



# A GPU-enabled acceleration algorithm for the CAM5 cloud microphysics scheme

Yan Hong<sup>1</sup> · Yuzhu Wang<sup>1</sup> · Xuanying Zhang<sup>1</sup> · Xiacong Wang<sup>2</sup> · He Zhang<sup>2</sup> · Jinrong Jiang<sup>3</sup>

Accepted: 27 April 2023 / Published online: 9 May 2023

© The Author(s), under exclusive licence to Springer Science+Business Media, LLC, part of Springer Nature 2023

## Abstract

The National Center for Atmospheric Research released a global atmosphere model named Community Atmosphere Model version 5.0 (CAM5), which aimed to provide a global climate simulation for meteorological research. Among them, the cloud microphysics scheme is extremely time-consuming, so developing efficient parallel algorithms faces large-scale and chronic simulation challenges. Due to the wide application of GPU in the fields of science and engineering and the NVIDIA's mature and stable CUDA platform, we ported the code to GPU to accelerate computing. In this paper, by analyzing the parallelism of CAM5 cloud microphysical schemes (CAM5 CMS) in different dimensions, corresponding GPU-based one-dimensional (1D) and two-dimensional (2D) parallel acceleration algorithms are proposed. Among them, the 2D parallel algorithm exploits finer-grained parallelism. In addition, we present a data transfer optimization method between the CPU and GPU to further improve the overall performance. Finally, GPU version of the CAM5 CMS (GPU-CMS) was implemented. The GPU-CMS can obtain a speedup of 141.69× on a single NVIDIA A100 GPU with I/O transfer. In the case without I/O transfer, compared to the baseline performance on a single Intel Xeon E5-2680 CPU core, the 2D acceleration algorithm obtained a speedup of 48.75×, 280.11×, and 507.18× on a single NVIDIA K20, P100, and A100 GPU, respectively.

**Keywords** High-performance computing · Graphics processing unit · Compute unified device architecture · Cloud microphysics

---

✉ Yuzhu Wang  
wangyz@cugb.edu.cn

Extended author information available on the last page of the article

## 1 Introduction

In order to do a more specific analysis and make a better understanding of global climate, the Climate and Global Dynamics Division of the National Center for Atmospheric Research (NCAR) has concentrated on developing climate simulation systems over the past twenty years. They developed the Community Atmosphere Model (CAM) which is a three-dimensional (3D) global atmospheric model. CAM has been substantially modified with a range of enhancements and improvements in the representation of physical processes since CAM version 4 (CAM4). The most significant improvement is that the combination of physical parameterization has been enhanced, making it possible for users to simulate full aerosol–cloud interactions, including cloud droplet activation by aerosols, the precipitation process due to dependent behavior of particle size, and the explicit radiative interaction of cloud particles [1–3]. In 2004, a new double-moment bulk microphysics scheme was described in [4], which predicts the number concentrations and mixing ratios of four hydrometeor substances: droplets, cloud ice, rain, and snow.

Obviously, calculations and specifications for the condensed phase optics (aerosols, liquid cloud droplets, hydrometeors, and ice crystals) taken from cloud microphysics consume tremendous time and computing resources. Hence, the demand for computing resources has been growing in tandem with the complexity of climate simulation systems. In the late 1990s, high-performance computing (HPC) came to be widely known in the field of scientific computing, with applications for petroleum exploration, weather forecasting, aerospace, and other computation-intensive research. Nowadays, the modern graphics processing unit (GPU) has become a reliable alternative to a central processing unit (CPU) in dealing with data-intensive, computation-intensive, and time-intensive problems by combining features of high parallelism, multi-threaded multicore processors, high-memory bandwidth, general-purpose computing, and low cost with compact size. Due to the advantages of the speed related to HPC, more and more meteorologists choose to use GPU to improve parallel computing efficiency [5–8]. The emergence of NVIDIA's Compute Unified Device Architecture (CUDA) increased the use of GPU in a wide range of scientific research [9].

The Chinese Academy of Sciences-Earth System Model (CAS-ESM) [10] uses the Institute of Atmospheric Physics of Chinese Academy of Sciences Atmospheric General Circulation Model version 4.0 (IAP AGCM4.0) [11], as its atmospheric component model. Here, the IAP AGCM4.0 uses the cloud microphysics scheme (CMS) of CAM version 5 (CAM5) as its microphysics parameterization scheme. The CMS process in the CAM5 global atmospheric model is very time-consuming, so developing an efficient parallel algorithm is a challenging and meaningful work. After further analyzing the code structure, we found that it is very suitable for parallel development. This paper proposed a GPU-based parallel algorithm that aims to accelerate the CAM5 CMS. After implementing the algorithm on the CUDA parallel computing platform, a GPU version of the CAM5 CMS, namely, GPU-CMS, was developed. The experimental results show that the

acceleration algorithm obtained a 507.18× speedup on a single A100 GPU without I/O transfer. Nevertheless, in heterogeneous computing, data transfer costs are still significant. Therefore, we propose a data transfer optimization method to reduce transfer time and further improve computing performance. GPU-CMS can achieve a speedup of 141.69× with I/O transfer. Our work has realized the parallelization of the overall code from two aspects of calculation and data transfer: the parallelism of the CAM5 CMS code has been fully exploited, and the data transfer time has been further compressed. Finally, the performance of the code has been evaluated and verified from various aspects.

The main contributions of our study are as follows:

- For the first time, we propose the GPU-based parallel acceleration algorithm of the CAM5 CMS process. Based on the CUDA computing platform, the one-dimensional (1D) parallel acceleration algorithm is proposed and implemented for CAM5 CMS in the horizontal direction. Then, an innovative two-dimensional (2D) parallel acceleration algorithm is proposed and implemented for CAM5 CMS in the horizontal and vertical directions. The proposed parallel algorithm shows excellent computational capability.
- A performance optimization method is proposed for time-consuming data transmission between CPU and GPU. The use of pinned memory technology reduces the I/O transfer time and further enhances the computing performance of GPU-CMS.

The rest of this paper is structured as follows. Section 2 illustrates the related work on accelerating physical parameterization schemes on GPU. Section 3 details the model and code structure of CAM5 CMS. Section 4 presents the GPU-based acceleration algorithms and their implementation. Section 5 evaluates the performance of GPU-CMS and verifies its correctness. Finally, Section 6 summarizes the full paper and makes a proposal for future research.

## 2 Related work

Modern GPUs have massive parallel microprocessors that can provide high-performance services for parallel computations in the areas of science and engineering. As a result, GPUs have become a powerful replacement for traditional microprocessors in HPC systems [12, 13]. For the parameterization methods of cloud microphysical processes on various models and platforms, many scholars have carried out research and experiments on GPU-based acceleration algorithms and achieved good acceleration results.

Mielikainen et al. [14] used a single GPU to accelerate the Weather Research and Forecast (WRF) Single Moment 5-class (WSM5) model, achieving a 389× speedup without I/O (Input/Output) transfer. When using 4 GPUs, the WRF WSM5 model got 357× and 1556× performance improvement with and without I/O transfer respectively. In the same year, the implementation of the GPU-based Kessler microphysics scheme proposed by Mielikainen et al. [15] achieved a 70×

speedup over the implementation of the CPU-based single-threaded scheme. When using 4 GPUs, the scheme achieved a performance improvement of 132 $\times$  speedup with I/O transfer. The obvious superiority of GPU can also be seen in the acceleration of the WRF Single Moment 6-class (WSM6) cloud microphysics scheme in the Global/Regional Assimilation and Prediction System (GRAPES) model proposed by Xiao et al., which obtained over 140 $\times$  speedup compared to the CPU serial version [16]. Similar to this system, the WRF Double-Moment 6-class (WDM6) microphysics scheme, which was proposed by Mielikainen et al., obtained a speedup of 150 $\times$  and 206 $\times$  with and without I/O transfer, respectively [17]. Huang et al. proposed a parallel design of a GPU-based WSM6 scheme that achieved a 216 $\times$  speedup using a single NVIDIA K40 GPU compared to the CPU part running on a single core [18]. Kim et al. proposed a scheme to accelerate the WSM6 microphysics of cross-scale prediction models on GPU using OpenACC instructions and verified the performance of the code on NVIDIA GPU Tesla V100. When transplanting the entire model to the GPU, a 5.71 $\times$  speedup was obtained without I/O transfer [19]. Wang et al. proposed an algorithm to accelerate the rapid radiative transfer model for general circulation models (RRTMG) shortwave radiation scheme using GPU technology (GPU-RRTMG\_SW). Without I/O transfer, the scheme on a single NVIDIA GeForce Titan V achieved a speedup of 38.88 $\times$  over the baseline performance on a single Intel Xeon E5-2680 CPU core [20]. Carlotto et al. proposed a GPU-based SW2D-GPU model (two-dimensional shallow water model) for the high computational cost of the shallow water model, which was implemented in parallel using CUDA C/C++. Compared to its serial version, the SW2D-GPU model achieves a 34 $\times$  speedup [21]. Cao et al. proposed a leap-format-based highly scalable 3D atmospheric general circulation model (AGCM-3DLF), which releases parallelism in all three physical dimensions. The model has good efficiency and scalability on different platforms. On the CAS-Xiandao1 supercomputer, AGCM-3DLF achieved a speed of 11.1 simulated years per day (SYPD) at a high resolution of 25 km [22].

The GPU was originally designed for graphics processing [23]. The CUDA parallel computing platform launched by NVIDIA makes full use of the computing power of the GPU and obtains a significant acceleration effect in the computation-intensive tasks of large-scale data parallel processing [24]. However, the CPU is suitable for processing control tasks with complex logic, and this complementarity facilitated the development of the CPU/GPU heterogeneous parallel computing system period. Due to NVIDIA's mature and stable CUDA platform, more and more scientific research and experiments have chosen CUDA as a development platform for parallel programming. In addition, many proven applications are developed in Fortran. As such, the Portland Group designed the CUDA Fortran language [25–27] to enable GPU acceleration for Fortran applications. Applications in fields such as meteorology and theoretical physics can be rewritten to take advantage of the computing power of GPUs.

From the above study, GPU is suitable for accelerating the physical parameterization schemes of climate system models. At present, there are no relevant studies about accelerating the CAM5 CMS using GPU. Therefore, we choose to implement the algorithm in CUDA Fortran to explore its GPU-based acceleration algorithm in

one and two dimensions, as well as to optimize the time-consuming data transfer process between CPU and GPU.

### 3 Model description

#### 3.1 CAM5 CMS

The two-moment bulk stratiform cloud microphysics scheme in the atmospheric circulation model was proposed by Morrison et al. [4, 28], which was applied in CAM5. In this study, we used this cloud microphysics scheme of CAM5. Cloud microphysics is a physical process that describes the mutual conversion of water, water vapor, and water condensate in clouds, which includes the collection/auto-conversion, condensation/evaporation, deposition/sublimation, freezing/melting, and nucleation/derivation of water vapor, water, and ice crystals in the air. This scheme can predict the number concentrations and mixing ratios of cloud water and cloud ice. The relationship between the specific various water species and the microphysical processes is shown in Fig. 1. Morrison et al. [4, 28] described the CAM5 CMS in detail. In atmospheric circulation models, cloud microphysical processes still occupy a large amount of computational time. Therefore, we analyzed the specific code structure in Sect. 3.2 and developed a GPU parallel acceleration scheme.

#### 3.2 Analysis of code structure

The code structure of CAM5 CMS in IAP AGCM4.0 of CAS-ESM is shown in Fig. 2. The subroutine *mmicro\_pcond* is the most time-consuming part of the CAM5 CMS, which mainly consists of multiple loop segments. Therefore, we split the *mmicro\_pcond* function into 3 sub-functions: *process1*, *process2*, and *process3*. In Fig. 2, “function” is an external function that is called to assist in the

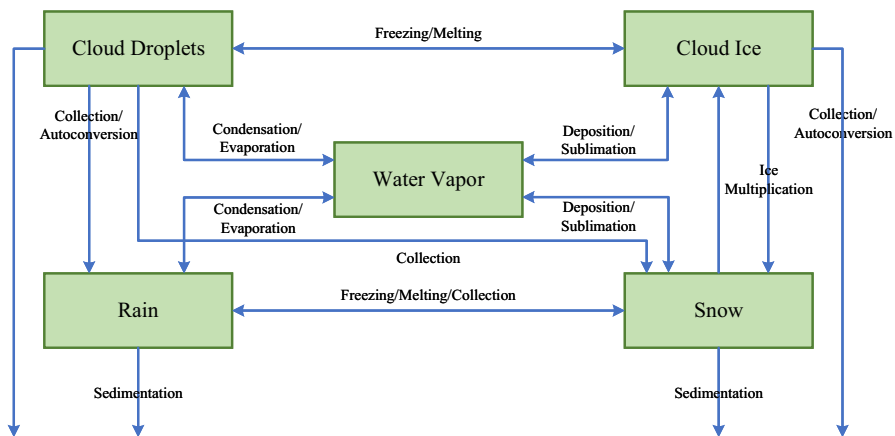
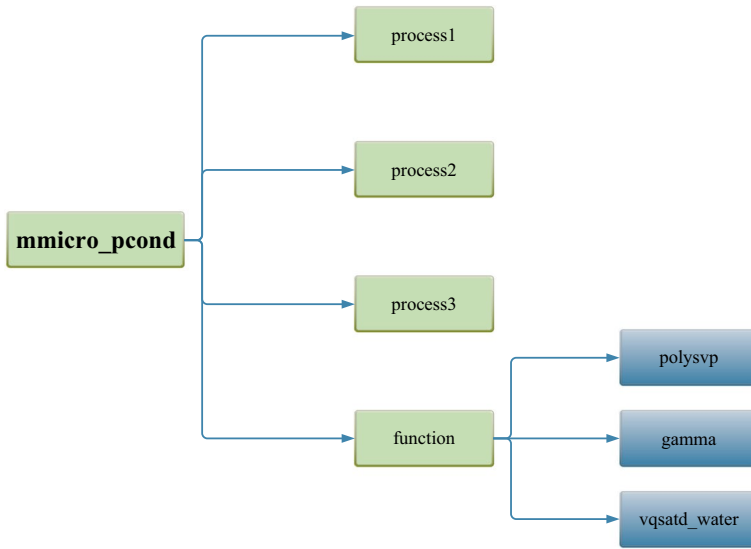


Fig. 1 Box diagram of the CAM5 CMS [4]



**Fig. 2** The code structure of CAM5 CMS in IAP AGCM4.0

calculation process. These external functions need to add device attributes to be called on the device during parallelism. In the *mmicro\_pcond* function, all three sub-functions have two or three loops for the computation process and weak data dependencies, making them suitable for parallel programming using the CUDA programming model.

The subroutine *process1* is used to initialize time-varying parameters and adjust air density for fall-speed parameters. The task of the subroutine *process2* includes (1) initializing sub-step microphysical tendencies, initializing diagnostic precipitation to zero, and initializing vertically integrated rain and snow tendencies; (2) calculating precipitation fractions based on maximum overlap assumption, obtaining in-cloud values of cloud water/ice mixing ratios and numbering concentrations for microphysical process calculations, calculating the potential for droplet activation if cloud water is present, and getting size distribution parameters based on in-cloud cloud water/ice; and (3) beginning microphysical process calculations, such as the auto conversion of cloud liquid water to rain, auto conversion of cloud ice to snow, heterogeneous freezing of cloud water, calculating evaporation/sublimation of rain and snow, and summing over sub-step for average process rates. Due to the complex computing process with a lot of matrix parameters, *process2* consumes a lot of time. The subroutine *process3* mainly calculates sedimentation for cloud water and ice. The external function “*polysvp*” is used to calculate the saturated vapor pressure. The external function “*gamma*” is used to simulate the particle spectral distribution density. The external function “*vqsatd\_water*” is used to calculate the saturated specific humidity. The serial computational procedure of the *mmicro\_pcond* on the CPU is shown in Algorithm 1.

**Algorithm 1** Computing procedure of *mmicro\_pcond*


---

```

1: subroutine mmicro_pcond (parameters)
2:   Initialize microphysical variables
   // Each sub-function has several loops of the calculation process
   // ncol is the number of horizontal columns
   // pver is the number of model layers
   // The specific calculation steps of process1
3:   Do calculation work
4:   do k=1, pver
5:     call vqsatd_water (parameters)
6:     do i=1, ncol
7:       Do calculation work
8:     end do
9:   end do
   //The gamma and polysvp functions are called repeatedly in process2 and
   process3
10:  call process2 (parameters)
11:  call process3 (parameters)
12:  Output and verification
13: end subroutine

```

---

## 4 GPU-enabled acceleration algorithm

### 4.1 Algorithm idea

Similar to the real earth, the CAM5 CMS is described in the form of three dimensions. The *x*-axis, *y*-axis, and *z*-axis represent the earth's longitude, latitude, and the model layers (vertical direction), respectively [29]. The CAM5 CMS model is usually computationally independent in the latitude and longitude dimensions (horizontal direction) and partially independent in the vertical direction, thus allowing for 1D and 2D parallelism in these two directions. In coding CAM5 CMS, the longitude and latitude dimensions are amalgamated into the first dimension of parameter arrays, and the vertical direction is in the second dimension. Data parallelism is primarily achieved by the thread and block indexes of the CUDA architecture that provides a unique global index for each thread, allowing multiple threads to perform the same computation on different data, that is, the single-instruction multi-data (SIMD) mode.

In CAS-ESM, the IAP AGCM4.0 is with a  $1.4^\circ \times 1.4^\circ$  horizontal resolution and 51 model layers in the vertical direction. Thus, the CAM5 CMS has  $256 \times 128 = 32,768$  horizontal grid points. When one GPU is used for acceleration, 32,768 horizontal grid points are computed per time step. In the structure of the program,

the *ncol* variable represents the size of the horizontal dimension. The number of total grid points is settled. When the size of *ncol* is set to 2048, 32,768 grid points will be divided into  $32,768/2048=16$  chunks, with 2048 points on each chunk. In other words, one GPU will complete total computations of 32,768 points through 16 iterations at each time step. This experiment simulates the integration process of CAM5 CMS for one model day, so the total number of iterations is  $16 \times 24$  hours = 384 times. Due to the limitation of global memory, the number of parallel threads that can be started at the same time on one GPU is limited. Moreover, the configuration varies from GPU to GPU, so the corresponding number of calculation iterations and running time will also be different.

As shown in Fig. 3, the execution model of CUDA is a three-layer thread hierarchy. Each kernel runs on a grid, which contain several thread blocks, and each block contains several threads. All threads in the same grid share the same global memory space. Collaboration between threads in the same thread block can be achieved by “synchronization” and “shared memory,” while threads in different thread blocks cannot cooperate. In the grid, we locate a thread by the two coordinate scalars: *blockIdx* (the ID number of the thread block in the grid) and *threadIdx* (the ID number of the thread within the thread block). In the CUDA programming model, the user implements the call of the kernel through the statement of “kernel  $\lll$  grid, block  $\ggg$ .” Among them, “grid” is the number of thread blocks, and “block” is the number of threads per block. And different computing performances can be achieved by modifying the execution configuration of the kernel, see Sect. 5.3 for the specific details of the experiment.

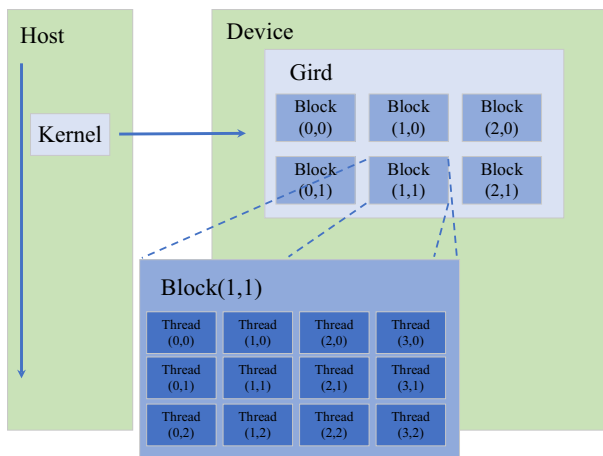


Fig. 3 Hierarchy of threads in CUDA



---

**Algorithm 2** Implementation of 2D Acceleration Algorithm for *mmicro\_pcond*


---

```

1: subroutine mmicro_pcond(parameters)
2:   Initialization parameters
   //Call the vqsatd_water function to calculate the saturation-specific
   humidity
3:   call vqsatd_water(parameters)
4:   Copy input data to GPU
   //grid is the number of thread blocks, block is threads per block
   // Due to the different data dependencies of the arrays of CAM-CMS
   codes in the first and second dimensions, 1D and 2D parallel algorithms are
   proposed. 1D means parallelism is achieved only in the first dimension, and
   2D means finer-grained parallelism is achieved in two dimensions.
   //Call the kernel process1 with 2D decomposition
5:   call process1 <<< grid, block >>>(parameters)
   // The gamma and polysvp functions are called repeatedly in process2
   and process3
   //Call the kernel process2 with 1D decomposition
6:   call process2 <<< grid, block >>>(parameters)
   //Call the kernel process3_1 with 2D decomposition
7:   call process3_1 <<< grid, block >>>(parameters)
   //Call the kernel process3_2 with 1D decomposition
8:   call process3_2 <<< grid, block >>>(parameters)
   //Call the kernel process3_3 with 2D decomposition
9:   call process3_3 <<< grid, block >>>(parameters)
   // Wait for the kernel execution to complete, call synchronous function
10:  istat=cudaDeviceSynchronize()
11:  Copy result to host
   // Judge whether the data calculation is completed
12:  If it is not completed goto 1
   // Output activated liquid and ice
13: end subroutine

```

---

In the heterogeneous computing process of CPU/GPU, we take the original program as the calling function and rewrite each loop in it as a sub-function called by the kernel. Moreover, the kernel is called on the host and executed on the device (the specific calculation flow of the 2D GPU-CMS acceleration algorithm based on the CPU/GPU heterogeneous process is shown in Algorithm 2). The CUDA Fortran execution flow of the GPU-CMS is illustrated below:

- (1) Define and allocate CPU memory for the arrays and variables that participate in the operation on the host. At the same time, define and allocate GPU memory for data that needs to be computed on the device, including input and output, as well as intermediate arrays and variables.

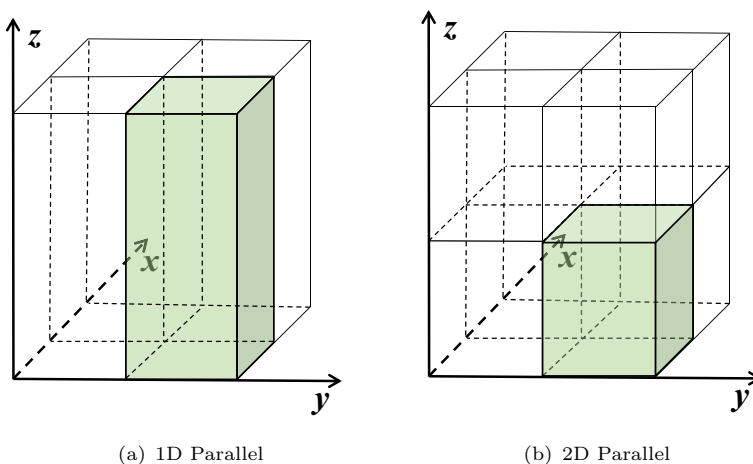
- (2) Initialize data, including input and output arrays and other parameters.
- (3) Transfer the data that was initialized into the global memory of the GPU.
- (4) Select the appropriate thread block and grid size, and call the kernel  $\llcorner grid, block \lrcorner$  function to calculate the microphysical process.
- (5) Transfer results from GPU to CPU.
- (6) Output the results and free the memory both on GPU and CPU.

## 4.2 Implementation of parallel algorithm

### 4.2.1 1D acceleration algorithm

In space, the CAM5 CMS is divided into a 3D grid of horizontal and model layers, as shown in Fig. 4. In Fig. 4, the  $x$ -axis,  $y$ -axis, and  $z$ -axis represent the earth's longitude, latitude, and model layers, respectively. In computation, a 3D grid in space corresponds to a two-dimensional array in the computation process. That is to say, in the CAM5 CMS code, the horizontal section composed of longitude and latitude corresponds to the first dimension of the array, and the model layer corresponds to the second dimension of the array.

Because of the independence of computation in the horizontal direction, the 1D parallel strategy of CAM5 CMS is with domain decomposition in the longitude and latitude dimensions (horizontal direction). It means that each CUDA parallel thread is tasked with a workload of a horizontal "column" (this is determined from the  $x$  and  $y$  axis), as shown in Fig. 4a. Here,  $ncol$  is the size of the first dimension of the array, which is also the number of "columns" participating in the parallel process. According to the sequential and dependent nature of the code structure, the parallel code is implemented as 3 kernels, which were also named *process1*, *process2*, and *process3*, with serial execution between the kernels. The implementation of 1D



**Fig. 4** 3D spatial structure of CAM5 CMS (longitude, latitude, and model layers) and its domain decomposition

acceleration algorithm of *process1* in the horizontal direction is illustrated in Algorithm 3. In addition, the implementation of the other kernels is similar to *process1*, so that will not be described here.

---

**Algorithm 3** Implementation of 1D acceleration algorithm of *process1* based on CUDA Fortran

---

```

1: attributes (global) subroutine process1(parameters)
   // Defining temporary variables and arrays
2:   real(kind=r8):: dum2,dumncu,lami_d(51),.....
   //The index of the x dimension uniquely identifies the thread ID
3:   i=(blockIdx%x-1)*blockDim%x+threadIdx%x
   //Removed a layer of loop
4:   if ((i≥1 .and. i≤ncol)) then
5:     do k=1,pver
6:       Do Assignment and calculation work
7:     end do
8:     do k=1,pver
9:       //Calculate the saturated vapor pressure
10:      call polysvp function
11:     Do calculation work
12:   end do
13: end if
14: end subroutine

```

---

According to the storage hierarchy of CUDA and the structural characteristics of the code, it is necessary to select the appropriate memory to deploy the program data in parallel programming. Therefore, it is essential for us to reduce the memory access latency. Generally speaking, in the CUDA storage structure, the larger the storage space, the slower the memory access speed, where the access speed from fast to slow is Registers  $\Rightarrow$  Caches  $\Rightarrow$  Shared Memory  $\Rightarrow$  Global Memory (Local Memory). First, the local variables in the kernel will be stored in the register, and when the register memory is exhausted, the array will be stored in the local memory, which is about as fast as global memory access. Second, each thread has its independent register and local memory, which can effectively avoid the conflict of memory writing between different threads. Therefore, we defined some of the intermediate arrays and variables as local variables instead of global variables (because global variables are generally stored in global memory). The memory access time of the modified program was optimized.

#### 4.2.2 2D acceleration algorithm

In CAS-ESM, *pver* is 51, which is the number of model layers of CAM5 CMS in the vertical direction. To achieve a higher level of parallelism on the GPU, more parallel threads were opened by making full use of GPU resources. So, a 2D parallel acceleration algorithm was introduced that can perform finer-grained parallelism.

The CAM5 CMS is partially independent in the vertical dimension. Therefore, we further modified the 1D GPU-CMS code to allow parallelism from both horizontal and vertical dimensions for the advance of GPU computational efficiency. As shown in Fig. 4b, each CUDA parallel thread was assigned a workload of a partial “column” (this is determined from the  $x$ ,  $y$ , and  $z$  axes) in the 2D domain decomposition algorithm of the GPU-CMS.

---

**Algorithm 4** Implementation of 2D acceleration algorithm of *process3\_1* based on CUDA Fortran

---

```

1: attributes(global) subroutine process3_1(parameters)
   // Defining temporary variables and arrays
2:   real(kind=r8):: unc, umc, lami_d, ...
   //Index of the  $x$  and  $y$  dimensions uniquely identify the thread ID
3:    $i=(\text{blockIdx}\%x-1)*\text{blockDim}\%x+\text{threadIdx}\%x$ 
4:    $k=(\text{blockIdx}\%y-1)*\text{blockDim}\%y+\text{threadIdx}\%y$ 
   //Removed two layers of loops
5:   if ( $(i \geq 1)$  .and.  $(i \leq ncol)$  .and.  $(k \geq 1)$  .and.  $(k \leq pver)$ ) then
   //Skip all calculations if no cloud water
6:     if (ltrue_d( $i$ ) .eq. 0) then
   //Assign default values for effective radius
7:       Do assignment work
8:       goto 500
9:     end if
10:    Do calculation work
11:    500 continue
12:  end if
13: end subroutine

```

---

On further analysis of the code structure, we found that *process1* supports 2D parallelism, and *process2* maintains 1D parallelism because there are data dependencies between the overall computing processes. Considering the data dependency and data synchronization problems of the *process3* kernel, we decoupled it preliminarily and divided it into three kernels named *process3\_1*, *process3\_2*, and *process3\_3*, respectively. Part of the code in *process3* could only be parallel in the horizontal direction due to data dependency, so the kernel *process3\_2* only performed 1D decomposition. The other code in *process3*, which is *process3\_1* and *process3\_2*, was able to use 2D decomposition. The implementation of 2D acceleration algorithm of *process3\_1* is illustrated in Algorithm 4. The detailed implementations of other kernels will not be described further here.

In the 1D acceleration algorithm, 1D temporary arrays were private to each thread (horizontal column) and were stored in registers or local memory. In the 2D acceleration algorithm (horizontal and vertical directions), register scalars are used instead of temporary arrays in local memory, such as the “lami\_d” variable in Algorithm 3 and Algorithm 4, to achieve the effect of register optimization. By modifying the code, memory usage was reduced, and access speed was also improved.

### 4.3 Optimization of data transfer

The peak bandwidth between device memory and GPU is much larger than the maximum transfer bandwidth between host memory and device video memory. For example, the memory bandwidth of the NVIDIA A100 GPU is 1555 GB/s, while the transfer rate of the PCIe 4.0×16 bus is only up to 64 GB/s. Therefore, transferring data on the PCIe bus between the CPU and the GPU has been the more time-consuming part when using the GPU for parallel acceleration. In this case, it is necessary to use the pinned memory technology to reduce the consumption of data transfer between the device and the host.

In CUDA architecture, there are two types of memory on the host: pageable memory and pinned memory. When allocating memory for variables on the host in the CUDA architecture, pageable memory is used by default, where data may be swapped to the hard disk. As shown in Fig. 5, the CUDA driver will allocate a temporary pinned memory when the data are sent to the device. Then, the data will be placed in this temporary space, and the device will read the data directly from the pinned memory. Pinned memory technology defines and allocates data directly in pinned memory, and the host operating system will not perform paging and swapping operations on this memory, ensuring that it always resides in physical memory. This approach effectively avoids additional overhead with the pinned host buffer, allowing for faster transfers. Since each pinned data requires an allocation of physical memory, which cannot be swapped to disk, it is necessary to use pinned memory appropriately. Excessive use will consume more memory space and reduce overall system performance.

In the process of simulating integration for one model day, the *mmicro\_pcond* function will be called repeatedly, and the data transmission between the host and the device will be performed many times, which will consume a lot of time. In our experiment, the input and output arrays of the GPU-CMS are pinned to maintain these arrays in the physical memory. The data transmission time has been significantly reduced, after the optimization of the pinned memory technology. The specific acceleration result is detailed in Sect. 5.5.

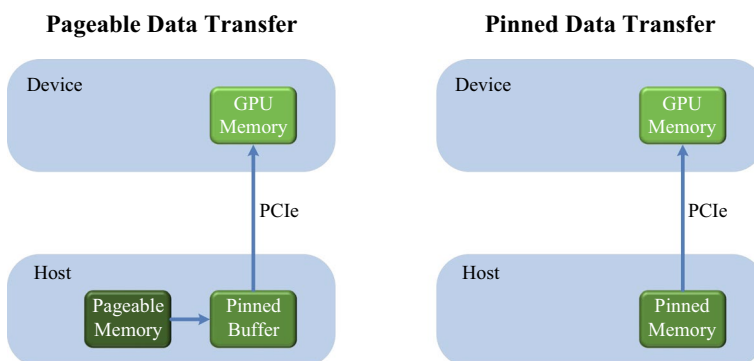


Fig. 5 Pageable memory (left) and pinned memory (right) data transfer

## 5 Results and discussion

To estimate the performance of GPU-CMS, we conducted experiments on K20, P100, and A100 GPU clusters, and Table 1 in Sect. 5.1 shows the detailed configuration. In this paper, we evaluate the overall performance of the code in terms of parallel acceleration algorithms, different experimental platforms, and I/O transfers. The runtime and speedup before and after GPU parallel programming were analyzed and compared using the runtime of the serial CAM5 CMS code on a single Intel Xeon E5-2680 CPU core as a benchmark. The work of verification was also carried out to ensure the accuracy of the code.

### 5.1 Experimental setup

The experiments were conducted on three GPU clusters: NVIDIA K20 cluster at the Computer Network Information Center of the Chinese Academy of Sciences, P100 cluster at the Inner Mongolia Super Brain of the Eastern Supercomputing Cloud, and A100 cluster at the N36 branch of the Beijing Super Cloud Computing Center. Their detailed configurations are listed in Table 1, including operating system, standard memory, memory bandwidth, and CUDA version. It is worth noting that the serial CAM5 CMS in the experiment ran on a single Intel Xeon E5-2680 v2 CPU core on the K20 cluster, and its GPU-CMS ran on a single GPU of the three clusters.

To fully explore the performance of the parallel acceleration algorithm proposed above, we conducted an ideal climate simulation experiment for one model day. In our experiment, the time step of the GPU-CMS is 1 h, and double precision is used to ensure accurate predictions. In this paper, we discuss the runtime of the parallel code with and without data transfer separately. The runtime including data transfer is counted to evaluate the overall performance of the code; the runtime excluding data transfer is counted to derive the best performance when the cloud microphysics is coupled with other physical processes. Without considering the data transfer time, the GPU-CMS runtime is shown in Eq. 1. If data transfer is considered, the runtime

**Table 1** Configurations of GPU clusters

	K20 cluster	P100 cluster	A100 cluster
Specification of CPU			
CPU	Intel Xeon E5-2680 v2@2.8GHz	Intel Xeon E5-2680 v4@2.4GHz	Intel Xeon Silver 4314@2.4GHz
Operating System	CentOS 6.4	Red Hat 4.8.5-39	Red Hat 4.8.5-39
Specification of GPU			
GPU	NVIDIA Tesla K20	NVIDIA Tesla P100	NVIDIA A100
CUDA Cores	2496	3584	6912
Standard Memory	5 GB	16 GB	40 GB
Memory Bandwidth	208 GB/s	732 GB/s	1555GB/s
CUDA Version	6.5	11.6	11.3

is calculated according to Eq. 2. Among them,  $T_{\text{calculation}}$  denotes the computation time of all kernels,  $T_{\text{mmicro\_pcond}}$  denotes the computation time of the most time-consuming *mmicro\_pcond* subroutine in the CAM5 CMS,  $T_{\text{HtoD}}$  denotes the data transfer time from the host to the device, and  $T_{\text{DtoH}}$  denotes the data transfer time from the device to the host.

$$T_{\text{calculation}} = T_{\text{process1}} + T_{\text{process2}} + T_{\text{process3}} \quad (1)$$

$$T_{\text{mmicro\_pcond}} = T_{\text{calculation}} + T_{\text{HtoD}} + T_{\text{DtoH}} \quad (2)$$

## 5.2 Influence of *ncol*

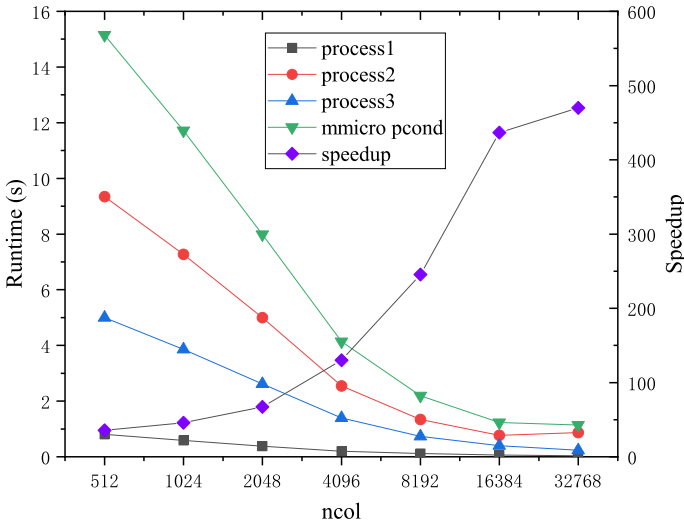
The number of units in the vertical dimension is only 51, so the effect of the variation of the parallelizable grid size in the horizontal dimension is mainly studied. The parallelizable horizontal “column” is represented by *ncol* in the code, so we can change *ncol* to set different horizontal points for one computation on one GPU. Increasing the value of *ncol* means that there are more threads in the horizontal dimension that can be parallelized, and the number of calls to the kernels on the host is relatively reduced, thus significantly improving the performance of the GPU-CMS code. Due to the difference in the size of GPU video memory, we chose to experiment on the best performing A100 GPU, and *ncol* was able to fetch from 512 to 32768. It needs to call the kernel 24 times on the host when GPU-CMS reaches the maximum value 32768 of *ncol*. Without considering I/O transfer, we fixed the thread block size to 128 (because the performance is best when the block size is 128, and the specific experiment is carried out in Sect. 5.3). In this experiment, the runtime and speedup of the 1D GPU-CMS obtained with the change of *ncol* value are studied. The results are shown in Fig. 6.

By analyzing the figure, the following conclusions can be drawn:

- (1) The performance improvement of GPU-CMS is often stronger as the horizontal resolution gets finer. Because the smaller *ncol* values do not have enough threads to take advantage of all GPU resources. However, as the *ncol* value exceeds 16384, the memory usage increases gradually, so the speedup does not improve significantly. There is even a slight increase in the runtime of the *process2* kernel.
- (2) The 1D GPU-CMS achieves optimal performance when *ncol* takes the value of 32768, where a speedup of 469.91× is obtained, and the runtime is compressed from 537.6662 s to 1.1442 s.

## 5.3 Comparison of shape and size of thread blocks

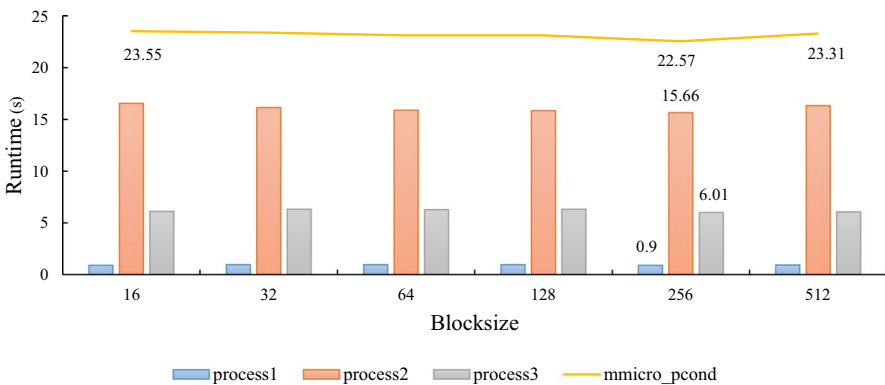
Apparently, there are a lot of factors that will have an impact on the runtime of the GPU-accelerated version. According to the CUDA structure, the block size is one of the main influencing factors. In other words, the number and dimensions of threads



**Fig. 6** The influence of *ncol* on the runtime (s) and speedup of GPU-CMS on a single A100 GPU, where the block size = 128

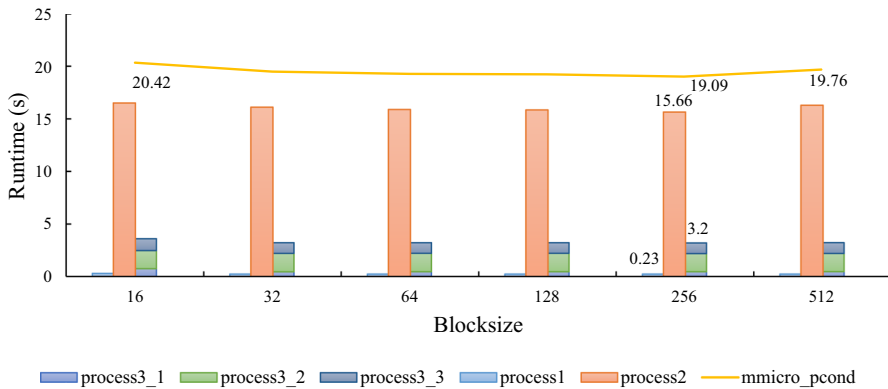
in one execution block will determine the computation time indirectly. Hence, we explore the optimal shape of the thread block. Generally, thread wrap is the minimal execution unit in a program while thread block is the basic activated unit. There are always 32 threads in a thread wrap. In a thread warp, all threads execute in a manner of single instruction multi-threading (SIMT). Therefore, the block size should be selected in multiples of 32 to set up the contrast experiments. A block of size 16 is selected in the experiment and used as a reference group to verify the selection rule.

The runtime of the CAM5 CMS on a single Intel Xeon E5-2680 v2 CPU is 537.6662 s. Without taking into account the I/O transfer, Figs. 7 and 8 portray the runtime of the 1D and 2D GPU-CMS on a single K20 GPU. Among them, the



**Fig. 7** The influence of block size on the runtime (s) of 1D GPU-CMS on a single K20 GPU, where the *ncol* = 2048





**Fig. 8** The influence of block size on the runtime (s) of 2D GPU-CMS on a single K20 GPU, where the y dimension size is 2 and  $ncol = 2048$

column represents the runtime of each kernel, and the polyline is the overall runtime of the *mmicro\_pcond* function. The stacked histogram section of Fig. 8 shows the total runtime of the *process3*, which was split into three kernel in 2D parallelism.

We attempted to explore the optimal shape of a thread block to examine the impact of different combinations of thread block settings on program performance, without considering the I/O transfer. In comparison, it is found that the optimal computational efficiency is obtained for a 1D thread block organization of  $256 \times 1$  and a 2D thread block organization of  $256 \times 2$ . Moreover, the 2D acceleration performance is significantly better than the 1D acceleration performance, and the specific speedup is shown in Table 2.

Compared with the 1D speedup, the speedup of 2D parallelism is significantly increased. The speedup of *process1* and *process3* was  $3.95 \times$  and  $1.88 \times$ , and the speedup of the *mmicro\_pcond* function is  $1.18 \times$ .

As we want to use the organization of threads efficiently to speed up our program, therefore, we considered it valuable to test the impact of thread block size on performance, in addition to thread block shape. We investigated the effect of  $x$ -dimension size on runtime in the 1D or 2D acceleration algorithm, where the  $y$ -dimension size of a 2D thread block is fixed to 2. Figs. 7 and 8 show that the best performance is achieved when the thread block size is 128 or 256 in the  $x$  dimension. Although the impact of the thread block size is not noticeable, the maximum difference is close

**Table 2** Runtime (s) and speedup of 1D and 2D GPU-CMS on a single K20 GPU

Dimension	Subroutines runtime (s)			<i>mmicro_pcond</i>
	<i>process1</i>	<i>process2</i>	<i>process3</i>	
1D	0.89923	15.6642	6.00807	22.5715
2D	0.22777	15.6642	3.20169	19.09366
Speedup	<b>3.95</b>	<b>1</b>	<b>1.88</b>	<b>1.18</b>

Here, the 1D block size = 256, 2D block size = (256,2)  $ncol = 2048$ ; the bold values are used to represent the speedup

to about 1 s, which has an influence on the performance of the algorithm. Theoretically, the memory access latency is hidden to some extent as the thread block size increases, and optimal performance is achieved. When the block size is too small, the threads will not be fully utilized, and the runtime will be relatively long, as in the case of block size 16 in Figs. 7 and 8. When the block size is too large, the thread requires too many resources, such as constant memory and shared memory, and the application storage space of the thread is larger than the hardware configuration, which may sometimes cause the kernel to fail to start. As shown in Figs. 7 and 8 when the block size is 512, although the kernel is successfully started, the runtime increases significantly. In this case, CUDA will ensure the resource supply by forcing the number of blocks to be reduced, which will also not fully utilize all threads and achieve the best performance. In summary, the highest computational efficiency can be achieved when the block size is 128/256. Therefore, the thread block size is set to (128,2) uniformly in the following experiments.

#### 5.4 Evaluation of different GPUs

With the continuous evolution of NVIDIA GPU architecture, upgrades, and breakthroughs are being made in the number of CUDA cores, video memory size, peak floating-point performance, and bandwidth. As a result, the different GPUs have different computational performances. We further examined how the performance of GPU-CMS varies with different GPUs on the three GPU clusters introduced in Sect. 5.1. At the same time, the portability of our parallel algorithm is verified.

The runtime and the corresponding speedup of the 2D GPU-CMS on the K20, P100, and A100 GPUs are shown in Tables 3 and 4 without the I/O transfer time. By analyzing the table, we can draw the following conclusions:

- (1) The maximum *ncol* of the K20 GPU can be taken to 4096 with optimal performance. P100 and A100 GPUs show the best performance at *ncol*=16384. Therefore, the experimental data with *ncol* of 4096 and 16384 were selected for comparison.
- (2) When *ncol*=4096, the speedup of the GPU-CMS on one K20, P100, and A100 are 48.75×, 163.17×, and 169.85×. When the value of *ncol* is small, P100 and

**Table 3** Runtime (s) and speedup of GPU-CMS on different GPUs *ncol* = 4096

Device	Subroutines runtime (s)			<i>mmicro_pcond</i>	Speedup
	<i>process1</i>	<i>process2</i>	<i>process3</i>		
CPU	39.7767	333.5353	164.3542	537.6662	–
K20	0.2245	8.3555	2.4483	11.0283	<b>48.75</b>
P100	0.0550	2.4782	0.8358	3.2951	<b>163.17</b>
A100	0.0246	2.5472	0.5937	3.1655	<b>169.85</b>

Here, the block size = (128,2); the bold values are used to represent the speedup

**Table 4** Runtime (s) and speedup of GPU-CMS on different GPUs,  $ncol = 16384$ 

Device	Subroutines runtime (s)			<i>mmicro_pcond</i>	Speedup
	<i>process1</i>	<i>process2</i>	<i>process3</i>		
CPU	39.7767	333.5353	164.3542	537.6662	–
P100	0.0535	1.3235	0.5425	1.9195	<b>280.11</b>
A100	0.0219	0.7545	0.2837	1.0601	<b>507.18</b>

Here, the block size = (128,2); the bold values are used to represent the speedup

A100 show similar performance. When the value of  $ncol$  is greater, the performance advantage of the A100 GPU will gradually increase.

- (3) When  $ncol=16384$ , the speedup of the GPU-CMS on one P100 and A100 can reach 280.11 $\times$  and 507.18 $\times$ . Since the A100's video memory size and memory bandwidth are much larger than those of the K20 and P100, the acceleration performance is better.

## 5.5 Optimization of I/O transfer

In the experiments of our study, two optimization methods are used to reduce unnecessary data transfer. First, storing the intermediate data on the device without passing it back to the host. Second, changing the initialization and definition of some data to the device. Nevertheless, the I/O transfer of some necessary data between CPU and GPU is inevitable. Therefore, the data transmission process needs to be optimized.

We discovered that the data transmission between CPU and GPU becomes the most time-consuming part of the GPU-CMS when the GPU-CMS computation time is compressed to 1.0601 s. We solved this issue by using pinned memory technology to optimize the code, and we got an improvement in the performance. Since the optimal speedup of calculation time is obtained on the A100 GPU. Hence, the A100 GPU is the best choice for us to continue the optimization work of data transfer. The runtime and speedup of the GPU-CMS on the A100 GPU are shown in Table 5, with and without considering the I/O transfer.

It can be seen from Table 6 that the transfer time from the host to the device gets a speedup of 1.62 $\times$ , and the transfer time from the device to the host gets a speedup of 1.68 $\times$ , after the optimization of pinned memory technology. The speedup of the

**Table 5** Runtime (s) of GPU-CMS I/O transfer on a single A100 GPU

States	Computing time (s)	I/O transfer		<i>mmicro_pcond</i>	Speedup
		HtoD	DtoH		
Without I/O	1.0601	–	–	1.0601	<b>507.18</b>
With I/O	1.0601	2.7085	1.7881	5.5567	<b>96.76</b>

Here, the block size = (128,2) and  $ncol = 16384$ ; the bold values are used to represent the speedup

**Table 6** Runtime (s) of GPU-CMS with the optimization on a single A100 GPU

States	Computing time (s)	I/O transfer		<i>mmicro_pcond</i>	Speedup
		HtoD	DtoH		
Without optimization	1.0601	2.7085	1.7881	5.5567	<b>96.76</b>
With optimization	1.0601	1.6704	1.0644	3.7949	<b>141.68</b>
Speedup	–	<b>1.62</b>	<b>1.68</b>	<b>1.46</b>	<b>1.46</b>

Here, the block size = (128,2) and *ncol* = 16384; the bold values are used to represent the speedup

entire GPU-CMS is increased by 1.46×, from 96.76× to 141.68×. In general, the pinned memory technology performs well in data transfer.

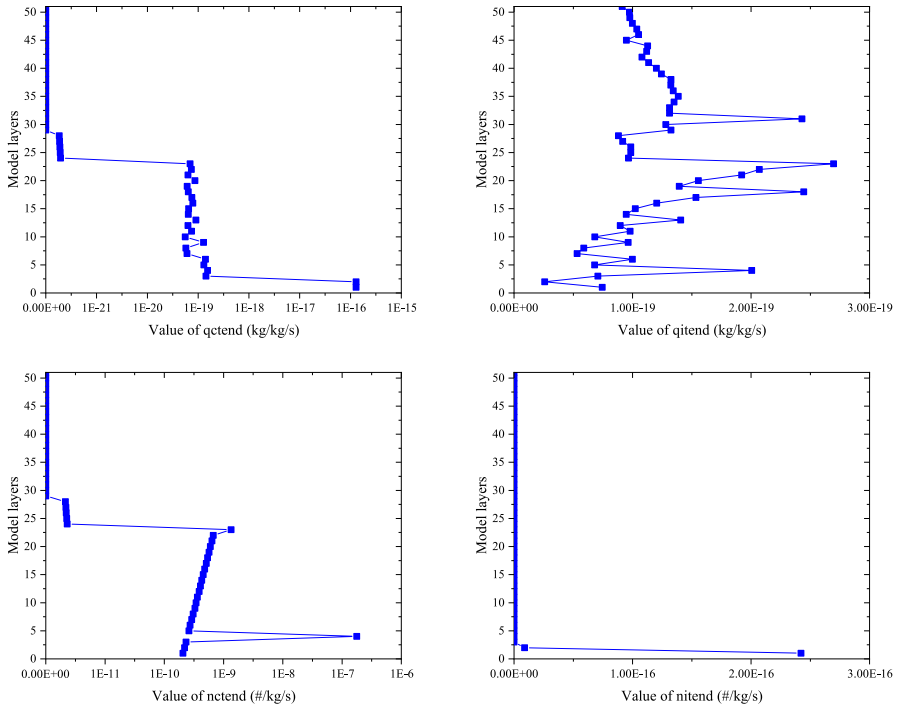
### 5.6 Verification

Clouds exert a significant impact on the short-wave and long-wave radiative transmission, acting as pipes for the conversion of water vapor into precipitation, and as an important part of heat transfer by releasing latent heat. Therefore, verifying the accuracy of the GPU-based cloud microphysical scheme is crucial for the overall atmospheric model operation [4]. Our work aims to parallelize the computation process of CAM5 CMS. Then, we focused on the computational error, and the model error was not in our consideration. The CAM5 CMS in this paper mainly predicts the number concentration and mixing ratio of cloud water and cloud ice. Therefore, the “*qctend*,” “*qitend*,” “*nectend*,” and “*nitend*” variables for the calculation of the concentration and mixing ratio increment process were selected for validation, examining the differences between the CPU code and the GPU code at each model layer. For validation, we ran the simulation for one model day using a horizontal grid size of 128×256=32768, a model layer count of 51, and an integration time step of 1 h.

$$\text{RMSD} = \sqrt{\frac{1}{n} \sum_{i=1}^n (x_i - \hat{x}_i)^2} \tag{3}$$

The root means square difference (RMSD) is selected to verify the correctness of the GPU-CMS code, which was used to calculate the absolute differences between runs on different devices (here refers to CPU and GPU) and to calculate the vertical distribution of the overall average horizontal dimension [19]. The detailed calculation is shown in Eq. 3. where, *i* = 1,..., 32768 is the index of the horizontal cells on each model layer, *n* is the number of horizontal grid points, and *x* is the sum of the calculated results of all horizontal cells on the vertical layer index *i* for a model day. (*x<sub>i</sub>* and *ŷ<sub>i</sub>* represent serial and parallel calculation results respectively). The experimental results are shown in Fig. 9 (RMSD diagram of the vertical layer).

We verified whether there is a significant difference between the results of the parallel GPU-CMS code and the original serial code. On the premise of ensuring correctness, the notable acceleration performance is more trustworthy. In CUDA



**Fig. 9** Time mean RMSD vertical profiles value for  $qctend$ ,  $qitend$ ,  $nctend$ , and  $nitend$  over one model day. The x-axis represents the values of the variables and the y-axis show the model layers of the model

parallel computing, the computation-intensive and data-independent codes are ported to the GPU for computation. Therefore, both CPU and GPU floating-point operations and math functions such as `sqrt` will introduce small calculation errors, but do not cause large deviations. From Fig. 9, it can be seen that the serial and parallel codes do not differ significantly during the integration of one model day. The “ $qctend$ ” and “ $nctend$ ” variables have errors in the first half of the model layers and no errors in the second half. Although “ $qitend$ ” seems to fluctuate greatly, the overall error range is around  $1.0E-19$ , which is reasonable. The “ $nitend$ ” variable is almost error-free. The time mean RMSD values for the “ $qctend$ ,” “ $qitend$ ,” “ $nctend$ ,” and “ $nitend$ ” variables were  $4.30E-18$ ,  $1.20E-19$ ,  $5.14E-09$ , and  $4.92E-18$ , with very small magnitudes. The error is small and within the acceptable range.

In order to further verify the correctness of the code, we integrated the code into the entire CAS-ESM system. Due to the limited running time of the platform, we can only simulate the integral running for seven model days. And the “PRECL” variable for the calculation of the precipitation rate was selected for validation (the rest of the variables are similar). Figure 10 is the result graph of the “PRECL” variable after seven model days of running in CAS-ESM system on the CPUs. And Fig. 11 is the meteorological error graph of the “PRECL” variable after seven model days of running in CAS-ESM system on the K20 GPUs. It can be seen from Fig. 11 that the error is within a reasonable range and is acceptable. Then, we can conclude that the

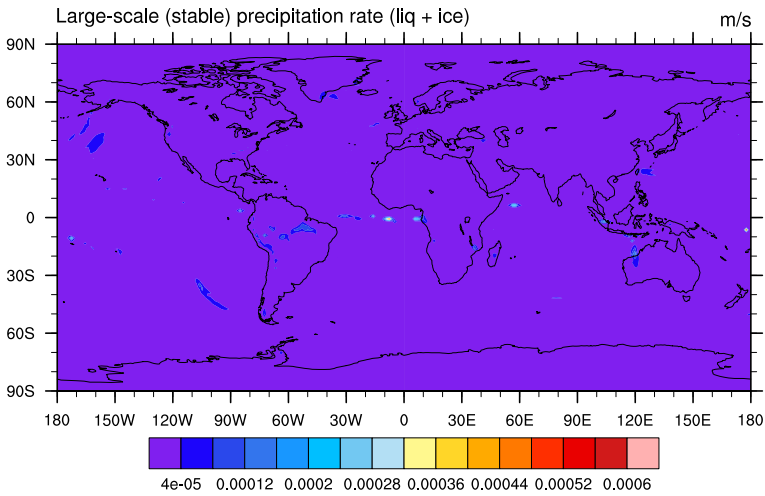


Fig. 10 Precipitation rate simulated by CAS-ESM CAM5 CMS on CPUs

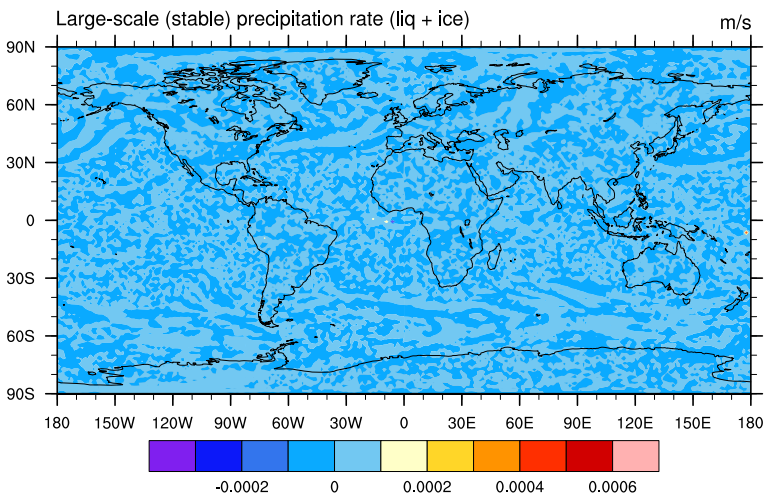


Fig. 11 The error range of precipitation rate simulated by CAS-ESM GPU-CMS on GPUs

error resulting from the modification of the code is small and within the acceptable range. Therefore, our GPU-CMS code is correct and efficient.

### 5.7 Discussion

- (1) Mielikainen et al. obtained a 70× speedup for the Kessler microphysics scheme on one single GPU [15], while our GPU-CMS obtained a speedup of 141.69×, which is obviously better. The CUDA C-based WSM6 scheme proposed by

Huang et al. obtained a speedup of 216 $\times$  on one single NVIDIA K40 GPU [18] with a horizontal grid size of 433 $\times$ 308. The horizontal grid size in this paper is 128 $\times$ 256. If the resolution increases, GPU-CMS will get a better speedup than 141.69 $\times$ . Increasing the resolution to do more in-depth experiments is one of our next works. Without considering the I/O transfer, Mielikainen et al.'s [14] single-GPU-based parallel method for Weather Research and Forecasting (WRF) Single Moment Class 5 (WSM5) achieves a 389 $\times$  speedup, while our GPU-CMS parallel work achieves a speedup of 507.18 $\times$ . At the same time, we compared our GPU-CMS work with Kim J Y, et al.'s [19] proposal to use OpenACC instructions to transplant the WSM6 microphysics of the cross-scale prediction model to the GPU for acceleration. The parallel performance of CUDA is obviously better than that of OpenACC. Of course, this is also determined by analyzing the specific code structure.

- (2) CUDA Fortran was chosen to implement parallelism in this experiment for the following reasons: (1) For the implementation on GPU, Fortran based on PGI is more concise in syntax than C language. (2) Because there is no error caused by different languages, CUDA Fortran is more accurate and has less error than CUDA C. While CUDA Fortran is easier to implement, CUDA C is more mature and generally performs better than CUDA Fortran. Therefore, we will continue to develop the CUDA C version of the code [30].
- (3) Without considering the I/O transfer, the acceleration performance of GPU-CMS will be better. With the integration of GPU-CMS into the whole CAS-ESM system, part of the data transfer process is reduced. At present, I/O transfer time still occupies most of the running time, which can be further reduced by CUDA stream and other technologies to achieve the maximum reduction of I/O transfer time.

## 6 Conclusion and future work

It is an entirely new challenge to accelerate the CAM5 CMS by using GPU. This paper presented the acceleration algorithm of the CAM5 CMS (namely GPU-CMS) on one GPU. First, the characteristics and code structure of the CAM5 CMS are analyzed. On the basis of this work, a parallel acceleration algorithm based on 1D region decomposition was proposed using the CUDA programming model. Second, the 2D parallel acceleration algorithm was further proposed. Third, the data transfer process between the host and the device was optimized using the pinned memory technology. As the experimental results in this paper show, our parallel algorithm is efficient. In order to test the acceleration speedup, implemented the original and improved cloud microphysics process (CAM5 CMS and GPU-CMS) in different experimental settings (NVIDIA K20, NVIDIA P100, and NVIDIA A100 GPU) and compared and analyzed. The experimental results indicated that the program can be performed better on the NVIDIA A100 GPU. During the computation of one model day, the 2D GPU-CMS on a single A100 GPU obtained a speedup of 141.69 $\times$  as compared to that in a single Intel Xeon E5-2680 CPU-core, reducing the runtime from 537.6662 s to 1.0601 s. Without considering I/O transmission, the speedup is

increased to 507.18×, which certainly expedites the computation of the CAM5 CMS model. In addition to obtaining better computational efficiency, it is very important to achieve accurate results. We carefully verified our code using RMSD and plotted the meteorological error by running seven model days in the CAS-ESM system, proving that the error of the code is within an acceptable range. In summary, it is feasible, cost-effective, and efficient to accelerate the computational process of the CAM5 CMS with GPU.

Indubitably, the current accelerated GPU-CMS still has some points that need to be enhanced. The future work will focus on the following two points: (1) The accelerated GPU-CMS currently only runs on a single GPU instead of multi-GPU. To completely utilize the thousands of CPU cores and GPUs in the device, the MPI+CUDA hybrid paradigm [31] and OpenMP+CUDA structure [32] should be considered to investigate multi-GPU acceleration algorithms for scalability. Obviously, the implementation of integrating this program onto multiple GPUs presents significant challenges. As a return, the algorithm based on multiple GPUs will achieve much better acceleration results. (2) The data transfer between CPU and GPU is still the most time-consuming part of the GPU-CMS. In this case, CUDA streaming technology can be used for asynchronous data transfer, which can overlap the kernel computation and data transfer process to achieve the purpose of hiding part of the data transfer time and reducing the data transfer time. In addition, the use of coalesced memory accesses techniques and mixed precision techniques to optimize GPU-CMS is also well worth investigating.

**Acknowledgements** This work was supported in part by the National Natural Science Foundation of China under Grant 41931183, in part by the National Key Research and Development Program of China under Grant 2016YFB0200800, and in part by the National Key Scientific and Technological Infrastructure project “Earth System Science Numerical Simulator Facility” (Earth Lab).

**Author contributions** YH helped in methodology, software, and writing—original draft; YW contributed to supervision, conceptualization, methodology, and writing—review and editing; XZ: Writing-original draft; XW, HZ, and JJ helped in writing—review and editing.

**Funding** Not applicable.

**Data availability** The data that support the findings of this study are available on request from the corresponding author.

## Declarations

**Conflict of interest** The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

**Ethical approval** Not applicable.

## References

1. Collins WD, Rasch PJ, Boville BA, Hack JJ, McCaa JR, Williamson DL, Kiehl JT, Briegleb B, Bitz C, Lin S-J, et al (2004) Description of the near community atmosphere model (cam 3.0). NCAR Tech. Note NCAR/TN-464+ STR 226, 1326–1334



2. Neale RB, Chen C-C, Gettelman A, Lauritzen PH, Park S, Williamson DL, Conley AJ, Garcia R, Kinnison D, Lamarque J-F et al (2010) Description of the ncar community atmosphere model (cam 5.0). NCAR Tech Note NCAR/TN-486+ STR 1(1):1–12
3. Conley AJ, Garcia R, Kinnison D, Lamarque J-F, Marsh D, Mills M, Smith AK, Tilmes S, Vitt F, Morrison H et al (2012) Description of the ncar community atmosphere model (cam 5.0). NCAR technical note 3
4. Morrison H, Curry J, Khvorostyanov V (2005) A new double-moment microphysics parameterization for application in cloud and climate models. part i: description. *J Atmos Sci* 62(6):1665–1677
5. Fan Z, Qiu F, Kaufman A, Yoakum-Stover S (2004) Gpu cluster for high performance computing. In: *SC'04: Proceedings of the 2004 ACM/IEEE Conference on Supercomputing*, pp 47–47. IEEE
6. Deng Z, Chen D, Hu Y, Wu X, Peng W, Li X (2012) Massively parallel non-stationary eeg data processing on gpgpu platforms with morlet continuous wavelet transform. *J Internet Serv Appl* 3(3):347–357
7. Chen D, Wang L, Tian M, Tian J, Wang S, Bian C, Li X (2013) Massively parallel modelling & simulation of large crowd with gpgpu. *J Supercomput* 63(3):675–690
8. Yuan Y, Shi F, Kirby JT, Yu F (2020) Funwave-gpu: multiple-gpu acceleration of a boussinesq-type wave model. *J Adv Model Earth Syst* 12(5):e01957
9. Sanders J, Kandrot E (2010) *CUDA by Example: an Introduction to General-purpose GPU Programming*. Addison-Wesley Professional
10. Xiao D, Tong-Hua S, Jun W, Ren-Ping L (2014) Decadal variation of the aleutian low-icelandic low seesaw simulated by a climate system model (cas-esm-c). *Atmos Oceanic Sci Lett* 7(2):110–114
11. Zhang H, Zhang M, Zeng Q-C (2013) Sensitivity of simulated climate to two atmospheric models: interpretation of differences between dry models and moist models. *Mon Weather Rev* 141(5):1558–1576
12. Owens JD, Houston M, Luebke D, Green S, Stone JE, Phillips JC (2008) Gpu computing. *Proc IEEE* 96(5):879–899
13. Nickolls J, Dally WJ (2010) The gpu computing era. *IEEE Micro* 30(2):56–69
14. Mielikainen J, Huang B, Huang H-LA, Goldberg MD (2012) Improved gpu/cuda based parallel weather and research forecast (wrf) single moment 5-class (wsm5) cloud microphysics. *IEEE J Select Topics Appl Earth Observ Remote Sensing* 5(4):1256–1265
15. Mielikainen J, Huang B, Wang J, Huang H-LA, Goldberg MD (2013) Compute unified device architecture (cuda)-based parallelization of wrf kessler cloud microphysics scheme. *Comput Geosci* 52:292–299
16. Xiao H, Sun J, Bian X, Dai Z (2013) Gpu acceleration of the wsm6 cloud microphysics scheme in grapes model. *Comput Geosci* 59:156–162
17. Mielikainen J, Huang B, Huang H-L, Goldberg M, Mehta A (2013) Speeding up the computation of wrf double-moment 6-class microphysics scheme with gpu. *J Atmos Oceanic Tech* 30(12):2896–2906
18. Huang M, Huang B, Gu L, Huang H-LA, Goldberg MD (2015) Parallel gpu architecture framework for the wrf single moment 6-class microphysics scheme. *Comput Geosci* 83:17–26
19. Kim JY, Kang J-S, Joh M (2021) Gpu acceleration of mpas microphysics wsm6 using openacc directives: performance and verification. *Comput Geosci* 146:104627
20. Wang Z, Wang Y, Wang X, Li F, Zhou C, Hu H, Jiang J (2021) Gpu-rrtmg\_sw: accelerating a short-wave radiative transfer scheme on gpu. *IEEE Access* 9:84231–84240
21. Carlotto T, Borges Chaffe PL, Innocente dos Santos C, Lee S (2021) Sw2d-gpu: a two-dimensional shallow water model accelerated by gpgpu. *Environ Modell Softw* 145:105205. <https://doi.org/10.1016/j.envsoft.2021.105205>
22. Cao H, Yuan L, Zhang H, Zhang Y, Wu B, Li K, Li S, Zhang M, Lu P, Xiao J (2023) Agcm-3dlf: accelerating atmospheric general circulation model via 3-d parallelization and leap-format. *IEEE Trans Parallel Distrib Syst* 34(3):766–780. <https://doi.org/10.1109/TPDS.2022.3231013>
23. Fung J, Mann S (2004) Computer vision signal processing on graphics processing units. In: *2004 IEEE International Conference on Acoustics, Speech, and Signal Processing*, vol. 5, pp 93. IEEE
24. Kirk D et al (2007) Nvidia cuda software and gpu parallel computing architecture. In: *ISMM* 7:103–104
25. Wolfe M et al (2012) *Cuda fortran programming guide and reference*. The Portland Group, Release
26. Ruetsch G, Fatica M (2013) *CUDA Fortran for Scientists and Engineers: Best Practices for Efficient CUDA Fortran Programming*, Elsevier

27. NVIDIA: CUDA Fortran Programming Guide and Reference. (2019). [Online]. available at <https://www.pgroup.com/resources/docs/19.1/pdf/pgi19cudaforg.pdf>
28. Morrison H, Gettelman A (2008) A new two-moment bulk stratiform cloud microphysics scheme in the community atmosphere model, version 3 (cam3). part i: description and numerical tests. *J Clim* 21(15):3642–3659
29. Wang Y, Zhao Y, Jiang J, Zhang H (2020) A novel gpu-based acceleration algorithm for a longwave radiative transfer model. *Appl Sci* 10(2):649
30. NVIDIA: “CUDA C Programming Guide v10.0.”. [Online]. [https://docs.nvidia.com/pdf/CUDA\\_C\\_Programming\\_Guide.pdf](https://docs.nvidia.com/pdf/CUDA_C_Programming_Guide.pdf) (2019)
31. Farhatuaini L, Pulungan R (2019) Parallelization of uniformization algorithm with cuda-aware mpi. In: 2019 7th International Conference on Information and Communication Technology (ICoICT), pp 1–6. IEEE
32. Czarnul P (2018) Parallelization of large vector similarity computations in a hybrid cpu+ gpu environment. *J Supercomput* 74(2):768–786

**Publisher's Note** Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Springer Nature or its licensor (e.g. a society or other partner) holds exclusive rights to this article under a publishing agreement with the author(s) or other rightsholder(s); author self-archiving of the accepted manuscript version of this article is solely governed by the terms of such publishing agreement and applicable law.

## Authors and Affiliations

**Yan Hong<sup>1</sup> · Yuzhu Wang<sup>1</sup> · Xuanying Zhang<sup>1</sup> · Xiaocong Wang<sup>2</sup> · He Zhang<sup>2</sup> · Jinrong Jiang<sup>3</sup>**

Yan Hong  
hongyan@email.cugb.edu.cn

Xuanying Zhang  
zxy13meredith@163.com

Xiaocong Wang  
wangxc@lasg.iap.ac.cn

He Zhang  
zhanghe@mail.iap.ac.cn

Jinrong Jiang  
jjr@sccas.cn

<sup>1</sup> School of Information Engineering, China University of Geosciences, Beijing 100083, China

<sup>2</sup> Institute of Atmospheric Physics, Chinese Academy of Sciences, Beijing 100029, China

<sup>3</sup> Computer Network Information Center, Chinese Academy of Sciences, Beijing 100190, China