives for the physics schemes in the model, to support both GPU and MIC. For the dynamic core part, the COSMO model uses a C++-based domain specific language to provide both CUDA and OpenMP backends. As newly developed numerical models, NIM and COSMO are relatively ahead in the transition, due to less constraints from the code legacy.

Different from the projects mentioned above, our work focuses on CAM 5.3, which is a complete atmospheric model with 560,000 lines of code, developed over the last few decades. Moreover, our target platform is the new Sunway processor, which is based on a hybrid many-core architecture with a few MPEs and an array of CPEs. Compared with GPU and other many-core accelerators, both the on-chip fast buffer and the available memory bandwidth are more limited, which make our porting significantly more challenging. For such a large code base that have been developed over the decades, we think a manual rewrite would not be a feasible solution. Instead, in our approach, we would minimize the manual refactoring (limited to the dynamic core part), and rely on source-to-source translation tool to remap the code onto the new architecture.

III. The Sunway TaihuLight Supercomputer

A. The Hardware System

As one of the two 100PF systems supported by China's National 863 High-Tech Research and Development Program in the 12th five-year plan, the Sunway TaihuLight supercomputer [17] is the successor of the Sunway Bluelight supercomputer hosted in the Jinan Supercomputer Center. Similar to the Sunway Bluelight system, the new Sunway supercomputer is also using China's homegrown processor designs.

The general architecture of the new Sunway heterogeneous processor [17] is shown in Figure 1. The processor includes 4 core-groups (CGs). Each CG includes one management processing element (MPE), one computing processing element (CPE) cluster with 8x8 CPEs, and one memory controller (MC). These 4 groups are connected via the network on chip (NoC). Each group has its own memory space, which is connected to the MPE and the CPE cluster through the MC. The processor connects to other outside devices through a system interface (SI).

The MPE is a complete 64-bit RISC core, which can run in both the user mode and the system mode. The MPE supports the complete interrupt functions, memory management, superscalar, and out of order issue / execution. Therefore, the MPE is an ideal core for handling management and communication functions.

In contrast, the CPE is also a 64-bit RISC core, but with limited functions. CPE can only run in the user mode, and does not support interrupt functions. The design goal is to achieve the maximum aggregated computing power, while minimizing the complexity of the micro-architecture. The CPE cluster is organized as an 8 by 8 mesh, with a mesh network to achieve low-latency register data communication among the 8 by 8 CPEs. The mesh also includes a mesh controller that handles interrupt and synchronization controls.

In terms of the memory hierarchy, each MPE has a 32KB L1 instruction cache and a 32KB L1 data cache, with a 256KB L2 cache for both instruction and data. Each CPE has its own 16KB L1 instruction cache, and a 64KB Scratch Pad Memory (SPM). The SPM can be configured as either a fast buffer that support precise control by the users or a software-emulated cache that achieves automatic data caching in a software manner. However, as the performance of the software-emulated cache is low, in most cases, we need a user-controlled buffering scheme to acheive good performance.

Combining the four CGs of MPE and CPE clusters, each Sunway processor provides a peak performance over 3 Tflops, with a performance-to-power ratio over 10 Gflops/Watt. While the computing performance and power efficiency is among the top when compared with existing GPU and MIC chips, the on-chip buffer size and the memory bandwidth is relatively limited. The four CGs are sharing an aggregated memory bandwidth of around 130 GB/s.

B. The Software System

On the software side, the TaihuLight system uses a customized 64-bit Linux as the operating system, with a set of compilation tools to support the development of applications on the new Sunway processor architecture.

The compilation tool set includes the basic compiler components, such as the C/C++, and Fortran compilers. In addition to that, there is also a parallel compilation tool that supports the OpenACC 2.0 syntax and targets the CPE clusters. The customized Sunway OpenACC tool supports mangement of parallel tasks, extraction of heterogeneous code, and description of data transfers. Moreover, according to the specific features of the Sunway processor architecture, the Sunway OpenACC tool has also made a number of syntax extensions from the original OpenACC 2.0 standard, such as a fine control over buffering of multi-dimensional array, and packing of distributed variables for data transfer (detailed in Section VI-C).

IV. Mapping CAM to the Sunway TaihuLight Supercomputer: Our General Methodology

A. General Workflow of CAM

CAM, which serves as the atmosphere component of the Community Earth System Model (CESM) [18], is the most computationally expensive component in typical configurations. The computation workflow of CAM can be divided into two phases: the dynamics and physics. The dynamics advances the evolutionary equations for the atmospheric flow, and the physics approximates sub-grid phenomena such

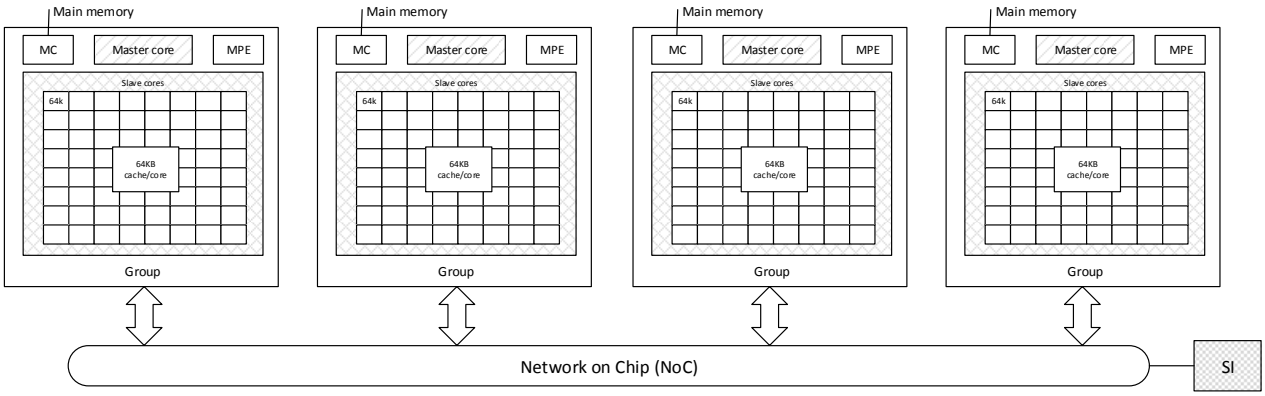

Figure 1. The general architecture of the new Sunway processor.

as clouds, precipitation processes, long/short-wave radiation, and turbulent mixing. As shown in Figure 2, the physics contains two phases. In phy_run1, radiation, shallow and deep advection are calculated, and in phy_run2, aerosol, and chemistry procedure are computed. State variables, such as temperature and precipitation, are passed through between two physic phases. The physics passes the traces, such as u, v, to the dynamics. After initialization, the physics and the dynamics are executed in turn during each simulation time-step.

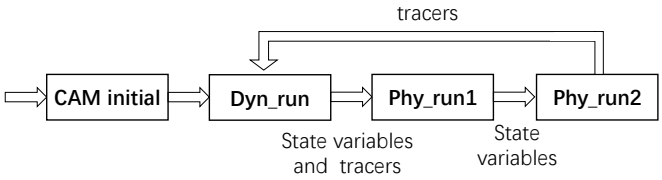

Figure 2. The general workflow of CAM.

the total mass over one year. We also observe an identical variation cycle when compared with the results on the Intel cluster.

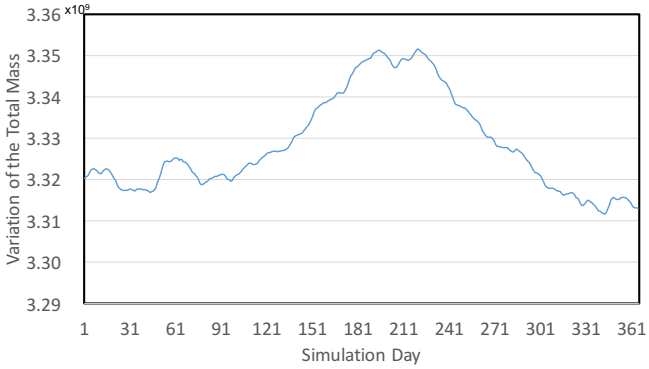

Figure 3. The variation of the total mass of a coupled CAM and CLM run on the new Sunway system.

B. Porting and Refactoring

As CAM has not been running on the Sunway architecture before, the first step of our porting is to verify the modeling results. As the current version of CAM has to be run in a coupled mode with the Common Land Model (CLM), using the F compset of CESM 1.2.1, we port both CAM and CLM onto the Sunway system, using only MPE to perform the computation.

After running the coupled CAM and CLM models on the Sunway system for the duration of three years, we compare results of major variables, as well as the conservation of mass and energy, to verify the correctness of our ported version. Compared with the results on the Intel clusters using the same modeling parameter configuration, we see almost identical distribution of key variables and an average relative error in the range of 10^{-4}. Figure 3 shows the variation of

Using the ported MPE-only version as the starting point, we then perform refactoring and optimization of both the dynamic core and the physics schemes to make utilization of the CPE clusters. During the process of expanding each kernel from MPE-only to MPE-CPE hybrid mode, we take the numerical result of the MPE-only mode as the true value, and ensure that the difference of the results in the MPE-CPE hybrid mode are within a similar range of the floating-point rounding errors.

Considering both the code volume and complexity, for the SE dynamic core, we take a manual approach to refactor and optimize the code. For the physics part, we rely on our source-to-source translation tools to perform loop transformation, and to reduce the memory footprint for the on-chip fast buffer (the SPM of each CPE).

V. Refactoring and Optimization of the SE Dynamic Core

A. Major Challenges and Our Solutions

The current CAM code is taking a hybrid parallelization scheme that combines MPI and OpenMP to parallelize the computation. As MPI is used for the inter-node parallelization and OpenMP is used for the intra-node parallelization, we generally apply a configuration with a large number of MPI processes (hundreds to thousands), and small number of OpenMP threads (8 to 16). In the Sunway system, we are dealing with groups with one MPE and 64 CPEs. Therefore, our first challenge is that a direct map from OpenMP to OpenACC would not provide a suitable level of parallelism for the CPE cluster.

The other challenge relates to the buffering of data. As mentioned in Section III-A, 64KB SPM of each CPE needs to be fully controlled by the user, either through a customized buffering scheme, or a software-emulated cache scheme. Our experiments show that the emulated software cache scheme is not efficient enough to provide performance benefits. Therefore, in most cases, we need to design a buffering scheme that loads the proper data into the SPM.

For the first challenge of achieving a suitable level of parallelism for CPE clusters, we do adjustments of both the computational sequence and the loop structures, so as to aggregate enough computations, and to enable the right number of parallel threads. For the second challenge of data buffering, on one hand, we refine the code to minimize the usage of intermediate variables; on the other hand, we design customized buffering schemes to determine when and where to load or unload data arrays.

In the following subsection, we take the $Euler\_step$ function (the most time consumption part of the SE dynamic core) as the example to demonstrate our refactorization and optimization schemes in details.

B. Refactorization of the $Euler\_step$ Function

The general structure of the $Euler\_step$ function is shown in block ① of Figure 4.

The $Euler\_step$ function is the basic forward Euler component used to construct the Strong Stability-Preserving (SSP) Runge-Kutta (RK) methods, consisting of the following three major parts:

1) computation of the biharmonic mixing term and its min/max values (later used as limiters);
2) the 2D advection step that updates each tracer's Qdp (the vertical integration result);
3) MPI communication for boundary exchange, including the preparation and package of data before that.

Inside these three major parts, there are mainly three types of loops, also shown in block ② of Figure 4:

- the ie loop processes each column that needs to be updated by the $Euler\_step$ function (48602 and 777602 for resolution of ne30 and ne120, respectively);
- the q loop iterates over each tracer (5, 25, and 108 tracers for different configurations);
- the k loop iterates over the vertical levels, which is 30 in the default configuration of CAM.

The first part of biharmonic value calculation consists of four stages, each of which is a three-level loop that iterates for ie, k, and q (shown in the upper part of block ②). In the original parallelization scheme, OpenMP is used for the second level loop of k. The value of k corresponds to the number of vertical levels in CAM, usually in the range of 30 to 50, which does not provide enough parallelism for the 64 CPEs. Moreover, while the separate four loops do not bring issues for the CPU cache hierarchy, for the user-controlled SPM of the Sunway CPE, separate loops require separate SPM loading operations, thus increasing both the loading cost and the programming complexity. To resolve this issue, we aggregate the four separate loops into one unified loop. To expose enough parallelism for the CPE cluster, we switch the loop levels of k and q to achieve consecutive memory access of the (k, q, ie) array, and collapse ie and q into one level of loop, with a larger number of iterations to assign to different CPEs (the exact transformation is shown in the upper part of block ③, and the left part of block ④, ⑤, and ⑥).

Compared with the biharmonic part, the second advection part is a more complex nested loop and consumes most of the time in the $Euler\_step$ function (the lower part of block ②). The outermost loop is an ie-loop. Within the ie-loop, we first have a k-loop that calculates the delta pressure and the initial velocity. Then, a two-level nested loop that iterates over q and k advances the objective function. In the end, a k-loop prepares the data for the boundary communication afterwards.

The major issue with the advection part is the distributed style of loop bodies, which increases the difficulty in both parallelization and data buffering. It is straightforward to aggregate the three k-loops in the two-level nested loop in the middle into one loop. However, due to variable dependencies, the first and the last k-loop cannot be easily congregated with the middle loop.

To resolve the above issue and to aggregate these separate loops, we firstly try the option to switch the q-loop and the k-loop in the middle two-level nested loop, so as to later aggregate the three k-loops into a larger one. While we manage to achieve an aggregated loop in this way, the resulting loop body involves too many variables to fit into the SPM of CPE.

Therefore, instead of aggregating the loops at the k level, we split the k-loops at the beginning and the end of the advection part into the two-level loop that iterates over q and k. The related cost is that these operations that used to be repeated for k times, now need to be performed for $q \cdot k$ times (as shown in the lower part of block ③). Although we pay the cost of repetitive computation, we can now reformat

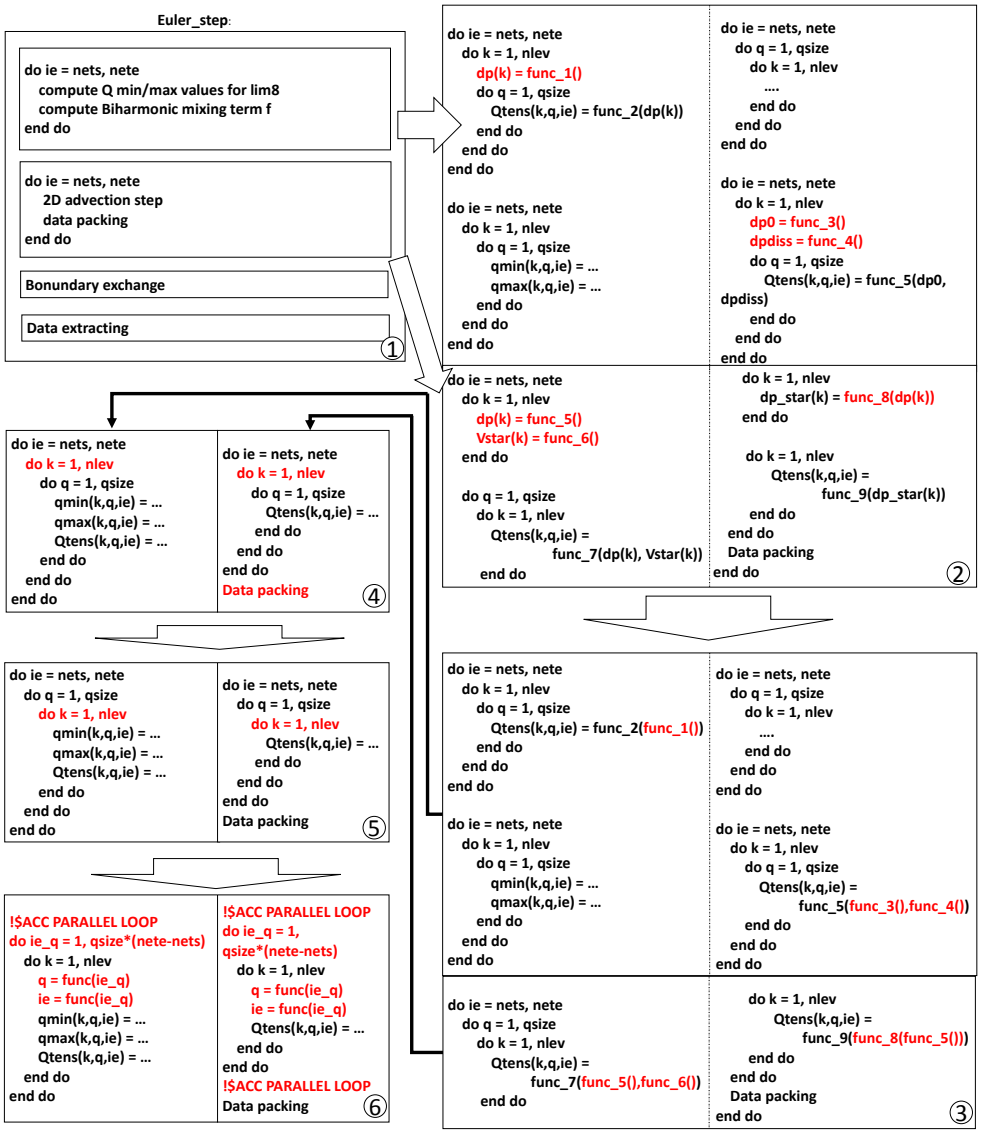

Figure 4. The general structure of the $Euler\_step$ function, and our corresponding refactor of the biharmonic computation part and the advection part.

the loops into a nested (ie, q, k) three-level loop. Similarly, we can then collapse the ie-loop and the q-loop to expose enough parallelism (as shown in the right part of block ④, ⑤, and ⑥). The required memory space of the intermediate variables of the resulting loop body is also reduced to fit into the SPM.

By performing the above transform of loops and the corresponding optimization for the buffering of local variables, we can achieve 22x speedup for the advection part, 10x speedup for the biharmonic part, and 7x speedup for the advection part, with a 2x to 4x speedup for the entire $Euler\_step$ function.

We apply similar refactorization and optimization schemes to the $compute\_and\_apply\_rhs$ function that computes a leapfrog timestep, and the $advance\_hypervis\_dp$ function that updates the temperature. In addition, we also refactorize and parallelize the parts that perform packing and unpacking of data elements for the halo communication stage, so as to improve the overall performance of the entire SE dynamic core.

VI. Refactoring and Optimization of the Physics Schemes

A. The General Parallelization Scheme

As shown in Figure 5, in most physics schemes, the computation is performed on the columns. Each column consists of the different vertical levels of the same surface

location. The computation of each column is completely independent, which provides the parallelism in the physics schemes. In order to achieve dynamic load balancing, CAM generally organize columns in different locations into one chunk, and assign each chunk to a different OpenMP thread.

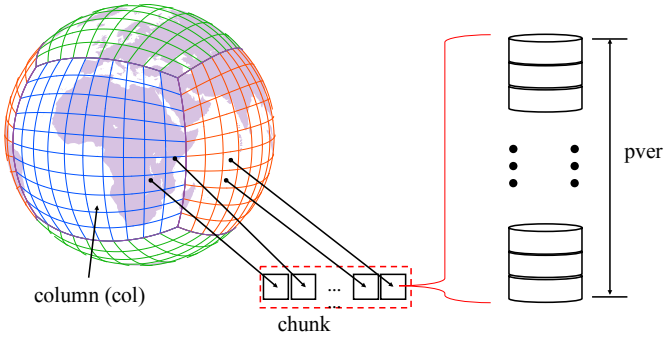

Figure 5. The organization of columns and chunks in the physics schemes of CAM.

In our design, we follow a similar approach to parallelize over different chunks. However, as the corresponding loop body is usually at the outermost level (such as the example of phy_run1 shown in Figure 7), the loop body would involve a huge volume of data variables and long sequences of instructions that could overwhelm the private instruction cache and SPM of each CPE.

Our solution is to expose the most suitable level of loop body to the Sunway OpenACC compiler, so as to provide a suitable level of intensity in both computation and data variables. To achieve this goal, we develop a loop transformation tool that moves the loop into the right level of the function, and serves as a preprocessor for the OpenACC compiler. To further reduce the storage space of variables and arrays that are related to the functions targeting CPE clusters, we also develop a variable storage analysis and reduction tool to minimize the required storage cost, and to fit the functions into the 64KB SPM of the CPE architecture.

In both tools, we utilize the ROSE source-to-source compiler [19] to construct a code translator to analyze the code and to generate the new code. By using the source-to-source transformation, we minimize the efforts and errors related to the manual rewriting of the programs.

B. The Loop Transformation Tool

Figure 6 shows a typical example of our loop transformation tool. For the target function call in a do-loop, we do the transform as follows: 1) lift the function call statement from the scope of do-loop to the upper scope and remove the do-loop statement; 2) push the loop index upper bound (m in the example) back into the function call argument list; and 3) find the array parameters containing the do-loop index (i in the figure) and replace the very index with ":", which means the entire dimension of this array is passed in. The function declaration, which is usually written in another source file, is located and the code is transformed by the following rules: 1) the parameter declarations are transformed in the same way with the function call statement; 2) the dimension of all the local parameters is expanded by 1 if not explicitly disabled by the developers; 3) for arrays whose dimension changes in the declarations, the array reference in the function body should also be changed correspondingly; and 4) move the original do-loop into the body of the function declaration and split the loop.

```
Do i = 1, m
   call F(A, B(i), C(i,:))
End do

Subroutine F(A, B, C)
   !parameter declaration
   real :: A, B
   real, dimension(:) :: C

   !local variable declaration
   real :: X, Y

   !execution
   X = 1
   Y = 1
   call lower1(X, C)
   call lower2(Y, C)
   B = X+Y
   C(:) = C(:) + X*Y
End
```

```
call F(A, B(:), C(:,:), m)

Subroutine F(A, B, C, m)
   !parameter declaration
   real :: A
   real, dimension(:) :: B
   real, dimension(:,:) :: C
   integer :: m

   !local variable declaration
   real, dimension(m) :: X, Y

   !execution
   do i = 1, m
      X(i) = 1
      Y(i) = 2
      call lower1(X(i), C(i, :))
   end do
   do i = 1, m
      call lower2(Y(i), C(i, :))
      B(i) = X(i)+Y(i)
      C(i, :) = X(i)*Y(i)
   end do
End
```

Figure 6. A typical example of the loop transformation tool.

A typical loop transform scenario is shown in Figure 7, which corresponds to the first part of the physics schemes, accounting for over 80% computation time of all the physics parts. In the original code, the OpenMP parallelization scheme happens on the outermost loop of different chunks (as shown in block ① of Figure 7). While such parallelization scheme fits the OpenMP threads on the multi-core CPU, the threads with a long sequence of computations and loads of intermediate variables can easily overwhelm the local faster buffer of the CPE, leading to the extremely low performance of the CPE threads. To resolve such issues, we apply the tool to move the chunk loop to the specific physics scheme function (block ② of Figure 7), and split the large loop into separate loops for each separate physics scheme (block ③ of Figure 7). If the function body at such a level is still too complicated to fit into the 64-KB SPM of the CPE, we can further apply the tool to move the chunk loop to an even deeper level of the call stack. Such as the function of zm_conv_tend function, we could further move the chunk loop to the four sub-components of the zm_conv_tend function (as shown in block ④ and ⑤ of Figure 7).

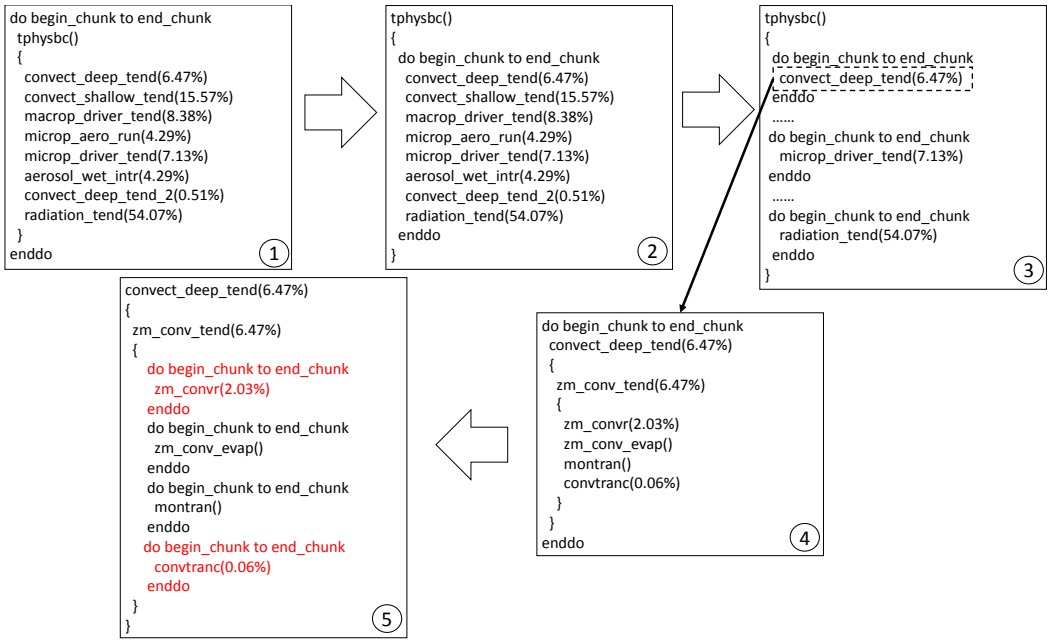

Figure 7. Loop permutation and split for the first part of physics schemes in CAM.

C. Variable Storage Space Analysis and Reduction Tool

Our variable analysis and reduction tool provides a number of basic functions: 1) to estimate the storage requirements of the variable and arrays in the current function region (how large storage is needed); 2) to identify the lifespan of the variables and arrays (how long is the storage needed). Based on the collected information, our tool can then determine whether the variables and arrays of each CPE thread can fit into the 64KB SPM.

In the cases that the 64KB SPM is not large enough to store the values, our tool performs a number of automated optimizations to reduce the storage space.

The most effective optimization is to reuse the memory space for intermediate arrays. In current physics modules, it is common that tens of local arrays are declared and referenced, of which the dimensions are identical. The poor coding style usually makes the SPM insufficient to hold all the local data. We first analyze the lifespan of these local arrays statement by statement in the scope of the function. The arrays whose lifespan do not overlap can reuse the same storage block. As shown in Figure 8, the original Fortran function accesses 7 intermediate arrays (A to G) during the computation process. By analyzing the lifespan of these 7 arrays, which are annotated by the lines above these arrays, we can determine that 4 arrays would provide sufficient space to store these 7 arrays in different stages of the execution process. The mapping of the original arrays and the new arrays can be calculated in advance as the shown by the array tables in the lower part of the figure. Our tool first adds the new array declarations in the function body, and turn the original local array declaration into pointers. Then the pointer assignment statements are inserted according to the mapping.

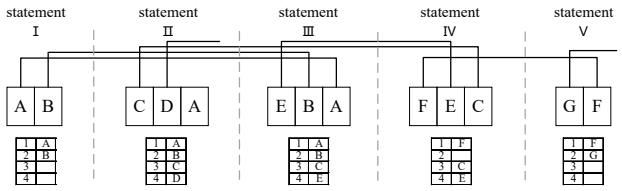

Figure 8. A general example of the loop transformation tool.

For the cases that the functions still involve too many variables to fit into the SPM, we can apply other methods to further reduce the variable storage space. One strategy is to perform a finer control over the multi-dimensional array. As shown in Figure 9, with the support of the Sunway OpenACC, we can specifically request to put a certain row of the 2D array A into the SPM for a given iteration of the inner loop. This feature is an expansion from the standard OpenACC syntax, which enables the CPE OpenACC thread to pre-load the specific row instead of the entire 2D array, thus significantly reducing the required data storage space.

Another expanded feature of the Sunway OpenACC that could potentially improve the data copy efficiency is the introduction of *pack*, *packin*, and *packout* clauses. The *pack*, *packin*, and *packout* clauses provide similar functions to the *copy*, *copyin*, and *copyout* clauses. However, for *pack* clauses, the compiler would go through the variable list,

```
double B[10]; //in
double A[10][10]; //inout

#pragma acc in(B)
for(i = 0; i < 10; i++){

        #pragma acc inout(A[i])
        for(j = 0; j < 10; j++){
                A[i][j] += B[j];
        }

}
```

Figure 9. Support for fine control over array variables: an example.

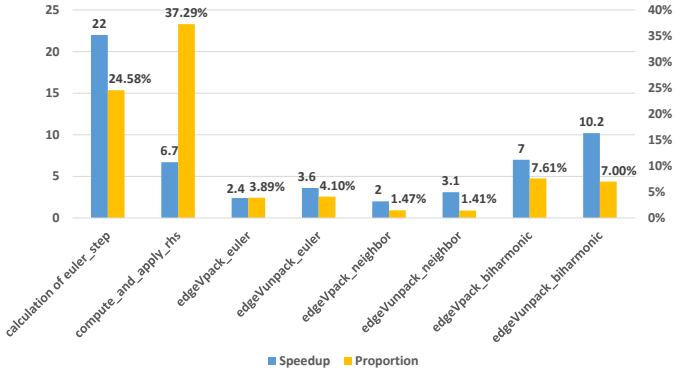

Figure 10. The speedup of major kernels in CAM-SE that we port onto the CPE clusters, and their proportions in the total runtime of the dynamic core part. The speedup is comparing the performance of the kernel running on 1 MPE and 64 CPEs against the performance of the kernel running on only 1 MPE.

pack the distributed variables into one continuous memory space, and perform the transfer of the data more efficiently. Our experiments demonstrate that the *pack* clauses can be quite helpful for the physics schemes that generally involves dozens of input and output parameters.

VII. Results

A. Results of the Kernels Running on CPE Clusters

In this section, we evaluate the individual functions/kernels that we port onto the Sunway processor, and analyze the performance benefits for using the CPE clusters.

Figure 10 shows the speedup of the major kernels in the CAM-SE dynamic core, as well as their proportions in the total runtime of the dynamic core part. The speedup we demonstrate here is comparing the computational performance of the hybrid version that uses both the MPE and the 8x8 CPE cluster against the starting-point version that only uses the MPE.

For the compute-intensive kernels, such as the major computational parts in the $advection\_step$ function (24.58% of the total runtime), and the $compute\_and\_apply\_rhs$ function (37.29% of the total run time), we can achieve a speedup of 7x to 22x. Especially for the most time-consuming $advection\_step$ function, we can achieve a significant speedup of 22x, which is a quite efficient utilization of the 64 CPEs.

For the other parts in these functions, a lot of the operations are memory copies that prepare the message for the following halo communication. These parts are mostly memory-bound, but the multi-threading by the 64 CPEs can still provide a speedup that ranges from 2x to 7x.

Similarly, Figure 11 shows the speedup of the major kernels in the physics schemes of CAM, as well as their proportions in the total runtime of the physics part. Different from the CAM-SE dynamic core, the physics schemes generally do not involve memory copy or communication operations, and are mainly dominated by computations. However, as the physics schemes involve a significantly large code base with different code styles for each scheme, we mainly rely on the loop transformation and variable reduction tools described in Section VI. As a result, the speedup we achieve for different kernels varies with the specific features of the kernel. The first three kernels in Figure 11 all include intensive computation operations. However, only the $microp\_mg1\_0$ kernel demonstrates a significant speedup of 14x, as the intermediate variables and arrays provide a nice fit to the SPM of the CPE clusters after the automated optimizations. In contrast, the speedup of the $convect\_shallow\_tend$ kernel is relatively low (1.6x), mainly because the intermediate variables and arrays (even after the reduction optimization) are too large to fit into the SPM. Therefore, the performance drops down significantly due to the frequent access to the main memory. For the case of the zm_convr kernel, as it involves a deep function call stack (as mentioned in Section VI-B), the performance improvement (7x) is not as high as the $microp\_mg1\_0$ kernel.

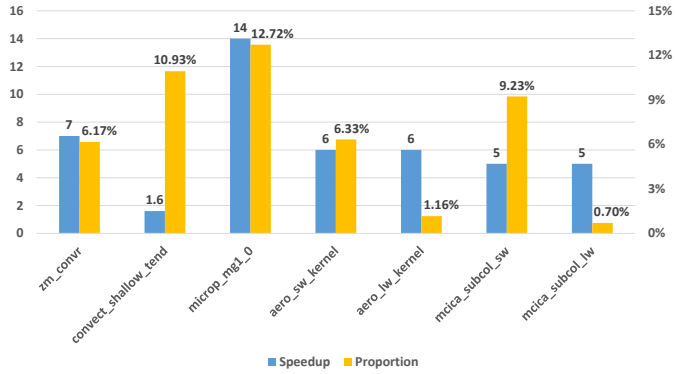

Figure 11. The speedup of major kernels in the physics schemes of CAM that we port onto the CPE clusters, and their proportions in the total runtime of the physics part. The speedup is comparing the performance of the kernel running on 1 MPE and 64 CPEs against the performance of the kernel running on only 1 MPE.

B. Performance of the SE Dynamic Core in Different Parallel Scales

Figure 12 demonstrates the execution time of the entire CAM-SE dynamic core that we can achieve with different parallel scales and different simulation configurations. The parallel scale is described by the number of CGs, as we generally run one MPI process for each CG. Each CG includes 1 MPE and 64 CPEs. For the simulation configuration, we change the resolution and the number of tracers. $ne30$ and $ne120$ refer to the 100-km and 25-km resolution configurations respectively. The number of tracers is denoted by $qsize$, which can be 25 (default configuration), or 108 (heavy chemistry configuration).

While we achieve up to 22x speedup for single kernels within the SE dynamic core, when we consider the entire SE (including computation, memory copy, communication, and the serial parts that simply can not take advantage of the CPEs), the speedup is between 2 times and 4 times. In general, the speedup drops down as the parallel scale increases, which corresponds to the increased portion of communication and reduced portion of computation for each CG. As for the number of tracers, a larger $qsize$ would bring a better speedup, as there would be a larger q to process for the CPE clusters (as discussed in Section V).

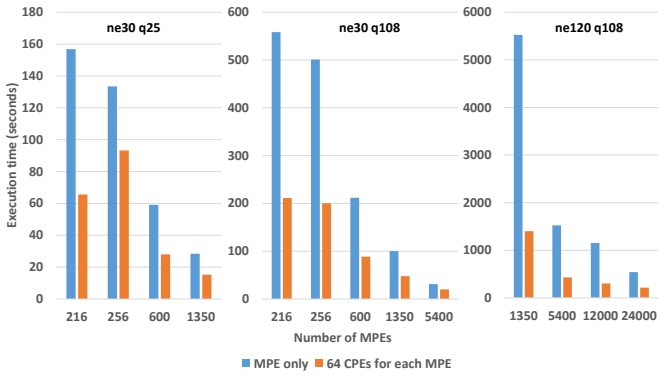

Figure 12. The execution time of the CAM-SE dynamic core that we port onto the CPE clusters with different parallel scales and different simulation configurations. We demonstrate both the time for running the CAM-SE dynamic core on both MPEs and CPE clusters and the time for running on only MPEs. The x axis describes the number of CGs (each CG includes 1 MPE and 64 CPEs). $ne30$ and $ne120$ refer to the 100-km and 25-km resolution configurations respectively. $qsize$ denotes the number of tracers used.

C. Performance of the Physics Part in Different Parallel Scales

In contrast to the dynamic core, as the physics schemes do not involve MPI communications, the performance improvement achieved from the CPE clusters does not change significantly with the parallel scale.

Figure 12 demonstrates the execution time of the entire physics part that we can achieve with different parallel scales and different resolution configurations. With the number of parallel CGs increasing, we see a constant performance improvement of two times by using both MPE and CPEs.

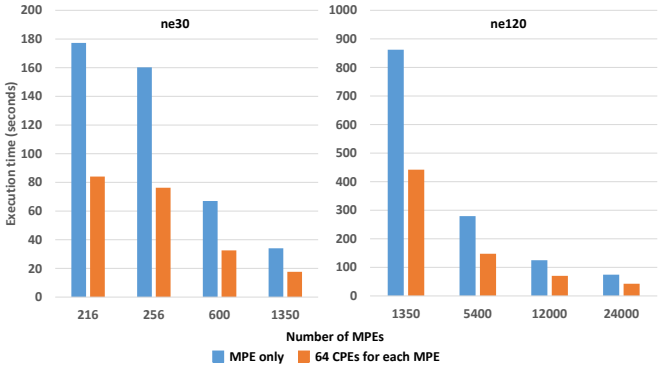

Figure 13. The execution time of the physics part that we port onto the CPE clusters with different parallel scales and different simulation configurations. Similarly, we demonstrate both the time for running on both MPEs and CPE clusters and the time for running on only MPEs. The x axis describes the number of CGs (each CG includes 1 MPE and 64 CPEs). $ne30$ and $ne120$ refer to the 100-km and 25-km resolution configurations respectively.

D. Speedup of the entire CAM model

Figure 14 shows the simulation speed of the CAM model (measured in Model Years Per Day (MYPD)) on the new Sunway supercomputer, with the number of CGs increasing from 1,024 to 24,000. Similar to previous reported results on other systems, the CAM model demonstrates a good scalability on the new Sunway supercomputer system, with the simulation speed increasing steadily with the number of CGs. For the large scale cases, we demonstrate the performances for using MPE only, and using both MPE and CPE clusters. As shown in the last two points in Figure 14, by using both the MPE and the CPE clusters, we can further improve the simulation speed by another 2x. When scaling the CAM model to 24,000 CGs (24,000 MPEs, and 1,536,000 CPEs), we can achieve a simulation speed of 2.81 MYPD.

E. Performance Result Analysis

In previous sections, we provide the general results about the performance of the refactored and optimized CAM-SE model on the Sunway TaihuLight system. The MPE and CPE hybrid architecture of the new Sunway system is largely different from previous heterogeneous systems. While MPE is like a CPU core, and the CPE cluster is like the many-core accelerator, both the CPU and the accelerator are now fused into one chip. The speedup of a complete CG (1 MPE and 64 CPEs) over the one MPE provides an important indication about how well our refactoring method and code-transformation tools can port the heavy codes of CAM-SE,

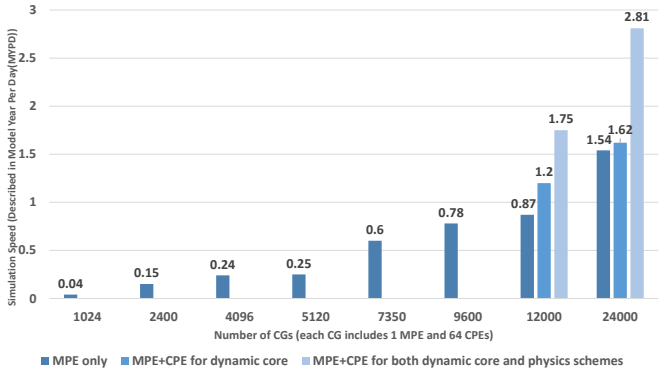

Figure 14. The simulation speed of the CAM model (measured in Model Years Per Day (MYPD)) on the new Sunway supercomputer, with the number of CGs increasing from 1,024 to 24,000. For the large-scale run with 12,000 and 24,000 CGs, we show the performances of the model for three scenarios: (1) using MPE only; (2) using MPE+CPE for the dynamic core; (3) using MPE+CPE for both the dynamic core and the physics schemes.

which were originally written for multi-core CPU, to the new hybrid chip.

Both the MPE and the CPE support 256-bit vector instructions. However, as the MPE supports dual issuing of instructions and the CPE only supports single issuing of instructions, the peak performance of one MPE (23.2 GFlops) is two times of the peak performances of one CPE (11.6 GFlops). Therefore, in terms of computing capability, each CG is equivalent to 33 MPEs. In terms of the memory bandwidth, the measured bandwidths for the MPE and the 64 CPEs are 5 GB/s and 20 GB/s respectively. Thus, each CG is only five times better than each MPE. These metrics provide a general guideline about the performance improvement that we can achieve when porting the code from one MPE to the complete CG with additional 64 CPEs.

The speedup numbers we achieve for various dynamic core and physics kernels align well with the ranges defined by the above metrics. For compute-intensive kernels (such as the $advection\_step$ function), we see a speedup of 22 times, which is close to the 33 theoretical bound. For memory-intensive kernels (such as $edgeVpack\_euler$ and $edgeVpack_neighbor$), the speedup is only 2 to 3 times, which is more bounded by the bandwidth.

In contrast to our work, NCAR, ORNL, Cray, and NVIDIA's collaborative effort on accelerating CAM-SE on Titan [8] was focused on the most expensive kernels in the dynamic core part. When moving from the multi-core CPU to the many-core GPU (with the peak computing performance improved by around 10 times and the peak memory bandwidth improved by around 5 times), the GPU accelerator would only bring around 6× speedup for expensive kernels, and 2× speedup for the entire model.

We originally expect a higher speedup for the kernels in the physics parts, which are generally more compute-intensive, and do not involve any communications. However, due to the complexity of the physics codes, we rely on our automated tools to perform the refactoring and the optimization in this part. We think that the current low speedup for certain kernels are partly due to the constraints of our current tools. Therefore, one of our future plans is to further improve our automated transformation tools, and to involve manual optimizations for certain expensive physics schemes, so as to further improve the performance of the physics part.

For the performance of the entire model, as the execution time is distributed in a large number of kernels, which sometimes involve both memory and communication operations, the ported CAM-SE demonstrates a clear memory-bound behavior. In our tested scenarios, porting onto CPEs only improves the performance by roughly 2 times, which we think is mostly constrained by the memory bandwidth and the communication parts that have not yet been fully optimized. The current model provides a simulation speed of 2.81 MYPD for the resolution of 25 km, using 24,000 CGs (24,000 MPEs and 1,536,000 CPEs), which is similar to the speed of around 2 MYPD for the high-resolution CESM runs at the Yellowstone supercomputer of NCAR [18]. While such a speed is still not good enough for scientists to perform experiments of a few hundreds or even thousands of years, we think it is a good starting point for our future efforts that would further improve the speed and efficiency through other algorithmic and architectural redesigns.

VIII. Conclusion

In this paper, we report our efforts on porting the CAM model to the Sunway TaihuLight many-core supercomputer. Due to the differences between the Sunway many-core processor (4 CGs, each of which consists of 1 MPE and 64 CPEs) and the traditional multi-core CPUs, and especially the 64KB SPM that needs to be explicitly controlled by the user, we perform an extensive refactor of CAM to expose the right level of parallelism to the 64 CPEs in each CG, and apply various optimization techniques to fit the involved variables and arrays into the 64KB SPM of each CPE. For single kernels in both the dynamic core and the physics schemes, we achieve 14x to 22x speedup for kernels that provide a suitable fit to both the computational and memory architecture of the CPE cluster. For kernels that are not suitable, we can still achieve around 2x to 7x speedup after applying the loop transformation tool and various variable storage reduction tool. Our refactored CAM model shows a good scalability on the Sunway TaihuLight supercomputer, and can efficiently use up to 24,000 MPEs and 1,536,000 CPEs, and provide a simulation speed of 2.81 MYPD when using the 25-km resolution.

While the speedup of the entire CAM model is not significant, it provides an important base for us to continue optimizing the performance of such a large and complicated

scientific simulation program. In our future work, we will continue working on our source-to-source translation tools to provide an automated workflow. The goal is to refine the refactor and optimization strategies for more suitable mappings between the algorithm and the architecture, and to improve the simulation speed of the model to a level that makes high-resolution simulation an applicable scientific tool on the Sunway system.

ACKNOWLEDGMENT

The authors are grateful to the reviewers for valuable comments that have greatly improved the paper. This work is partially supported by National key research and development plan of China under Grant No. 2016YFA0602103, China Special Fund for Meteorological Research in the Public Interest under Grant No. GYHY201306062, Natural Science Foundation of China under Grant Nos. 61303003, 41374113, 91530103, 91530323, 51190101, and 61361120098. The corresponding author is Wei Xue (email: xuewei@tsinghua.edu.cn).

REFERENCES

[1] J. Drake, I. Foster, J. Michalakes, B. Toonen, and P. Worley, "Design and Performance of a Scalable Parallel Community Climate Model," *Parallel Computing*, pp. 1571–1591, 1995.

[2] S. Shingu, H. Takahara, H. Fuchigami, M. Y. Y. Tsuda, M. Yamada, Y. Tsuda, and et. al., "A 26.58 Tflops Global Atmospheric Simulation with the Spectral Transform Method on the Earth Simulator," in *In Proceedings of the ACM / IEEE Supercomputing SC2002 conference*, 2002.

[3] X.-J. Yang, X.-K. Liao, K. Lu, Q.-F. Hu, J.-Q. Song, and J.-S. Su, "The TianHe-1A Supercomputer: Its Hardware and Software," *Journal of Computer Science and Technology*, vol. 26, no. 3, pp. 344–351, 2011.

[4] X. Liao, L. Xiao, C. Yang, and Y. Lu, "MilkyWay-2 supercomputer: system and application," *Frontiers of Computer Science*, vol. 8, no. 3, pp. 345–356, 2014.

[5] J. Michalakes and M. Vachharajani, "GPU acceleration of numerical weather prediction," in *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, April 2008, pp. 1–7.

[6] R. Kelly, "GPU Computing for Atmospheric Modeling," *Computing in Science and Engineering*, vol. 12, no. 4, pp. 26–33, 2010.

[7] Z. Wang, X. Xu, N. Xiong, L. Yang, and W. Zhao, "GPU Acceleration for GRAPES Meteorological Model," in *High Performance Computing and Communications (HPCC), 2011 IEEE 13th International Conference on*, Sept 2011, pp. 365–372.

[8] I. Carpenter, R. Archibald, K. Evans, J. Larkin, P. Micikevicius, M. Norman, and et. al., "Progress towards accelerating HOMME on hybrid multi-core systems," *International Journal of High Performance Computing Applications*, vol. 27, no. 3, pp. 335–347, 2013.

[9] T. Shimokawabe, T. Aoki, C. Muroi, J. Ishida, K. Kawano, T. Endo, and et. al., "An 80-Fold Speedup, 15.0 TFlops Full GPU Acceleration of Non-Hydrostatic Weather Model ASUCA Production Code," in *High Performance Computing, Networking, Storage and Analysis (SC), 2010 International Conference for*, Nov 2010, pp. 1–11.

[10] S. Xu, X. Huang, Y. Zhang, H. Fu, L.-Y. Oey, F. Xu, and et. al., "gpuPOM: a GPU-based Princeton Ocean Model," *Geoscientific Model Development Discussions*, vol. 7, no. 6, pp. 7651–7691, 2014.

[11] M. Govett, J. Middlecoff, and T. Henderson, "Directive-Based Parallelization of the NIM Weather Model for GPUs," in *Accelerator Programming using Directives (WACCPD), 2014 First Workshop on*, Nov 2014, pp. 55–61.

[12] B. Cumming, C. Osuna, T. Gysi, M. Bianco, X. Lapillonne, O. Fuhrer, and T. C. Schulthess, "A review of the challenges and results of refactoring the community climate code COSMO for hybrid Cray HPC systems," *Proceedings of Cray User Group*, 2013.

[13] J. M. Dennis, M. Vertenstein, P. H. Worley, A. A. Mirin, A. P. Craig, R. Jacob, and et. al., "Computational performance of ultra-high-resolution capability in the Community Earth System Model," *International Journal of High Performance Computing Applications*, vol. 26, no. 1, pp. 5–16, 2012.

[14] R. B. Neale and et al., "Description of the NCAR Community Atmosphere Model (CAM 5.0)," Natl.Cent. for Atmos. Res., Boulder, Colo., Tech. Rep. Note NCAR/TN-4861STR.

[15] J. M. Dennis, J. Edwards, K. J. Evans, O. Guba, P. H. Lauritzen, A. A. Mirin, and et. al., "CAM-SE: A scalable spectral element dynamical core for the Community Atmosphere Model," *International Journal of High Performance Computing Applications*, vol. 26, no. 1, pp. 74–89, 2012.

[16] J. Linford, J. Michalakes, M. Vachharajani, and A. Sandu, "Multi-core acceleration of chemical kinetics for simulation and prediction," in *High Performance Computing Networking, Storage and Analysis, Proceedings of the Conference on*, Nov 2009, pp. 1–11.

[17] H. Fu, J. Liao, J. Yang, L. Wang, Z. Song, X. Huang, C. Yang, W. Xue, F. Liu, F. Qiao, W. Zhao, X. Yin, C. Hou, C. Zhang, W. Ge, J. Zhang, Y. Wang, C. Zhou, and G. Yang, "The sunway taihulight supercomputer: system and applications," *Science China Information Sciences*, pp. 1–16, 2016. [Online]. Available: http://dx.doi.org/10.1007/s11432-016-5588-7

[18] R. J. Small, J. Bacmeister, D. Bailey, A. Baker, S. Bishop, F. Bryan, J. Caron, J. Dennis, P. Gent, H.-m. Hsu, M. Jochum, D. Lawrence, E. Muoz, P. diNezio, T. Scheitlin, R. Tomas, J. Tribbia, Y.-h. Tseng, and M. Vertenstein, "A new synoptic scale resolving global climate simulation using the community earth system model," *Journal of Advances in Modeling Earth Systems*, vol. 6, no. 4, pp. 1065–1094, 2014. [Online]. Available: http://dx.doi.org/10.1002/2014MS000363

[19] D. Quinlan and C. Liao, "The ROSE Source-to-Source Compiler Infrastructure," in *Cetus Users and Compiler Infrastructure Workshop, in conjunction with PACT 2011*, October 2011.