Computational benefit of GPU optimization for the atmospheric chemistry modeling

Jian Sun[1,*], Joshua S. Fu[1,2], John B. Drake[1], Qingzhao Zhu[1], Azzam Haidar[3], Mark Gates[3], Stanimire Tomov[3] and Jack Dongarra[3]

[1]Department of Civil and Environmental Engineering, University of Tennessee, Knoxville, TN, 37996, USA

[2]Climate Change Science Institute and Computational Sciences and Engineering Division, Oak Ridge National Laboratory, Oak Ridge, TN, 37831, USA

[3]Innovative Computing Laboratory, University of Tennessee, Knoxville, TN, 37996, USA

[*]now at: Atmospheric Science and Global Change Division, Pacific Northwest National Laboratory, Richland, Washington, USA

Correspondence to: Joshua S. Fu (jsfu@utk.edu)

Key points:

- A combination of fully interleaved memory layout, CUDA streams and constant memory yields the best performance on the GPU
- The Intel compiler with block-interleaved memory layout provides the best performance on the CPU.
- The GPU version achieves a factor of ~11.7× speed-up for computation alone and ~3.82× speed-up when the data transfer is considered

Abstract

Global chemistry-climate models are computationally burdened as the chemical mechanisms become more complex and realistic. Optimization for a GPU may make longer global simulation with regional detail possible, but limited study has been done to explore the potential benefit for the atmospheric chemistry modeling. Hence in this study, the second-order Rosenbrock solver of the chemistry module of CAM4-Chem is ported to the GPU to gauge potential speed-up. We find that on the CPU, the fastest performance is achieved using the Intel compiler with a block interleaved memory layout. Different combinations of compiler and memory layout lead to ~11.02× difference in the computational time. In contrast, the GPU version performs the best when using a combination of fully interleaved memory layout with block size equal to the warp size, CUDA streams for independent kernels and constant memory. Moreover, the most efficient data transfer between CPU and GPU is gained by allocating the memory contiguously during the data initialization on the GPU. Compared to one CPU core, the speed-up of using one GPU alone reaches a factor of ~11.7× for the computation alone and ~3.82× when the data transfer between CPU and GPU is considered. Using one GPU alone is also generally faster than the multithreaded implementation for 16 CPU cores in a compute node and the single source solution (OpenACC). The best performance is achieved by the implementation of the hybrid CPU/GPU version but rescheduling the workload among the CPU cores is required before the practical CAM4-Chem simulation.

Key points: GPU, CUDA, compiler, memory layout, data transfer, hybrid

1. Introduction

Physics and chemistry are closely coupled in the framework of earth system modeling. Most radiatively active compounds (e.g., CH_4, O_3, and aerosols) in Earth's current atmosphere are also chemically active. The atmospheric chemistry is an essential component of climate [Tian and Chipperfield, 2006; IPCC AR5 Chapter 8, 2013; Collins et al., 2017], which includes the homogeneous (e.g., gas-phase species) and heterogeneous (e.g., gas and aerosol) reactions, aerosol and acid (nitrogen + sulfur) deposition, and cloud-aerosol interactions in the atmosphere. A robust representation of atmospheric chemistry including the chemical reaction with other species (e.g., gaseous species, aerosols and water) and photolysis (interaction with solar radiation) is thus crucial to determine the burden and lifetime of chemically active compounds [Su et al., 2011; Dameris and Jöckel, 2013; Lamarque et al., 2013; Tilmes et al., 2016].

Due to the complex chemical mechanism and strong variability of temporal and spatial patterns, atmospheric chemistry is usually characterized by the significant non-linearity [Kleinman et al., 2001] and a wide range of temporal and spatial scales [Isaksen et al., 2009], which makes it challenging to model. High resolution global models capture some of the same scales as mesoscale weather and regional air quality models so an argument can be made that global models should exercise more comprehensive chemistry. This prompts us to review and improve all the computational methods of global chemical simulation. Currently, a first-order implicit solver is widely used in the global chemistry-climate models [Austin et al., 2003; Horowitz et al., 2003; Schraner et al., 2008; Emmons et al., 2010]. The first-order implicit solver is unconditional stable [Kinnison et al., 2007], but may suffer from low computational efficiency and low accuracy. Sun et al. [2017] implemented a second-order Rosenbrock (ROS-2) solver in the global chemistry-climate model (CAM4-Chem), replacing the original first-order implicit solver. The results showed that utilizing the same optimized subroutine structure, the ROS-2 solver achieved ~2× speed-up on the CPU over the original first-order implicit solver. This speed-up results from avoiding a re-evaluation of the Jacobian matrix and LU factorization during the two-stage computation. For the overall performance, the chemistry takes ~24% of the total computational time for the atmospheric component (CAM) and the chemical solver itself dominates ~52% of the computational time for the chemistry. Thus ~2× speed-up from the chemical solver is likely to save ~6% of the total computational time for CAM.

In addition to improving the numerical method of the solver, new computer architectures demand a review of the optimization strategy. The heterogeneous architecture of supercomputers has developed rapidly now including multi-node parallelism and graphics processing units (GPUs). Optimizing for the GPU, considerable speed-up was achieved for both regional [Michalakes and Vachharajani, 2008; Linford et al., 2009] and global models [Korwar et al., 2013; Xu et al., 2015]. Networks of chemical reactions required to model atmospheric chemistry and other applications were accelerated two or more orders of magnitude on GPUs using a new algebraically-stabilized fast explicit approach for kinetic integration [Haidar et al., 2016]. Implicit integration methods, while more stable and accurate, result in solving systems of equations. For these implicit solvers, the GPU is highly efficient in solving large and dense matrix systems [Abdelfattah et al., 2018], but the atmospheric chemistry problem is characterized by small (size less than 100 x 100) and sparse (10% non-zero elements) matrix systems. However, it was still possible to utilize the GPU efficiently when a large number of small matrices were solved simultaneously and independently [Dong et al., 2014; Abdelfattah et al., 2017; Haidar et al., 2018]. Alvanos and Christoudias [2017] recently used the GPU accelerators to speed up the chemistry module of the global chemistry-climate model ECHAM/MESSy Atmospheric Chemistry (EMAC) by a factor of 1.75×. However, the CUDA codes in their study were parsed from the Fortran codes generated by the Kinetic PreProcessor (KPP). It was incompatible with the global chemistry-climate models such as CAM4-Chem that did not use KPP. In addition, little information was provided to explore the optimal configurations of the CUDA kernels. Hence in this work, we will port the ROS-2 chemical solver of CAM4-Chem [Sun et al., 2017] to the GPU to solve the chemistry as a box model and examine a series of optimization strategies. The goal is to investigate whether the chemistry box model can benefit from the GPU and its associated most optimized configuration. We use CUDA as it is the native, vendor-supported language for the NVIDIA GPUs on Titan, which provides the best possible performance. OpenACC may provide some measure of portability to other GPU platforms, and similar techniques should be applicable to those technologies. Nevertheless, at best it could match CUDA's performance, not exceed it. In addition, OpenACC is not compatible with all the compilers so its application will be restricted for the machines with Intel compiler only. The CUDA codes are translated from the original Fortran codes in CAM4-Chem by the Perl scripts, with minor modifications for the header files and interface. These scripts can be further integrated into the chemistry preprocessor so that it will be more feasible for the community. The investigation in this study will inform the software engineering choices that developers need to make to effectively optimize global chemistry-climate models for high resolution and comprehensive atmospheric chemical mechanisms.

2. Methodology

2.1. Data structure

In CAM4-Chem, the default finite volume dynamic core uses a latitude × longitude × vertical-level grid over the global sphere [Lin, 2004; Mirin and Worley, 2011]. In order to achieve parallelization, domain decomposition is involved, which divides the global domain into different subdomains. Each subdomain contains k chunks. There are m columns inside each chunk and each column consists of n vertical layers. The parameters k, m and n are determined by the grid resolution and for the 1° x 1° horizontal resolution, k = 26, m = 16 and n = 26 by default. Each subdomain is assigned as an MPI task, and OpenMP directives are used for thread-level parallelism when looping over chunks in each subdomain [Worley and Drake, 2005]. Therefore, each OpenMP thread handles exactly the chemistry computation inside one chunk. For the chemistry at each grid point inside a chunk, a system of ordinary differential equation (ODE) is solved at each time step, which takes the following form:

$$\frac{\mathrm{Dy}}{\mathrm{Dt}} = \mathrm{P(y)} - \mathrm{L(y)} + \mathrm{I(y)} \qquad (1)$$

where y is the vector of volume mixing ratios for the chemical species at a given grid cell; the right hand side source terms include the production P(y) and loss L(y) due to chemical reactions and the external forcing (i.e., lightning and aircraft emissions). To solve the ODE system above, the second-order Rosenbrock method is applied [Verwer et al., 1999]:

$$(\mathrm{I} - \mathrm{h}\gamma\mathrm{A})\mathrm{k}_1 = \mathrm{F}(\mathrm{y}^\mathrm{n}) \qquad (2)$$

$$(\mathrm{I} - \mathrm{h}\gamma\mathrm{A})\mathrm{k}_2 = \mathrm{F}(\mathrm{y}^\mathrm{n} + \mathrm{hk}_1) - 2\mathrm{k}_1 \qquad (3)$$

$$\mathrm{y}^{\mathrm{n}+1} = \mathrm{y}^\mathrm{n} + \frac{3}{2}\mathrm{hk}_1 + \frac{1}{2}\mathrm{hk}_2 \qquad (4)$$

where I is an N x N identity matrix; h is the time step size; γ is a constant parameter; $\mathrm{A} = \left.\frac{\partial \mathrm{F(y)}}{\partial \mathrm{y}}\right|_{\mathrm{y}=\mathrm{y}^\mathrm{n}}$ is the Jacobian matrix at time t= t^n; y^n and $\mathrm{y}^{\mathrm{n}+1}$ are the species mixing ratios at time t = t^n and $\mathrm{t}^{\mathrm{n}+1}$, respectively; vectors k_1 and k_2 are the intermediate solutions at each stage. In the real implementation, the dimension of the species concentration array is declared as (ncol, pver, gas_pcnst), where ncol is the number of columns allocated to a given chunk, pver is the number of vertical layers allocated to a given column and gas_pcnst is the total number of species. The second-order Rosenbrock method solves an ODE system following the Equation (2) to (4) for a given column and vertical layer. Therefore, the loop structure to solve the ODE above would look like the one shown in Figure 1. Since the chemistry in CAM4-Chem is treated independently among different columns and vertical layers, it behaves exactly like a box model for a given column and vertical layer. Therefore, the chemistry is characterized with potentially massive parallelism, which makes it suitable for the computation on the GPU. To simplify the codes, the chemistry box model in the rest of this study will collapse the two loops inside one chunk as one loop, with chunk size equal to the number of loop iterations. The number of loop iterations is calculated by the equation below:

$$\text{Number of loop iterations} = \text{ncol} \times \text{pver} \tag{5}$$

The number of loop iterations is tunable and examined for a wide range in this study. The main motivation is to mimic the real scenarios with different horizontal resolutions, which leads to different number of columns in each chunk. This will change the corresponding number of loop iterations for the chemistry update. A larger number of loop iterations corresponds to more columns inside a chunk. This typically represents a scenario with fine horizontal resolution in CAM4-Chem, which may target at the purpose of numerical weather prediction. On the other hand, the smaller number of loop iterations corresponds to fewer columns inside a chunk. This typically represents a scenario with coarse horizontal resolution in CAM4-Chem, which may target at the purpose of long-term climate simulation. Note that the number of loop iterations in a chunk is ncol × pver for each CPU core on a Titan node. Thus we are comparing one CPU core with one GPU throughout the rest of this study to make the number of loop iterations identical.

Figure 1 here.

2.2. Architecture

The Titan supercomputer at Oak Ridge National Laboratory is used for the computational performance and analysis. Each Titan compute node contains one AMD Opteron™ 6274 (Interlagos) CPU (16 cores) and one NVIDIA Tesla™ K20X (Kepler) GPU connected through a PCI express 2.0 interface. The AMD Opteron™ 6274 CPU supports the 4-wide Fused Multiply-Add (FMA) vector instructions. On the CPU, three major compilers (GNU: gcc/4.9.3, Intel: intel/17.0.0.098 and PGI: pgi/17.9.0) are all supported by Titan. GPU computation is organized into thread blocks, where each thread block has one or more warps of 32 threads each, and all the instructions (e.g., addition) are issued at the warp level. This execution model is called single instruction multiple thread (SIMT). A function launched on the GPU is called a kernel. Each Kepler K20X GPU contains 14 streaming multiprocessors (SMX) that can run up to 2048 threads, 16 thread blocks, or 64 warps. Note that there are several other important limits imposed on the Kepler K20X GPU such as a maximum of 255 registers per thread or 65,536 per SMX, 48 kilobytes shared memory per SMX, and 64 kilobytes constant memory per GPU. Violating any of these limits will lead to kernel launch failure.

2.3. Memory layout

The memory layout is known to play a critical role in achieving good computational performance [Dongarra et al., 2016]. The strided and interleaved memory layouts are generally competitive for problems of very small sizes that are available in the fast GPU-accelerated implementation of the standard basic linear algebra subroutines (cuBLAS) and other research [Dong et al., 2014; Abdelfattah et al., 2016; Gates, et al., 2017; Haidar et al., 2018]. Therefore, their effects on our problems will be investigated. In Fortran, arrays are stored column-wise, so the strided memory layout (SML) stores the matrices consecutively as shown in Figure 2a. All the elements in a matrix will be stored together before moving to the next matrix. Instead, the fully interleaved memory layout (FIML) stores the i-th entries of all matrices consecutively (Figure 2b). By storing each matrix contiguously, it is clear that the SML can access two elements in the same matrix quickly. In contrast, the elements in the same matrix are not stored consecutively in the FIML. For example, if the number of matrices is N, moving from the first element to the second element in the same matrix requires a jump of N memory locations, which could hinder opportunities to reuse cached data, and thus reduce the overall performance when N is large. Despite this drawback, the FIML is expected to benefit from vectorization --- an implicit single instruction multiple data (SIMD) parallelization for a single core processor, where the code is transformed into SIMD vector operations (e.g., addition) that can be executed in parallel as single instructions. On a modern multi-core computational architecture, achieving a high-level vectorization is important for obtaining excellent performance. The SML may not utilize vectorization as effortlessly as the FIML. For instance, consider the following four calculations:

$$a_1 = r_1 * b_1 + c_1 \tag{6}$$

$$a_2 = r_2 * b_2 + c_2 \tag{7}$$

$$a_3 = r_3 * b_3 + c_3 \tag{8}$$

$$a_4 = r_4 * b_4 + c_4 \quad (9)$$

where a, b, c are the volume mixing ratios of three species and r is the reaction rate for different chemical systems. For the SML, it can only use one Advanced Vector Extensions (AVX) register and thus will take four clock cycles to complete the four addition instructions. However, for the FIML, it only takes one clock cycle since it can use four AVX registers due to its data storage structure. There is an intermediate approach between these two memory layouts, called the block interleaved memory layout (BIML). In the BIML, instead of interleaving all N matrices, the first K matrices are interleaved, then the next K matrices, and so on (Figure 2c). The main benefit of the "block interleaved" approach is that when moving from the first element to the second element in the same matrix, it requires a jump of only K memory locations instead of N memory locations for the FIML. Meanwhile, the BIML can readily utilize vector instructions, unlike the SML. In this study, we will explore the performance of these three memory layouts and see which one fits the best for the atmospheric chemistry modeling. Note that the SML and FIML can be treated as a special case of BIML. Take the array of volume mixing ratios in CAM4-Chem for example. Denoting N as the number of loop iterations, the array is declared as Array(gas_pcnst, N) for the SML, Array(N, gas_pcnst) for the FIML and Array(K, gas_pcnst, N / K) for the BIML. Converting the data array between two memory layouts is done efficiently by changing the parameter K in the main driver of chemistry box model, and this is not included in the measurement of wall-clock time. For the BIML, K is a tuning parameter and the optimal choice is shown as 4 for both GNU and Intel compiler but 8 for PGI compiler (Figure 3). This is reasonable as there is no benefit beyond matching the native vector length of the processor (e.g., 4 for AVX, 32 (warp size) for NVIDIA GPUs), while a longer vector will decrease cache performance, particularly for L1 cache due to its small size. Therefore, these optimal block sizes will be used for the corresponding compilers throughout the following analysis.

Figure 2 here.

Figure 3 here.

3. Results and discussion

3.1 Basic analysis

Before proceeding to investigate the computational performance, it is necessary to understand the computational rate and memory bandwidth requirements of the chemistry box model. The main functions of the chemistry box model include formation of the Jacobian matrix, LU factorization and solve, and formation of the right hand side source term. The analysis of floating point operations (FLOP) for one loop iteration indicates that formation of the Jacobian matrix and LU (both factorization and solve) should consume roughly 70% of the total computational time (Table 1). Note that the calculations of FLOP for LU solve and formation of the right hand side source term have been multiplied by a factor of two, due to the two-stage computation in the ROS-2 method (the same for the calculation of data copy later). On the other hand, formation of the Jacobian matrix requires a significantly higher amount of data copy (both copy in and copy out) than other functions. This is due to the fact that formation of the Jacobian matrix consists of two sub-functions, which calculate the linear and non-linear components of the Jacobian matrix separately. However, in the real architecture, caching will significantly reduce the cost of accessing data from main memory. In order to take the cache effect into account, the Performance Application Programming Interface (PAPI v5.5) is used to measure the L2 cache misses. The data transfer per second (unit: GB/s) for the whole program and each function of the chemistry box model is then estimated by:

$$\text{Date transfer per second} = \frac{\text{L2 cache miss}\times\text{size of cache line}}{\text{time}} \qquad (10)$$

where the size of cache line is 64 bytes for the processor on Titan. Besides the L2 cache misses, the computational rate (unit: GFLOP per second (GFLOPS = 1E+9 FLOPS)) is also measured by PAPI for 128, 1,024 and 10,240 loop iterations. The results generally vary among different number of loop iterations, compilers and memory layouts (Figure 4) for the data transfer and computational rate (Figure 5). Considering the AMD Opteron$^{\text{TM}}$ 6274 processor on Titan, the theoretical peak computational rate of one CPU core is estimated by:

$$\begin{aligned}\text{Peak computational rate} &= 2.2\ (\text{GHz}) \times 8\ (\text{double precision FLOP per cycle for FMA4})\\ &= 17.6\ \text{GFLOPS} \qquad (11)\end{aligned}$$

The practical peak memory bandwidth is measured by the STREAM benchmark program provided by the University of Virginia (https://www.cs.virginia.edu/stream/) and the results show that when running with one thread, the memory bandwidth on Titan varies from 6.86 GB/s to 12.68 GB/s for different operations, such as copy and scale, and compilers.

Compared with these hardware limitations, the whole chemistry box model reaches only up to 8% of the theoretical peak computational rate (PGI compiler with BIML) and 27.58% of the practical peak memory bandwidth (GNU compiler with FIML). However, these percentages increase to 12.21% (LU solve, GNU compiler with BIML) and 50.76% (Init, GNU compiler with BIML) for the individual functions. It is worth noting that in modern computer processors, there is also a technique named "cache prefetching" that fetches data into cache before it is needed. This will further increase the data transfer as measured by L2 cache misses for the functions above. Therefore, the analysis here reveals that the chemistry box model performance is not limited by the computational rate but by the memory bandwidth. Since the GPU has both more floating point cores and higher memory bandwidth than the CPU [Mantell et al., 2016; Alvanos and Christoudias, 2017], we believe that it is still promising to gain some computational benefit from the GPU. Note that L2 cache misses are used as an indication of cache performance and memory bandwidth. Although the limitation for the chemistry box model is the bandwidth to the main memory and L3 cache, no counters for measuring the L3 cache hits and misses are available on Titan. Besides, the data transfer between L1 and L2 cache is much faster than the data transfer between L2 and L3 cache. Therefore, it is more appropriate to use the PAPI events for L2 cache instead of L1 cache as an indication of cache performance and memory bandwidth.

Figure 4 here.

Figure 5 here.

3.2 CPU

For the chemistry box model on the CPU, the two most dominant factors that may affect the computational performance are the compilers and memory layouts. On Titan, three major compilers are examined in this study with the flags that enable AVX vectorization ("-O3 -fopenmp -mavx" for GNU, "-O3 -qopenmp -mavx" for Intel and "-O3 -openmp -Mvect=simd:256" for PGI). In addition, memory alignment flags ("-Mcache_align -fastsse" for PGI) or directives ("__assume_aligned" for Intel and "__builtin_assume_aligned" for GNU) are used to further assist vectorization. The results show that the fastest computational time for a given number of loop iterations is achieved by the BIML for all the three compilers (Figure 6). Using SML and FIML require ~1.42× and ~2.89× the computational time of BIML for the GNU compiler (Figure 6a), ~1.75× and ~6.95× for the Intel compiler (Figure

6b), and ~1.54× and ~7.68× for the PGI compiler (Figure 6c), respectively. In addition, the Intel compiler with BIML slightly outperforms among all the configurations for different number of loop iterations. The assembly files (*.s) for the individual functions are further investigated for the three compilers and memory layouts. Many "packed double" vector instructions are generated for all the three compilers with FIML and BIML, while the instructions are mainly non-vectorized "scalar double" for all the three compilers with SML. This highlights the benefit of using FIML and BIML to assist efficient vectorization. However, using FIML generally yields poor computational performance here, which may be related to its higher data transfer per second shown in Figure 4. On the other hand, using SML still achieves fast computational performance, which is due to the fact that all the data is already loaded into cache during data initialization (Figure 4). Therefore, the data transfer between cache and main memory is significantly reduced for the remaining functions. Using BIML is able to vectorize the loops efficiently and reduce the data copy at the same time, thus leading to the fastest computational performance for all the three compilers.

Figure 6 here.

The total wall-clock time of each function in the chemistry box model shows that for GNU compiler, LU factorization consumes the highest amount of time for both SML and BIML (Figure 7a and 7c) while formation of the Jacobian matrix and LU solve become the most computationally expensive parts for FIML, depending on the number of loop iterations (Figure 7b). For the Intel compiler (Figure 7d, 7e and 7f), formation of the Jacobian matrix costs the highest amount of time for all three memory layouts, except the FIML with larger number of loop iterations, where LU factorization dominates the consumption of time. For the PGI compiler (Figure 7g, 7h and 7i), formation of the Jacobian matrix and LU factorization take similar computational time for SML. However, formation of the Jacobian matrix is more computationally expensive than other functions when using BIML, and LU factorization consumes more computational time for FIML with larger number of loop iterations. It is clear that formation of the Jacobian matrix and LU (both factorization and solve) cost more than 70% of the total wall-clock time in most configurations, which is consistent with the previous analysis in Sections 3.1. It also proves that the compilers, together with the choice of memory layout, can affect the computational performance significantly. For a given compiler, the largest difference between different memory layouts can be as high as a factor of 11.02 (i.e., PGI: FIML vs. BIML). For a fixed memory layout, the largest difference between different compilers can also be around a factor of 3.01 (i.e., FIML: GNU vs. PGI).

Figure 7 here.

3.3 GPU

3.3.1 Strided vs. interleaved vs. block interleaved memory layout

The CUDA platform used in this study is cudatoolkit/7.5 on Titan. The NVIDIA visual profiler (NVVP) results of the GPU version of chemistry box model for SML, FIML and BIML show that the LU factorization kernel has already hit the limit of 255 registers per thread. Therefore, a maximum 256 threads (correspondingly 12.5% occupancy) can be launched simultaneously in each SMX for this kernel in order to not exceed the threshold of 65,536 registers per SMX. Since there are 14 SMXs per GPU on Titan and instructions are executed at the warp (size = 32) level, the number of loop iterations is therefore chosen as a multiple of 14 x 32 = 448. Similar to Section 3.1, we first examine the impact of different block sizes for the BIML on the GPU, ranging from 4 to 256. It turns out that using a thread count smaller than 32 leads to a clear increase of computational time, especially for large number of loop iterations (Figure 8). This is reasonable as all the computation on the GPU is executed at the warp level, which equals to 32 threads per block. Using fewer than 32 threads per block will fail to fully use the resources and thus perform worse. On the other hand, using 32, 64, 128 and 256 threads will launch 8, 4, 2 and 1 thread blocks per SMX, respectively. In this study, they behave very close to each other and using 32 threads per block may outperform slightly. Therefore, 32 is used as the optimal block size for the rest of this study. For SML, the computational time is 2.27× to 4.13× the data transfer time between CPU and GPU (Figure 9a and 9b). Except for the data deallocation (Figure 9c), the time of both computation and data transfer grows linearly with the number of loop iterations. In contrast, for both FIML and BIML, the computational time is less than the data transfer time except when doing 448 loop iterations. For the summed time (Figure 9d), using SML requires 1.65× to 2.80× total wall-clock time as that of using BIML and 1.89× to 3.18× total wall-clock time as that of using FIML. This difference is mainly caused by the shorter computational time using BIML and FIML. On the GPU, each thread works on exactly one matrix and the memory bandwidth is much higher than that on the CPU. GPU cores usually access the global memory directly instead of through the memory hierarchy on the CPU. Hence, FIML and BIML seem to benefit more from the SIMT model than SML. In particular, FIML achieves the fastest computation, which is different from what is observed on the CPU (see Section 3.2). The total wall-clock time of each function shown in Figure 10 further confirms that using SML will spend a significant amount of time on formation of the Jacobian matrix, formation of the right hand side source term and update of intermediate and final solutions, which can be done very efficiently using FIML. Even for the LU functions (factorization and solve), the actual wall-clock time using FIML is also smaller than the one using SML. According to Figure 10a, 10b and 10d, the better computational performance of FIML over BIML is mainly attributed to the fast computation of formation of the Jacobian matrix, LU factorization and formation of the right hand side source term using FIML. This suggests that FIML is the best choice of memory layout on the GPU for the chemistry box model and thus will be used for the following examinations in this study.

Figure 8 here.

Figure 9 here.

Figure 10 here.

3.3.2 Multiple kernels vs. one kernel

In Section 3.3.1, the GPU version of chemistry box model is implemented by launching each function as an individual kernel. The NVVP results show that there is a clear overhead time between two separate kernels. In order to avoid the overhead time, it is possible to assemble all the functions into one kernel that is launched just once. It seems when the number of loop iterations is smaller than 3,584, using one kernel costs about 96% to 98% of the total wall-clock time as that of using the multiple kernels (Figure 11). However, when the number of loop iterations is larger than 3,584, using one kernel takes ~1.1× total wall-clock time as that of using the multiple kernels. Although the overhead time between two functions is eliminated by implementing the one kernel version, this does not necessarily always speed up the computation. One potential reason is that the compiler can optimize each kernel separately in the multiple kernels version, while it may not be feasible for the one kernel version. For example, in the one kernel version, the maximum number of threads per SMX is limited by the LU factorization, which is 256 in this study (see Section 3.3.1). However, in the multiple kernels version, the theoretical occupancies for functions like formation of the

Jacobian matrix and the right hand side source term can reach as high as 18.8% and 25%, respectively. Therefore, more threads per SMX can be involved for these functions and the computational time is thus reduced, especially for the large number of loop iterations.

Figure 11 here.

3.3.3 Shared and constant memory

As mentioned in Section 2.2, there are 48 kilobytes shared memory per SMX and its memory latency is relatively low compared to the GPU's global DRAM memory. Porting some frequently visited arrays to shared memory should increase the memory access speed and thus save some computational time. For the chemistry box model, two arrays (solution vector and intermediate solution vectors) are good candidates for the shared memory:

$$\text{Size of solution vector} = 103\ (\text{number of species}) \times 32\ (\text{numer of threads}) \times 8\ (\text{bytes for double precision number}) = 25.75\ \text{kilobytes} \tag{12}$$

$$\text{Size of intermediate solution vectors} = 95\ (\text{number of extracted species}) \times 32\ (\text{number of threads}) \times 2\ (\text{number of stages}) \times 8\ (\text{bytes for double precision number}) = 47.5\ \text{kilobytes} \tag{13}$$

We port these two vectors separately to the shared memory and compare their performances with the one kernel version (note that only the one kernel version is able to effectively exploit the shared memory). The NVVP results show that when using shared memory, the theoretical occupancy reaches just 1.6% (corresponding to 1.6% × 2048 (maximum number of threads per SMX) = 32 threads) and thus only one thread block can be launched per SMX. The results show that when the number of loop iterations is 448, each SMX will launch only one thread block and using shared memory is faster in this case (Figure 12). In particular, porting the intermediate solution vectors to shared memory can save up to 26% of the computational time since it could almost fully use the shared memory. However, when the number of loop iterations increases to 898 or larger, the shared memory version is slower than the no shared memory version by ~4.4× (blue line in Figure 12) and ~3.5× (red line in Figure 12) when the number of loop iterations equals to 3,584. This is caused by the fact that for the no shared memory version, each SMX can have 256 threads working simultaneously and the total number of working threads is 14 × 256 = 3,584. Therefore, the no shared memory version can launch 8 thread blocks per SMX (256 (maximum number of threads per SMX) / 32 (number of threads per block) = 8), while the shared memory version can launch only one thread block per SMX as mentioned above. This is also consistent with the observation that the computational time of shared memory version grows linearly with the number of loop iterations, while the computational time of the no shared memory version increases slightly within 3,584 loop iterations (mainly due to the overhead time) but dramatically between 3,584 and 5,376 loop iterations.

Figure 12 here.

Besides the shared memory, there are also 64 kilobytes constant memory that reside in the GPU's global DRAM memory and can be broadcast among all the SMXs. For the chemistry box model in this study, there are two integer mapping arrays used to extract the 95 reaction-active species from the total 103 chemical species and permute it to an appropriate order with fewer fill-in values [Sun et al., 2017]. In the previous implementation, one copy of these arrays is generated for each thread, which is clearly not necessary since all the values in the mapping arrays are constant. Therefore, some computational time can be saved by storing the mapping arrays in the constant memory where all the SMXs can access them simultaneously. The results show that for the shared memory version which stores the intermediate solution vectors, using constant memory for the mapping arrays could save up to 6.7% of the computational time (blue and green lines in Figure 13). For the no shared memory version, using the constant memory can also save 3.6% to 5.2% of the computational time (black and red lines in Figure 13). This small improvement is still impressive and worth implementation considering the small size of the mapping arrays (2 (number of arrays) × 95 (elements in each array) × 4 (bytes of an integer for a 64-bit system) = 760 bytes).

Figure 13 here.

3.3.4 Stream

A CUDA stream refers to a queue of work such as kernel launches and memory copies. Operations in the same stream are ordered and can't be overlapped, while operations in different streams can be run in parallel if there are no data dependencies between streams. According to the numerical steps in Section 2.1, formation of the Jacobian matrix can be done in parallel with the initialization of local data and formation of the right hand side source term in the first stage. Hence, some computational time can be saved by involving CUDA streams here. Note that the streamed version works only with multiple kernels, so we also compare with the computational time of the one kernel version with constant memory (the fastest version so far) to see whether we really benefit from the streamed multiple kernels version. The results indicate that using the streamed multiple kernels with constant memory (red line in Figure 14) is likely to save about 4% (7,168 loop iterations) to 16% (448 loop iterations) of the computational time, compared to the previous multiple kernels version (blue line in Figure 14). It is also faster than the one kernel version with constant memory (black line in Figure 14) but may only save 1.8% (3,584 loop iterations) to 11.2% (5,376 loop iterations) of the computational time.

Figure 14 here.

3.3.5 Memory copy

In the previous sections, we mainly focus on the optimization of the computational time. As observed in Figure 9, the data transfer between CPU and GPU can also consume a significant amount of time and its time is even higher than the computational time when using FIML and BIML (Figure 9a and 9b). Therefore, optimizing the data transfer between CPU and GPU is likely to save additional total wall-clock time. Three strategies are investigated here: 1) calling "cudaMalloc" for each array separately; 2) allocating a large space for all the arrays, like mixing ratios of chemical species and reaction rates, so that they are contiguous in the memory locations; 3) using pinned memory by calling the function "cudaMallocHost" before "cudaMemcpy". The results show that when allocating all the arrays contiguously in the memory locations (red line in Figure 15), it saves about 40% of the time for the 448 loop iterations, compared to the baseline (blue line in Figure 15) where each array is allocated separately. When the number of loop iterations increases, the percent of saved time decreases to as low as 10% for the 7,168 loop iterations. When using the pinned memory, it costs ~1.1× data transfer time as that of baseline for the 448 loop iterations but increases to ~2.1× for the 7,168 loop iterations (black line in Figure 15). The NVVP results indicate that for all the three cases, the copy rate can reach 5.75 GB/s for the 448 loop iterations. When the number of loop iterations grows to 7,168, the copy rate reduces to ~3 GB/s for the non-pinned memory but increases slightly to 6 GB/s for the pinned memory. However, it also suggests that for the pinned memory, it spends much more time on the "cudaMallocHost" and thus eliminates the benefit of the fast copy rate for different number of loop iterations. On the other hand, it is worth our efforts to allocate a contiguous memory space for all the input arrays, especially for the small number of loop iterations.

Figure 15 here.

Based on the results above, it seems that the most efficient GPU version of the chemistry box model is using the streamed multiple kernels with constant memory and the contiguous memory allocation for all the arrays. We further compare it with the fastest CPU version (Intel compiler with BIML) and the results show that the CPU version (black line in Figure 16) requires ~2.33× computational time as that of GPU version (blue line in Figure 16) for the 448 loop iterations. This factor increases rapidly with the number of loop iterations and reaches up to 11.7× for the 7,168 loop iterations. When the time for the data transfer between CPU and GPU (green line in Figure 16) and “cudaFree” (pink line in Figure 16) is considered, the total wall-clock time of CPU version is still ~1.29× as that of GPU version for the 448 loop iterations and grows to ~3.82× for the 7,168 loop iterations. This clearly shows that the GPU version is superior to the CPU version for the computation alone. When the number of loop iterations is larger than 1,792 and the data transfer time is higher than the computational time alone, the GPU version is still faster than the CPU version and this speed-up is considerable as long as the number of loop iterations is large enough.

Figure 16 here.

3.4 Application

3.4.1 Study of the effect of the multithreading for CPU

In the previous context, the comparison was made between one GPU and one CPU core. In Section 2.1 we have mentioned that the parallel design for a CPU implementation would be applied to loop over the chunks inside one subdomain. This means, that if we use OpenMP as a multithreading framework to easily implement the parallelization as described above, each OpenMP thread will have to solve independent loop iterations of chemistry. Therefore, in this section, we will report on the comparison of the computational performance between 16 CPU cores in a node and one GPU for the Titan architecture. In Figure 17, we illustrate the performance obtained by one CPU core (solid black line) and the parallel multithreaded implementation using OpenMP (solid blue line). The comparison of total wall-clock time for various numbers of loop iterations shows that running 16 CPU cores in parallel is very attractive and it could achieve a factor of ~4.17× speed-up when the number of loop iterations is larger than 28,672. We could not simply gain a factor of 16× speed-up here mainly due to the fact that the chemistry box model is found to be memory bound (e.g., limited by the bandwidth to the main memory instead of the computational intensity) as described and analyzed in Section 3.1. Using 16 CPU cores could increase the computational capacity but would not resolve the issue of being bandwidth bound. The GPU performance result is depicted by the solid green line in Figure 17. Compared to using 16 CPU cores, using one GPU achieves up to 1.33× speed-up when the number of loop iterations is smaller than 14,336 but both implementation become very competitive with each other when the number of loop iterations further grows. This is mainly due to the data transfer between CPU and GPU. When the number of loop iterations is large, the profiling of the GPU execution shows that 70% of the time is spent on the data transfer between CPU and GPU. Thus, even if the GPU provide 4× ratio of memory bandwidth and is about 4× faster, only ~30% speed-up is observed.

3.4.2 Analysis of the portability design by using OpenACC

In this section we investigate the code portability (e.g., single source (CPU/GPU) solution). For that we decide to take advantage of the OpenACC framework to add "!$acc" directives to make the CPU Fortran codes portable to GPU and can run on GPU. We evaluate the computational performance of OpenACC with PGI compiler in this study. In order to minimize the data transfer, we specify two types of data in the OpenACC data region: the data that only needs to be copied from the host (CPU) and the one that only need to be allocated on the device (GPU). We also specify the number of thread blocks and the number of threads per thread block to help the compiler better parallelize the codes. The result depicted in Figure 17 shows that OpenACC (solid pink line) requires ~16× total wall-clock time compared to the CUDA version for 448 loop iterations. Nevertheless, the difference reduces gradually with the increase of the number of loop iterations and OpenACC variant is able to achieve very similar computational performance as the CUDA variant when the number of loop iterations is 57,344 or lager. For all the cases, OpenACC could not beat CUDA with respect to the computational performance, which is consistent with the previous literature [Hoshino et al., 2013; Li et al., 2016; Memeti et al., 2017]. On the other hand, using OpenACC indeed saves significant amount of time to modify the pure CPU codes and it is easier to switch the codes between CPU and GPU version with the compiler flags.

3.4.3 Development of a hybrid CPU/GPU implementation

Last, we investigate a hybrid CPU/GPU version (OpenMP + CUDA) for the chemistry box model. In the hybrid implementation, one OpenMP thread is assigned to launch the CUDA kernels while the remaining 15 OpenMP threads are still active and will contribute to the computation. The total number of loop iterations is split accordingly between CPUs and GPU based on the power of each hardware. Different number of loop iterations are assigned for CPU and GPU computation in order to balance the workload. Our experiments show that for the loop iterations smaller than 3,584, the hybrid CPU/GPU version (red line in Figure 17) requires up to 1.34× total wall-clock time compared to using one GPU alone. This is due to the fact that the data layout differs for the CPUs and the GPU computation (i.e., CPUs work the best with BIML but GPU works the best with FIML). Therefore, when both are working together, a data translation is required and it slows down the hybrid CPU/GPU version when the number of loop iterations is small. However, when the number of loop iterations further increases, the hybrid CPU/GPU version begins to outperform over either the multithreaded CPU variant or the GPU only variant. It could reach ~1.75× speed-up when the number of loop iterations is 57,344. This is equivalent to ~1.95× speed-up compared to using either 16 CPU cores alone or one GPU alone. Note that the CPU version of chemistry box model in this study has already been optimized for the memory layout and compilers. Therefore, the speed-up of the hybrid CPU/GPU version over the default CPU version of chemistry box model from CAM4-Chem (the strided version shown in Figure 6) is even higher.

3.4.4 Practical scenario

Referring to the practical CAM4-Chem simulation with 1° × 1° horizontal resolution, there are 416 loop iterations (16 columns × 26 levels) inside one chunk. Since each CPU core in a compute node is assigned independent chunks, there are totally 6,656 (416 loop iterations per CPU core × 16 CPU cores) loop iterations involved in the chemistry update when the 16 CPU cores are running simultaneously. For this particular scenario, using either one GPU alone or the hybrid CPU/GPU implementation will provide better computational performance than other CPU only approaches. If the horizontal resolution is refined to 0.5° × 0.5°, the vertical layer is refined to 72 layers, and the computational resource remains the same, the loop iterations per chunk will increase rapidly to 73,728 and using hybrid CPU/GPU version is clearly the best choice. This makes the GPU only or the hybrid CPU/GPU implementation of the chemistry module very promising in the global simulation, especially for the fine grid resolution. In addition, although we use Titan as a testbed, this optimized configuration can be implemented on other supercomputing platforms as well. However, overlapping the computation in the CAM4-Chem requires more work since rescheduling for workload balance leads to modification of the main data structure of CAM4-Chem. This is beyond the scope of this study, but we would like to work on that in the future.

Figure 17 here.

4. Conclusion

The growing complexity of the global chemistry-climate model increases the computational burden, challenging progress in model development for high resolution simulations. The strong computational power and fast memory access of the GPU in modern supercomputer architectures provides an opportunity to accelerate the computation. The global chemistry-climate model is a natural fit for massive data and instruction parallelism. However, programming for a GPU is difficult and error-prone and limited studies have been done to explore its corresponding benefit to the components of global chemistry-climate model. Therefore, in this study, we port the ROS-2 solver in the chemistry module of the global chemistry-climate model (CAM4-Chem) to the GPU and seek potential speed-up compared with the CPU version. The basic analysis of the chemistry box model reveals that it is not bounded by the computational rate, but by the data access from the CPU to the main memory. Both parts can be further accelerated by the GPU.

For the CPU version, different compilers and memory layouts play an important role in the computational time. All the three compilers yield the fastest computation by using block interleaved memory layout (BIML), while the Intel compiler with BIML further outperforms over the other two compilers. Formation of the Jacobian matrix and LU (both factorization and solve) are shown to be the most time-consuming parts during the chemistry update (around 70%) for most configurations. In contrast, the GPU version benefits more from the fully interleaved memory layout (FIML). In addition, the computational time of the GPU version increases slowly with the number of loop iterations, opposite to the quick increase of computational time for the CPU version. Tuning the kernel's block size, it is shown that similar performance can be achieved as long as the block is a multiple of the warp size. But it will be slower if the block is smaller than the warp size since it doesn't fully utilize all the threads in a warp. The multiple kernels version provides better performance for larger number of loop iterations, while the one kernel version runs faster for smaller number of loop iterations. The shared memory version yields better performance only for the 448 loop iterations but runs much slower for the larger number of loop iterations, as it is constrained by the small size of shared memory. Nevertheless, the GPU version can always improve slightly from the usage of constant memory. The best performance of the GPU version is achieved using CUDA streams, which enable the simultaneous execution of independent kernels. The data transfer between CPU and GPU, which is also a critical limitation for the overall performance, is done most efficiently by a contiguous allocation when declaring the arrays on the GPU.

Using the most optimized configurations from those experiments, the GPU version shows up to 11.7× speed-up of the computational time compared to the optimized CPU version for the memory layout and compiler (Intel with BIML, using one CPU core). When the data transfer between CPU and GPU is considered, the speed-up can still be as high as 3.82×. Even compared to using 16 CPU cores, using one GPU is again overall faster, especially when the number of loop iterations is not too large (~1.33× speed-up). In addition, using OpenACC requires fewer modifications of the original CPU codes and provides a single source solution to different heterogeneous architectures. However, there is no computational benefit gained compared to the optimized CUDA version in this study. The best performance is achieved by the implementation of the hybrid CPU/GPU version, which is slightly slower when the number of loop iterations is small but clearly outperforms over other approaches for large number of loop iterations (~6.75× speed-up against one CPU core, ~1.95× speed-up against 16 CPU cores and ~1.75× speed-up against one GPU alone). This is the most promising strategy to be introduced into CAM4-Chem but more efforts are required to reschedule the workload among the CPU cores in a compute node.

Acknowledgment

The diagnostic simulations in this study use the resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy (contract DE-AC05-00OR22725). The CESM project is supported by the National Science Foundation and the Office of Science (BER) of the U.S. Department of Energy. The GPU material is also based on the work supported by the National Science Foundation under Grants No. OAC 1740250. The authors want to thank Dr.

Jean-Francois Lamarque for his support from the previous university subproject of the DOE SciDAC project ‘‘Chemistry in CESM-SE: Evaluation, Performance and Optimization’’ (UCAR subaward Z12-93537 to University of Tennessee - Knoxville). The source code for the model used in this study, the CAM4-Chem, is freely available at http://www.cesm.ucar.edu/models/cesm1.2/. The CPU and GPU codes for the chemistry box model are available from the authors upon request (jsfu@utk.edu).

Reference

Abdelfattah, A., A. Haidar, S. Tomov and Jack Dongarra (2016), Performance, Design, and Autotuning of Batched GEMM for GPUs, In: Kunkel J., Balaji P., Dongarra J. (eds) High Performance Computing, ISC High Performance 2016, Lecture Notes in Computer Science, vol 9697, 21-38, Springer, Cham, doi: 10.1007/978-3-319-41321-1_2.

Abdelfattah, A., A. Haidar, S. Tomov and J. J. Dongarra (2017), Factorization and Inversion of a Million Matrices using GPUs: Challenges and Countermeasures, Procedia Computer Science, 108: 606-615, doi: 10.1016/j.procs.2017.05.250.

Abdelfattah, A., A. Haidar, S. Tomov and J. Dongarra (2018), Analysis and Design Techniques towards High-Performance and Energy-Efficient Dense Linear Solvers on GPU, EEE Transactions on Parallel & Distributed Systems, doi: 10.1109/TPDS.2018.2842785.

Alvanos, M. and T. Christoudias (2017), GPU-accelerated atmospheric chemical kinetics in the ECHAM/MESSy (EMAC) Earth system model (version 2.52), *Geosci. Model Dev.*, 10(10), 3679-3693, doi: 10.5194/gmd-10-3679-2017.

Austin, J., Shindell, D., Beagley, S. R., Brühl, C., Dameris, M., Manzini, E., Nagashima, T., Newman, P., Pawson, S., Pitari, G., Rozanov, E., Schnadt, C. and T. G. Shepherd (2003), Uncertainties and assessments of chemistry-climate models of the stratosphere, Atmos. Chem. Phys., 3(1), 1-27, doi: 10.5194/acp-3-1-2003.

Collins, W. J., Lamarque, J.-F., Schulz, M., Boucher, O., Eyring, V., Hegglin, M. I., Maycock, A., Myhre, G., Prather, M., Shindell, D. and Smith, S. J. (2017), AerChemMIP: quantifying the effects of chemistry and aerosols in CMIP6, Geosci. Model Dev., 10(2), 585-607, doi:10.5194/gmd-10-585-2017.

Dameris, M. and P. Jöckel (2013), Numerical Modeling of Climate-Chemistry Connections: Recent Developments and Future Challenges, Atmosphere, 4(2), 132-156, doi: 10.3390/atmos4020132.

Dong, T., A. Haidar, P. Luszczek, J. A. Harris, S. Tomov and J. Dongarra (2014), LU factorization of small matrices: Accelerating Batched DGETRF of the GPU, *2014 IEEE Intl. Conf. on High Performance Computing and Communications, 2014 IEEE 6th Intl. Symp. on Cyberspace Safety and Security, 2014 IEEE 11th Intl. Conf. on Embedded Software and Syst. (HPCC, CSS, ICESS)*, Paris, France, pp. 157-160, doi: 10.1109/HPCC.2014.30.

Dongarra, J., I. Duff, M. Gates, A. Haidar, S. Hammarling, N. J. Higham, J. Hogg, P. Valero-Lara, S. D. Relton, S. Tomov and M. Zounon (2016), A proposed API for Batched Basic Linear Algebra Subprograms, MIMS EPrint 2016.25, Manchester Institute for Mathematical Sciences, The University of Manchester, UK, http://eprints.ma.man.ac.uk/2464/.

Emmons, L. K., S. Walters, P. G. Hess, J.-F. Lamarque, G. G. Pfister, D. Fillmore, C. Granier, A. Guenther, D. Kinnison, T. Laepple, J. Orlando, X. Tie, G. Tyndall, C. Wiedinmyer, S. L. Baughcum and S. Kloster (2010), Description and evaluation of the Model for Ozone and Related chemical Tracers, version 4 (MOZART-4), *Geosci. Model Dev.*, **3**, 43-67, doi:10.5194/gmd-3-43-2010.

Gates, M., J. Kurzak, P. Luszczek, Y. Pei and J. Dongarra (2017), Autotuning batch Cholesky factorization in CUDA with interleaved layout of matrices, *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, Lake Buena Vista, FL, pp. 1408-1417, doi: 10.1109/IPDPSW.2017.18.

Haidar, A, B. Brock, S. Tomov, M. Guidry, J. J. Billings, D. Shyles and J. Dongarra (2016), Performance Analysis and Acceleration of Explicit Integration for Large Kinetic Networks using Batched GPU Computations, 2016 IEEE High Performance Extreme Computing Conference (HPEC), Waltham, MA, doi: 10.1109/HPEC.2016.7761605.

Haidar, A., A. Abdelfattah, M. Zounon, S. Tomov and J. Dongarra (2018), A Guide for Achieving High Performance with Very Small Matrices on GPU: A Case Study of Batched LU and Cholesky Factorizations, IEEE Transactions on Parallel and Distributed Systems, 29(5), 973-984, doi: 10.1109/TPDS.2017.2783929.

Horowitz, L. W., S. Walters, D. L. Mauzerall, L. K. Emmons, P. J. Rasch, C. Granier, X. X. Tie, J.-F. Lamarque, M. G. Schultz, G. S. Tyndall, J. J. Orlando and G. P. Brasseur (2003), A global simulation of tropospheric ozone and related tracers: Description and evaluation of MOZART, version 2, J. Geophys. Res., 108(D24), 4784-4812, doi:10.1029/2002JD002853.

Hoshino, T., N. Maruyama, S. Matsuoka, and R. Takaki (2013), CUDA vs OpenACC: Performance case studies with kernel benchmarks and a memory-bound CFD application, 13th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid'13), Delft, Netherlands, pp 136-143, doi: 10.1109/CCGrid.2013.12.

Isaksen, I. S. A., Granier, C., Myhre, G., Berntsen, T. K., Dalsøren, S. B., Gauss, M., Klimont, Z., Benestad, R., Bousquet, P., Collins, W., Cox, T., Eyring, V., Fowler, D., Fuzzi, S., Jöckel, P., Laj, P., Lohmann, U., Maione, M., Monks, P., Prevot, A.S.H., Raes, F., Richter, A., Rognerud, B., Schulz, M., Shindell, D., Stevenson, D. S., Storelvmo, T., Wang, W.-C., van Weele, M., Wild, M. and D. Wuebbles (2009), Atmospheric composition change: Climate–Chemistry interactions, *Atmos. Environ.*, 43(33), 5138-5192, doi: 10.1016/j.atmosenv.2009.08.003.

Kinnison, D. E., G. P. Brasseur, S. Walters, R. R. Garcia, D. R. Marsh, F. Sassi, V. L. Harvey, C. E. Randall, L. Emmons, J.-F. Lamarque, P. Hess, J. J. Orlando, X. X. Tie, W. Randel, L. L. Pan, A. Gettelman, C. Granier, T. Diehl, U. Niemeier and A. J. Simmons (2007), Sensitivity of chemical tracers to meteorological parameters in the MOZART-3 chemical transport model, *J. Geophys. Res.*, **112**, D20302, doi: 10.1029/2006JD007879.

Kleinman, L. I., Daum, P. H., Lee, Y. N., Nunnermacker, L. J., Springston, S. R., WeinsteinLloyd, J. and J. Rudolph (2001), Sensitivity of ozone production rate to ozone precursors, *Geophys. Res. Lett.*, 28(15), 2903-2906, doi: 10.1029/2000GL012597.

Korwar, S. K., S. Vadhiyar and R. S. Nanjundiah (2013), GPU-enabled efficient executions of radiation calculations in climate modeling, 20th Annual International Conference on High Performance Computing, Bangalore, 353-361, doi:10.1109/HiPC.2013.6799141.

Lamarque, J.-F., Shindell, D. T., Josse, B., Young, P. J., Cionni, I., Eyring, V., Bergmann, D., Cameron-Smith, P., Collins, W. J., Doherty, R., Dalsoren, S., Faluvegi, G., Folberth, G., Ghan, S. J., Horowitz, L. W., Lee, Y. H., MacKenzie, I. A., Nagashima, T., Naik, V., Plummer, D., Righi, M., Rumbold, S. T., Schulz, M., Skeie, R. B., Stevenson, D. S., Strode, S., Sudo, K., Szopa, S., Voulgarakis, A. and G. Zeng (2013), The Atmospheric Chemistry and Climate Model Intercomparison Project (ACCMIP): overview and description of models, simulations and climate diagnostics, Geosci. Model Dev., 6(1), 179-206, doi: 10.5194/gmd-6-179-2013.

Li, X., P.-C. Shih, J. Overbey, C. Seals and A. Lim (2016), Comparing programmer productivity in OpenACC and CUDA: an empirical investigation, International Journal of Computer Science, Engineering and Applications (IJCSEA), 6(5): 1-15, doi: 10.5121/ijcsea.2016.6501.

Lin, S. (2004), A “Vertically Lagrangian” Finite-Volume Dynamical Core for Global Models, *Mon. Wea. Rev.*, 132, 2293-2307, doi: 10.1175/1520-0493(2004)132<2293:AVLFDC>2.0.CO;2.

Linford, J. C., J. Michalakes, M. Vachharajani and A. Sandu (2009), Multi-core acceleration of chemical kinetics for simulation and prediction, *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, SC '09*, Association for Computing Machinery, New York, NY, **7**, 1-11, doi: 10.1145/1654059.1654067.

Mantell, R. G., C. E. Pitt and D. J. Wales (2016), GPU-Accelerated Exploration of Biomolecular Energy Landscapes, *J. Chem. Theory Comput.*, 12(12), 6182-6191, doi: 10.1021/acs.jctc.6b00934.

Memeti, S., L. Li, S. Pllana, J. Kolodziej and C. Kessler (2017), Benchmarking OpenCL, OpenACC, OpenMP, and CUDA: programming productivity, performance, and energy consumption, Proceedings of the 2017 workshop on adaptive resource management and scheduling for cloud computing, ACM, New York, NY, USA, ARMS-CC '17, pp 1-6, doi: 10.1145/3110355.3110356.

Michalakes, J. and M. Vachharajani (2008), GPU Acceleration of Numerical Weather Prediction, Parallel Processing Letters, 18(4), 531-548, doi: 10.1142/S0129626408003557.

Mirin, A. A. and P. H. Worley (2011), Improving the performance scalability of the community atmosphere model, *The International Journal of High Performance Computing Applications*, 26(1), 17-30, doi: 10.1177/1094342011412630.

Myhre, G., D. Shindell, F.-M. Bréon, W. Collins, J. Fuglestvedt, J. Huang, D. Koch, J.-F. Lamarque, D. Lee, B. Mendoza, T. Nakajima, A. Robock, G. Stephens, T. Takemura and H. Zhang, 2013: Anthropogenic and Natural Radiative Forcing. In: Climate Change 2013: The Physical Science Basis. Contribution of Working Group I to the Fifth Assessment Report of the Intergovernmental Panel on Climate Change [Stocker, T.F., D. Qin, G.-K. Plattner, M. Tignor, S.K. Allen, J. Boschung, A. Nauels, Y. Xia, V. Bex and P.M. Midgley (eds.)]. Cambridge University Press, Cambridge, United Kingdom and New York, NY, USA.

Schraner, M., Rozanov, E., Schnadt Poberaj, C., Kenzelmann, P., Fischer, A. M., Zubov, V., Luo, B. P., Hoyle, C. R., Egorova, T., Fueglistaler, S., Brönnimann, S., Schmutz, W. and T. Peter (2008), Technical Note: Chemistry-climate model SOCOL: version 2.0 with improved transport and chemistry/microphysics schemes, Atmos. Chem. Phys., 8(19), 5957-5974, doi: 10.5194/acp-8-5957-2008.

Su, H., Cheng, Y. F., Oswald, R., Behrendt, T., Trebs, I., Meixner, F. X., Andreae, M. O., Cheng, P., Zhang, Y. H. and U. Pöschl (2011), Soil Nitrite as a Source of Atmospheric HONO and OH Radicals, Science, 333(6049), 1616-1618, doi: 10.1126/science.1207687.

Sun, J., J. S. Fu, J. Drake, J.-F. Lamarque, S. Tilmes, and F. Vitt (2017), Improvement of the prediction of surface ozone concentration over conterminous U.S. by a computationally efficient second-order Rosenbrock solver in CAM4-Chem, J. Adv. Model. Earth Syst., 9, doi:10.1002/2016MS000863.

Tian, W. S. and M. P. Chipperfield (2006), A new coupled chemistry–climate model for the stratosphere: The importance of coupling for future O_3-climate predictions, *Quarterly Journal of the Royal Meteorological Society,* 131(605), 281-303, doi: 10.1256/qj.04.05.

Tilmes, S., Lamarque, J.-F., Emmons, L. K., Kinnison, D. E., Marsh, D., Garcia, R. R., Smith, A. K., Neely, R. R., Conley, A., Vitt, F., Val Martin, M., Tanimoto, H., Simpson, I., Blake, D. R. and N. Blake (2016), Representation of the Community Earth System Model (CESM1) CAM4-chem within the Chemistry-Climate Model Initiative (CCMI), Geosci. Model Dev., 9(5), 1853-1890, doi: 10.5194/gmd-9-1853-2016.

Verwer, J. G., E. J. Spee, J. G. Blom and W. Hundsdorfer (1999), A second-order rosenbrock method applied to photochemical dispersion problems, *SIAM J. Sci. Comput.*, **20**(4), 1456–1480, doi:10.1137/S1064827597326651.

Worley, P. H. and J. B. Drake (2005), Performance Portability in the Physical Parameterizations of the Community Atmospheric Model, *The International Journal of High Performance Computing Applications,* 19(3), 187-201, doi: 10.1177/1094342005056095.

Xu, S., Huang, X., Oey, L.-Y., Xu, F., Fu, H., Zhang, Y. and Yang, G. (2015), POM.gpu-v1.0: a GPU-based Princeton Ocean Model, Geosci. Model Dev., 8(9), 2815-2827, doi:10.5194/gmd-8-2815-2015.1

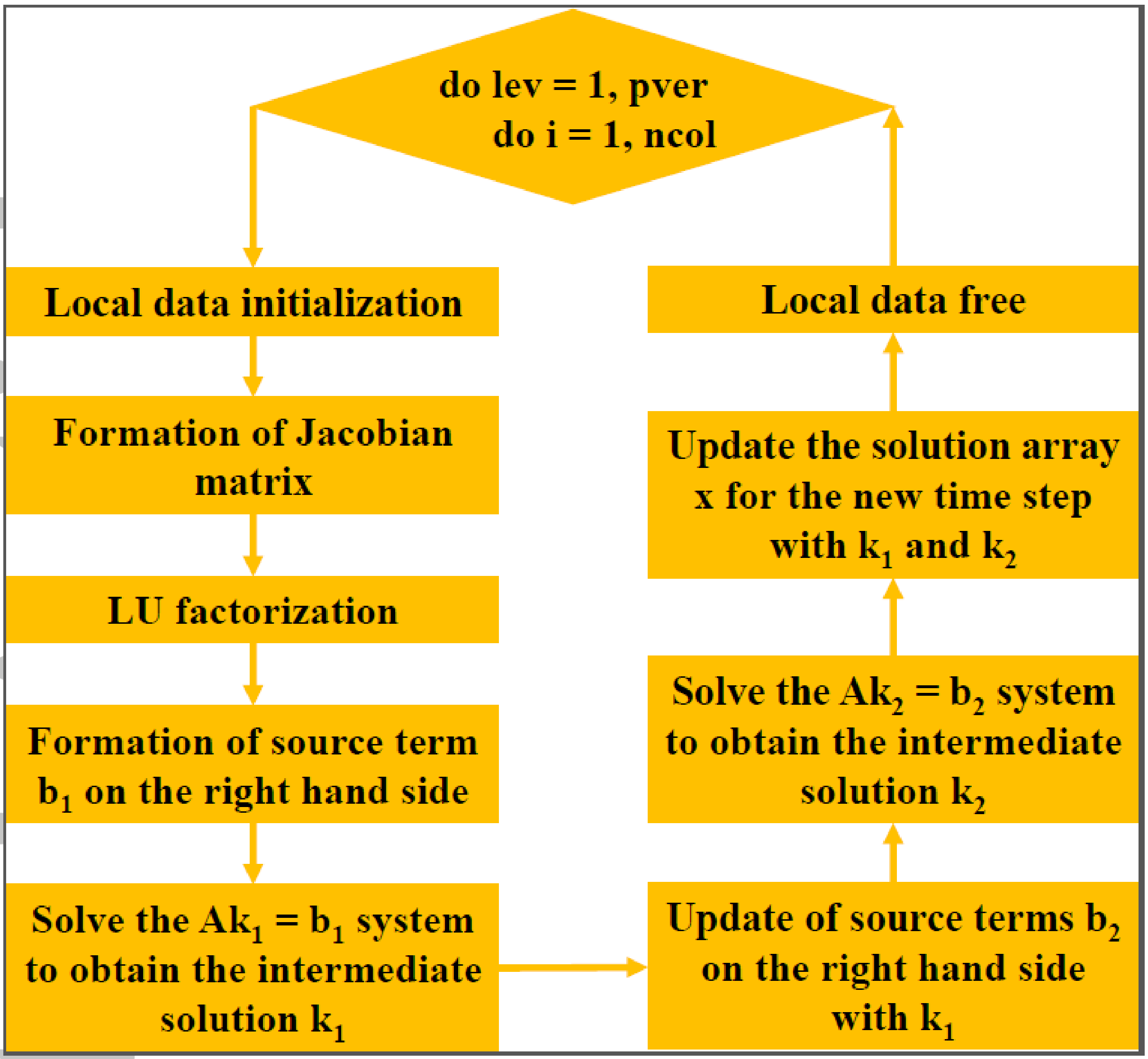
do lev = 1, pver
do i = 1, ncol
Local data initialization
Formation of Jacobian matrix
LU factorization
Formation of source term b_1 on the right hand side
Solve the $Ak_1 = b_1$ system to obtain the intermediate solution k_1
Update of source terms b_2 on the right hand side with k_1
Solve the $Ak_2 = b_2$ system to obtain the intermediate solution k_2
Update the solution array x for the new time step with k_1 and k_2
Local data free

Figure 1. The flow chart of chemistry update. The diamond refers to the number of loop iterations and the rectangular shape refers to the individual functions involved in the chemistry update.

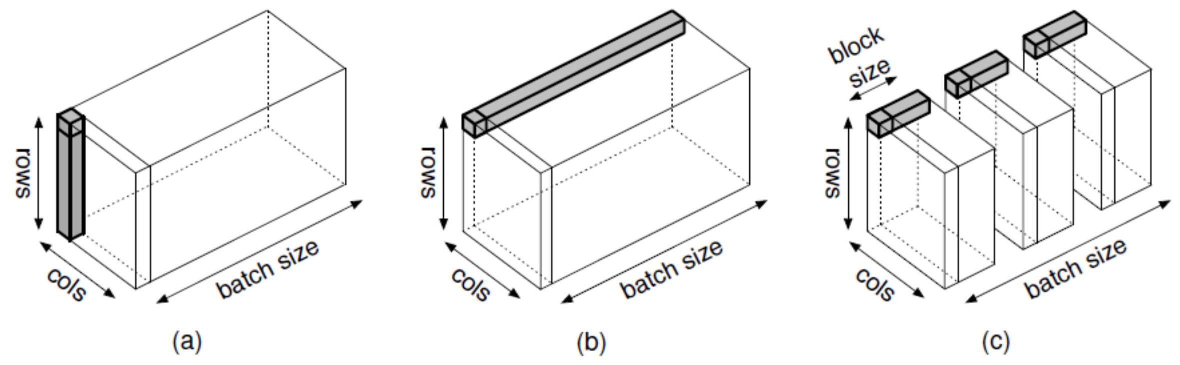

Figure 2. The storage of matrices in Fortran using the (a) strided, (b) fully interleaved and (c) block interleaved memory layout.

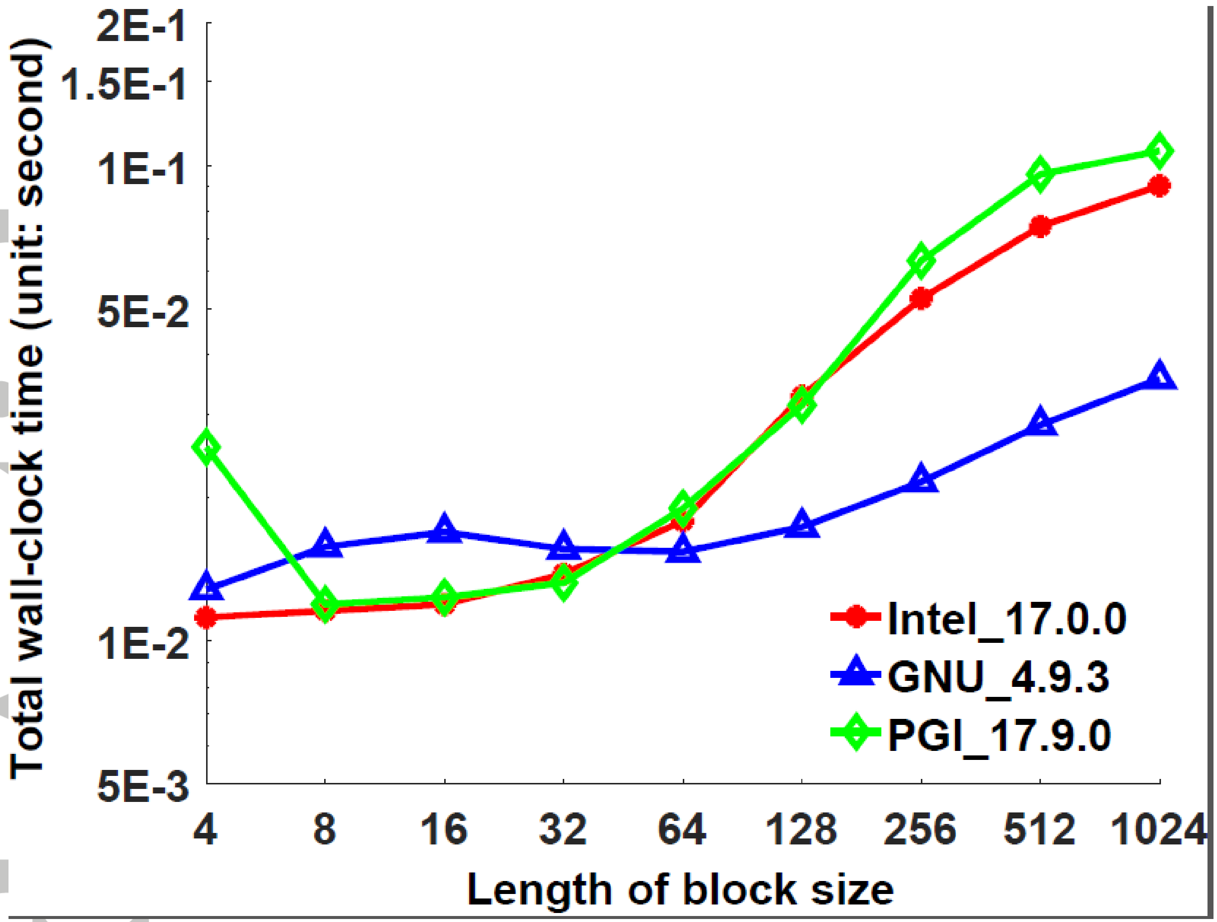

Figure 3. Impact of block size (X-axis) on the total wall-clock time (Y-axis, log scale, unit: second) for the block interleaved memory layout (number of loop iterations = 1,024). Different colors refer to different compilers (red: Intel, blue: GNU, green: PGI).

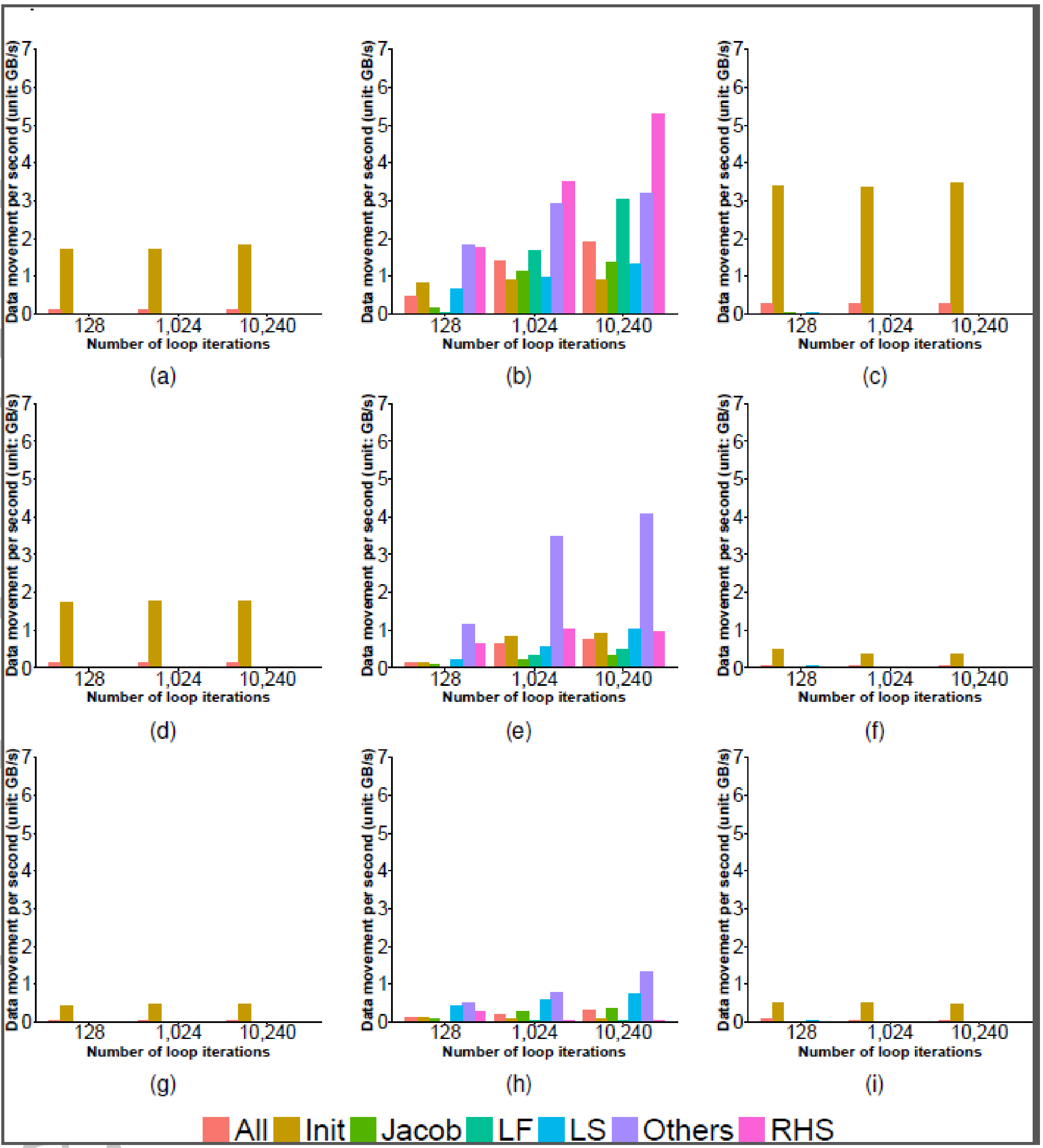

Figure 4. Data transfer per second (unit: GB/s) for (a-c) GNU, (d-f) Intel and (g-i) PGI compiler using different memory layouts (left panel: strided memory layout; middle panel: fully interleaved memory layout; right panel: block interleaved memory layout). Different colors refer to the different functions in the chemistry box model (All: the whole chemistry box model, Init: data initialization, Jacob: formation of the Jacobian matrix, LF: LU factorization, LS: LU solve, Others: update of intermediate and final solutions, RHS: formation of the right hand side source term).

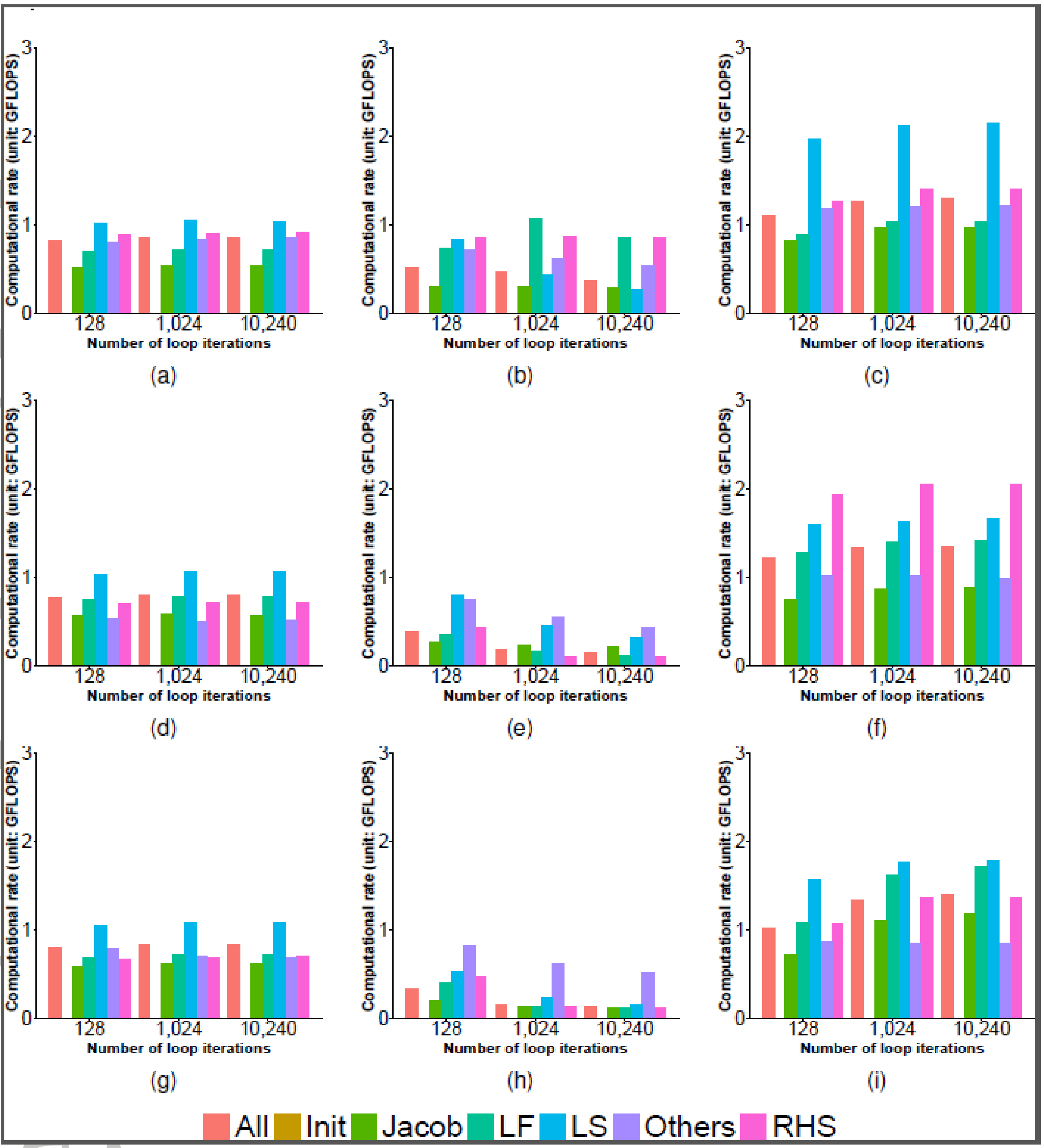

Figure 5. The same as Figure 4 but for the computational rate (unit: GFLOPS).

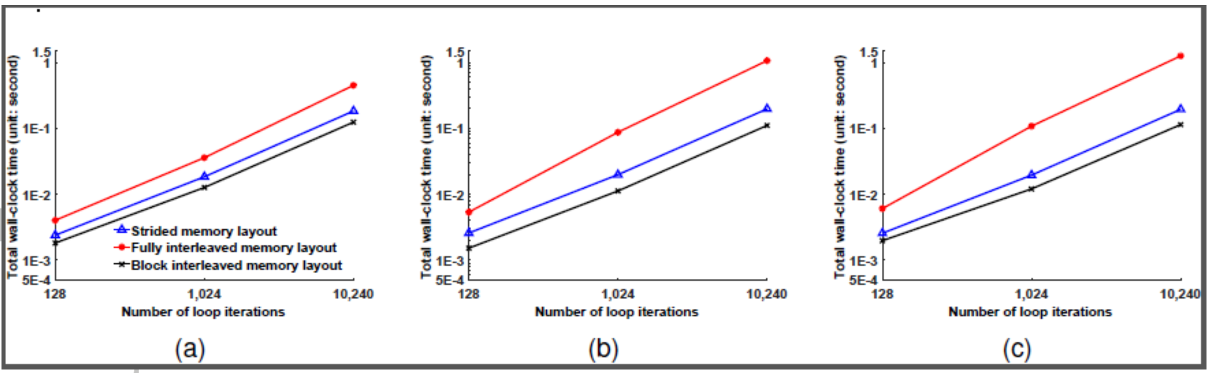

Figure 6. The total wall-clock time (Y-axis, log scale, unit: second) of the chemistry box model with different number of loop iterations (X-axis, log scale) for (a) GNU, (b) Intel, and (c) PGI compiler. Different colors refer to different memory layouts (blue: strided, red: fully interleaved, black: block interleaved).

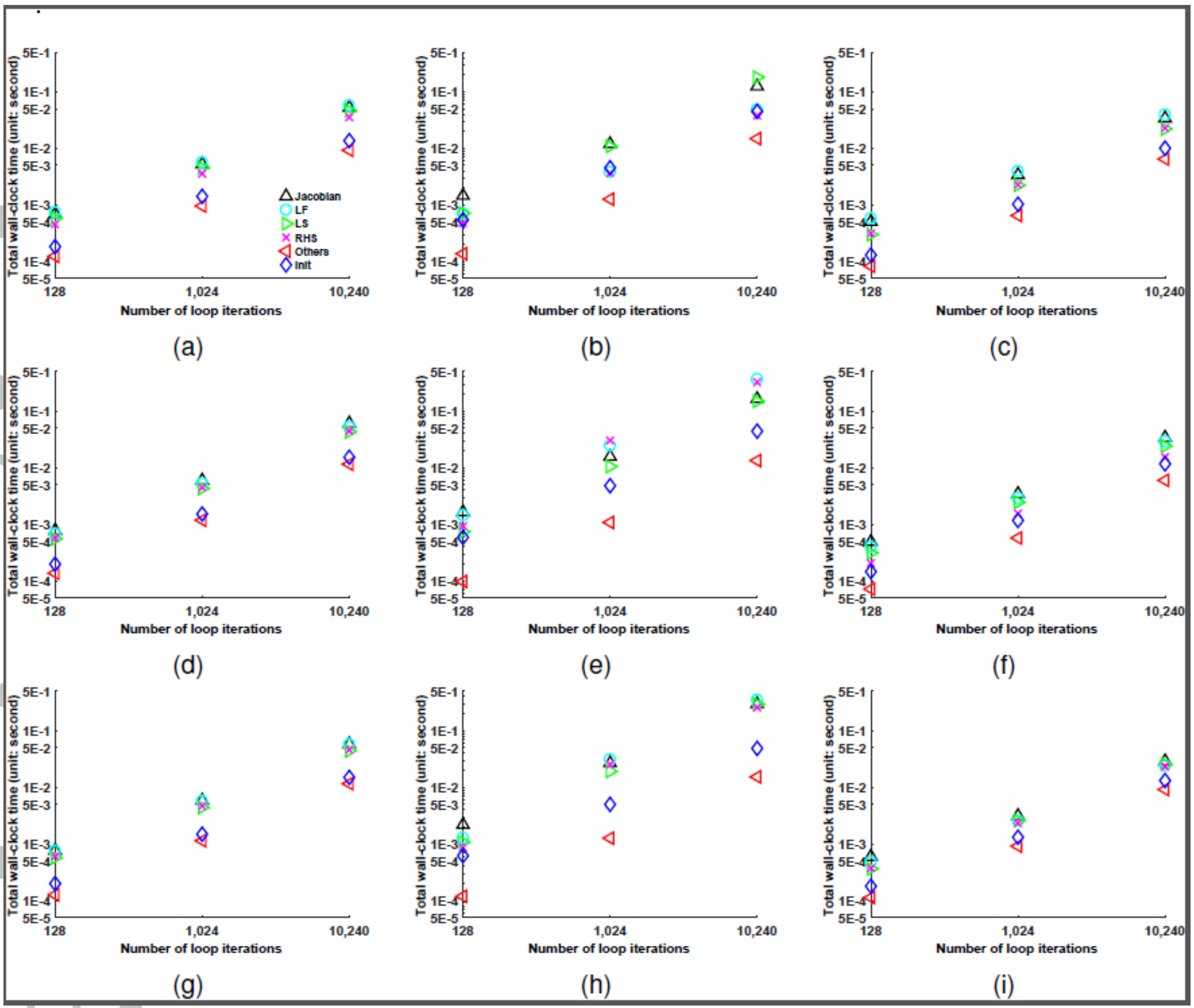

Figure 7. The total wall-clock time (Y-axis, log scale, unit: second) of each function (black: formation of the Jacobian matrix, pink: LU factorization, green: LU solve, pink: formation of the right hand side source term, red: update of intermediate and final solutions, blue: cudaFree) in the GPU version of chemistry box model with different number of loop iterations (X-axis) for different compilers ((a-c): GNU, (d-f): Intel and (g-i): PGI) and memory layouts (left-panel: strided, middle-panel: fully interleaved, right-panel: block interleaved).

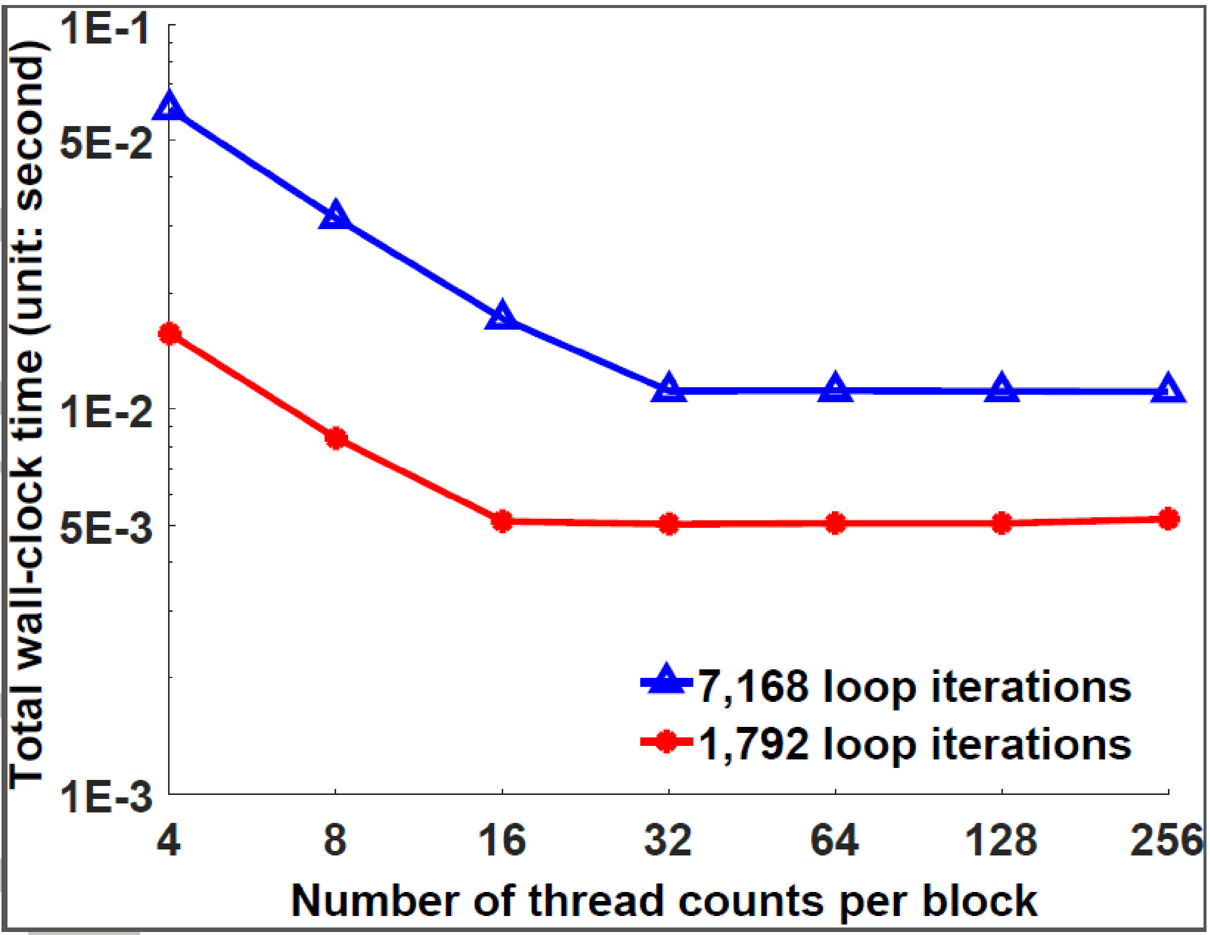

Figure 8. The total wall-clock time (Y-axis, log scale, unit: second) of the GPU version of chemistry box model with different number of threads per block (X-axis), using the block interleaved memory layout (blue: 7,168 loop iterations; red: 1,792 loop iterations).

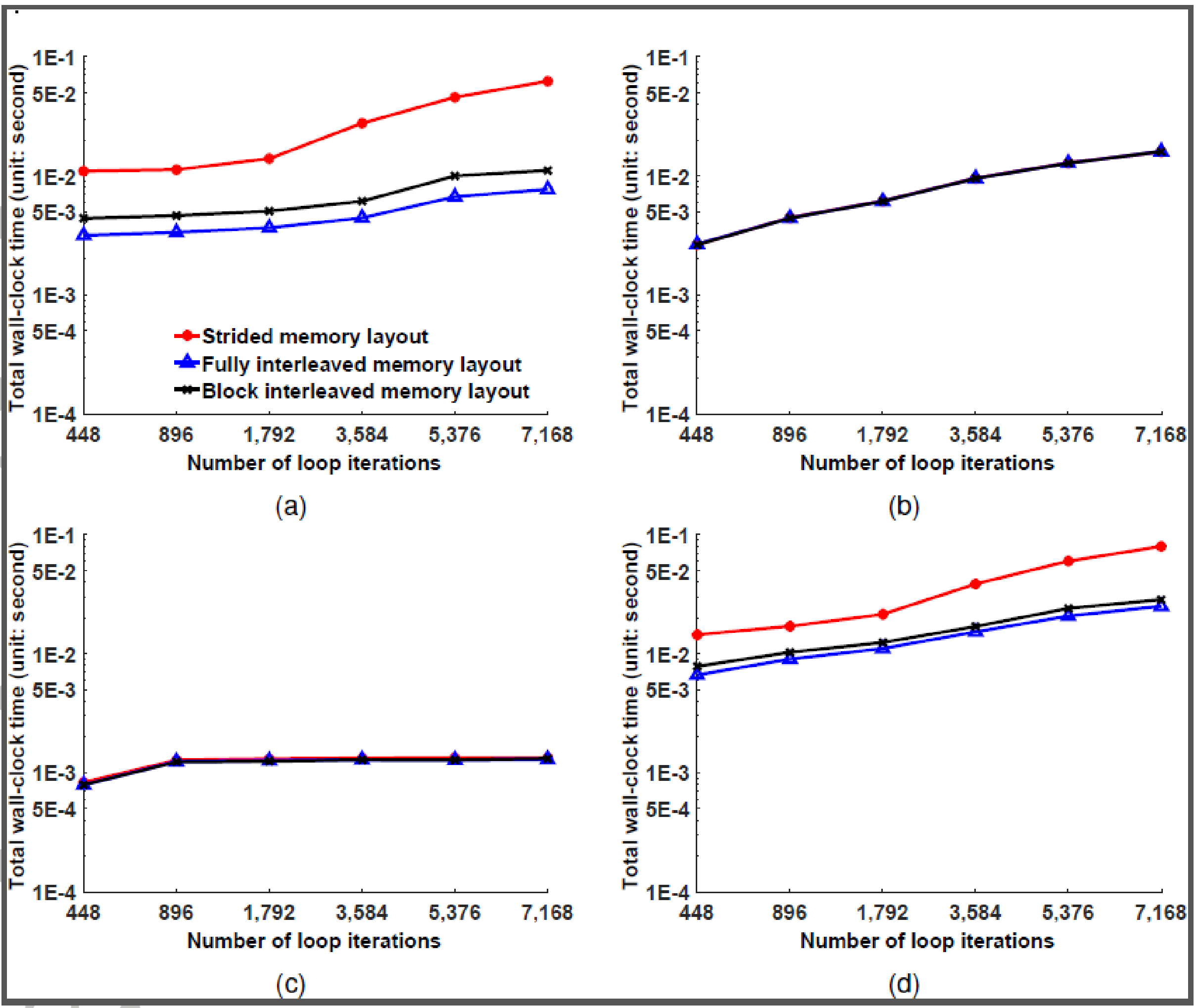

Figure 9. The total wall-clock time (Y-axis, log scale, unit: second) of the GPU version of chemistry box model with different number of loop iterations (X-axis) for the (a) computation alone, (b) data transfer between CPU and GPU, (c) data deallocation on the GPU and (d) summed time from the three above (red: strided memory layout, blue: fully interleaved memory layout, black: block interleaved memory layout).

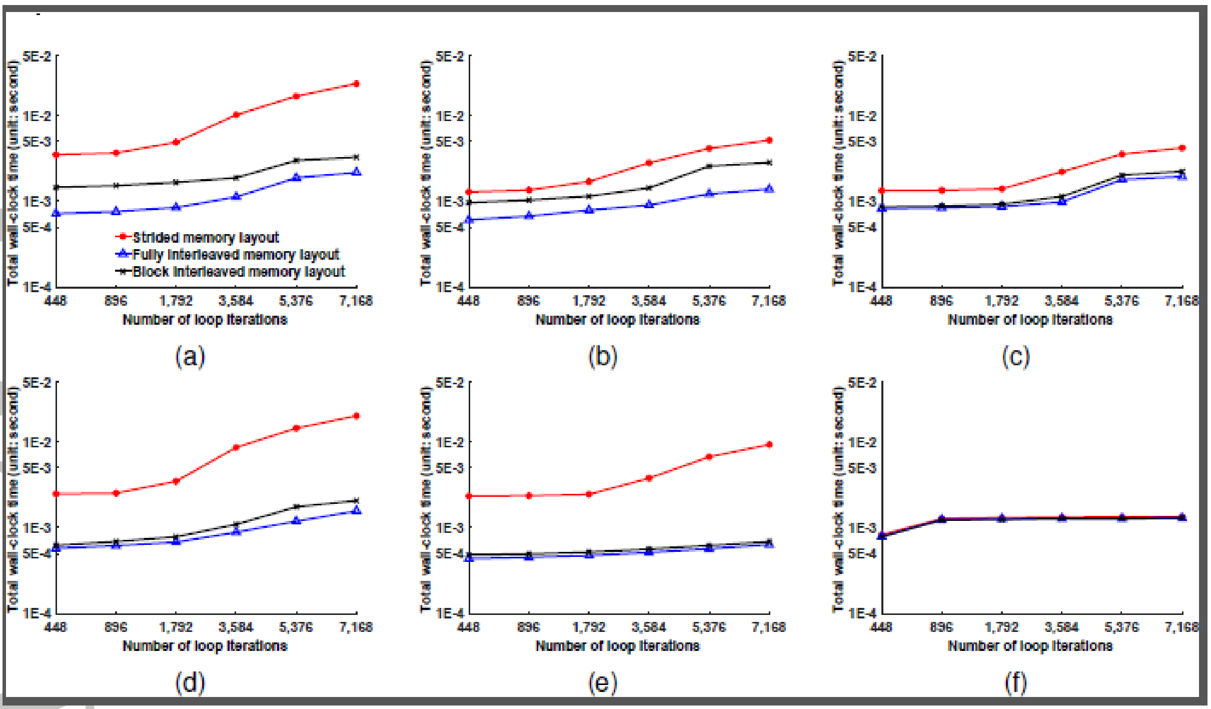

Figure 10. The total wall-clock time (Y-axis, log scale, unit: second) of each function (a: formation of the Jacobian matrix, b: LU factorization, c: LU solve, d: formation of the right hand side source term, e: update of intermediate and final solutions, f: cudaFree) in the GPU version of chemistry box model (red: strided memory layout, blue: fully interleaved memory layout, black: block interleaved memory layout).

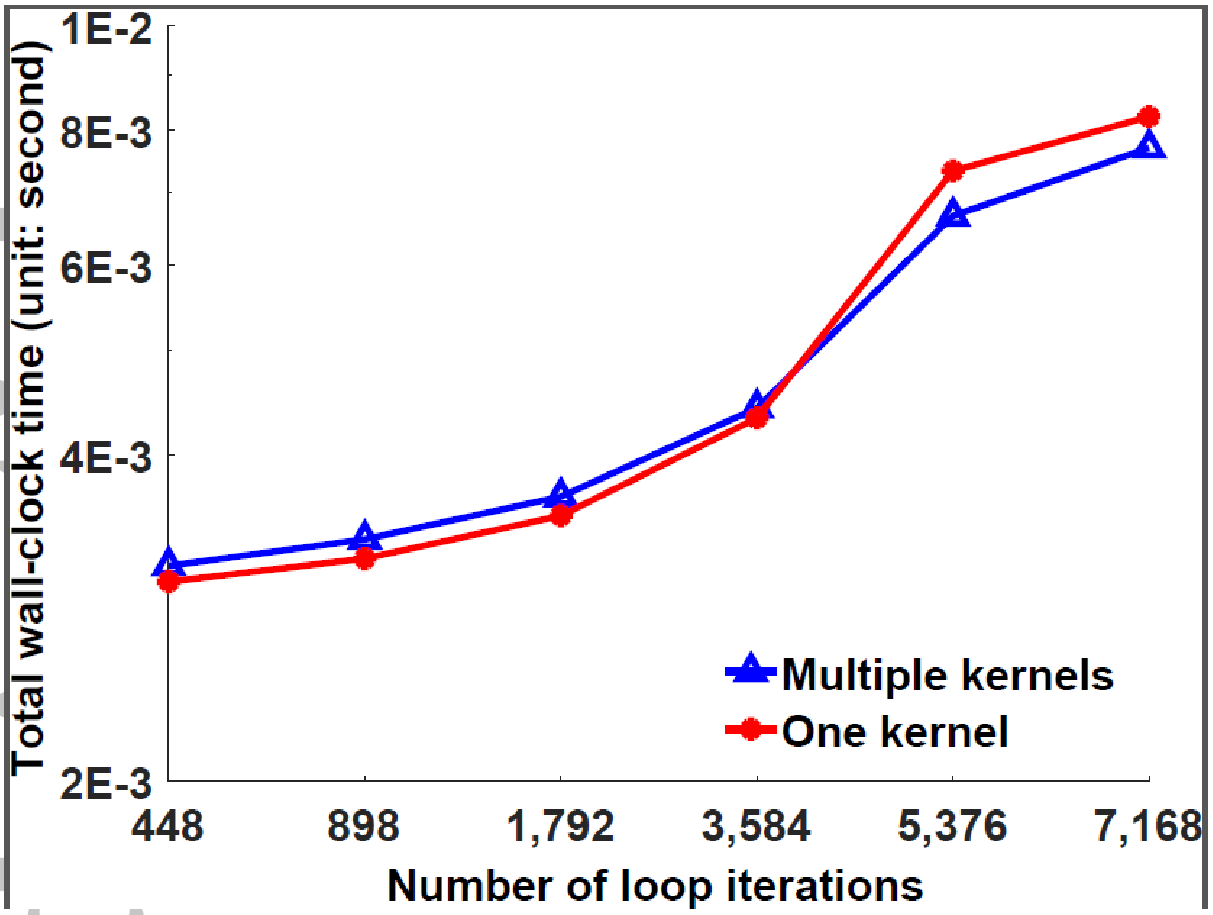

Figure 11. The computational time (Y-axis, log scale, unit: second) of the GPU version of chemistry box model with different number of loop iterations (X-axis) for the multiple kernels (blue) and one kernel (red).

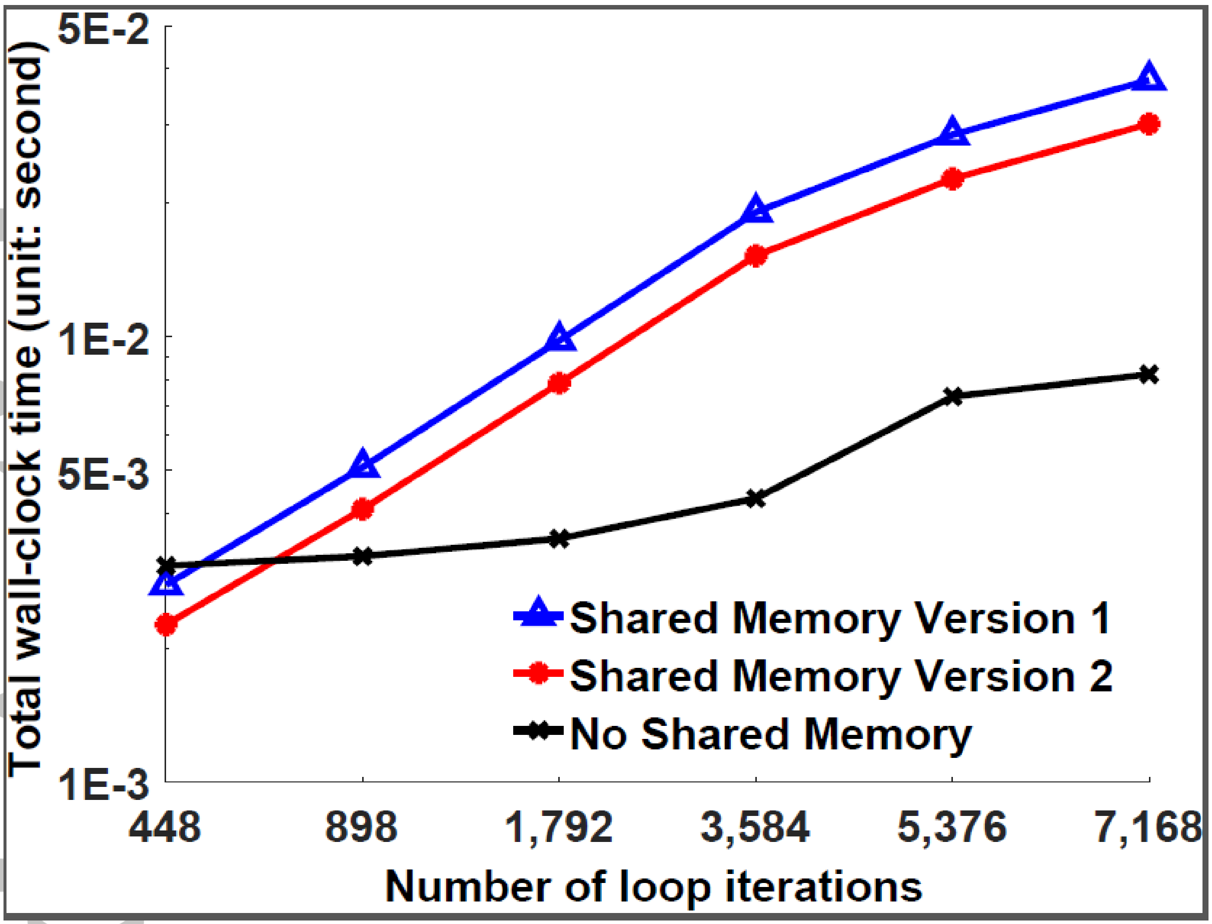

Figure 12. The computational time (Y-axis, log scale, unit: second) of the GPU version of chemistry box model with different number of loop iterations (X-axis) using the shared memory for the solution vector (blue), the shared memory for the intermediate solution vectors (red) and no shared memory (black).

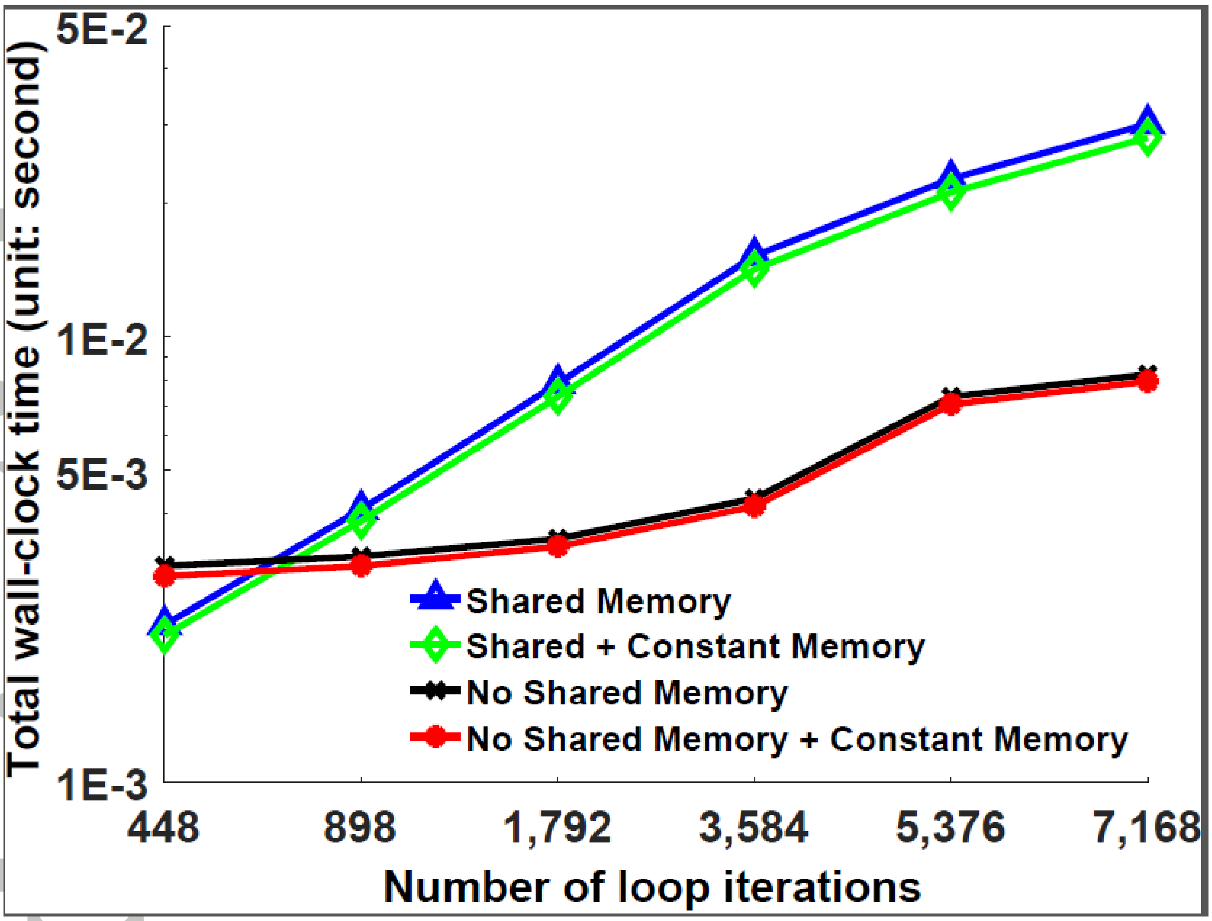

Figure 13. The computational time (Y-axis, log scale, unit: second) of the GPU version of chemistry box model with different number of loop iterations (X-axis) using the shared memory for the intermediate solution vectors (blue), shared memory plus constant memory (green), no shared memory (black) and no shared memory but constant memory (red).

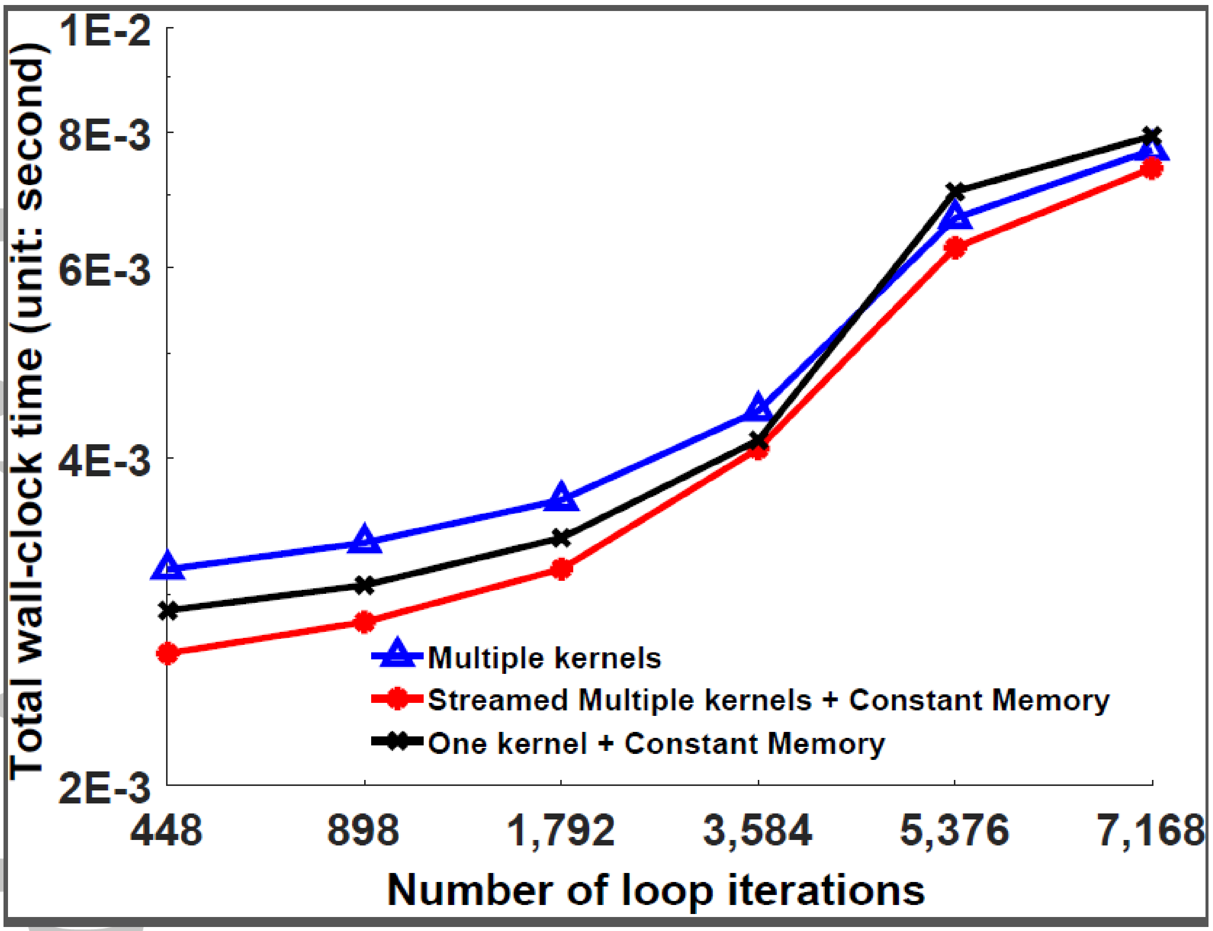

Figure 14. The computational time (Y-axis, log-scale, unit: second) of the GPU version of chemistry box model with different number of loop iterations (X-axis) for the multiple kernels (blue), one kernel with constant memory (black) and streamed multiple kernels with constant memory (red).

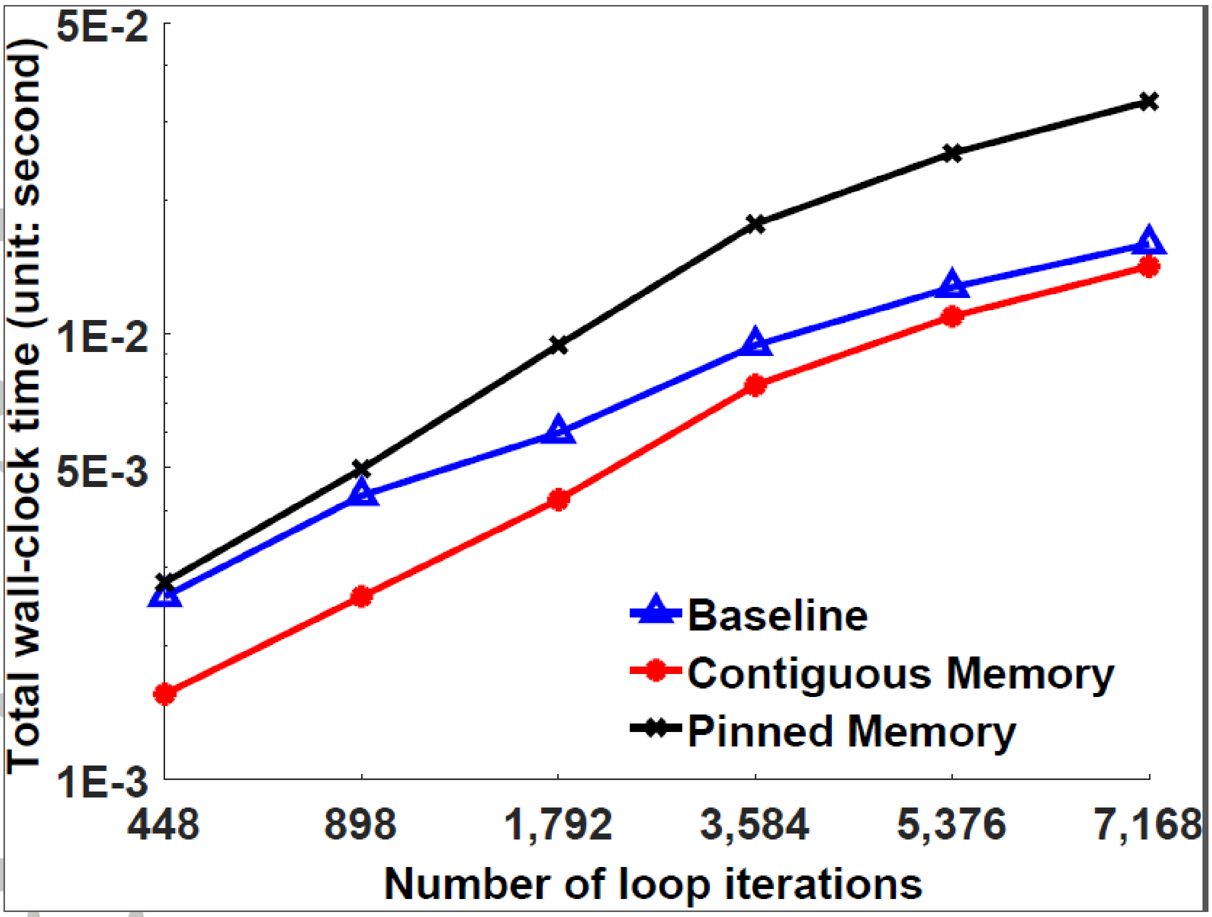

Figure 15. The data transfer time (Y-axis, log scale, unit: second) of the GPU version of chemistry box model with different number of loop iterations (X-axis) using the separate memory allocation (blue), the contiguous memory allocation (red) and the pinned memory (black).

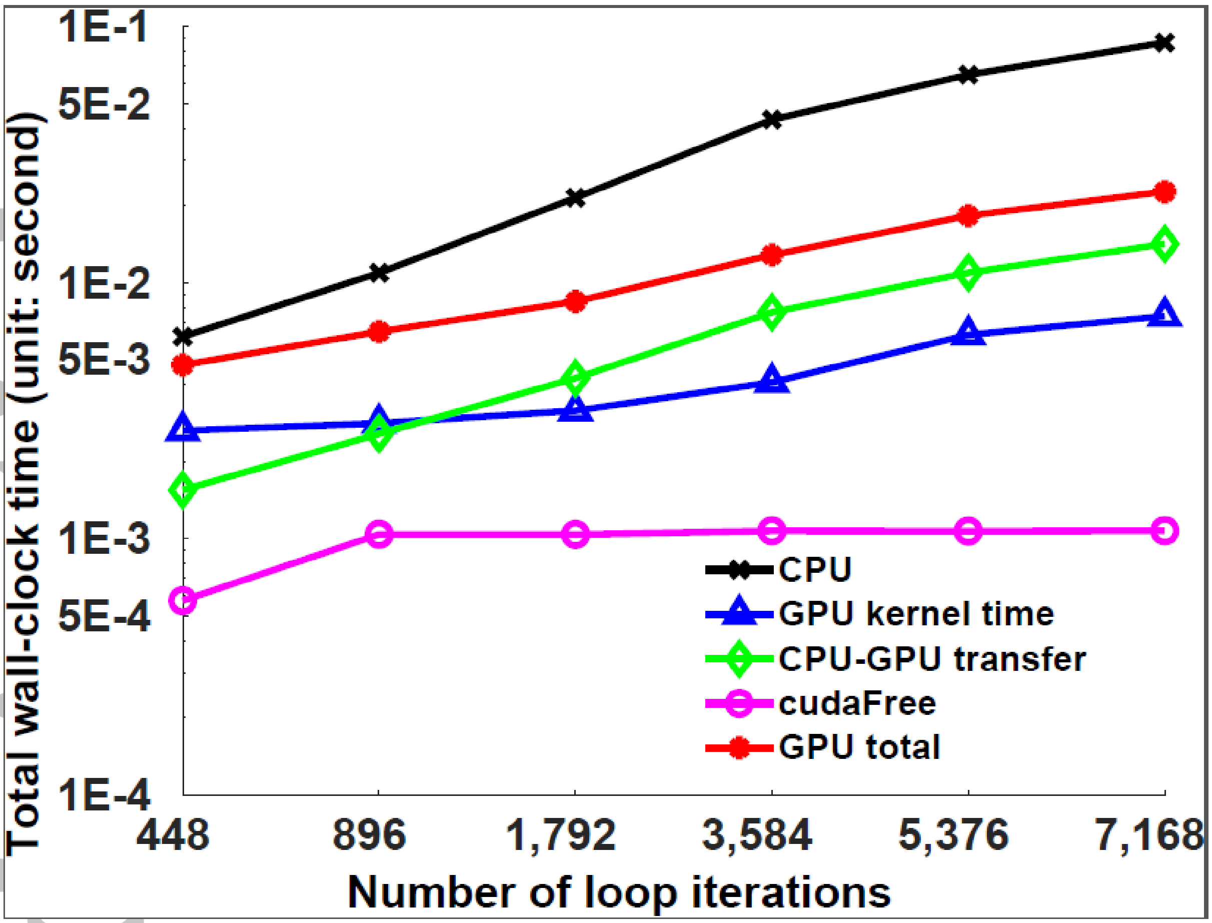

Figure 16. The total wall-clock time (Y-axis, log scale, unit: second) of the chemistry box model with different number of loop iterations (X-axis). Different colors refer to different metrics of time (black: total wall-clock time of CPU version, red: total wall-clock time of GPU version, blue: time of computation alone, green: time of data transfer between CPU and GPU, pink: time of data deallocation on the GPU).

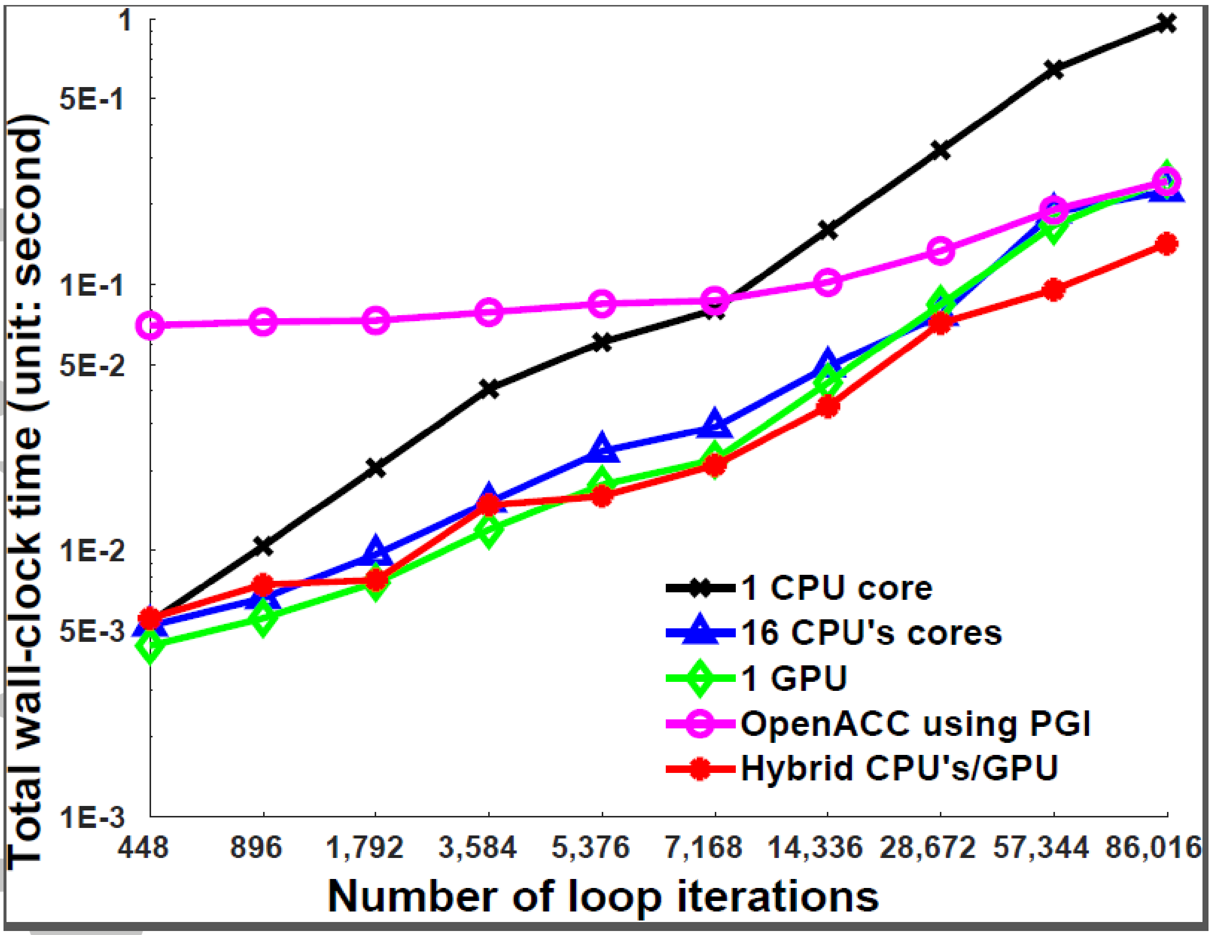

Figure 17. The total wall-clock time (Y-axis, log scale, unit: second) of the chemistry box model with different number of loop iterations (X-axis). Different colors refer to different approaches (black: one CPU core, blue: 16 CPU cores, green: one GPU alone, pink: OpenACC, red: hybrid CPU/GPU).

Table 1. The floating point operations (FLOP) and data copy (unit: KB) per loop iteration for the main functions of the chemistry box model.

Function	FLOP	Copy in	Copy out
Formation of the Jacobian matrix	3,070	25.64	13.18
LU factorization	4,075	6.59	6.59
LU solve	3,116	14.7	1.52
Formation of the right hand side source term	4,086	6.69	3.04