


Article

A Novel GPU-Based Acceleration Algorithm for a Longwave Radiative Transfer Model

Yuzhu Wang ^{1,*} , Yuan Zhao ¹, Jinrong Jiang ² and He Zhang ³¹ School of Information Engineering, China University of Geosciences, Beijing 100083, China; zy@cugb.edu.cn² Computer Network Information Center, Chinese Academy of Sciences, Beijing 100190, China; jjr@scas.cn³ Institute of Atmospheric Physics, Chinese Academy of Sciences, Beijing 100029, China; zhanghe@mail.iap.ac.cn

* Correspondence: wangyz@cugb.edu.cn

Received: 27 November 2019; Accepted: 11 January 2020; Published: 16 January 2020



Abstract: Graphics processing unit (GPU)-based computing for climate system models is a longstanding research area of interest. The rapid radiative transfer model for general circulation models (RRTMG), a popular atmospheric radiative transfer model, can calculate atmospheric radiative fluxes and heating rates. However, the RRTMG has a high calculation time, so it is urgent to study its GPU-based efficient acceleration algorithm to enable large-scale and long-term climatic simulations. To improve the calculative efficiency of radiation transfer, this paper proposes a GPU-based acceleration algorithm for the RRTMG longwave radiation scheme (RRTMG_LW). The algorithm concept is accelerating the RRTMG_LW in the *g-point* dimension. After implementing the algorithm in CUDA Fortran, the G-RRTMG_LW was developed. The experimental results indicated that the algorithm was effective. In the case without I/O transfer, the G-RRTMG_LW on one K40 GPU obtained a speedup of $30.98\times$ over the baseline performance on one single Intel Xeon E5-2680 CPU core. When compared to its counterpart running on 10 CPU cores of an Intel Xeon E5-2680 v2, the G-RRTMG_LW on one K20 GPU in the case without I/O transfer achieved a speedup of $2.35\times$.

Keywords: high performance computing; graphics processing unit; compute unified device architecture; radiation transfer

1. Introduction

The radiative process, one of the important atmospheric physics processes, is often used for calculating atmospheric radiative fluxes and heating rates [1]. To simulate the radiative process, several radiative transfer models were developed, such as the line-by-line radiative transfer model (LBLRTM) [2], and the rapid radiative transfer model (RRTM) [3]. The RRTM that is a validated model computing longwave and shortwave radiative fluxes and heating rates, uses the correlated-k method to provide the required accuracy and computing efficiency [4], but it still demands enormous computing resources for long-term climatic simulation. To address this issue, as an accelerated version of RRTM, the rapid radiative transfer model for general circulation models (RRTMG) provides improved efficiency with minimal loss of accuracy for atmospheric general circulation models (GCMs) [5]. As a coupled climate system model comprising eight separate component models and one central coupler, the Chinese Academy of Sciences–Earth System Model (CAS-ESM) [6,7] was developed by the Institute of Atmospheric Physics (IAP) of Chinese Academy of Sciences (CAS). As the atmospheric component model of the CAS-ESM, the IAP Atmospheric General Circulation Model Version 4.0 (IAP AGCM4.0) [8,9] used the RRTMG as its radiative parameterization scheme.

Radiative transfer is relatively time-consuming [10,11]. The RRTMG improves the computing efficiency of radiative transfer, but it is still so computationally expensive that it cannot be performed with shorter time steps or finer grid resolutions in operational models [12]. To greatly improve the computational performance, it is beneficial to use high-performance computing (HPC) technology to accelerate the RRTMG.

At present, HPC is widely employed in earth climate system models [13–15]. With the rapid development of HPC technology, due to the features of multithreaded many-core processor, high parallelism, high memory bandwidth, and low cost, the modern graphics processing unit (GPU) has substantially outpaced its central processing unit (CPU) counterparts in dealing with data- and computing-intensive problems [16–19]. Currently, increasing numbers of atmospheric applications were accelerated by the GPUs [20,21]. For example, the WSM5 microphysics scheme from the Weather Research and Forecasting (WRF) model obtained a $206\times$ speedup on a GPU [22].

In view of the booming GPU capability, our previous study used a GPU for accelerating the RRTMG longwave radiation scheme (RRTMG_LW). In this study, the GPU-based acceleration algorithms with one-dimensional (1D) and two-dimensional (2D) domain decompositions for the RRTMG_LW were proposed [23]. However, the RRTMG_LW did not achieve an excellent speedup on a GPU. Therefore, the present paper focuses on the implementation of better GPU-based accelerating methods for the RRTMG_LW. To further accelerate the RRTMG_LW in the CAS-ESM, a GPU-based acceleration algorithm in the *g-point* dimension is proposed. The proposed algorithm enables massively parallel calculation of the RRTMG_LW in the *g-point* dimension. Then, an optimized version of the RRTMG_LW is built by successfully adopting GPU technology, resulting in G-RRTMG_LW. The experimental results demonstrated that the G-RRTMG_LW on one K40 GPU obtained a $30.98\times$ speedup.

The main contributions of this study are as follows:

- (1) To further accelerate the RRTMG_LW with a massively parallel computing technology, a GPU-based accelerating algorithm in the *g-point* dimension is proposed. The aim is to explore the parallelization of the RRTMG_LW in the *g-point* dimension. The proposed algorithm adapts well to the advances in multi-threading computing technology of GPUs and can be generalized to accelerate the RRTMG shortwave radiation scheme (RRTMG_SW).
- (2) The G-RRTMG_LW was implemented in CUDA (NVIDIA's Compute Unified Device Architecture) Fortran and shows excellent computational capability. To some extent, the more efficient computation of the G-RRTMG_LW supports real-time computing of the CAS-ESM. Moreover, the heterogeneous computing of the CAS-ESM is implemented.

The remainder of this paper is organized as follows. Section 2 presents representative approaches that aim at improving the computational efficiency of physical parameterization schemes. Section 3 introduces the RRTMG_LW model and GPU environment. Section 4 details the CUDA-based parallel algorithm in the *g-point* dimension for the RRTMG_LW. Section 5 describes the parallelization implementation of the G-RRTMG_LW. Section 6 evaluates the performance of the G-RRTMG_LW in terms of runtime efficiency and speedup, and discusses some of the problems arising in the experiment. The last section concludes the paper with a summary and proposal for future work.

2. Related Work

There were many successful attempts at using GPUs to accelerate physical parameterization schemes and climatic system models. This section describes the most salient work along this direction.

The WRF Goddard shortwave radiance was accelerated on GPUs using CUDA C [24]. Via double precision arithmetic and with data I/O, the shortwave radiance obtained a $116\times$ speedup on two NVIDIA GTX 590 s [25]. The WRF five-layer thermal diffusion scheme was accelerated using CUDA C, and a $311\times$ speedup was obtained on one Tesla K40 GPU [26]. The WRF Single Moment 6-class

microphysics scheme was also accelerated using CUDA C, and obtained a $216\times$ speedup on one Tesla K40 GPU [27].

The WRF long-wave RRTM code was ported to GPUs using CUDA Fortran [28]. The RRTM on Tesla C1060 GPUs attained a $10\times$ speedup [29]. The RRTM longwave radiation scheme (RRTM_LW) on the GTX480 obtained a $27.6\times$ speedup compared with the baseline wall-clock time [1]. The CUDA Fortran version of the RRTM_LW in the GRAPES_Meso model was developed. It adopted some optimization methods for enhancing the computational efficiency, and obtained a $14.3\times$ speedup [10].

The Fortran code of the WRF RRTMG_LW was rewritten in the C programming language, and then its GPU parallelization was implemented using CUDA C. With I/O transfer, the RRTMG_LW achieved a $123\times$ speedup on one Tesla K40 GPU [30]. The Fortran code of the RRTMG_SW was also rewritten in the C programming language. Furthermore, the RRTMG_SW achieved a $202\times$ speedup on one Tesla K40 GPU compared with its single-threaded Fortran counterpart running on Intel Xeon E5-2603 [31].

In a significantly different approach from the previous work, this study first proposes a new and detailed parallel algorithm in the *g-point* dimension for the CAS-ESM RRTMG_LW. Rewriting the original Fortran code of the RRTMG_LW would take considerable time, so CUDA Fortran rather than CUDA C was adopted in the parallelization implementation. The major concerns addressed by the proposed algorithm include the following: (a) runtime efficiency, and (b) common processes and technologies of GPU parallelization.

3. Model Description and GPU Overview

3.1. RRTMG_LW Model

As a critical process affecting our planet's climate, radiative transfer is the transport of energy by electromagnetic waves through a gas. Atmospheric and Environmental Research (AER) developed the RRTM and RRTMG. Their calculations exhibit an effective accuracy equivalent to that provided by the LBLRTM, but their computational cost is lower. In view of these advantages, the RRTM or RRTMG was adopted as the radiative transfer schemes of many climate models, such as the WRF and GRAPES [32].

The RRTM uses the correlated k-distribution method to calculate the broad-band radiative fluxes. In this method, the radiative spectrum is first divided into bands. Because of the rapid variation of absorption lines within the bands of gas molecules, the values of the absorption intensities within each band are further binned into a cumulative distribution function of the intensities. This distribution function is then discretized by using *g* intervals for integration within each band to obtain the band radiative fluxes, which are further integrated across the bands to obtain the total radiative flux to calculate atmospheric radiative heating or cooling. The *g* points are the discretized absorption intensities within each band.

The RRTM_LW is the RRTM for infrared radiation (longwave, LW); RRTM_SW is the RRTM for solar radiation (shortwave, SW). RRTM_LW has 16 bands and 256 *g* points. RRTM_SW has 16 bands and 224 *g* points. To speed up the calculations for climate and weather models, the spectral resolutions of RRTM_LW and RRTM_SW are further coarsened for applications in GCMs as RRTMG_LW and RRTMG_SW. RRTMG_LW has 16 bands and 140 *g* points. RRTMG_SW has 16 bands and 112 *g* points [33].

Below, we briefly describe the radiation flux and heating/cooling rate for calculating radiative transfer through a planetary atmosphere. The spectrally averaged outgoing radiance from an atmospheric layer is expressed using the following formula:

$$I_v(\mu) = \frac{1}{v_2 - v_1} \int_{v_1}^{v_2} dv \left\{ I_0(v) + \int_{T_v}^1 [B(v, \theta(T')) - I_0(v)] dT' \right\}. \quad (1)$$

In this expression, μ is the zenith direction cosine; v is the wavenumber; θ is temperature; v_1 and v_2 are the beginning and ending wavenumbers of the spectral interval, respectively; $B(v, \theta)$ is the

Planck function at v and θ ; I_0 is the radiance incoming to the layer; T_v is the transmittance for the layer optical path; T'_v is the transmittance at a point along the optical path in the layer.

Equation (1) can derive the following Equation (2) under the necessary assumptions. In Equation (2), g is the fraction of the absorption coefficient; $Beff(g, T_g)$ is an effective Planck function for the layer; ρ is the absorber density in the layer; P is layer pressure; Δz is the vertical thickness of the layer; φ is the angle of the optical path in the azimuthal direction; $k(g, P, \theta)$ is the absorption coefficient at P and θ .

$$I_g(\mu, \varphi) = \int_0^1 dg \left\{ Beff(g, T_g) + [I_0(g) - Beff(g, T_g)] \exp[-k(g, P, \theta) \frac{\rho \Delta z}{\cos \varphi}] \right\}. \quad (2)$$

The monochromatic net flux is

$$F_v = F_v^+ - F_v^-, \quad (3)$$

where $F_v^+ = 2\pi \int_0^1 I_v(\mu) \mu d\mu$ and $F_v^- = 2\pi \int_0^{-1} I_v(\mu) \mu d\mu$.

The total net flux is obtained by integrating over v

$$F_{net} = F_{net}^+ - F_{net}^-. \quad (4)$$

The radiative heating (or cooling) rate is expressed as

$$\frac{d\theta}{dt} = -\frac{1}{c_p \rho} \frac{dF_{net}}{dz} = \frac{g}{c_p} \frac{dF_{net}}{dP}, \quad (5)$$

where c_p is the specific heat at a constant pressure; P is pressure; g is the gravitational acceleration; ρ is the air density in a given layer [30].

3.2. RRTMG_LW Code Structure

Figure 1 indicates the profiling graph of the original Fortran RRTMG_LW. Here, the subroutine *rad_rrtmg_lw* is the driver of longwave radiation code. The subroutine *mcica_subcol_lw* is used to create Monte-Carlo Independent Column Approximation (McICA) stochastic arrays for cloud physical or optical properties. The subroutine *rrtmg_lw* is the driver subroutine for the RRTMG_LW. The subroutine *rrtmg_lw* (a) calls the subroutine *inatm* to read in the atmospheric profile from the GCM for use in the RRTMG_LW, and to define other input parameters; (b) calls the subroutine *cldprmc* to set cloud optical depth for the McICA based on the input cloud properties; (c) calls the subroutine *setcoef* to calculate information needed by the radiative transfer routine that is specific to this atmosphere, especially some of the coefficients and indices needed to compute the optical depths by interpolating data from stored reference atmospheres; (d) calls the subroutine *taumol* to calculate the gaseous optical depths and Planck fractions for each of the 16 spectral bands; (e) calls the subroutine *rtrnmc* (for both clear and cloudy profiles) to perform the radiative transfer calculation using the McICA to represent sub-grid scale cloud variability; and (f) passes the necessary fluxes and heating rates back to the GCM.

Algorithm 1 shows the computing procedure of *rrtmg_lw*. As depicted in Figure 1, *rrtmg_lw* took most computing time of *rad_rrtmg_lw*, so the study target was to use GPUs for accelerating the *inatm*, *cldprmc*, *setcoef*, *taumol*, and *rtrnmc* subroutines.

3.3. GPU and CUDA Fortran

There are an array of streaming multiprocessors (SMs) inside a GPU. Several streaming processors inside each SM share the control logic and instruction cache. CUDA, a general purpose parallel computing architecture, fosters a software environment to make full use of many cores of GPUs in a massively parallel fashion. Functions or subroutines are defined as “kernels” by CUDA, and then they are executed on the GPU. Before running on the GPU, these kernels need to be invoked by the CPU. In the three-level hierarchy of CUDA, each kernel has a grid that is at the highest level. Each grid

consists of thread blocks. Inside each thread block, data is efficiently shared by a group of threads through a fast shared memory [34]. CUDA supports many high-level languages, such as C/C++ and Fortran. NVIDIA and PGI jointly developed CUDA Fortran [28]. CUDA Fortran extends Fortran in memory management statements, declaration statements, CUDA runtime APIs, and kernel execution syntaxes [35].

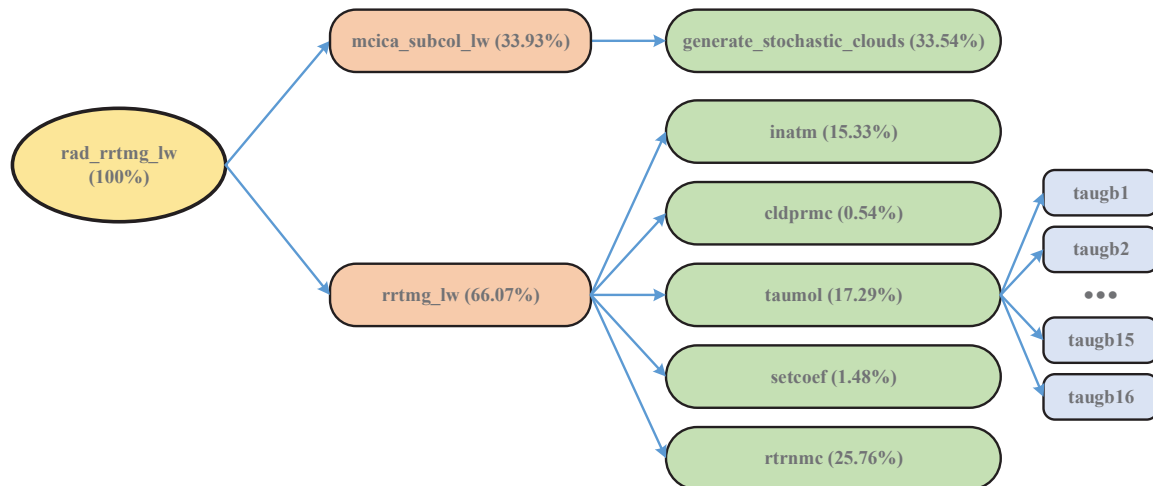


Figure 1. Profiling graph of the original RRTMG_LW code in the CAS-ESM.

Algorithm 1: Computing procedure of original *rrtmg_lw*.

```

subroutine rrtmg_lw(parameters)
  // ncol is the number of horizontal columns
  1. do iplon=1, ncol
  2.   call inatm(parameters)
  3.   call cldprmc(parameters)
  4.   call setcoef(parameters)
  5.   call taumol(parameters)
  6.   if aerosol is active then
     // combine gaseous and aerosol optical depths
  7.     taut(k, ig) = taug(k, ig) + taua(k, ngb(ig))
  8.   else
  9.     taut(k, ig) = taug(k, ig)
  10.  end if
  11.  call rtrnmc(parameters)
  12.  Transfer fluxes and heating rate to output arrays
  13. end do
end subroutine

```

4. GPU-Enabled Acceleration Algorithm

In this section, the 2D acceleration algorithm of the RRTMG_LW is introduced. Then, the parallel strategy of the RRTMG_LW in the *g-point* dimension is described. Finally, a CUDA-based acceleration algorithm is proposed.

4.1. 2D Acceleration Algorithm

The RRTMG_LW uses a collection of three-dimensional (3D) cells to describe the atmosphere. Its 1D acceleration algorithm with a domain decomposition in the horizontal direction assigns the workload of one “column,” shown in Figure 2, to each CUDA thread. Here, the *x*-axis

represents longitude, the y -axis represents latitude, and the z -axis represents the vertical direction. The RRTMG_LW in spatial structure has three dimensions, but the x and y dimensions in its CUDA code implementation are merged into one dimension to easily write the code. In the CAS-ESM, the IAP AGCM4.0 has a $1.4^\circ \times 1.4^\circ$ horizontal resolution and 51 levels in the vertical direction, so the RRTMG_LW has $n_x \times n_y = 256 \times 128$ horizontal grid points. Thus, the first dimension of 3D arrays in its code has 256×128 elements at most.

To make full use of the GPU performance, the 2D acceleration algorithm with a domain decomposition in the horizontal and vertical directions for the RRTMG_LW was proposed in our previous study [23]. Figure 3 illustrates the 2D domain decomposition for the RRTMG_LW accelerated on the GPU. The 2D acceleration algorithm is illustrated in Algorithm 2. Because of data dependency, the acceleration of *cldprmc* and *rtrnmc* in the vertical direction is unsuitable, while *inatm*, *setcoef*, and *taumol* are able to accelerate in the vertical direction. In the 1D acceleration, n is the number of threads in each thread block, while $m = \lceil (\text{real})ncol/n \rceil$ is the number of blocks used in each kernel grid. In the 2D acceleration, *tBlock* defines the number of threads used in each thread block of the x , y , and z dimensions by the derived type *dim3*. Furthermore, *grid* defines the number of blocks in the x , y , and z dimensions by *dim3*.

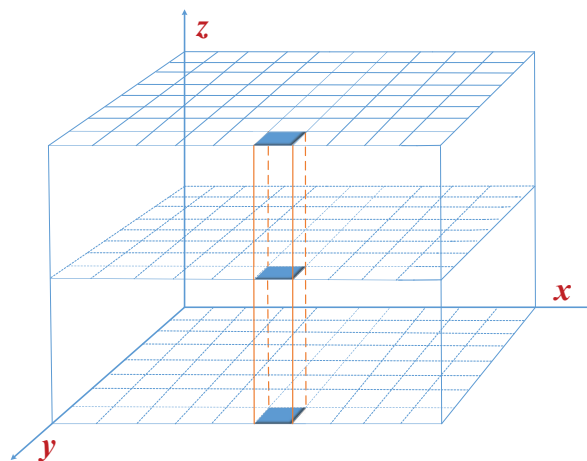


Figure 2. Spatial structure of RRTMG_LW.

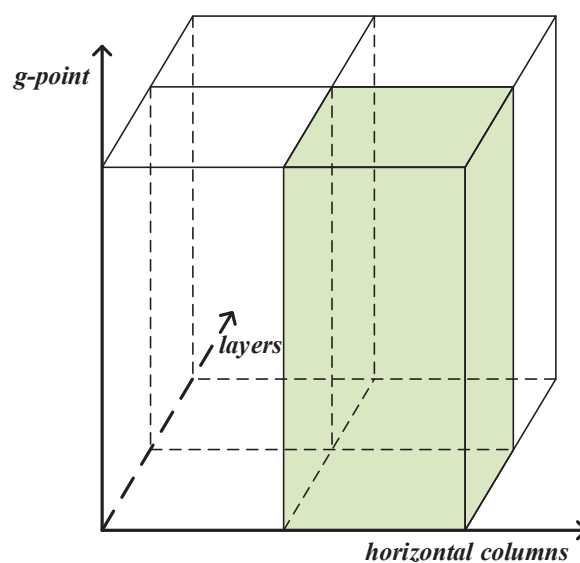


Figure 3. Schematic diagram of 2D decomposition for the RRTMG_LW in the GPU acceleration.

Algorithm 2: 2D acceleration algorithm.

```

subroutine rrtmg_lw_d2(parameters)
1. Copy input data to GPU device
   //Call inatm_d1 with 2D decomposition
2. call inatm_d1 <<< grid, tBlock >>>(parameters)
   //Call inatm_d2 with 2D decomposition
3. call inatm_d2 <<< grid, tBlock >>>(parameters)
   //Call inatm_d3 with 1D decomposition
4. call inatm_d3 <<< m, n >>>(parameters)
   //Call inatm_d4 with 2D decomposition
5. call inatm_d4 <<< grid, tBlock >>>(parameters)
   //Call cl DPRMC_d with 1D decomposition
6. call cl DPRMC_d <<< m, n >>>(parameters)
   //Call setcoef_d1 with 2D decomposition
7. call setcoef_d1 <<< grid, tBlock >>>(parameters)
   //Call setcoef_d2 with 1D decomposition
8. call setcoef_d2 <<< m, n >>>(parameters)
   //Call taumol_d with 2D decomposition
9. call taumol_d <<< grid, tBlock >>>(parameters)
   //Call rtrnmc_d with 1D decomposition
10. call rtrnmc_d <<< m, n >>>(parameters)
11. Copy result to host
   //Judge whether atmospheric horizontal profile data is completed
12. if it is not completed goto 1
end subroutine

```

4.2. Parallel Strategy

In the RRTMG_LW, the total number of g points, ng , is 140. Therefore, there are iterative computations for each g point in *inatm*, *taumol*, and *rtrnmc*. For example, the computation of 140 g points is executed by a do-loop in the GPU-based acceleration implementation of 1D *rtrnmc_d*, as illustrated in Algorithm 3. To achieve more fine-grained parallelism, 140 CUDA threads can be assigned to run the kernels *inatm_d*, *taumol_d*, and *rtrnmc_d*. Thus, the parallel strategy is further accelerating *inatm_d*, *taumol_d*, and *rtrnmc_d* in the g -point dimension. Moreover, the parallelization between the kernels should also be considered in addition to that within the kernels.

It is noteworthy that the first dimension of 3D arrays in the CUDA code represents the number of horizontal columns, the second dimension represents the number of model layers, and the third dimension represents the number of g points. If one GPU is applied, in theory, $nx \times ny \times nz \times ng = 256 \times 128 \times 51 \times 140$ CUDA threads will be required for each kernel in the new parallel method.

4.3. Acceleration Algorithm

Figure 4 illustrates the domain decomposition in the g -point dimension for the RRTMG_LW accelerated on the GPU. The acceleration algorithm in the g -point dimension for the RRTMG_LW, is illustrated in Algorithm 4. The algorithm is described as follows:

- (1) In the acceleration algorithm, *inatm* consists of five kernels (*inatm_d1*, *inatm_d2*, *inatm_d3*, *inatm_d4*, and *inatm_d5*). Due to data dependency, a piece of code in *inatm* can be parallel only in the horizontal or vertical direction, so the kernel *inatm_d4* uses a 1D decomposition. The kernels *inatm_d1*, *inatm_d2*, and *inatm_d5* use a 2D decomposition in the horizontal and vertical directions. The kernel *inatm_d3* uses a composite decomposition in the horizontal and vertical directions and g -point dimension. Due to the requirement of data synchronization, *inatm_d1* and *inatm_d2* cannot be merged into one kernel.

- (2) The kernel *cldprmc_d* still uses a 1D decomposition.
- (3) Similarly, the kernel *setcoef_d1* uses a 2D decomposition, and the kernel *setcoef_d2* uses a 1D decomposition.
- (4) The kernel *taumol_d* uses a composite decomposition in the horizontal and vertical directions and *g-point* dimension. In *taumol_d*, 16 subroutines with the device attribute are invoked.
- (5) Similarly, *rtrnmc* consists of 11 kernels (*rtrnmc_d1*–*rtrnmc_d11*). Here, *rtrnmc_d1*, *rtrnmc_d4*, *rtrnmc_d8*, *rtrnmc_d10*, and *rtrnmc_d11* use a 1D decomposition. Furthermore, *rtrnmc_d2* and *rtrnmc_d9* use a 2D decomposition in the horizontal and vertical directions. In addition, *rtrnmc_d5* and *rtrnmc_d6* use a 2D decomposition in the horizontal direction and *g-point* dimension. Finally, *rtrnmc_d3* and *rtrnmc_d7* use a composite decomposition in the horizontal and vertical directions and *g-point* dimension.

Algorithm 3: Implementation of 1D *rtrnmc_d*.

```

attributes(global) subroutine rtrnmc_d(parameters)
1. iplon=(blockIdx%x-1)×blockDim%x+threadIdx%x
2. if (iplon≥1 .and. iplon≤ncol) then
3.   Initialize variable arrays
4.   do lay = 1, nlayers
5.     do ig = 1, ngptlw
6.       do some corresponding work
7.     end do
8.   end do
   //Loop over frequency bands
   //istart is beginning band of calculation
   //iend is ending band of calculation
9.   do iband = istart, iend
   //Downward radiative transfer loop
10.  do lay = nlayers, 1, -1
11.    do some corresponding work
12.  end do
13. end do
14.end if
end subroutine

```

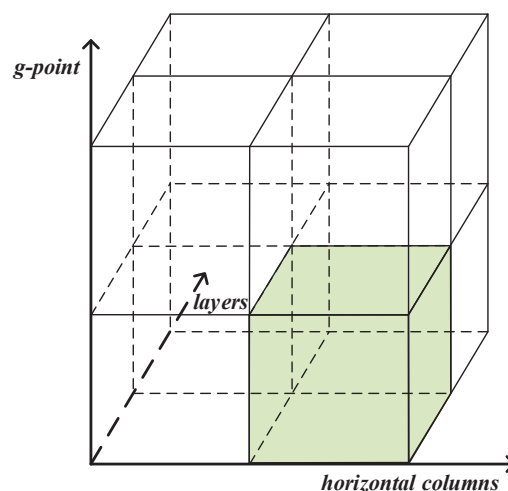


Figure 4. Schematic diagram of the decomposition in the *g-point* dimension for the RRTMG_LW in the GPU acceleration.

Algorithm 4: Acceleration in *g-point* dimension.

```

subroutine rrtmg_lw_d3(parameters)
1. Copy input data to GPU device
   //Call inatm_d1 with 2D decomposition
2. call inatm_d1<<< grid1,tBblock1 >>>(parameters)
   //Call inatm_d2 with 2D decomposition
3. call inatm_d2<<< grid1,tBblock1 >>>(parameters)
   //Call inatm_d3 with g-point acceleration
4. call inatm_d3<<< grid2,tBblock2 >>>(parameters)
   //Call inatm_d4 with 1D decomposition
5. call inatm_d4<<< m,n >>>(parameters)
   //Call inatm_d5 with 2D decomposition
6. call inatm_d5<<< grid1,tBblock1 >>>(parameters)
   //Call cldprmc_d with 1D decomposition
7. call cldprmc_d<<< m,n >>>(parameters)
   //Call setcoef_d1 with 2D decomposition
8. call setcoef_d1<<< grid,tBblock >>>(parameters)
   //Call setcoef_d2 with 1D decomposition
9. call setcoef_d2<<< m,n >>>(parameters)
   //Call taumol_d with g-point acceleration
10.call taumol_d<<< grid2,tBblock2 >>>(parameters)
   //Call rtrnmc_d1 with 1D decomposition
11.call rtrnmc_d1<<< m,n >>>(parameters)
   //Call rtrnmc_d2 with 2D decomposition
12.call rtrnmc_d2<<< grid1,tBblock1 >>>(parameters)
   //Call rtrnmc_d3 with g-point acceleration
13.call rtrnmc_d3<<< grid2,tBblock2 >>>(parameters)
   //Call rtrnmc_d4 with 1D decomposition
14.call rtrnmc_d4<<< m,n >>>(parameters)
   //Call rtrnmc_d5 with 2D decomposition in horizontal and g-point dimensions
15.call rtrnmc_d5<<< grid3,tBblock3 >>>(parameters)
   //Call rtrnmc_d6 with 2D decomposition in horizontal and g-point dimensions
16.call rtrnmc_d6<<< grid3,tBblock3 >>>(parameters)
   //Call rtrnmc_d7 with g-point acceleration
17.call rtrnmc_d7<<< grid2,tBblock2 >>>(parameters)
   //Call rtrnmc_d8 with 1D decomposition
18.call rtrnmc_d8<<< m,n >>>(parameters)
   //Call rtrnmc_d9 with 2D decomposition
19.call rtrnmc_d9<<< grid1,tBblock1 >>>(parameters)
   //Call rtrnmc_d10 with 1D decomposition
20.call rtrnmc_d10<<< m,n >>>(parameters)
   //Call rtrnmc_d11 with 1D decomposition
21.call rtrnmc_d11<<< m,n >>>(parameters)
22.Copy result to host
   //Judge whether atmospheric horizontal profile data is completed
23.if it is not completed goto 1
end subroutine

```

5. Algorithm Implementation

In this section, the acceleration algorithm implementation is described. The implementations of 1D *cldprmc_d* and 2D *setcoef_d* were described in our previous paper, so this section only considers the implementations of *inatm_d*, *taumol_d*, and *rtrnmc_d* with an acceleration in the *g-point* dimension.

5.1. *Inatm_d*

The implementations of CUDA-based 2D *inatm_d1*, *inatm_d2*, and *inatm_d5* are similar to the procedure in Algorithm 5. Here, *threadIdx%x* identifies a unique thread inside a thread block in the *x* dimension, *blockIdx%x* identifies a unique thread block inside a kernel grid in the *x* dimension, and *blockDim%x* identifies the number of threads in a thread block in the *x* dimension. In the same way, *threadIdx%y*, *blockIdx%y*, and *blockDim%y* identify the corresponding configuration in the *y* dimension. Please note that the “%” symbol in Fortran is the access to fields of a structure and not the modulus operator. In addition, *iplon*, the coordinate of the horizontal grid points, represents the ID of a global thread in the *x* dimension, which can be expressed as $iplon=(blockIdx%x-1)\times blockDim%x+threadIdx%x$. The coordinate of the vertical direction, *lay*, also represents the ID of a global thread in the *y* dimension, which can be expressed as $lay=(blockIdx%y-1)\times blockDim%y+threadIdx%y$. Thus, a grid point in the horizontal and vertical directions can be identified by the *iplon* and *lay* variables as 2D linear data.

The implementation of *inatm_d3* is illustrated in Algorithm 6. Here, *threadIdx%z* identifies a unique thread inside a thread block in the *z* dimension, *blockIdx%z* identifies a unique thread block inside a kernel grid in the *z* dimension, and *blockDim%z* identifies the number of threads in a thread block in the *z* dimension. The coordinate of the *g* points, *ig*, also represents the ID of a global thread in the *x* dimension, which can be expressed as $ig=(blockIdx%x-1)\times blockDim%x+threadIdx%x$. Furthermore, *inatm_d3* is used to assign a value to the four 3D arrays.

The implementation of 1D *inatm_d4* is illustrated in Algorithm 7.

Algorithm 5: Implementation of 2D *inatm_d*.

```
attributes(global) subroutine inatm_d(parameters)
1. iplon=(blockIdx%x-1)×blockDim%x+threadIdx%x
2. lay=(blockIdx%y-1)×blockDim%y+threadIdx%y
   //nlayers=nlay + 1, nlay is number of model layers
3. if ((iplon≥1 .and. iplon≤ncol) .and. (lay≥1 .and. lay≤nlayers-1 or nlayers or nlayers+1)) then
4.   Initialize variable arrays or do some computational work for them
5. end if
end subroutine
```

Algorithm 6: Implementation of *inatm_d3*.

```
attributes(global) subroutine inatm_d3(parameters)
1. ig=(blockIdx%x-1)×blockDim%x+threadIdx%x
2. iplon=(blockIdx%y-1)×blockDim%y+threadIdx%y
3. lay=(blockIdx%z-1)×blockDim%z+threadIdx%z
   //ngptlw is total number of reduced g-intervals
4. if ((iplon≥1 .and. iplon≤ncol) .and. (lay≥1 .and. lay≤nlayers-1) .and. (ig≥1 .and. ig≤ngptlw)) then
5.   cldfmc(ig, iplon, lay) = cldfmc_d(ig, iplon, nlayers-lay)
6.   taucmc(ig, iplon, lay) = taucmc_d(ig, iplon, nlayers-lay)
7.   ciwpmc(ig, iplon, lay) = ciwpmc_d(ig, iplon, nlayers-lay)
8.   clwpmc(ig, iplon, lay) = clwpmc_d(ig, iplon, nlayers-lay)
9. end if
end subroutine
```

5.2. *taumol_d*

The implementation of *taumol_d* is illustrated in Algorithm 8. Here, the implementations of *taugb1_d* and *taugb2_d* with the device attribute are also described. The implementations of the other 14 subroutines (*taugb3_d*–*taugb16_d*) are similar to those of *taugb1_d* and *taugb2_d*.

Algorithm 7: Implementation of 1D *inatm_d4*.

```

attributes(global) subroutine inatm_d4(parameters)
1. iplon=(blockIdx%x-1)×blockDim%x+threadIdx%x
2. if (iplon≥1 .and. iplon≤ncol) then
3.   Initialize variable arrays
4.   do lay = 1, nlayers-1 or nlayers
5.     do some corresponding computational work
6.   end do
7. end if
end subroutine

```

Algorithm 8: Implementation of *taumol_d*.

```

attributes(global) subroutine taumol_d(parameters)
1. iplon=(blockIdx%x-1)×blockDim%x+threadIdx%x
2. lay=(blockIdx%y-1)×blockDim%y+threadIdx%y
3. ig=(blockIdx%z-1)×blockDim%z+threadIdx%z
4. if ((iplon≥1 .and. iplon≤ncol) .and. (lay≥1 .and. lay≤nlayers) .and. (ig≥1 .and. ig≤ngptlw)) then
5.   call taugb1_d(parameters)
6.   call taugb2_d(parameters)
7.   call taugb3_d(parameters)
8.   call taugb4_d(parameters)
9.   call taugb5_d(parameters)
10.  call taugb6_d(parameters)
11.  call taugb7_d(parameters)
12.  call taugb8_d(parameters)
13.  call taugb9_d(parameters)
14.  call taugb10_d(parameters)
15.  call taugb11_d(parameters)
16.  call taugb12_d(parameters)
17.  call taugb13_d(parameters)
18.  call taugb14_d(parameters)
19.  call taugb15_d(parameters)
20.  call taugb16_d(parameters)
21.end if
end subroutine
attributes(device) subroutine taugb1_d(parameters)
  //Lower atmosphere loop
  //laytrop is tropopause layer index, ngs1 is an integer parameter used for 140 g-point model
1. if ((iplon≥1 .and. iplon≤ncol) .and. (lay≥1 .and. lay≤laytrop(iplon)) .and. (ig≥1 .and. ig≤ngs1))
  then
2.   do some computational work
3. end if
  //Upper atmosphere loop
4. if ((iplon≥1 .and. iplon≤ncol) .and. (lay≥laytrop(iplon)+1 .and. lay≤nlayers) .and. (ig≥1 .and.
  ig≤ngs1)) then
5.   do some computational work
6. end if
end subroutine
attributes(device) subroutine taugb2_d(parameters)
  //ngs2 is an integer parameter used for 140 g-point model
1. if ((iplon≥1 .and. iplon≤ncol) .and. (lay≥1 .and. lay≤laytrop(iplon)) .and. (ig≥ngs1+1 .and.
  ig≤ngs2)) then
2.   do some computational work
3. end if
4. if ((iplon≥1 .and. iplon≤ncol) .and. (lay≥laytrop(iplon)+1 .and. lay≤nlayers) .and. (ig≥ngs1+1 .and.
  ig≤ngs2)) then
5.   do some computational work
6. end if
end subroutine

```

5.3. *rtrnmc_d*

The implementations of 1D *rtrnmc_d1*, *rtrnmc_d4*, *rtrnmc_d8*, *rtrnmc_d10*, and *rtrnmc_d11* are similar to the procedure in Algorithm 3. The 2D *rtrnmc_d2* and *rtrnmc_d9* are used to assign a value to some arrays; their implementations are not described further here. The implementations of *rtrnmc_d3*, 2D *rtrnmc_d5*, and 2D *rtrnmc_d6* are illustrated in Algorithm 9. The implementation of *rtrnmc_d7* is similar to that of *rtrnmc_d3*.

Algorithm 9: Implementation of *rtrnmc_d*.

```

attributes(global) subroutine rtrnmc_d3(parameters)
1. iplon=(blockIdx%x-1)×blockDim%x+threadIdx%x
2. igc=(blockIdx%y-1)×blockDim%y+threadIdx%y
3. lay=(blockIdx%z-1)×blockDim%z+threadIdx%z
4. if ((iplon≥1 .and. iplon≤ncol) .and. (lay≥0 .and. lay≤nlayers) .and. (igc≥1 .and. igc≤ngptlw)) then
5.   urad(iplon, igc, lay) = 0.0
6.   drad(iplon, igc, lay) = 0.0
7.   clrurad(iplon, igc, lay) = 0.0
8.   clrdrad(iplon, igc, lay) = 0.0
9. end if
end subroutine

attributes(global) subroutine rtrnmc_d5(parameters)
1. iplon=(blockIdx%x-1)×blockDim%x+threadIdx%x
2. igc=(blockIdx%y-1)×blockDim%y+threadIdx%y
3. if ((iplon≥1 .and. iplon≤ncol) .and. (igc≥1 .and. igc≤10)) then
   //Loop over frequency bands
4.   iband=1
   //Downward radiative transfer loop
5.   do lay = nlayers, 1, -1
6.     do some corresponding work
7.   end do
8. end if
9. if ((iplon≥1 .and. iplon≤ncol) .and. (igc≥11 .and. igc≤22)) then
10.  iband=2
11.  do lay = nlayers, 1, -1
12.    do some corresponding work
13.  end do
14.end if
15. When iband=3 ~ 9, the algorithms are similar to that in the case of iband=1 or 2.
end subroutine

attributes(global) subroutine rtrnmc_d6(parameters)
1. iplon=(blockIdx%x-1)×blockDim%x+threadIdx%x
2. igc=(blockIdx%y-1)×blockDim%y+threadIdx%y
3. if ((iplon≥1 .and. iplon≤ncol) .and. (igc≥109 .and. igc≤114)) then
4.   iband=10
5.   do lay = nlayers, 1, -1
6.     do some corresponding work
7.   end do
8. end if
9. When iband=11 ~ 16, the algorithms are similar to that in the case of iband=10.
end subroutine

```

6. Result and Discussion

Experimental studies were conducted to evaluate and compare the performance of the proposed algorithm with the solutions above. The results are described below.

6.1. Experimental Setup

To fully investigate the proposed algorithm, this paper conducted an ideal global climate simulation for one model day. The time step of the RRTMG_LW was 1 h.

The experiment platforms include two GPU clusters (K20 and K40 clusters). The K20 cluster at the Computer Network Information Center of CAS has 30 GPU nodes. Each GPU node has two Intel Xeon E5-2680 v2 processors and two NVIDIA Tesla K20 GPUs. Twenty CPU cores in each GPU node share 64 GB DDR3 system memory through QuickPath Interconnect. The basic compiler is the PGI Fortran compiler Version 14.10 that supports CUDA Fortran. The K40 cluster at China University of Geosciences (Beijing) has four GPU nodes, each with two NVIDIA Tesla K40 GPUs. Table 1 lists their detailed configurations. The serial RRTMG_LW was executed on an Intel Xeon E5-2680 v2 processor of K20 cluster. The G-RRTMG_LW was executed on one K20 or K40 GPU.

Table 1. Configurations of GPU clusters.

Specification of CPU	K20 Cluster	K40 Cluster
CPU	E5-2680 v2@2.8GHz	E5-2695 v2@2.4GHz
Operating System	CentOS 6.4	Red Hat Enterprise Linux Server 7.1
Specification of GPU	K20 Cluster	K40 Cluster
GPU	Tesla K20	Tesla K40
CUDA Cores	2496	2880
Standard Memory	5 GB	12 GB
Memory Bandwidth	208 GB/s	288 GB/s
CUDA Version	6.5	9.0

6.2. Influence of Block Size

The serial runtime of the subroutine *rrtmg_lw* on one core of an Intel Xeon E5-2680 v2 processor, which accounts for 66.07% of the total computing time of the subroutine *rad_rrtmg_lw*, is 647.12 s in this simulation, as shown in Table 2. Here, the computing time of the RRTMG_LW on the CPU or GPU, T_{rrtmg_lw} , is calculated with the following formula:

$$T_{rrtmg_lw} = T_{inatm} + T_{cldprmc} + T_{setcoef} + T_{taumol} + T_{rtrnmc},$$

where T_{inatm} is the computing time of the subroutine *inatm* or kernel *inatm_d*; moreover, $T_{cldprmc}$, $T_{setcoef}$, T_{taumol} , and T_{rtrnmc} are the corresponding computing times of the other kernels. For exploring whether/how the number of threads in a thread block may affect the computation performance, the execution time of the G-RRTMG_LW with a tuned number of threads was measured over one GPU. Taking the case of no I/O transfer as an example, Figure 5 portrays the runtime of the G-RRTMG_LW on one K20 GPU. Indeed, the number of threads per block, or block size, affected the computation performance to some extent. The G-RRTMG_LW achieved optimal performance when the block size was 128. The G-RRTMG_LW on one K40 GPU resulted in a similar rule, as shown in Figure 6. Some conclusions and analysis are drawn as below.

- (1) Generally, increasing the block size can hide some memory access latency to some extent and improve the computational performance of parallel algorithms. Therefore, kernels with simple computation usually show optimal performance when the block size is 128 or 256. Thus, with a large amount of calculation, the kernel *taumol* with an acceleration in the *g-point* dimension achieved optimal performance on one K20 GPU when the block size was 128.
- (2) The runtime and speedup of the RRTMG_LW 2D acceleration algorithm on the K20 and K40 GPUs are shown in Table 2. From Table 2, Figures 5 and 6, the kernel *taumol* with an acceleration in the *g-point* dimension costs more computational time than its 2D version did. This is because there is much redundant computing in the current *taumol*. For example, each thread in 2D *taumol*

ran the code shown in Table 3, but each thread in the current *taumol* still had to run the code although there were more threads. Therefore, the current *taumol*, with more threads than its 2D counterpart, did not take full advantage of having plenty of threads.

- (3) The *rtrnmc* with an acceleration in the *g-point* dimension on one K20 GPU and one K40 GPU both achieved optimal performance when the block size was 512. During its current acceleration, more threads were assigned, so that each thread had fewer calculations and required fewer hardware resources. When the block size was 512, the assigned hardware resources of each thread were in a state of equilibrium, so in this case, the current *rtrnmc* showed better performance.

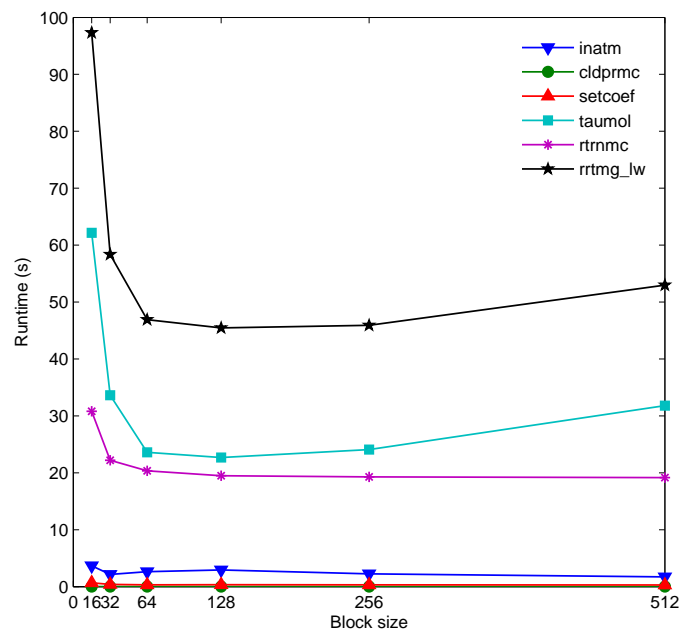


Figure 5. Runtime of the G-RRTMG_LW on one K20 GPU.

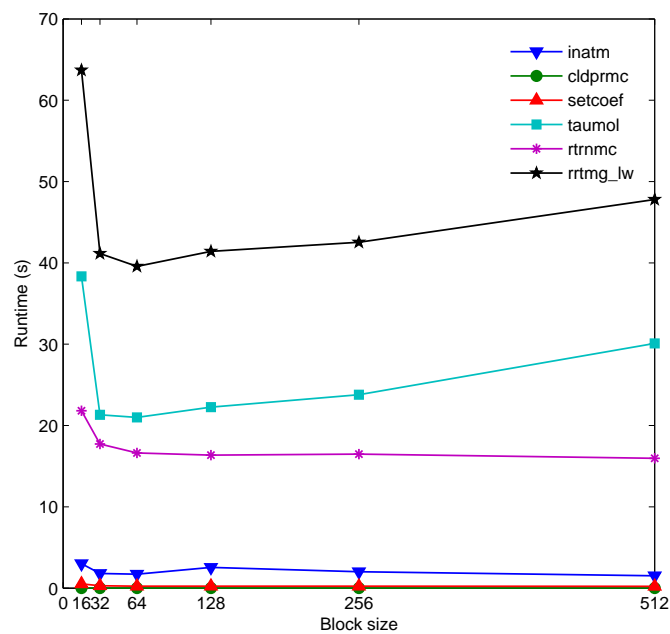


Figure 6. Runtime of the G-RRTMG_LW on one K40 GPU.

Table 2. Runtime and speedup of the RRTMG_LW 2D acceleration algorithm on one GPU. Here, the block size = 128 and $ncol = 2048$; *inatm*, *cldprmc*, and *rtrnmc* are with a 1D decomposition; *setcoef* and *taumol* are with a 2D decomposition.

Subroutines	CPU Time (S)	K20 Time (S)	Speedup
<i>inatm</i>	150.48	18.2060	8.27
<i>cldprmc</i>	5.27	0.0020	2635.00
<i>setcoef</i>	14.52	0.3360	43.21
<i>taumol</i>	169.68	4.1852	40.54
<i>rtrnmc</i>	252.80	21.5148	11.75
<i>rtrmg_lw</i>	647.12	44.2440	14.63

Subroutines	CPU Time (S)	K40 Time (S)	Speedup
<i>inatm</i>	150.48	14.9068	10.09
<i>cldprmc</i>	5.27	0.0020	2635
<i>setcoef</i>	14.52	0.2480	58.55
<i>taumol</i>	169.68	3.1524	53.83
<i>rtrnmc</i>	252.80	16.6292	15.20
<i>rtrmg_lw</i>	647.12	34.9384	18.52

Table 3. A piece of code in *taumol*.

```

if((iplon ≥ 1 .and. iplon ≤ ncol) .and. (lay ≥ laytrop(iplon)+1 .and. lay ≤ nlayers) .and. (ig ≥ 1 .and. ig ≤ ngl)) then
  ind0_1 = ((jp(iplon, lay) - 13) * 5 + (jt(iplon, lay) - 1)) * nspb(1) + 1
  ind1_1 = ((jp(iplon, lay) - 12) * 5 + (jt1(iplon, lay) - 1)) * nspb(1) + 1
  indf_1 = indfor(iplon, lay)
  indm_1 = indminor(iplon, lay)
  pp_1 = pavel(iplon, lay)
  corradj_1 = 1._r8 - 0.15_r8 * (pp_1 / 95.6_r8)
  scalen2_1 = colbrd(iplon, lay) * scaleminorn2(iplon, lay)
end if

```

6.3. Evaluations on Different GPUs

The runtime and speedup of the G-RRTMG_LW on the K20 and K40 GPUs are shown in Table 4. Due to the poor performance of the *taumol* with an acceleration in the *g-point* dimension, its 2D version was still used here. The speedups of the *inatm* and *rtrnmc* with an acceleration in the *g-point* dimension on one K20 GPU were $87.37\times$ and $13.20\times$, respectively. Using one K20 GPU in the case without I/O transfer, the G-RRTMG_LW achieved a speedup of $25.47\times$ as compared to its counterpart running on one CPU core of an Intel Xeon E5-2680 v2. Whereas, using one K40 GPU in the case without I/O transfer, the G-RRTMG_LW achieved a speedup of $30.98\times$. Some conclusions and analysis are drawn below.

- (1) The K40 GPU had a higher core clock and memory clock, more cores, and stronger floating-point computation power than the K20 GPU did. Thus, the G-RRTMG_LW on the K40 GPU showed better performance.
- (2) According to the testing, it was found that the transposition of four 3D arrays in the 1D or 2D *inatm* required most of the computational time. Because of discontinuous access for these arrays using a do-loop form, the transposition cost too much time. However, the *inatm* with an acceleration in the *g-point* dimension has no do-loops, as shown in Algorithm 6, so it can show an excellent performance improvement.
- (3) There are 11 kernels in the current *rtrnmc*, but only two of them have a composite decomposition in the horizontal, vertical and *g-point* dimensions. The two kernels only cost about 17% computational time, so the current *rtrnmc* did not achieve a noticeable performance improvement compared with its 1D version.

Table 4. Runtime and speedup of the G-RRTMG_LW on one GPU. Here, the block size = 512 and $ncol = 2048$; *inatm* and *rtrnmc* are with an acceleration in the *g-point* dimension; *cldprmc* is with a 1D decomposition; *setcoef* and *taumol* are with a 2D decomposition.

Subroutines	CPU Time (S)	K20 Time (S)	Speedup
<i>inatm</i>	150.48	1.7224	87.37
<i>cldprmc</i>	5.27	0.0020	2635.00
<i>setcoef</i>	14.52	0.3360	43.21
<i>taumol</i>	169.68	4.1852	40.54
<i>rtrnmc</i>	252.80	19.1580	13.20
<i>rtrmg_lw</i>	647.12	25.4036	25.47
Subroutines	CPU Time (S)	K40 Time (S)	Speedup
<i>inatm</i>	150.48	1.5140	99.39
<i>cldprmc</i>	5.27	0.0020	2635
<i>setcoef</i>	14.52	0.2480	58.55
<i>taumol</i>	169.68	3.1524	53.83
<i>rtrnmc</i>	252.80	15.9712	15.83
<i>rtrmg_lw</i>	647.12	20.8876	30.98

When compared to its counterpart running on 10 CPU cores of an Intel Xeon E5-2680 v2, the speedup of the G-RRTMG_LW on the K20 GPU is shown in Table 5. In this case, using one K20 GPU in the case without I/O transfer, the G-RRTMG_LW achieved a speedup of $2.35\times$. This shows that running the RRTMG_LW on one K20 GPU still has a better computing performance than on 10 cores of a CPU.

Table 5. Runtime and speedup of the G-RRTMG_LW on one GPU when compared to its counterpart running on 10 CPU cores of an Intel Xeon E5-2680 v2. Here, the block size = 512 and $ncol = 2048$; *inatm* and *rtrnmc* are with an acceleration in the *g-point* dimension; *cldprmc* is with a 1D decomposition; *setcoef* and *taumol* are with a 2D decomposition.

Subroutines	CPU Time (S)	K20 Time (S)	Speedup
<i>rtrmg_lw</i>	59.79	25.4036	2.35

6.4. I/O Transfer

I/O transfer between the CPU and GPU is inevitable. The runtime and speedup of the G-RRTMG_LW with I/O transfer on the K20 and K40 GPUs are shown in Table 6. The I/O transfer time on the K20 cluster was 61.39 s, while it was 59.3 s on the K40 cluster. Using one K40 GPU in the case with I/O transfer, the G-RRTMG_LW achieved a speedup of $8.07\times$. Some conclusions and analysis are drawn below.

- (1) In the simulation with one model day, the RRTMG_LW required integral calculations 24 times. During each integration for all 128×256 grid points, the subroutine *rtrmg_lw* had to be invoked 16 ($128 \times 256 / ncol$) times when $ncol$ is 2048. Due to the memory limitation of the GPU, the maximum value of $ncol$ on the K40 GPU was 2048. This means that the G-RRTMG_LW was invoked repeatedly $16 \times 24 = 384$ times in the experiment. For each invocation, the input and output of the 3D arrays must be updated between the CPU and GPU, so the I/O transfer incurs a high communication cost.
- (2) The computational time of the G-RRTMG_LW in the case without I/O transfer was fairly short, so the I/O transfer cost exhibited a huge bottleneck for the maximum level of performance improvement of the G-RRTMG_LW. In the future, compressing data and improving the network bandwidth will be beneficial for reducing this I/O transfer cost.

Table 6. Runtime and speedup of the G-RRTMG_LW with I/O transfer on one GPU, where the block size = 512 and *ncol* = 2048.

Subroutines	CPU Time (S)	K20 Time (S)	Speedup
<i>rrtmg_lw</i>	647.12	86.79	7.46
Subroutines	CPU Time (S)	K40 Time (S)	Speedup
<i>rrtmg_lw</i>	647.12	80.19	8.07

6.5. Error Analysis

During accelerating the computational performance of a climate system model, it is of vital importance to ensure that the model on one GPU can generate the same results within a small tolerance threshold. In this simulating experiment, Figure 7 illustrates the impact on the longwave flux at the top of the atmosphere in a clear sky. The outgoing longwave flux by running entirely the CAS-ESM RRTMG_LW on the CPU is shown in Figure 7a. The longwave flux differences between the simulations running the CAS-ESM RRTMG_LW only on the CPU and running the CAS-ESM RRTMG_LW (G-RRTMG_LW) on the K20 GPU is shown in Figure 7b. The results show that there are minor and negligible differences. Besides the impact of running the G-RRTMG_LW on the GPU, the impact of the slight physics change by running the G-RRTMG_LW code on the GPU also results in these differences.

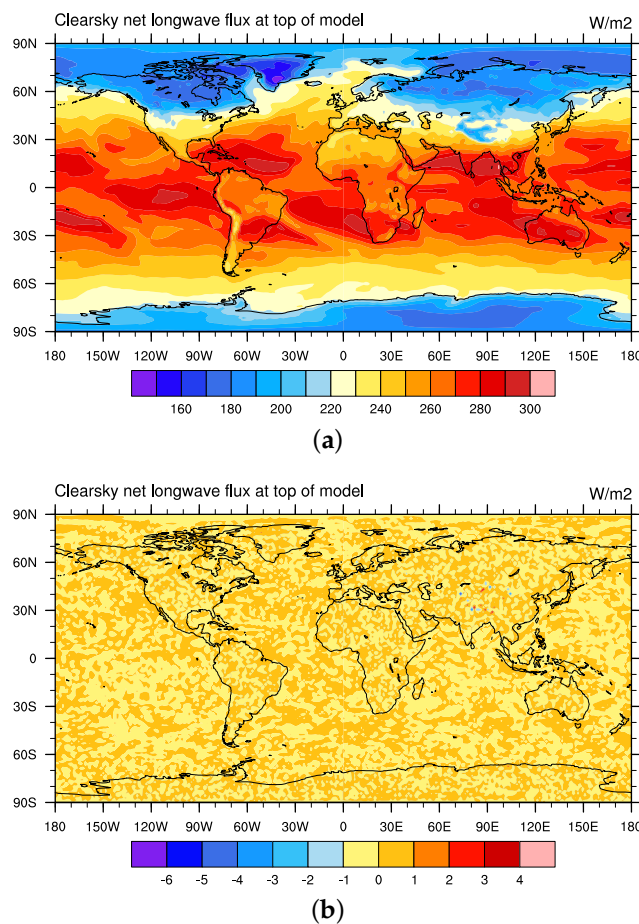


Figure 7. Impact on the longwave flux at the top of the atmosphere in a clear sky. (a) Longwave flux simulated by the CAS-ESM RRTMG_LW on the CPU; (b) Longwave flux differences between the simulations running the CAS-ESM RRTMG_LW only on the CPU and running the CAS-ESM RRTMG_LW on the GPU.

7. Conclusions and Future Work

It is exceedingly challenging to use GPUs to accelerate radiation physics. In this work, a GPU-based acceleration algorithm of the RRTMG_LW in the *g-point* dimension was proposed. Then, the acceleration algorithm was implemented using CUDA Fortran. Finally, G-RRRTMG_LW, the GPU version of the RRTMG_LW, was developed. The results indicated that the algorithm was effective. During the climate simulation for one model day, the G-RRRTMG_LW on one K40 GPU achieved a speedup of $30.98\times$ as compared with a single Intel Xeon E5-2680 CPU-core counterpart. Its runtime decreased from 647.12 s to 20.8876 s. Running the G-RRRTMG_LW on one GPU presented a better computing performance than on a CPU with multiple cores. Furthermore, the current acceleration algorithm of the RRTMG_LW displayed better calculation performance than its 2D algorithm did.

The future work will include three aspects. First, the proposed algorithm will be further optimized. For example, read-only arrays in CUDA code will be considered for inclusion in the CUDA texture memory, rather than in the global memory. Second, the I/O transfer in the current G-RRRTMG_LW still costs a great deal of time, so the methods of reducing the I/O transfer between the CPU and GPU will be studied. Third, the G-RRRTMG_LW currently only runs on one GPU. The CAS-ESM often runs on several CPUs and nodes, so the acceleration algorithm on multiple GPUs will be studied. An MPI+CUDA hybrid paradigm will be adopted to run the G-RRRTMG_LW on multiple GPUs.

Author Contributions: Conceptualization, Y.W.; Methodology, Y.W., Y.Z. and J.J.; Supervision, Y.W.; Validation, Y.W. and Y.Z.; Writing—original draft, Y.W.; Writing—review & editing, Y.W. and H.Z. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported in part by the National Key Research and Development Program of China under Grant 2016YFB0200800, in part by the National Natural Science Foundation of China under Grant 61602477, in part by the China Postdoctoral Science Foundation under Grant 2016M601158, in part by the Fundamental Research Funds for the Central Universities under Grant 2652017113, and in part by the Open Research Project of the Hubei Key Laboratory of Intelligent Geo-Information Processing under Grant KLIIGIP-2017A04.

Acknowledgments: The authors would like to acknowledge the contributions of Minghua Zhang for insightful suggestions on algorithm design.

Conflicts of Interest: The authors declare no conflict of interest. The funders had no role in the design of the study; in the collection, analyses, or interpretation of data; in the writing of the manuscript, or in the decision to publish the results.

References

1. Lu, F.; Cao, X.; Song, J.; Zhu, X. GPU computing for longwave radiation physics: A RRTM_LW scheme case study. In Proceedings of the IEEE 9th International Symposium on Parallel and Distributed Processing with Applications Workshops (ISPAW), Busan, Korea, 26–28 May 2011; pp. 71–76.
2. Clough, S.A.; Iacono, M.J. Line-by-line calculation of atmospheric fluxes and cooling rates II: Application to carbon dioxide, ozone, methane, nitrous oxide and the halocarbons. *J. Geophys. Res. Atmos.* **1995**, *100*, 16519–16535. [[CrossRef](#)]
3. Mlawer, E.J.; Taubman, S.J.; Brown, P.D.; Iacono, M.J.; Clough, S.A. Radiative transfer for inhomogeneous atmospheres: RRTM, a validated correlated-k model for the longwave. *J. Geophys. Res. Atmos.* **1997**, *102*, 16663–16682. [[CrossRef](#)]
4. Clough, S.A.; Shephard, M.W.; Mlawer, E.J.; Delamere, J.S.; Iacono, M.J.; Cady-Pereira, K.; Boukabara, S.; Brown, P.D. Atmospheric radiative transfer modeling: A summary of the AER codes. *J. Quant. Spectrosc. Radiat. Transf.* **2005**, *91*, 233–244. [[CrossRef](#)]
5. Iacono, M.J.; Delamere, J.S.; Mlawer, E.J.; Shephard, M.W.; Clough, S.A.; Collins, W.D. Radiative forcing by long-lived greenhouse gases: Calculations with the AER radiative transfer models. *J. Geophys. Res. Atmos.* **2008**, *113*. [[CrossRef](#)]
6. Wang, Y.; Jiang, J.; Ye, H.; He, J. A distributed load balancing algorithm for climate big data processing over a multi-core CPU cluster. *Concurr. Comput. Pract. Exp.* **2016**, *28*, 4144–4160. [[CrossRef](#)]
7. Wang, Y.; Hao, H.; Zhang, J.; Jiang, J.; He, J.; Ma, Y. Performance optimization and evaluation for parallel processing of big data in earth system models. *Clust. Comput.* **2017**, *22*, 2371–2381. [[CrossRef](#)]

8. Zhang, H.; Zhang, M.; Zeng, Q. Sensitivity of simulated climate to two atmospheric models: Interpretation of differences between dry models and moist models. *Mon. Weather. Rev.* **2013**, *141*, 1558–1576. [[CrossRef](#)]
9. Wang, Y.; Jiang, J.; Zhang, H.; Dong, X.; Wang, L.; Ranjan, R.; Zomaya, A.Y. A scalable parallel algorithm for atmospheric general circulation models on a multi-core cluster. *Future Gener. Comput. Syst.* **2017**, *72*, 1–10. [[CrossRef](#)]
10. Zheng, F.; Xu, X.; Xiang, D.; Wang, Z.; Xu, M.; He, S. GPU-based parallel researches on RRTM module of GRAPES numerical prediction system. *J. Comput.* **2013**, *8*, 550–558. [[CrossRef](#)]
11. Iacono, M.J. *Enhancing Cloud Radiative Processes and Radiation Efficiency in the Advanced Research Weather Research and Forecasting (WRF) Model*; Atmospheric and Environmental Research: Lexington, MA, USA, 2015.
12. Morcrette, J.J.; Mozdzyński, G.; Leutbecher, M. A reduced radiation grid for the ECMWF Integrated Forecasting System. *Mon. Weather. Rev.* **2008**, *136*, 4760–4772. [[CrossRef](#)]
13. Xue, W.; Yang, C.; Fu, H.; Wang, X.; Xu, Y.; Liao, J.; Gan, L.; Lu, Y.; Ranjan, R.; Wang, L. Ultra-scalable CPU-MIC acceleration of mesoscale atmospheric modeling on tianhe-2. *IEEE Trans. Comput.* **2015**, *64*, 2382–2393. [[CrossRef](#)]
14. Wang, Y.; Jiang, J.; Zhang, J.; He, J.; Zhang, H.; Chi, X.; Yue, T. An efficient parallel algorithm for the coupling of global climate models and regional climate models on a large-scale multi-core cluster. *J. Supercomput.* **2018**, *74*, 3999–4018. [[CrossRef](#)]
15. Cracknell, A.P.; Varotsos, C.A. New aspects of global climate-dynamics research and remote sensing. *Int. J. Remote. Sens.* **2011**, *32*, 579–600. [[CrossRef](#)]
16. Deng, Z.; Chen, D.; Hu, Y.; Wu, X.; Peng, W.; Li, X. Massively parallel non-stationary EEG data processing on GPGPU platforms with Morlet continuous wavelet transform. *J. Internet Serv. Appl.* **2012**, *3*, 347–357. [[CrossRef](#)]
17. Chen, D.; Wang, L.; Tian, M.; Tian, J.; Wang, S.; Bian, C.; Li, X. Massively parallel modelling & simulation of large crowd with GPGPU. *J. Supercomput.* **2013**, *63*, 675–690.
18. Chen, D.; Li, X.; Wang, L.; Khan, S.U.; Wang, J.; Zeng, K.; Cai, C. Fast and scalable multi-way analysis of massive neural data. *IEEE Trans. Comput.* **2015**, *64*, 707–719. [[CrossRef](#)]
19. Candell, F.; Petit, S.; Sahuquillo, J.; Duato, J. Accurately modeling the on-chip and off-chip GPU memory subsystem. *Future Gener. Comput. Syst.* **2018**, *82*, 510–519. [[CrossRef](#)]
20. Norman, M.; Larkin, J.; Vose, A.; Evans, K. A case study of CUDA FORTRAN and OpenACC for an atmospheric climate kernel. *J. Comput. Sci.* **2015**, *9*, 1–6. [[CrossRef](#)]
21. Schalkwijk, J.; Jonker, H.J.; Siebesma, A.P.; Van Meijgaard, E. Weather forecasting using GPU-based large-eddy simulations. *Bull. Am. Meteorol. Soc.* **2015**, *96*, 715–723. [[CrossRef](#)]
22. Mielikainen, J.; Huang, B.; Huang, H.L.; Goldberg, M.D. Improved GPU/CUDA based parallel weather and research forecast (WRF) single moment 5-class (WSM5) cloud microphysics. *IEEE J. Sel. Top. Appl. Earth Obs. Remote. Sens.* **2012**, *5*, 1256–1265. [[CrossRef](#)]
23. Wang, Y.; Zhao, Y.; Li, W.; Jiang, J.; Ji, X.; Zomaya, A.Y. Using a GPU to accelerate a longwave radiative transfer model with efficient CUDA-based methods. *Appl. Sci.* **2019**, *9*, 4039. [[CrossRef](#)]
24. NVIDIA, CUDA C Programming Guide v10.0. Technical Document (2018). Available online: https://docs.nvidia.com/pdf/CUDA_C_Programming_Guide.pdf (accessed on 1 October 2019).
25. Mielikainen, J.; Huang, B.; Huang, H.L.; Goldberg, M.D. GPU acceleration of the updated Goddard shortwave radiation scheme in the weather research and forecasting (WRF) model. *IEEE J. Sel. Top. Appl. Earth Obs. Remote. Sens.* **2012**, *5*, 555–562. [[CrossRef](#)]
26. Huang, M.; Huang, B.; Chang, Y.L.; Mielikainen, J.; Huang, H.L.; Goldberg, M.D. Efficient parallel GPU design on WRF five-layer thermal diffusion scheme. *IEEE J. Sel. Top. Appl. Earth Obs. Remote. Sens.* **2015**, *8*, 2249–2259. [[CrossRef](#)]
27. Huang, M.; Huang, B.; Gu, L.; Huang, H.L.; Goldberg, M.D. Parallel GPU architecture framework for the WRF Single Moment 6-class microphysics scheme. *Comput. Geosci.* **2015**, *83*, 17–26. [[CrossRef](#)]
28. NVIDIA, CUDA Fortran Programming Guide and Reference. Technical Document (2019). Available online: <https://www.pgroup.com/resources/docs/19.1/pdf/pgi19cudaforug.pdf> (accessed on 1 October 2019).
29. Ruetsch, G.; Phillips, E.; Fatica, M. GPU acceleration of the long-wave rapid radiative transfer model in WRF using CUDA Fortran. In Proceedings of the Many-Core and Reconfigurable Supercomputing Conference, Roma, Italy, 22–24 March 2010. Available online: <https://pdfs.semanticscholar.org/6844/d70506d1f79ce7a70fa505a4625febd2dec2.pdf> (accessed on 1 October 2019).

30. Price, E.; Mielikainen, J.; Huang, M.; Huang, B.; Huang, H.L.; Lee, T. GPU-accelerated longwave radiation scheme of the Rapid Radiative Transfer Model for General Circulation Models (RRTMG). *IEEE J. Sel. Top. Appl. Earth Obs. Remote. Sens.* **2014**, *7*, 3660–3667. [[CrossRef](#)]
31. Mielikainen, J.; Price, E.; Huang, B.; Huang, H.L.; Lee, T. GPU compute unified device architecture (CUDA)-based parallelization of the RRTMG shortwave rapid radiative transfer model. *IEEE J. Sel. Top. Appl. Earth Obs. Remote. Sens.* **2016**, *9*, 921–931. [[CrossRef](#)]
32. Wang, Z.; Xu, X.; Xiong, N.; Yang, L.T.; Zhao, W. GPU acceleration for GRAPES meteorological model. In Proceedings of the IEEE International Conference on High Performance Computing and Communications, Banff, AB, Canada, 2–4 September 2011; pp. 365–372.
33. Mlawer, E.J.; Iacono, M.J.; Pincus, R.; Barker, H.W.; Oreopoulos, L.; Mitchell, D.L. Contributions of the ARM program to radiative transfer modeling for climate and weather applications. *Ams Meteorol. Monogr.* **2016**, *57*, 15.1–15.19. [[CrossRef](#)]
34. Chen, D.; Li, D.; Xiong, M.; Bao, H.; Li, X. GPGPU-aided ensemble empirical-mode decomposition for EEG analysis during anesthesia. *IEEE Trans. Inf. Technol. Biomed.* **2010**, *14*, 1417–1427. [[CrossRef](#)]
35. Lu, F.; Song, J.; Cao, X.; Zhu, X. CPU/GPU computing for long-wave radiation physics on large GPU clusters. *Comput. Geosci.* **2012**, *41*, 47–55. [[CrossRef](#)]



© 2020 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<http://creativecommons.org/licenses/by/4.0/>).