Performance Comparison between GPU Based Code and CPU Based Code in Simplified CFD Problems

By Honggang Wang
Honggangwang1979@gmail.com

Abstract:

Based on the fact that the problem size of a typical CFD problem depends mainly on the mesh size, the number of time steps/ number of iterations, and the work load per mesh cell per time step/iteration, a generalized and simplified pseudo CFD application is proposed. Six versions of GPU based parallel code (NormalCuda, CudaMg, CudaMgAuto, CudaDevice, CudaDeviceAuto, and OpenAcc) and one CPU based parallel code (CpuOMP) are adopted to analyze the performance of different CPU/GPU combinations compared to the pure CPU serial code (CpuFortran). The sensitivities of the GPU speedup performance to the three parameters characterizing a typical CFD problem are analyzed, with the results showing that: 1)GPU generally works better for larger mesh size, time steps/number of iteration, and work load; 2)GPU based parallel methods significantly outperform the CPU based parallel method of CpuOmp; 3) Of the 6 GPU based parallel methods, usually the top three are CudaDeviceAuto, CudaMgAuto, and OpenAcc. 4) The method presented in this article can be used to guide the selection of parallel methods and CPU/GPU pairs.

Keywords: CPU, GPU, CUDA, OpenMP, CFD, Performance

Abbreviations: CPU, Central Processing Unit; GPU, Graphic Processing Unit; CUDA, Compute Unified Device Architecture; CFD, Computational fluid dynamics; HPC, High Performance Computing; SPP, Single-Presicion Performance; TFLOPS, Trillion Floating-Point Operations per Second

1. Introduction

For large scale CFD problems, while CPU based paralleling methods like MPI and OpenMP are still prevail today in HPC and/or cloud computation[1], GPU based paralleling methods are drawing more attentions due to their advantages over CPU based methods: increased performance, reduced hardware costs, and smaller power budget [2]. For example, it is reported that the computing performance of a NVIDA A100 GPU equals to that of 272 Intel Xeon Gold 6242 cores or 400 Intel Xeon Platinum 8380 cores, and the hardware cost of a GPU server can be 7 times less than that of an equivalent CPU cluster, with a bonus of 4 times lower power consumption [2]. A detailed analysis of the relative performance and cost of GPU and CPU architecture for a full aircraft RANS simulation using the CFD code zCFD provides extra evidence to support the continued focus on CFD code development to support GPUs [3]. When turning to 2022, two large commercial CFD vendors, Ansys and Siemens, launched GPU accelerated support for their flagship CFD tools, suggesting the arriving of a new age of CFD [4].

The performance improvement of using GPU varies due to the diversity in selecting the CFD applications, serial/parallel methods, and CPU/GPU pairs. For example, an open source CFD tool, OpenFoam, has been GPU accelerated by SimFlow, showing a speedup of about 2 times when comparing one Tesla K20X GPU to 8 Intel E5-2679 cores for a pisoForm/LES problem with 4M mesh cells [5]. A wildfire simulation tool, vFireLib, has been ported to GPU, presenting an increased speedup times proportional to the mesh size when comparing a GTX-970 GPU to a Intel i7-3770K CPU [6].

Other researchers comparing one GPU to one CPU have reported a speedup times of 20 to 300 for their respective CFD codes and paralleling methods[7,8,9,10].

As to the performance comparison between different paralleling methods, Mikhail Khalilov, etc. compared the performance of CUDA, OpenMP and OpenACC on state-of-the-art Nvidia Tesla V100 GPU in various typical scenarios that arise in scientific programming, showing that CUDA has the highest performance, followed by OpenACC and OpenMP, and the performance gaps among these three methods increase with the growing of problem size [11] . However, it is not clear how the variables memory are allocated and used in the CUDA code, which has considerable effects on the GPU performance. Matthew Norman, etc., have conducted a case study of CUDA FORTRAN and OpenACC, reporting a speedup of 1.56 times of CUDA code over the OpenACC code [12]. T. Hoshino, etc., have reported that OpenACC in general is approximately 50% slower than CUDA, but for some applications it can reach up to 98% (of the CUDA performance) with careful manual optimizations [13]. B. Cloutier, etc., studied the performance differences among CUDA FORTRAN, CUDA C and OpenACC for the incompressible Navier-Stokes equation with a variety of grid sizes, they found that maximum speedup times of 30, 23, and 25, respectively, could be achieved at the largest grid size of 4096 x 4096 [10]. Antonio J. Rueda, etc., stated that in their hydrological applications OpenACC could be 3.5 times slower in average than CUDA codes (although still 2 to 6 times faster than a CPU implementation) [14].

The studies shown above displayed a variety of GPU performance speedup times over CPU, due to differences in the chosen CFD applications, paralleling methods, and CPU/GPU pairs. With the intent of studying the performance enhancement of GPU based paralleling methods over CPU based serial/paralleling methods in the general area of CFD problems, in this article we propose a simplified pseudo CFD application with its problem size characterized by mesh cell number (or mesh size), time steps or number of iterations, and work load per time step/iteration per cell. This same simplified CFD problem is tested over 4 various CPU/GPU pairs with 6 GPU based paralleling methods, one CPU based paralleling method, and one CPU based serial method, removing the uncertainties associated with various adopted CFD applications to only show how well a GPU and a paralleling method are. The programming language used is FORTRAN.

2. Design of a general, simplified CFD application

Although there are a large range of CFD applications with a variety of numerical methods, in general the problem size of a CFD application depends on three factors: (1) mesh size (or mesh cell number), which defines the physical domain and the simulation granularity. (2) time steps for unsteady CFD problems or number of iteration for steady CFD problems, or both (for unsteady problems where iteration is needed to achieve convergence for each time step). For the sake of simplicity, we will only use time step in the following discussion. (3) work load per mesh cell per time step, which defines the number of operations needed to timely update the values or statuses of local variables like density, concentrations, velocities, temperature, pressure, etc.. These operations are intrinsically serial and cannot be further parallelized. For example, the following figure shows CloverLeaf's ideal gas kernel and flux kernel[15,16] which can be substituted in our pseudo CFD application with work loads of 12 and 8 (one operator counts as one operation) :

```
!CloverLeaf's Idea Gas Kernel
!....
Do k=y_min, y_max
    Do j=x_min, x_max
        v = 1.0_8/density(j,k)
        pressure(j,k) = (1.4_8-1.0_8)*density(j,k)*energy(j,k)
        pressurebyenergy = (1.4_8-1.0_8)*density(j,k)
        pressurebyvolume = -density(j,k)*pressure(j,k)
        sound_speed_squared = v*v* &
                (pressure(j,k)*pressurebyenergy-presurebyvolume)
        soundspeed(j,k) = SQRT(sound_speed_squared)
    EndDo
EndDo

!CloverLeaf's Flux calculation Kernel
!....
Do k=y_min, y_max
    Do j=x_min, x_max+1
        vol_flux_x(j,k) = 0.25_8*dt*xarea(j,k)* &
                (xvel0(j,k)+xvel0(j,k+1)+xvel1(j,k)+xvel1(j,k+1))
    EndDo
EndDo
!....
```

Figure 1 CloverLeaf's Ideal Gas Kernel and Flux Calculation Kernel.

The serial CPU code of the simplified CFD application code is shown below:

```
!loop over time steps
DO t = 1, T_step

        !Within each time step, loop over each cell
        DO i=1,N_i
                DO j=1,N_j
                        DO k=1,N_k

                                !For each cell, loop over work loads
                                do m=1, M_load
                                        c(i,j,k) = a(i,j,k) + b(i,j,k)
                                end do

                        End DO
                End DO
        End DO

End DO
```

Figure 2 CpuFortran - A CPU based serial fortran code shows the simplified CFD application. T_step is the time steps of the simulation, N_i/N_j/N_k are the three dimensions of the mesh, and M_load is the work load per each cell per each time step.

This compiling command used is:

nvfortran -g -m64 -O0 -Wall -Werror -traceback -Mrecursive -o CpuFortran CpuFortran.f90

3. Seven paralleling methods

3. 1 CPU based OpenOMP paralleling method (CpuOMP)

OpenMP is an implementation of multithreading that supports multi-platform shared-memory programming in C, C++, and FORTRAN [17]. The following figure shows the OpenMP realization of our simplified CFD application:

```
DO t = 1, T_step

        !$omp parallel do shared(N_i, N_j, N_k, M_load, a, b, c) collapse(3)
        DO i=1,N_i
                DO j=1,N_j
                        DO k=1,N_k
                                do m=1, M_load
                                        c(i,j,k) = a(i,j,k) + b(i,j,k)
                                end do
                        End DO
                End DO
        End DO
        !$end omp parallel do

End DO
```

Figure 3 CpuOMP - A CPU based OpenMP realization of our simplified CFD application. As a higher level language,

OpenMP uses a serial of directives to finish the tasks of thread creation, workload distribution, data environment management, and thread synchronization, etc.[14]. The number of threads can be set by exporting the environment variable OMP_NUM_THREADS, but the maximum performance can only be met when the number of threads equal to the product of the number of cores and hyper threads.

The compiling command used is:

nvfortran -g -mp -m64 -O0 -Minfo -Wall -Werror -traceback -Mrecursive -o OpenMP OpenMP.f90

3. 2 Normal CUDA FORTRAN paralleling method (NormalCuda)

Normal CUDA FORTRAN code uses the traditional or native CUDA method of defining variables in both the host and the device sides and exchanging the variables between the host and the device, as shown below:

```
use cudafor
implicit none

!define host variable
real*8 :: a(N_i,N_j,N_k), b(N_i,N_j,N_k), c(N_i,N_j,N_k)
!define device variable
real*8, device :: D_a(N_i,N_j,N_k), D_b(N_i,N_j,N_k), D_c(N_i,N_j,N_k)

integer :: N_i, N_j, N_k, M, istat, i, T_steps

!define block dimensions
integer*4 ::  blockDim_x=16, blockDim_y=16, blockDim_z=4
integer*4 ::  gridDim_x, gridDim_y, gridDim_z !define grid dimensions

type(dim3) :: grid, block

!Calculate the grid dimension based on the mesh size
gridDim_x = (N_i+blockDim_x-1)/blockDim_x + 1
gridDim_y = (N_j+blockDim_y-1)/blockDim_y + 1
gridDim_z = (N_k+blockDim_z-1)/blockDim_z + 1
grid = dim3(gridDim_x, gridDim_y, gridDim_z)
block= dim3(blockDim_x, blockDim_y, blockDim_z)

!initialize a, b, c

D_a = a
D_b = b
D_c = c

DO i=1, T_steps
    call cube_add<<<grid,block>>>(D_a, D_b, D_c, N_i, N_j, N_k, M)
    istat = cudaDeviceSynchronize()
END DO

c = D_c

!....
```

a) Host code

```
1  attributes(global) subroutine cube_add(a, b, c, N_i, N_j, N_k, M_load)
2
3      implicit none
4
5      real*8 :: a(N_i,N_j,N_k),b(N_i,N_j,N_k),c(N_i,N_j,N_k)
6      integer*4, value :: N_i, N_j, N_k, M_load
7
8      integer*4 :: i, j, k,m
9      integer*4 :: idx_i, idx_j, idx_k
10     integer*4 :: stride_i, stride_j, stride_k
11     integer*4 :: total_does
12
13     !blockIdx%x starts from 1
14     idx_i = (blockIdx%x-1) * blockDim%x + threadIdx%x
15     idx_j = (blockIdx%y-1) * blockDim%y + threadIdx%y
16     idx_k = (blockIdx%z-1) * blockDim%z + threadIdx%z
17
18     stride_i = blockDim%x * gridDim%x
19     stride_j = blockDim%y * gridDim%y
20     stride_k = blockDim%z * gridDim%z
21
22     DO i=idx_i,N_i, stride_i
23         DO j=idx_j,N_j, stride_j
24             DO k=idx_k,N_k, stride_k
25                 do m=1, M_load
26                     c(i,j,k) = a(i,j,k) + b(i,j,k)
27                         total_does=total_does+1
28                 end do
29             End DO
30         End DO
31     End DO
32         !print *, 'total_does =', total_does
33 end subroutine cube_add
```

b) Device code

Figure 4 NormalCuda. ***a) Line 1 include the cudafor library needed for CUDA FORTRAN; Line 5 and 7 defines the host and device variables, respectively; Line 12 and 13 defines block and grid dimensions, respectively; Line 15 to 22 define and assign the grid and block dimensions; Line 26 to 28 move the host variables of a, b, and c to the corresponding device variables D_a, D_b, and D_c; Line 31 calls the device code (which is called kernel) as shown in b); Line 32 is to synchronize the GPU threads, and line 35 copies the updated device variable back to the host variable. Note that a and b are only input variables and do not need to be copied out. More details can be found in [18] b) Device code of NormalCuda. In case that the length of data in each dimension is larger than the size of assigned GPU thread dimension, stride is used so that it is possible for one thread to process more than one data points.***

The compiling command used is:

nvfortran -g -m64 -O0 -Wall -Werror -gpu=ccnative -cuda -traceback -Mrecursive -o NormalCuda NormalCuda.cuf

3. 3 CUDA paralleling method using managed variables (CudaMg)

Managed variables use Unified Memory which is a component of the CUDA programming model. First introduced in CUDA 6.0, Unified Memory defines a managed memory space in which all processors see a single coherent memory image with a common address space [19]. This hardware/software technology allows applications to allocate data that can be read or written from code running on either CPUs or GPUs [20]. The use of managed variables reduces the programming efforts by removing the explicit data exchange between the host side and the device side, as shown in the following **Figure 5**.

```
use cudafor
implicit none

!define host variable
real*8 , managed :: a(N_i,N_j,N_k), b(N_i,N_j,N_k), c(N_i,N_j,N_k)
!define device variable
!real*8, device :: D_a(N_i,N_j,N_k), D_b(N_i,N_j,N_k), D_c(N_i,N_j,N_k)

integer :: N_i, N_j, N_k, M, istat, i, T_steps

!define block dimensions
integer*4 ::  blockDim_x=16, blockDim_y=16, blockDim_z=4
integer*4 ::  gridDim_x, gridDim_y, gridDim_z !define grid dimensions

type(dim3) :: grid, block

!Calculate the grid dimension based on the mesh size
gridDim_x = (N_i+blockDim_x-1)/blockDim_x + 1
gridDim_y = (N_j+blockDim_y-1)/blockDim_y + 1
gridDim_z = (N_k+blockDim_z-1)/blockDim_z + 1

grid = dim3(gridDim_x, gridDim_y, gridDim_z)
block= dim3(blockDim_x, blockDim_y, blockDim_z)

!.........
!initialize a, b, c
!..........

DO i=1, T_steps
    call cube_add<<<grid,block>>>(a, b, c, N_i, N_j, N_k, M)
    istat = cudaDeviceSynchronize()
END DO

!....
```

Figure 5 Host code of CudaMg. Compared to the host code of NormalCuda, only one set of managed variables of a, b, and c is defined, leaving the system to manage which side the variables should be and when to move data between the host side and the device side.

The device code of CudaMg is same to that of NormalCuda.

This compiling command used is:

nvfortran -g -m64 -O0 -Wall -Werror -gpu=ccnative -cuda -traceback -Mrecursive -o CudaMg CudaMg.cuf

3. 4 CUDA paralleling method using managed variables and conditional compilation (CudaMgAuto)

This method uses the CUDA's conditional compilation function, which further simplifies both the host side and the device side codes since the compiler is now responsible for automatically creating the kernel code, as shown below:

```
!@cuf use cudafor
implicit none

!@cuf attributes(managed) :: a, b, c
real*8 :: a(N_i,N_j,N_k),b(N_i,N_j,N_k),c(N_i,N_j,N_k)
integer*4, value :: N_i, N_j, N_k, M_load
integer*4 :: i, j, k,m, istat

DO i=1, T_steps
    !$cuf kernel do(3) <<<*,*>>>
    DO i=1,N_i
            DO j=1,N_j
               DO k=1,N_k
                    do m=1, M_load
                        c(i,j,k) = a(i,j,k) + b(i,j,k)
                    end do
               End DO
            End DO
    End DO
End DO

istat = cudaDeviceSynchronize()
```

Figure 6 CudaMgAuto - both managed variables and conditional compilation are used. A source line that begins with the !@cuf sentinels the rest of the line appears as a statement [15]. Line 1 indicates the use of cudafor library, line 4 indicates the variables of a, b, c to be defined next line are managed variables, and line 9 tells the compiler to create the kernel code for the following 3 layers of do loops. By using the nvfortran compiler, !@cuf will be recognized if the source file name ending with .cuf or .CUF, or if "-cuda" is specified on the compiling command line.

The compiling command used is:

nvfortran -g -m64 -O0 -Wall -Werror -gpu=ccnative -Minfo=accel -cuda -traceback -Mrecursive -o CudaMgAuto CudaMgAuto.cuf

3. 5 CUDA paralleling method using device variables(CudaDevice)

Everything in CudaDevice code is same to that in CudaMg except the variable definition in line 5 of Figure 5 changes from "managed" to "device". This means that all the three major variables of a, b, and c are all-time residing in the device side, and therefore there is to need to exchange the data between the host side and the device side.

The compiling command used is:

nvfortran -g -m64 -O0 -Wall -Werror -gpu=ccnative -cuda -traceback -Mrecursive -o CudaDevice CudaDevice.cuf

3. 6 CUDA paralleling method using device variables and conditional compilation (CudaDeviceAuto)

Similarly, compared to CudaMgAuto, in the CudaDeviceAuto code only the variable definition is changed from "managed" to "device" in line 4 of Figure 6.

The compiling command used is:

nvfortran -g -m64 -O0 -Wall -Werror -gpu=ccnative -cuda -traceback -Mrecursive -o CudaDeviceAuto CudaDeviceAuto .cuf

3. 7 Paralleling method with OpenACC

OpenACC is a programming standard designed to simplify parallel programming of heterogeneous CPU/GPU systems [21]. Similar to OpenMP and conditional compilation discussed above, OpenACC uses directives as its primary mode of programming [18]. The realization of our simplified Pseudo CFD application in OpenACC is shown below:

```
1 !$acc data copy(a, b, c)
2 DO t = 1, T_step
3
4         !$acc kernels
5         DO i=1,N_i
6             DO j=1,N_j
7                 DO k=1,N_k
8                     do m=1, M_load
9                         c(i,j,k) = a(i,j,k) + b(i,j,k)
10                    end do
11                End DO
12            End DO
13        End DO
14        !$acc end kernels
15
16 End DO
17 !$acc end data
```

Figure 7 OpenACC realization. In line 1 the host side variables of a, b, and c are copied into the device side, and then in line 17 the device side variables of a, b, and c are copied back to the host side. Line 4 and Line 14 make a kernel scope for the compiler to look for any potential parallelism. Note that in the kernel scope, only the 3 mesh do loops are able to be parallelized since the work load do loop has data dependence on c(i,j,k) and cannot be parallelized.

The compiling command used is:

nvfortran -g -m64 -O0 -acc -gpu=ccnative -Minfo=accel -o OpenAcc OpenAcc.f90

Where -acc tells the compiler to recognize the lines starting with "!$acc". Therefore, OpenAcc is also a conditional compilation method different from CUDA.

4. Performance analysis of various CPU/GPU pairs and parallel methods

4. 1 CPU/GPU pairs and compiler

Four GPUs as well as the associated CPUs are select to investigate the performance of GPU when the three major factors of mesh size, time steps, and work load change.

(1) P5000

CPU: Intel(R) Xeon(R) CPU E5-2623 v4 @ 2.60GHz, 8 cores

GPU: Quadro P5000, 16Gb Memory with bandwidth up to 288Gb/s, 2560 CUDA cores, more details can be found in [22].

(2) RTX4000

CPU: Intel(R) Xeon(R) Silver 4215R CPU @ 3.20GHz, 8 cores

GPU: Quadro RTX 4000, 8Gb Memory with bandwidth up to 416Gb/s , 2304 CUDA cores, 288 Tensor Cores, more details can be found in [23]

(3) RTX5000

CPU: Intel(R) Xeon(R) Silver 4215R CPU @ 3.20GHz, 8 cores

GPU: Quadro RTX 5000, 16Gb Memory with bandwidth up to 448Gb/s, 3072 CUDA cores, 384 Tensor Cores, more details can be found in [24]

(4) A4000

CPU: Intel(R) Xeon(R) Gold 5315Y CPU @ 3.20GHz, 8 cores

GPU: NVIDIA RTX A4000, 16Gb Memory with bandwidth of 448Gb/s, 6144 CUDA cores, 192 Tensor Cores, more details can be found in [25]

In all the CPU/GPU pairs, nvhpc-23-7 is installed which provides the compiler of nvfortran used in our study.

4. 2 Tests setting up

For each of the 4 CPU/GPU pairs, performance tests are conducted when each of the three problem size factors changes. The testing cases are:

(1) PTM (Performance Test for Mesh size)

In this group, time step and work load are set to constant values of 50 and 5, respectively, and only the mesh size (same in each of the 3 dimensions) varies in the set of 10, 20, 50, 100, 200, and 500.

(2) PTT (Performance Test for Time step)

In this group, mesh size and work load are set to constant values of 50 and 5, respectively, and only the time step varies in the set of 10, 20, 50, 100, 200, and 500

(3) PTL (Performance Test for work Load)

In this group, mesh size and time step are both set to constant value of 50, and only the work load varies in the set of 1, 2, 5, 10, 20, and 50

4. 3 Performance Comparison between the eight pseudo CFD application codes per CPU/GPU pair when each factor changes

(1) Mesh size

Table 1 Simulation time (s) for various mesh size in P5000

Types	PTM10	PTM20	PTM50	PTM100	PTM200	PTM500
CpuFortran	0.0018	0.0172	0.3271	3.4152	74.2440	1421.1420
NormalCuda	0.0098	0.0159	0.1096	0.7068	4.9891	76.0624
CudaMg	0.0108	0.0141	0.1038	0.6868	4.9254	75.9827
CudaMgAuto	0.0038	0.0097	0.0929	0.6916	4.8174	68.3299
CudaDevice	0.0100	0.0125	0.1028	0.6844	4.8954	75.2176
CudaDeviceAuto	0.0036	0.0090	0.0918	0.6901	4.7959	68.1323
OpenAcc	0.0033	0.0086	0.0964	0.7602	6.0297	91.0776
CpuOmp	0.0024	0.0172	0.3077	3.2370	29.3063	496.2524

Table 2 Simulation time (s) for various mesh size in RTX4000

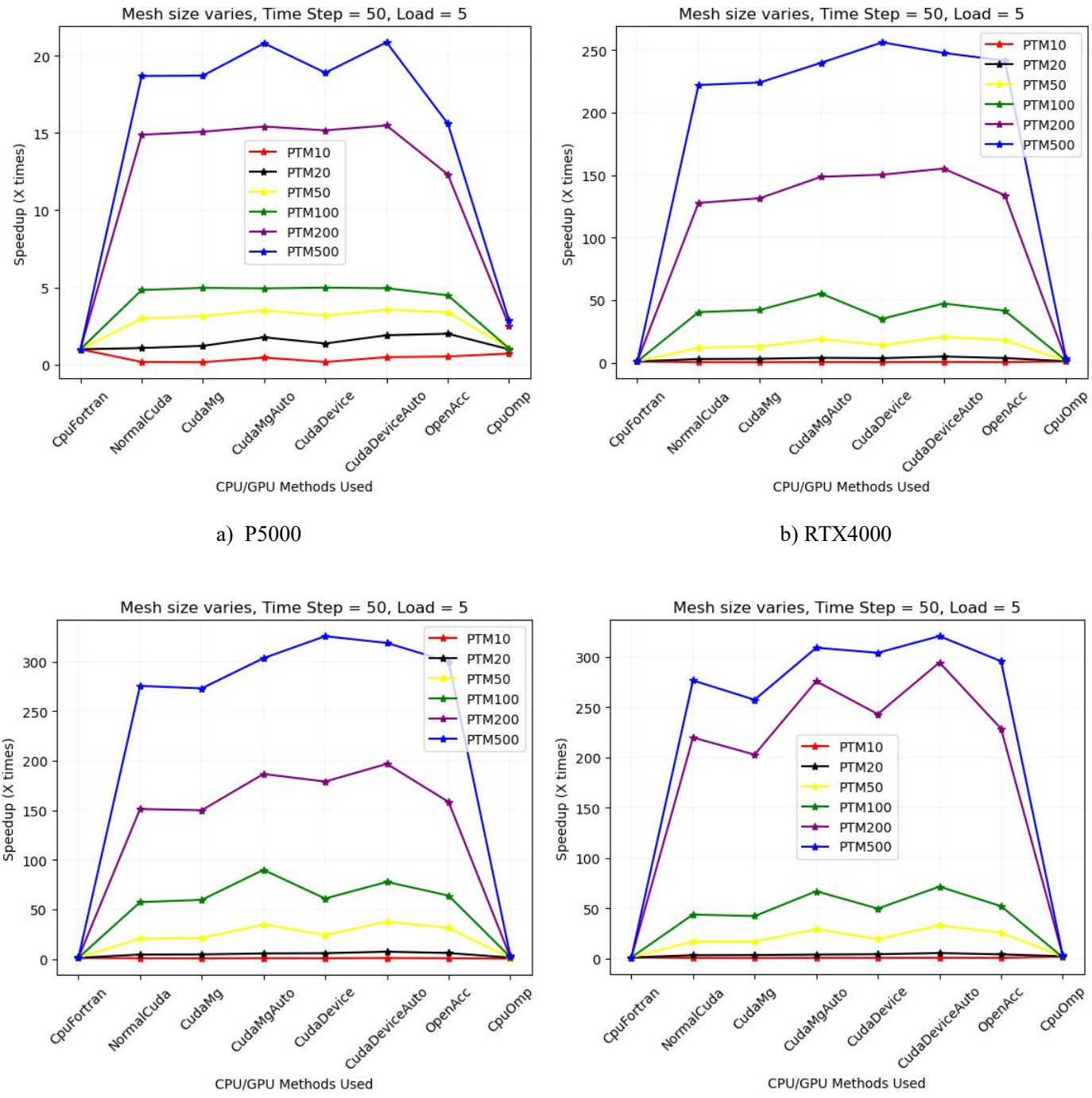
Types	PTM10	PTM20	PTM50	PTM100	PTM200	PTM500
CpuFortran	0.0013	0.0110	0.2203	2.8264	49.8891	1313.2060
NormalCuda	0.0029	0.0037	0.0184	0.0699	0.3906	5.9151
CudaMg	0.0031	0.0035	0.0170	0.0669	0.3795	5.8624
CudaMgAuto	0.0025	0.0028	0.0116	0.0511	0.3357	5.4779
CudaDevice	0.0028	0.0030	0.0157	0.0805	0.3318	5.1278
CudaDeviceAuto	0.0021	0.0022	0.0107	0.0597	0.3215	5.3024
OpenAcc	0.0028	0.0029	0.0122	0.0679	0.3725	5.4390
CpuOmp	0.0012	0.0097	0.1636	1.7206	23.1322	404.2322

Table 3 Simulation time (s) for various mesh size in RTX5000

Types	PTM10	PTM20	PTM50	PTM100	PTM200	PTM500
CpuFortran	0.0012	0.0110	0.2113	2.7734	45.1567	1261.1560
NormalCuda	0.0019	0.0025	0.0103	0.0485	0.2986	4.5784
CudaMg	0.0023	0.0024	0.0101	0.0465	0.3011	4.6200
CudaMgAuto	0.0018	0.0020	0.0061	0.0309	0.2421	4.1564
CudaDevice	0.0019	0.0019	0.0088	0.0455	0.2524	3.8732
CudaDeviceAuto	0.0014	0.0015	0.0056	0.0357	0.2295	3.9561
OpenAcc	0.0019	0.0019	0.0068	0.0434	0.2851	4.2038
CpuOmp	0.0038	0.0086	0.2381	1.4678	23.1406	392.4486

Table 4 Simulation time (s) for various mesh size in A4000

Types	PTM10	PTM20	PTM50	PTM100	PTM200	PTM500
CpuFortran	0.0011	0.0088	0.1729	2.2606	62.7138	1233.6920
NormalCuda	0.0022	0.0027	0.0102	0.0517	0.2856	4.4637
CudaMg	0.0024	0.0026	0.0103	0.0537	0.3095	4.7971
CudaMgAuto	0.0019	0.0023	0.0060	0.0338	0.2276	3.9937
CudaDevice	0.0020	0.0021	0.0089	0.0456	0.2582	4.0603
CudaDeviceAuto	0.0016	0.0016	0.0053	0.0317	0.2131	3.8501
OpenAcc	0.0020	0.0021	0.0067	0.0435	0.2745	4.1751
CpuOmp	0.0006	0.0041	0.1015	1.7734	21.1167	370.1656



a) P5000

b) RTX4000

c)RTX5000

d)A4000

Figure 8 GPU speedup times compared to CPU for different CPU/GPU pairs when mesh size changes.

From the tables and the figure above, the following observations can be made:

1) For all the four CPU/GPU pairs, the performance speedup times monotonically increases with mesh size for all the six GPU parallel methods. For the CPU based parallel method, namely OpenMP, the performance speedup times only increases monotonically with mesh size for RTX4000. For the other three CPU/GPU pairs, there are points at which the performance speedup times decrease. This means that when the mesh size increases, the CPU based parallel method, CpuOmp, does not guarantee a better scale-up capability than the pure serial CPU based method, while all the six GPU based parallel methods do. Note that each of the CPUs has 8 cores, and the number of OpenMP threads has been set to 8 to get the highest performance, therefore increase the number of OpenMP threads will not work better.
2) For all the four CPU/GPU pairs, CpuOmp can only gain a performance speedup times of less than 4, whereas in the worst case of P5000 the GPU based parallel methods can reach a factor of 20, and in the best case of RTX5000 a factor of 325.

3) Of the six GPU parallel methods, the unified memory method of CudaMg which does not use the conditional compilation has the lowest performance, and the traditional GPU CUDA method of NormalCuda slightly outperforms CudaMg.
4) Surprisingly, the unified memory method of CudaMgAuto which uses conditional compilation shows great performances, which are comparable to the best methods of CudaDevice or CudaDeviceAuto (The two device methods keep all the large variables on the device side to save the data exchanging time, that is why they have the highest performances since data movement between host and device is usually the bottleneck of the GPU based parallel methods).
5) The two GPU parallel methods using conditional compilation functions, namely CudaMgAuto and CudaDeviceAuto, have shown good performance compared to their counterparts of CudaMg and CudaDevice, respectively. This means that considering the easiness they provide to the programmer, the two conditional compilation methods are promising.
6) Differently from some of the existing publications showing significant lower performance of OpenACC than the other CUDA based parallel methods, our performance tests with a variety of mesh sizes and CPU/GPU pairs indicates that the performance of OpenACC is comparable to that of the other CUDA based parallel methods. In the worst case of mesh size 500 in P5000, the fastest method of CudaDeviceAuto is only 1.34 times faster than the OpenACC.
7) All the GPU based parallel methods outperform CPU based parallel method of CpuOmp for all the mesh sizes except the smallest size of 10.
8) It can be inferred from the test result that there exists a critical mesh size below which no parallel method could outperform the CPU based serial method. This is reasonable in that every parallel method will inevitably introduce some overhead time and thus the mesh size needs to be large enough to make a parallel method profitable.
9) Without running the tests to see how many CPU cores can match the capability of a GPU, we can estimate this number based on the performance gap between the 8 core CPUs and the GPUs. For example, the GPU RTX4000's CudaDevice parallel method outperforms its linked CPU's CpuOMP parallel method by about 80 times at mesh size of 500, which leads to 8 * 80 = 640 CPU cores assuming a linear scalability.

(2) Time step

Table 5 Simulation time (s) for various time steps in P5000

Types	PTT10	PTT20	PTT50	PTT100	PTT200	PTT500
CpuFortran	0.0726	0.1311	0.3426	0.6975	1.2673	3.3268
NormalCuda	0.0238	0.0491	0.1104	0.2207	0.4413	1.0931
CudaMg	0.0231	0.0460	0.1040	0.2070	0.4135	1.0292
CudaMgAuto	0.0220	0.0410	0.0930	0.1844	0.3668	0.9138
CudaDevice	0.0229	0.0451	0.1028	0.2046	0.4111	1.0252
CudaDeviceAuto	0.0204	0.0405	0.0933	0.1844	0.3654	0.9136
OpenAcc	0.0224	0.0438	0.0979	0.1936	0.3873	0.9676
CpuOmp	0.0991	0.0771	0.2287	0.4809	1.0058	1.9414

Table 6 Simulation time (s) for various time steps in RTX4000

Types	PTT10	PTT20	PTT50	PTT100	PTT200	PTT500
CpuFortran	0.0469	0.0868	0.2195	0.4262	0.8835	2.1736
NormalCuda	0.0046	0.0081	0.0183	0.0355	0.0647	0.1172
CudaMg	0.0045	0.0076	0.0170	0.0328	0.0593	0.0879
CudaMgAuto	0.0028	0.0049	0.0112	0.0229	0.0402	0.0598

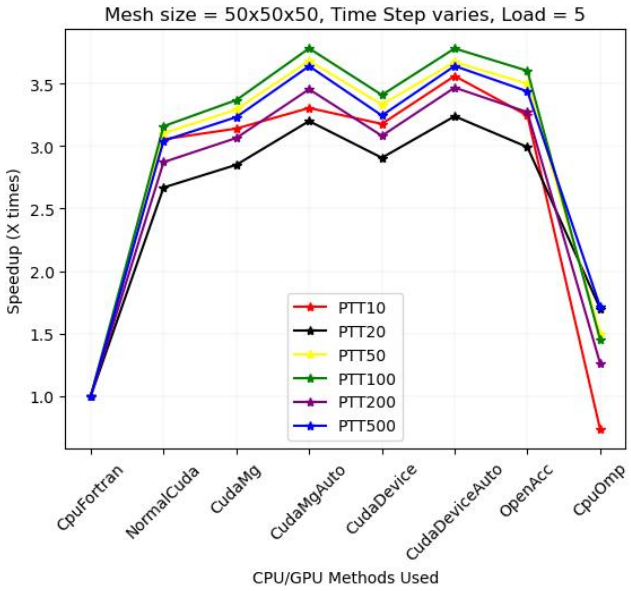
CudaDevice	0.0032	0.0064	0.0157	0.0313	0.0579	0.0866
CudaDeviceAuto	0.0022	0.0043	0.0107	0.0213	0.0427	0.0588
OpenAcc	0.0038	0.0059	0.0122	0.0229	0.0447	0.0605
CpuOmp	0.0643	0.0681	0.1262	0.2095	0.5907	1.5399

Table 7 Simulation time (s) for various time steps in RTX5000

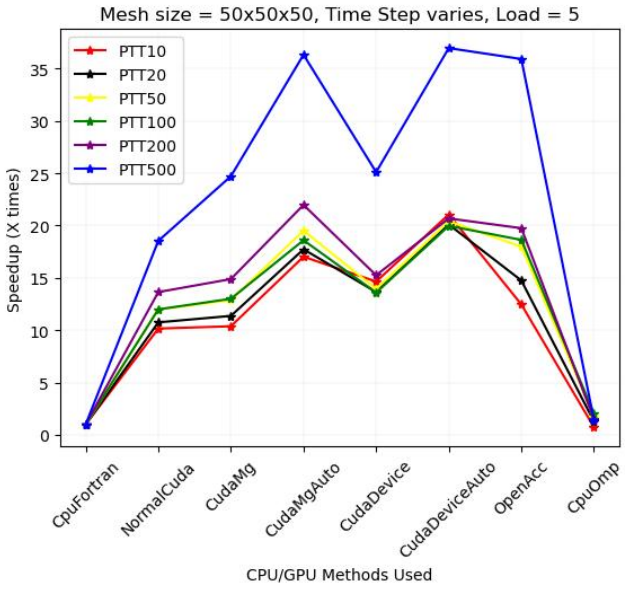
Types	PTT10	PTT20	PTT50	PTT100	PTT200	PTT500
CpuFortran	0.0446	0.0840	0.2462	0.4961	0.9175	2.0963
NormalCuda	0.0030	0.0048	0.0104	0.0195	0.0379	0.0854
CudaMg	0.0031	0.0047	0.0101	0.0191	0.0363	0.0819
CudaMgAuto	0.0018	0.0028	0.0061	0.0117	0.0227	0.0494
CudaDevice	0.0018	0.0036	0.0088	0.0176	0.0358	0.0778
CudaDeviceAuto	0.0012	0.0023	0.0056	0.0112	0.0222	0.0485
OpenAcc	0.0025	0.0036	0.0069	0.0119	0.0228	0.0478
CpuOmp	0.0608	0.0728	0.2021	0.3709	0.5670	1.5004

Table 8 Simulation time (s) for various time steps in A4000

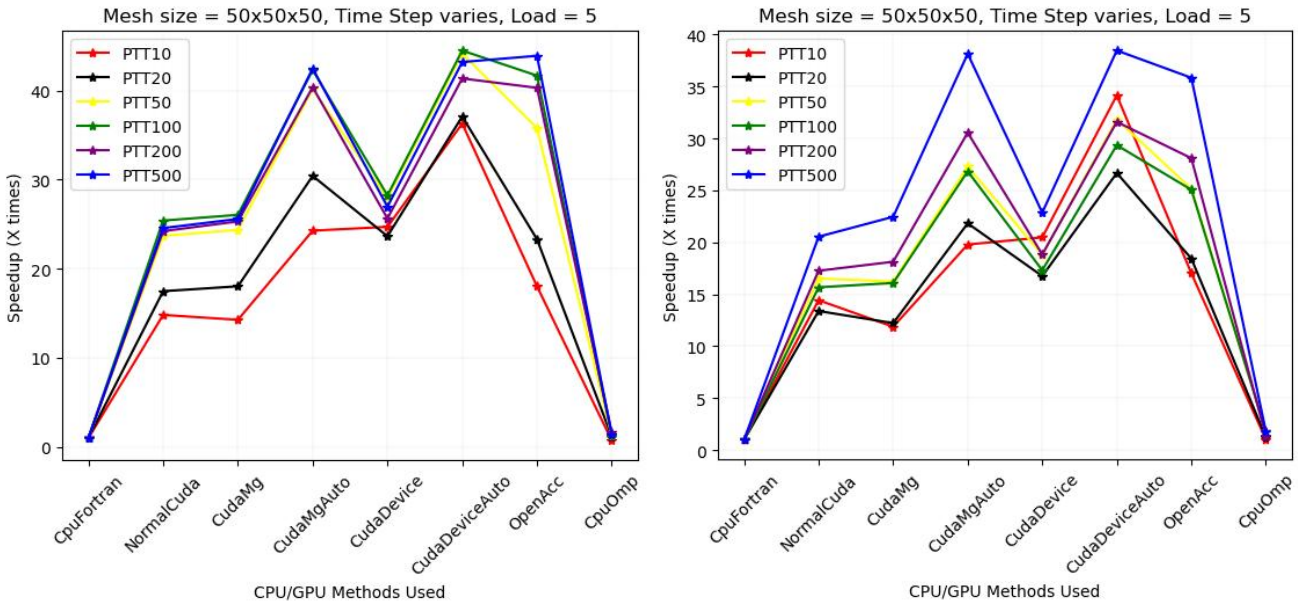
Types	PTT10	PTT20	PTT50	PTT100	PTT200	PTT500
CpuFortran	0.0372	0.0602	0.1675	0.3086	0.6673	1.7005
NormalCuda	0.0026	0.0045	0.0101	0.0197	0.0386	0.0828
CudaMg	0.0031	0.0049	0.0103	0.0192	0.0368	0.0757
CudaMgAuto	0.0019	0.0028	0.0061	0.0115	0.0218	0.0446
CudaDevice	0.0018	0.0036	0.0089	0.0178	0.0354	0.0744
CudaDeviceAuto	0.0011	0.0023	0.0053	0.0105	0.0211	0.0442
OpenAcc	0.0022	0.0033	0.0067	0.0123	0.0237	0.0474
CpuOmp	0.0367	0.0484	0.0929	0.1692	0.4909	0.9518



a)P5000



b)RTX4000



c)RTX5000 d)A4000

Figure 9 GPU speedup times compared to CPU when time step changes

From the tables and the figure above, the following observations can be made:

1) When the time step grows from 10 to 500, the performance speedup times of the 6 GPU based parallel methods have factors of about 4 to 40, which are much lower than the case of mesh size (~20 to 300). The reason of this is that the mesh size parts of the codes are within the GPU kernel and parallelized, whereas the time step parts of the codes are outside the GPU kernel and not parallelized.
2) Ideally speaking, the performance speedup factors for each of the 6 GPU based parallel methods should not change when the time step increases (since time steps codes are not parallelized). However, for each of the 4 CPU/GPU pairs, we did see the changes, which may come from: 1) The variance in CPUs' and/or GPUs' runtime speed, memory management, I/O schedule, and branch predictor [26], etc.. 2) The reduced average overhead, like the time needed to move data between the host side and the device side, etc. Since the two causes may both exist in NormalCuda, CudaMg, CudaMgAuto, and OpenAcc, but only the first one may exist in CudaDevice and CudaDeviceAuto, we observed more centralized performance speedup factors for CudaDevice and CudaDeviceAuto, especially in RTX5000.
3) It is not guaranteed that the performance speedup times of the GPU based parallel methods will increase with the time steps. In our tests, this tendency generally exists in RTX4000, RTX5000, and A400, but not in P5000.
4) All the GPU based parallel methods outperform CPU based parallel method of CpuOmp for every value of time step.

(3) Work load

Table 9 Simulation time for various work loads (s) in P5000

Types	PTL1	PTL2	PTL5	PTL10	PTL20	PTL50
CpuFortran	0.1343	0.1520	0.3387	0.5406	0.9543	2.0322
NormalCuda	0.0430	0.0568	0.1096	0.2000	0.3758	0.9055
CudaMg	0.0358	0.0511	0.1035	0.1941	0.3680	0.8976
CudaMgAuto	0.0247	0.0400	0.0924	0.1797	0.3558	0.8763
CudaDevice	0.0350	0.0498	0.1009	0.1901	0.3677	0.8972
CudaDeviceAuto	0.0231	0.0400	0.0915	0.1788	0.3514	0.8763
OpenAcc	0.0504	0.0592	0.0972	0.1634	0.2999	0.6990
CpuOmp	0.1138	0.1757	0.2490	0.2775	0.4979	0.9890

Table 10 Simulation time (s) for various work loads in RTX4000

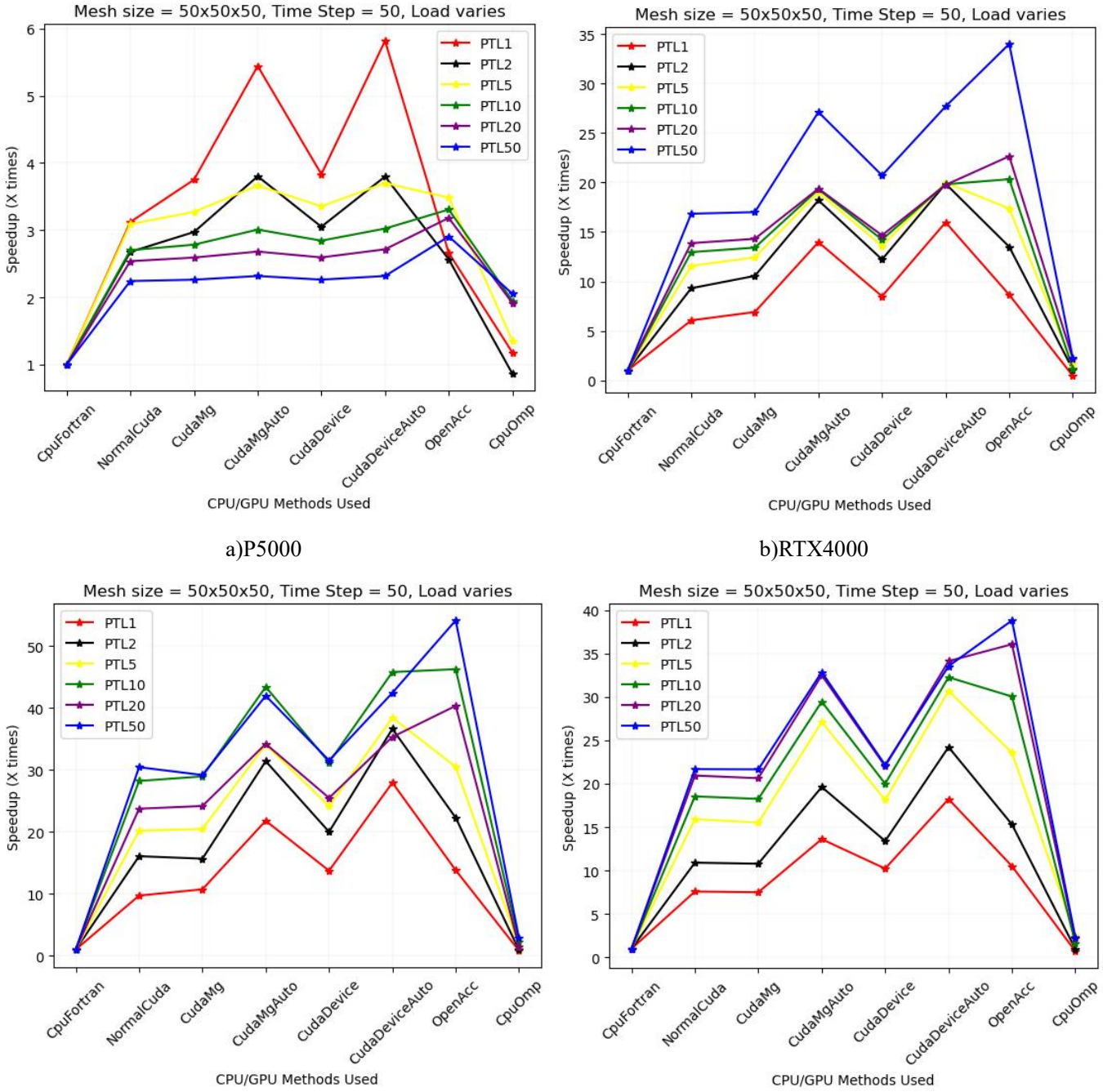
Types	PTL1	PTL2	PTL5	PTL10	PTL20	PTL50
CpuFortran	0.0615	0.1136	0.2133	0.3777	0.7063	1.5884
NormalCuda	0.0101	0.0122	0.0184	0.0291	0.0509	0.0943
CudaMg	0.0089	0.0108	0.0172	0.0282	0.0494	0.0934
CudaMgAuto	0.0044	0.0062	0.0112	0.0196	0.0364	0.0586
CudaDevice	0.0072	0.0093	0.0157	0.0266	0.0482	0.0767
CudaDeviceAuto	0.0038	0.0057	0.0107	0.0191	0.0358	0.0574
OpenAcc	0.0071	0.0084	0.0123	0.0186	0.0312	0.0467
CpuOmp	0.1371	0.1045	0.1320	0.3209	0.3264	0.7122

Table 11 Simulation time (s) for various work loads in RTX5000

Types	PTL1	PTL2	PTL5	PTL10	PTL20	PTL50
CpuFortran	0.0614	0.1131	0.2089	0.4458	0.6369	1.8152
NormalCuda	0.0063	0.0070	0.0103	0.0158	0.0268	0.0596
CudaMg	0.0057	0.0072	0.0102	0.0154	0.0263	0.0622
CudaMgAuto	0.0028	0.0036	0.0061	0.0103	0.0186	0.0433
CudaDevice	0.0045	0.0057	0.0087	0.0143	0.0250	0.0576
CudaDeviceAuto	0.0022	0.0031	0.0054	0.0097	0.0180	0.0428
OpenAcc	0.0044	0.0051	0.0068	0.0096	0.0158	0.0335
CpuOmp	0.0762	0.1229	0.1352	0.2034	0.4422	0.6435

Table 12 Simulation time (s) for various work loads in A4000

Types	PTL1	PTL2	PTL5	PTL10	PTL20	PTL50
CpuFortran	0.0440	0.0731	0.1618	0.2983	0.5797	1.3549
NormalCuda	0.0058	0.0067	0.0102	0.0161	0.0277	0.0625
CudaMg	0.0059	0.0068	0.0104	0.0163	0.0281	0.0626
CudaMgAuto	0.0032	0.0037	0.0060	0.0101	0.0178	0.0412
CudaDevice	0.0043	0.0055	0.0089	0.0149	0.0263	0.0611
CudaDeviceAuto	0.0024	0.0030	0.0053	0.0092	0.0170	0.0404
OpenAcc	0.0042	0.0048	0.0069	0.0099	0.0161	0.0349
CpuOmp	0.0626	0.0750	0.0636	0.1803	0.2676	0.5888



a)P5000 b)RTX4000

c)RTX5000 d)A4000

Figure 10 GPU speedup times compared to CPU when work load changes

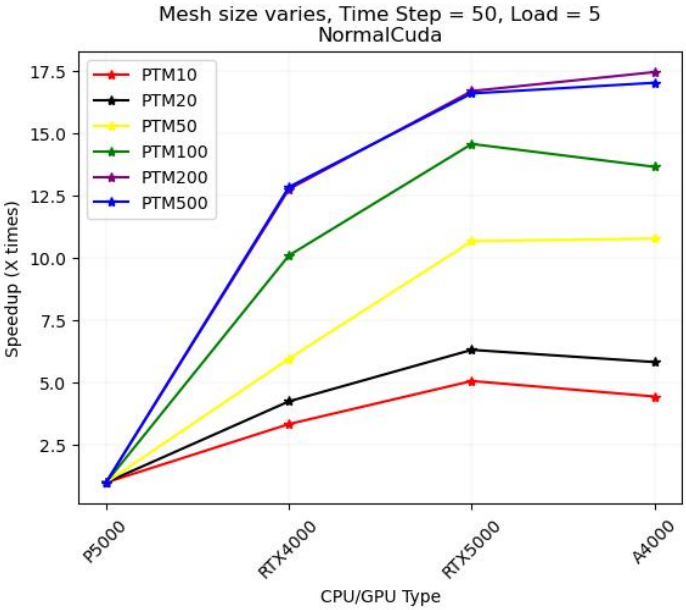
From the tables and the figure above, the following observations can be made:

1) When the work load grows from 1 to 50, the performance speedup times of the 6 GPU based parallel methods have factors of about 6 to 50, which are much lower than the case of mesh size (~20 to 300). The reason of this is that although the work load parts of the codes are inside the GPU kernel but they are not parallelized due to the existence of data dependency. Furthermore, the performance speedup times are slightly higher than the case of time step, which is due to the reduced average overhead from the synchronization of GPU threads.
2) Same to the case of time step, the performance speedup factors do not always increase with the work load.
3) Same to the case of time step, all the 6 GPU based parallel methods outperform the CPU based parallel method of CpuOmp

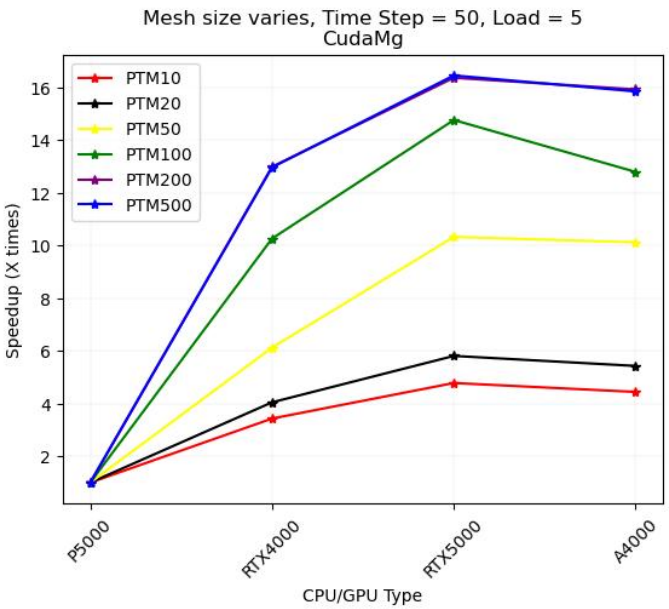
4. 4 GPU Performance comparison for each GPU parallel method

In this section, the performance of P5000 has been selected as the baseline (speedup times = 1), and performances of other CPU/GPU pairs are normalized based on it.

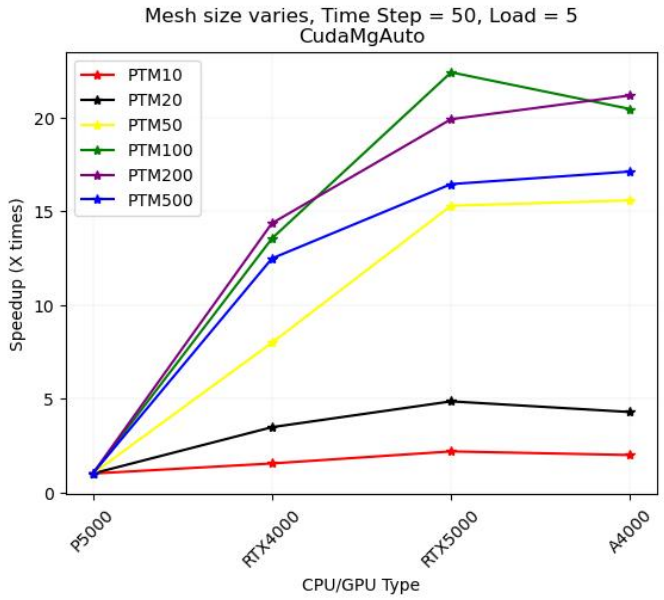
(1) Mesh size



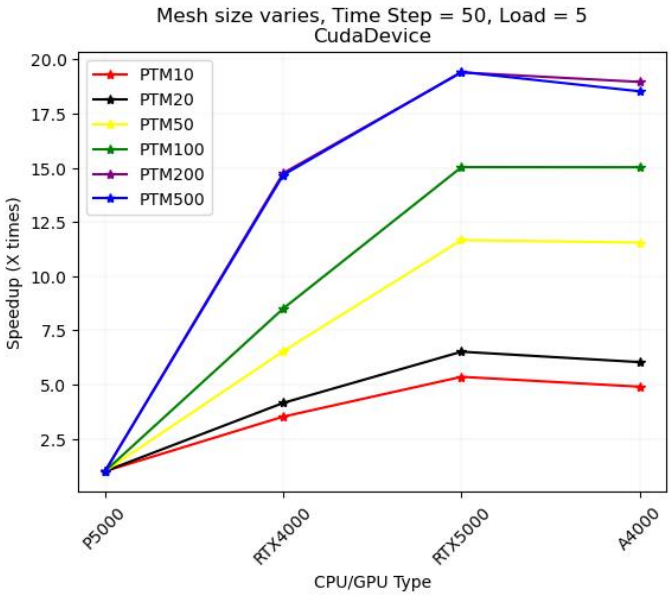
a) NormalCuda



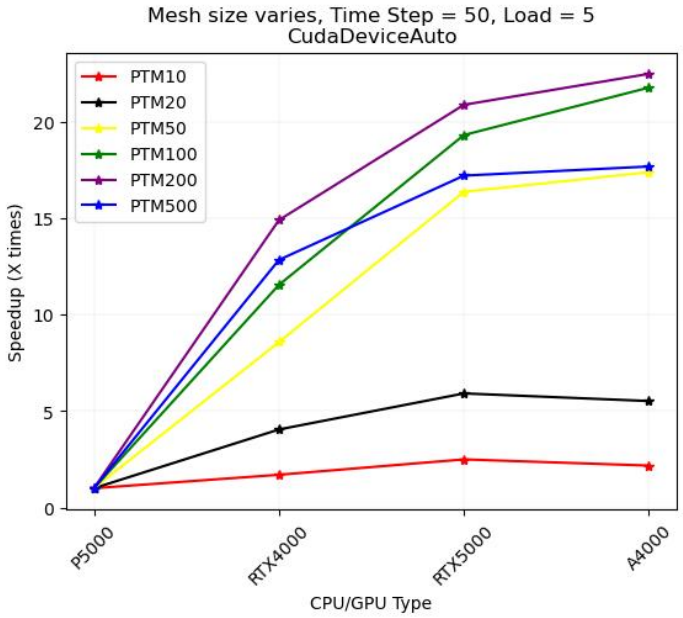
b) CudaMg



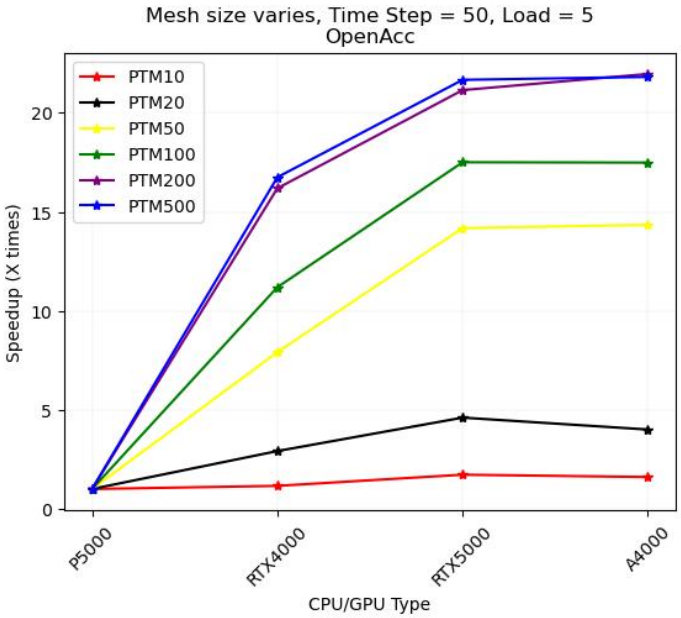
c) CudaMgAuto



d) CudaDevice



e) CudaDeviceAuto



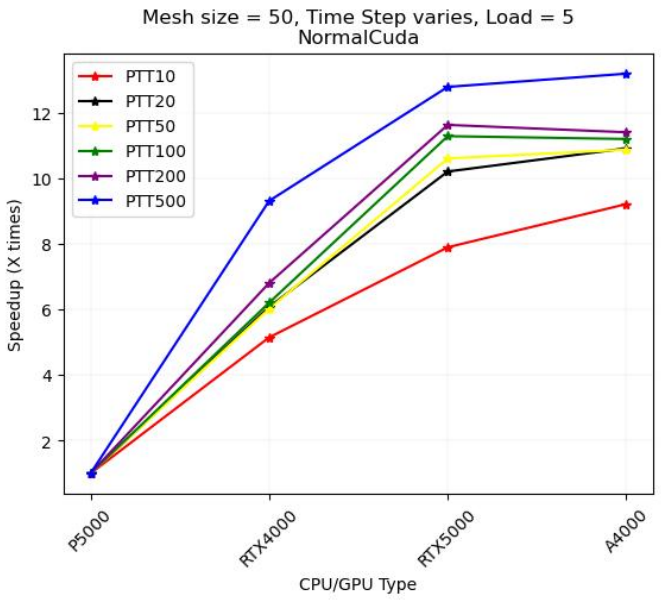
f) OpenAcc

Figure 11 GPU performance comparison for various GPU parallel methods when mesh size changes

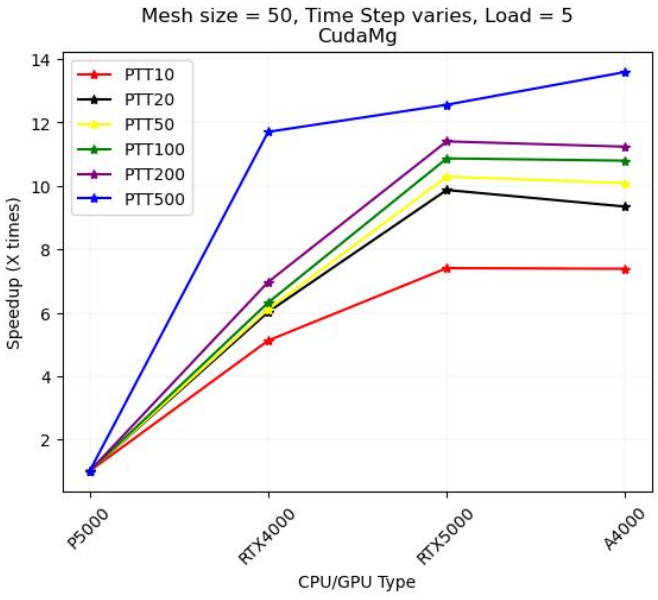
From the figure above, the following observations can be made:

1) RTX5000 and A4000 have very close performance and they all outperform RTX4000 which in turn outperforms P5000.
2) In general, It is not guaranteed that the performance speedup times of one GPU over another GPU will increase with the mesh size for every GPU based parallel method. For example, OpenAcc in RTX5000 hits highest performance for mesh size of 500, but CudaDeviceAuto in RTX500 hits highest performance for mesh size of 200.

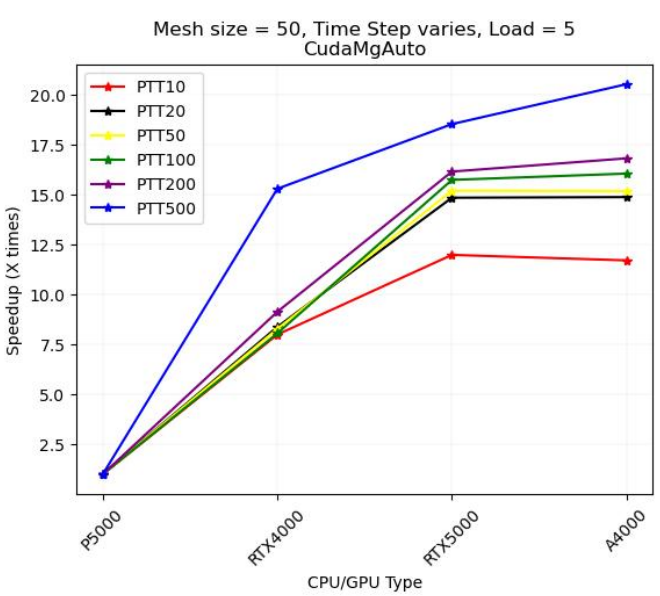
(2) Time step



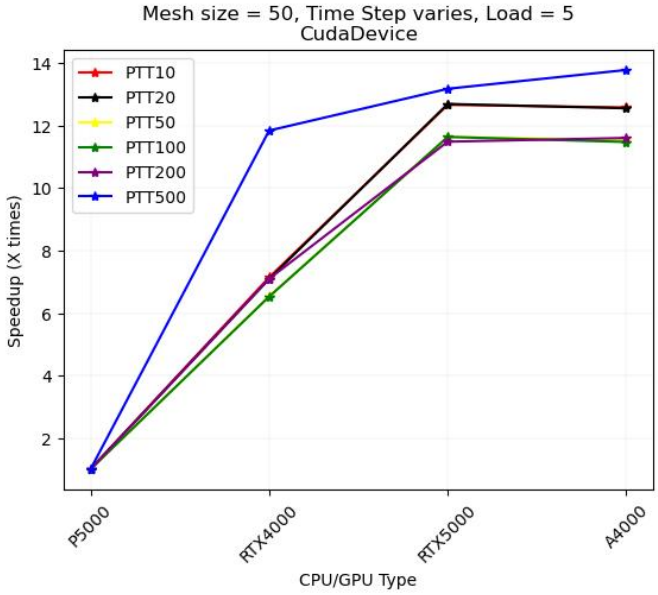
a)NormalCuda



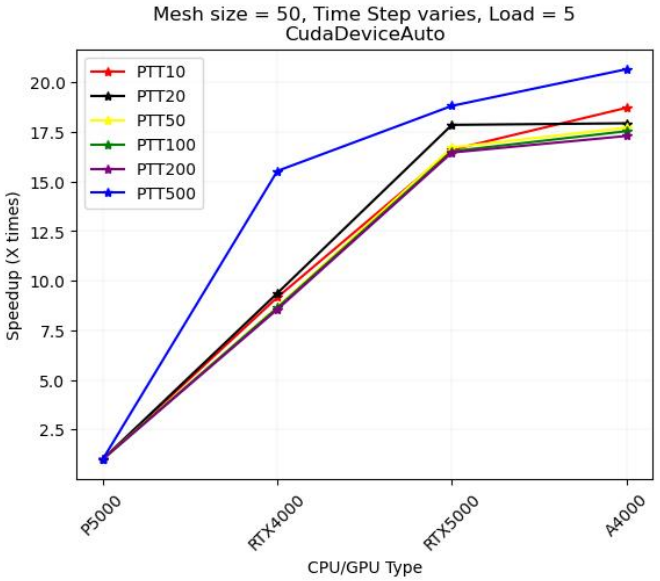
b) CudaMg



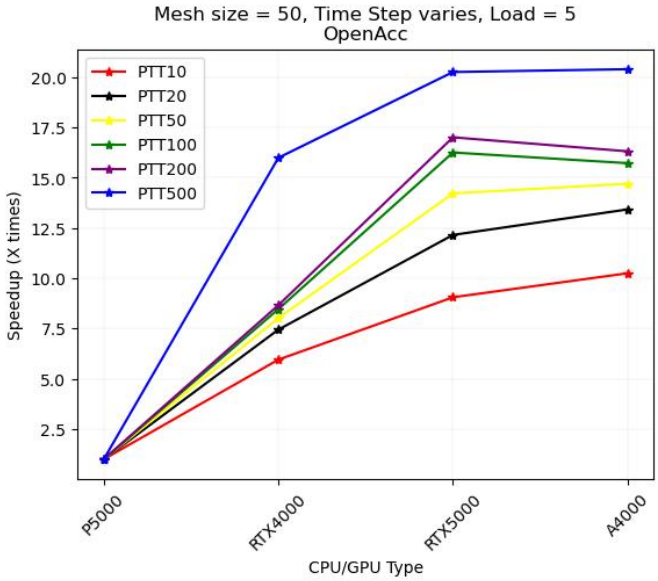
c) CudaMgAuto



d) CudaDevice



e) CudaDeviceAuto



f) OpenAcc

Figure 12 GPU performance comparison for various GPU parallel methods when time step changes

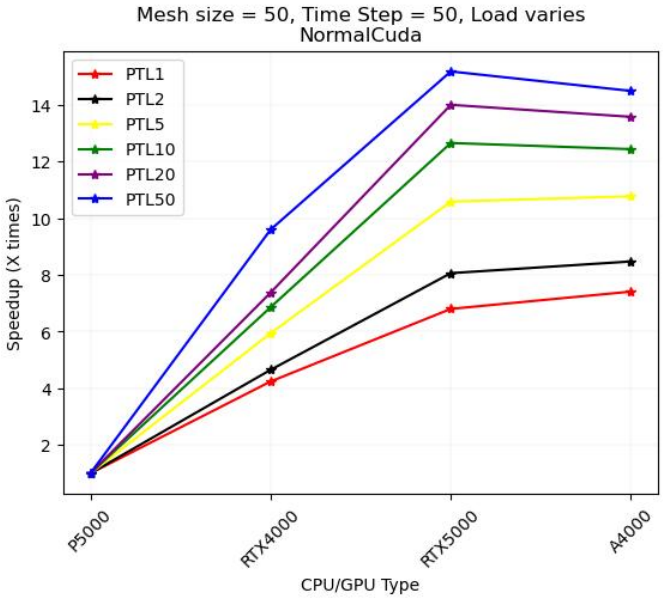
From the figure above, the following observations can be made:

1) The performance speed up generally (not strictly) increases with the time step for each GPU parallel method

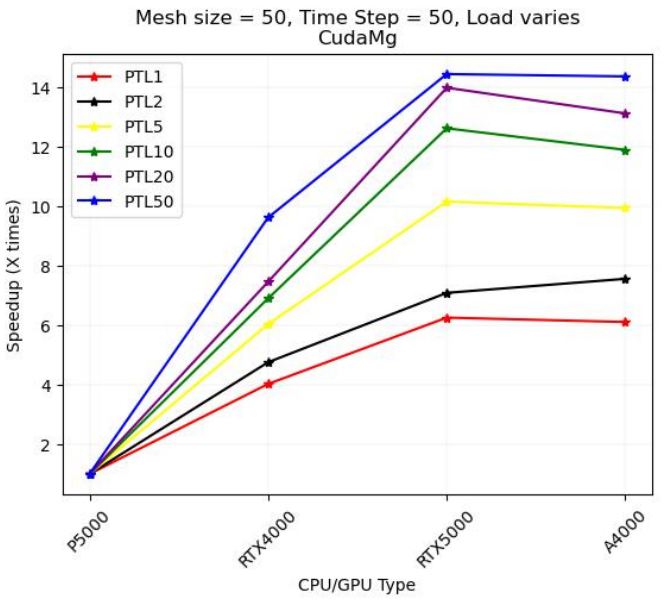
2) The performance speed up times compared to P5000 are generally smaller than the case of mesh size for each GPU parallel method.

3) In general, A4000 has the highest performance, followed by RTX5000 and RTX4000

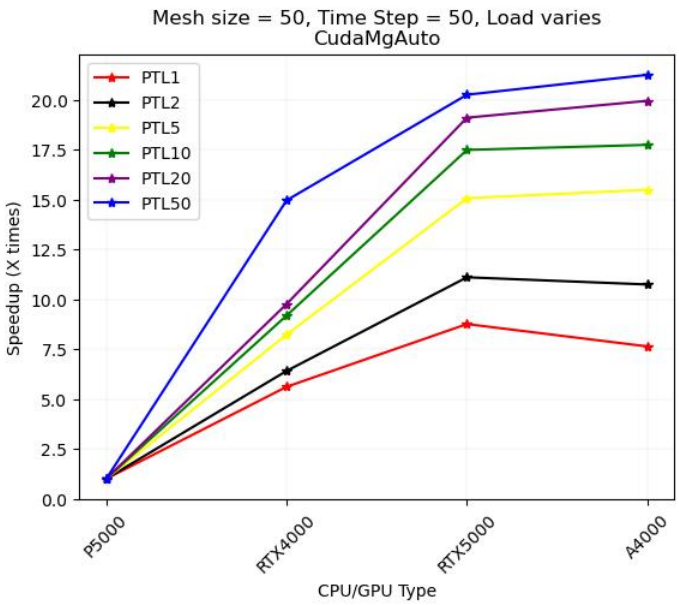
(3) Work load



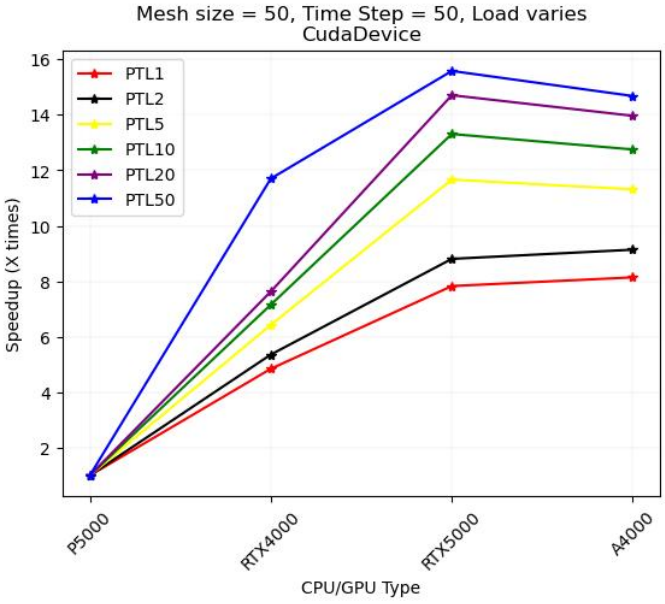
a)NormalCuda



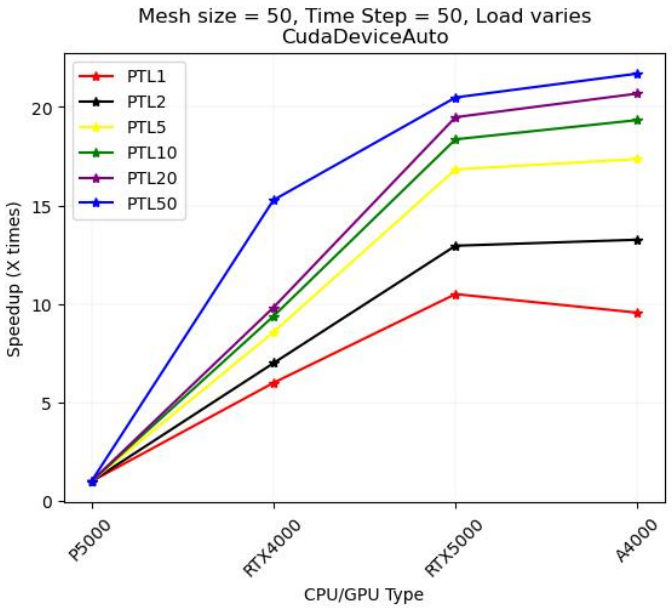
b) CudaMg

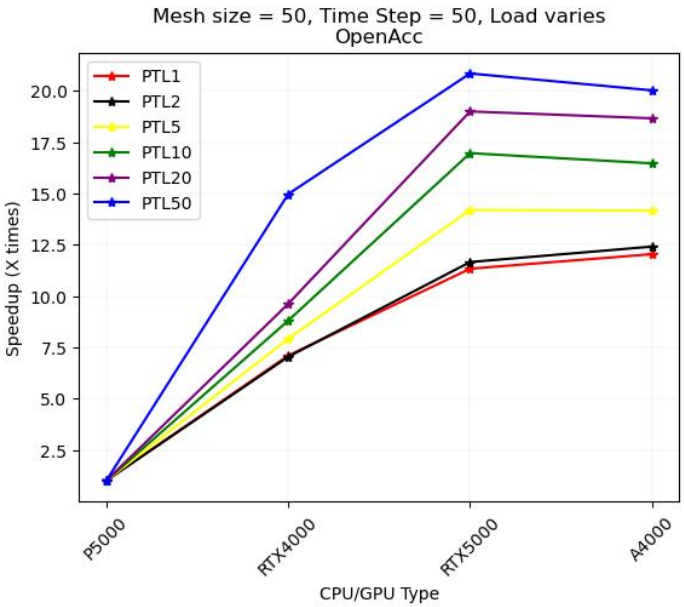


a)CudaMgAuto



b) CudaDevice





a)CudaDeviceAuto b) OpenAcc

Figure 13 GPU performance comparison for various GPU parallel methods when work load changes.

From the figure above, the following observations can be made:

1) Unlike the cases of mesh size and time step, the performance speed up monotonically increases with the work load for each GPU parallel method.

2) The performance speed up times compared to P5000 are generally smaller than the case of mesh size but very close to the case of time step for each GPU parallel method.

3) In four of the six GPU parallel methods, RTX5000 slightly outperforms A4000.

5. Conclusions

The current researches on the performance of different parallel methods applied in CFD problems show quite different results due to the variance in the adopted CFD applications. To remove this bias, in this article we characterize a general CFD application with three problem size factors: mesh size, time step, and work load, and conduct a serials of tests to investigate the performance gap between different parallel methods and different CPU/GPU pairs. The following general conclusions can be made based on our test results:

(1) In general, all the GPU based parallel methods significantly outperform the CPU based parallel method of CpuOmp. In other word, one single GPU can equal tens of hundreds of CPU cores, depending on the type of GPU, the GPU parallel methods, and which problem size factors dominate a CFD application. This is consistent with the results reported by [2].

(2) Although the two device parallel methods, CudaDevice and CudaDeviceAuto, tend to achieve higher performance for larger mesh size, noticeably the conditional compilation method of CudaMgAuto and OpenAcc, which require less porting efforts, have demonstrated very competitive performances for changing time step and work load. The conclusions made by [12,14] that OpenAcc is more than 1.5 times slower than the (best) CUDA based parallel methods have been only confirmed in the cases with very small time steps and/or small work loads, see Figure 9 c) and d), and Figure 10 a). To some extent our results are closer to that reported by [10, 13] which showed smaller performance gap between OpenACC and CUDA. This finding can guide us to choose different parallel methods based on what factors are dominating a CFD application. For example, we can choose device parallel methods for CFD applications with large number of mesh cells although the porting efforts may increase, and choose easier methods of CudaMgAuto and/or OpenAcc for CFD applications with medium mesh size but much larger time step and work load.

(3) The selection of suitable CPU/GPU pairs (especially the GPU types) is very important. Sometimes the computation capability of single-precision performance (SPP) with the unit of TFLOPS alone cannot decide which GPU has better performance when being applied in specific CFD applications. For example, although the RTX4000 has a SPP of 7.1 TFLOPS which is smaller than that of P5000 (8.9 TFLOPS [27]), our tests show the former has significantly higher performance than the latter, probably due to the former's higher memory bandwidth (RTX4000 has an up to 416 GB/s memory bandwidth, while P5000 has an up to 288 GB/s memory bandwidth), and/or the higher speed of its paired CPU (the CPU sitting with RTX4000 has a speed of 3.2GHz, while that sitting with P5000 has a speed of 2.6GHz). Before starting to port a specific CFD application, our method presented in this article can be employed to measure the real CPU/GPU pairs' performance if it is not straightforward to tell which is better.

6. Discussions

While applying the method we proposed in this article, It should be beard in mind that realistic CFD applications may be quite complicate and need apprehensive analysis before starting to port them into GPU based codes. This may include but not limited to:

(1) While the determination of the mesh size is straightforward, the calculation of average time steps needed to finish the simulations and average work loads may not be a trivial thing.

(2) Some CFD codes may incorporate considerable data dependency which may prevent the parallelism. For example, the use of implicit numerical methods in CFD to speed up the simulation may also introduce considerable data dependency

between different time steps [28, 29]. Therefore some decoupling work may be necessary before we can port the codes into GPU.

(3) It is important to estimate the profitability of porting a CPU based CFD application to a GPU based version, since only the parallelizable part of the code can be parallelized, and thus to what extent the porting can enhance the simulation performance will be limited by the percentage of parallelizable code.

Acknowledgements

Thanks for PaperSpace to provide the author with low cost CPU/GPU pairs to conduct the main part of this research. Also thanks for Google Colab and AWS Studiolab Sagemaker to grant the author free CPU/GPU pairs to conduct the initial tests.

Reference

[1] Weicheng Xue, Charles W. Jackson, Christoper J. Roy, An improved framework of GPU computing for CFD applications on structured grids using OpenACC, Journal of Parallel and Distributed Computing, Volume 156, 2021, Pages 64-85, ISSN 0743-7315, https://doi.org/10.1016/j.jpdc.2021.05.010.

[2] https://www.ansys.com/blog/unleashing-the-power-of-multiple-gpus-for-cfd-simulations, accessed 10/26/2023

[3] Jamil Appa, Mike Turner and Neil Ashton, Performance of CPU and GPU HPC Architectures for off-design aircraft simulations, AIAA 2021-0141, https://doi.org/10.2514/6.2021-0141

[4] https://developer.nvidia.com/blog/computational-fluid-dynamics-revolution-driven-by-gpu-acceleration/ , accessed 10/26/2023

[5] https://sim-flow.com/rapid-cfd-gpu/

[6] Rui Wu, Connor Scully-Allison, Chase Carthen, Andy Garcia, Roger Hoang, Christopher Lewis, Ronn Siedrik Quijada , Jessica Smith , Sergiu M. Dascalu, Frederick C. Harris Jr, vFirelib: A GPU-based fire simulation and isualization tool, SoftwareX 23 (2023) 101411, https://doi.org/10.1016/j.softx.2023.101411

[7] Vanka, S & Shinn, Aaron & Sahu, Kirti. (2011). Computational Fluid Dynamics Using Graphics Processing Units: Challenges and Opportunities. ASME 2011 International Mechanical Engineering Congress and Exposition, IMECE 2011. 6. 10.1115/IMECE2011-65260.

[8] Erich Elsen, Patrick LeGresley, Eric Darve, Large calculation of the flow over a hypersonic vehicle using a GPU, Journal of Computational Physics, Volume 227, Issue 24, 2008, Pages 10148-10161, ISSN 0021-9991, https://doi.org/10.1016/j.jcp.2008.08.023.

[9] Yue Xiang, Bo Yu, Qing Yuan, Dongliang Sun, GPU Acceleration of CFD Algorithm: HSMAC and SIMPLE, Procedia Computer Science, Volume 108, 2017, Pages 1982-1989, ISSN 1877-0509, https://doi.org/10.1016/j.procs.2017.05.124.

[10] B. Cloutier, B. K. Muite and P. Rigge, "Performance of FORTRAN and C GPU Extensions for a Benchmark Suite of Fourier Pseudospectral Algorithms," 2012 Symposium on Application Accelerators in High Performance Computing, Argonne, IL, USA, 2012, pp. 145-148, doi: 10.1109/SAAHPC.2012.24.

[11] Mikhail Khalilov and Alexey Timoveev, 2021 J. Phys.: Conf. Ser. 1740 012056, DOI 10.1088/1742-6596/1740/1/012056

[12] Matthew Norman, Jeffrey Larkin, Aaron Vose, Katherine Evans, A case study of CUDA FORTRAN and OpenACC for an atmospheric climate kernel, Journal of Computational Science, Volume 9, 2015, Pages 1-6, ISSN 1877-7503, https://doi.org/10.1016/j.jocs.2015.04.022.

[13] T. Hoshino, N. Maruyama, S. Matsuoka and R. Takaki, "CUDA vs OpenACC: Performance Case Studies with Kernel Benchmarks and a Memory-Bound CFD Application," 2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing, Delft, Netherlands, 2013, pp. 136-143, doi: 10.1109/CCGrid.2013.12.

[14] Antonio J. Rueda, José M. Noguera, Adrián Luque, A comparison of native GPU computing versus OpenACC for implementing flow-routing algorithms in hydrological applications, Computers & Geosciences, Volume 87, 2016, Pages 91-100, ISSN 0098-3004, https://doi.org/10.1016/j.cageo.2015.12.004.

[15] Mallinson, Andrew C., Beckingsale, David A., Gaudin, W. P., Herdman, J. A., Levesque, J. M. and Jarvis, Stephen A. (2013) Cloverleaf : preparing hydrodynamics codes for exascale. In: A New Vintage of Computing : CUG2013, Napa, CA, 6 - 9 May 2013. Published in: A New Vintage of Computing : Preliminary Proceedings

[16] GREG RUETSCH, A CUDA FORTRAN PORT OF CLOVERLEAF, GPU Technology Conference, 2015, https://on-demand.gputechconf.com/gtc/2015/presentation/S5379-Greg-Ruetsch.pdf

[17] https://en.wikipedia.org/wiki/OpenMP, accessd on 10/27/2023

[18] https://docs.nvidia.com/hpc-sdk/compilers/cuda-fortran-prog-guide/, accessed 10/30/2023

[19] CUDA C++ Programming Guide, https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#um-unified-memory-programming-hd, accessed 10/31/2023

[20] https://developer.nvidia.com/blog/unified-memory-cuda-begi

nners/, accessed on 10/31/2023

[21] https://en.wikipedia.org/wiki/OpenACC#cite_note-1, accessed on 10/31/2023

[22] https://images.nvidia.com/content/pdf/quadro/data-sheets/192195-DS-NV-Quadro-P5000-US-12Sept-NV-FNL-WEB.pdf , accessed on 11/02/2023

[23] https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/quadro-product-literature/quadro-rtx-4000-data sheet-us-nvidia-1060942-r2-web.pdf , accessed on 11/02/2023

[24] https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/quadro-product-literature/quadro-rtx-5000-data -sheet-us-nvidia-704120-r4-web.pdf , accessed on 11/02/2023

[25] https://www.nvidia.com/content/dam/en-zz/Solutions/gtcs21/rtx-a4000/nvidia-rtx-a4000-datasheet.pdf , accessed on 11/02/2023

[26]https://en.wikipedia.org/wiki/Branch_predictor, accessed 11/05/2023

[27] https://www.pny.com/nvidia-quadro-p5000, accessed on 11/05/2023

[28] https://www.simscale.com/blog/implicit-vs-explicit-fem/#:~:text=Explicit%20FEM%20is%20used%20to%20calculate%20the%20state%20of%20a,states%20of%20the%20given%20system. accessed on 11/05/2023

[29] V. Venkatakrishnan, IMPLICIT SCHEMES AND PARALLEL COMPUTING IN UNSTRUCTURED GRID CFD, Lecture notes prepared for 26th Computational Fluid Dynamics Lecture Series Program of the von Karman Institute (VKI) for Fluid Dynamics, Rhode-SaintGenese, Belgium, 13-17 March 1995.