JCAMECH

Vol. 48, No. 2, December 2017, pp 161-170
DOI: 10.22059/jcamech.2017.238226.166

Accelerating high-order WENO schemes using two heterogeneous GPUs

Hossein Mahmoodi Darian[1,*]

[1]*Faculty of Engineering Science, College of Engineering, University of Tehran, Tehran, Iran*

Received: 21 July. 2017, Accepted: 1 Sep. 2017

Abstract

A double-GPU code is developed to accelerate WENO schemes. The test problem is a compressible viscous flow. The convective terms are discretized using third- to ninth-order WENO schemes and the viscous terms are discretized by the standard fourth-order central scheme. The code written in CUDA programming language is developed by modifying a single-GPU code. The OpenMP library is used for parallel execution of the code on both the GPUs. Data transfer between GPUs which is the main issue in developing the code, is carried out by defining halo points for numerical grids and by using a CUDA built-in function. The code is executed on a PC equipped with two heterogeneous GPUs. The computational times of different schemes are obtained and the speedups with respect to the single-GPU code are reported for different number of grid points. Furthermore, the developed code is analyzed by CUDA profiling tools. The analyze helps to further increase the code performance.

Keywords:

Multi-GPU, CUDA, OpenMP, WENO schemes, Compressible viscous flow

[*] Corresponding Author. Tel.: +98 2161112158; Fax: +98 2161112174
Email Address: hmahmoodi@ut.ac.ir

1. Introduction

In recent years by the appearance of many-core GPUs, there has been a growing interest in utilizing graphics processing units (GPU) in scientific computations. In the area of computational fluid dynamics (CFD), researchers have been exploiting this capability to reduce the computational time of the simulations. From the many recent studies, we may mention [1-3].

Using high-order methods are necessary to effectively resolve complex flow features such as turbulent or vortical flows [4]. For shock-containing flows, linear high-order methods are not suitable and instead high-order shock capturing schemes such as weighted essentially non-oscillatory (WENO) schemes [5, 6] should be used. However, these schemes are complex and have more computational cost than linear high-order schemes, such as compact or non-compact schemes [7-10].

Athanasios et al. [11] ported a Navier–Stokes solver on GPU, which used high-order WENO schemes for computing the convective fluxes. They used domain decomposition technique to distribute grid points between thread blocks inside the GPU. In [12, 13] the GPU implementation of high-order shock capturing WENO schemes is studied in very detail for multi-dimensional linear wave equation, Euler equations of gas dynamics and Navier-Stokes equations. The scope of this work is to investigate the acceleration of these schemes (third- to ninth-order) using more than one GPU. The programming language is CUDA. For utilizing several GPUs simultaneously, one may use MPI (Message Passing Interface) or OpenMP (Open Multi-Processing) libraries. When the GPUs reside on several PCs, MPI is the only choice. However, when the GPUs reside on a single PC, using OpenMP is preferred because of its simple code execution and debugging. Due to computational resources, only two GPUs are utilized in this research. Since both the GPUs reside on a single PC, the OpenMP library is used for parallel execution.

2. Test Problem

A viscous test case is used to verify and assess the developed code. The test case, known as shock-mixing layer interaction, is generally used in literature to examine the performance of shock capturing schemes for interaction of shock waves and shear layers [13-16]. The governing equations are the two-dimensional compressible Navier-Stokes equations in the non-dimensional form:

$$
\begin{aligned}
&U_t + F_x + G_y = F_x^v + G_y^v \\
&U = \begin{pmatrix} \rho \\ \rho u \\ \rho v \\ E \end{pmatrix},\ F = \begin{pmatrix} \rho u \\ \rho u^2 + p \\ \rho uv \\ (E+p)u \end{pmatrix},\ G = \begin{pmatrix} \rho v \\ \rho vu \\ \rho v^2 + p \\ (E+p)v \end{pmatrix},\ F^v = \begin{pmatrix} 0 \\ \tau_{xx} \\ \tau_{xy} \\ u\tau_{xx} + v\tau_{xy} - q_x \end{pmatrix},\ G^v = \begin{pmatrix} 0 \\ \tau_{yx} \\ \tau_{yy} \\ u\tau_{yx} + v\tau_{yy} - q_y \end{pmatrix} \\
&E = \rho\left(e + \frac{u^2+v^2}{2}\right), \quad p = \rho e(\gamma - 1) \\
&\tau_{xx} = \frac{4}{3}\frac{\mu}{\mathrm{Re}}u_x, \quad \tau_{xy} = \tau_{yx} = \frac{\mu}{\mathrm{Re}}(u_y + v_x), \quad \tau_{yy} = \frac{4}{3}\frac{\mu}{\mathrm{Re}}u_y \\
&q_x = -\frac{\mu}{(\gamma-1)\,\mathrm{Re}\,\mathrm{Pr}\,M_\infty^2}T_x, \quad q_y = -\frac{\mu}{(\gamma-1)\,\mathrm{Re}\,\mathrm{Pr}\,M_\infty^2}T_y \\
&\mu = \frac{1+\bar{c}}{\bar{c}+T}T^{3/2}, \quad \bar{c} = \frac{110.3}{300}, \quad T = \gamma M_\infty^2 \frac{p}{\rho}
\end{aligned}
\tag{1}
$$

where ρ, u, v, p, and e denote the density, x-velocity, y-velocity, pressure and internal energy per unit mass, respectively. The computational domain is $L_x \times L_y = 200 \times 40$ and the flow properties are $\gamma = 1.4$, $\mathrm{Pr} = 0.72$, $\mathrm{Re} = 1500$ and $M_\infty = 5.625$. An oblique shock originating from the upper-left corner interacts with a shear layer where the vortices arise from the shear layer instability. This oblique shock is deflected by the shear layer and then reflects from the bottom slip wall. Simultaneously, an expansion fan forms above the shear layer and at the downstream, a series of shock waves form around the vortices.

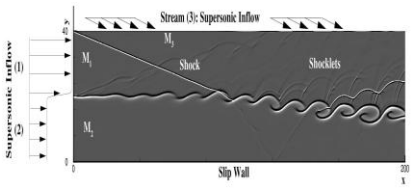

Figure 1 Schematic view of the shock/mixing-layer interaction configuration (By courtesy of Ref. [16] authors).

Figure 1 .s the schematic view of the flowshow The left inflow boundary condition is specified with a hyperbolic tangent profile for the x-velocity component and a fluctuating profile for the y-velocity component

$$u=\frac{1}{u_r}\left(2.5+0.5\tanh(2\hat{y})\right),\quad v=\frac{1}{u_r}\sum_{k=1}^{2}a_k\cos(2\pi kt/T+\phi_k)\exp(-\hat{y}^2/2) \tag{2}$$
$$u_r=3,\quad \hat{y}=y-\frac{L_y}{2}$$

with a period of $T=\lambda/u_c$, a convective velocity of $u_c=2.68/u_r$ and a wavelength of $\lambda=30$. Other constants are given by $b=10$, $a_1=a_2=0.05$, $\varphi_1=0$ and $\varphi_2=\pi/2$. Also, the inlet density and pressure are as follows

$$\rho=\frac{1}{\rho_r}\begin{cases}0.3626 & \hat{y}<0\\ 1.6374 & \hat{y}\ge 0\end{cases}\quad p=\frac{0.3327}{\rho_r u_r^2},\quad \rho_r=1.6374$$

The upper inflow boundary conditions are

$$u=\frac{2.9709}{u_r},v=\frac{-0.1367}{u_r},\rho=\frac{2.1101}{\rho_r},p=\frac{0.4754}{\rho_r u_r^2}$$

and the lower boundary is a slip wall and the outflow boundary has been arranged to be supersonic everywhere. For more details on the initial and boundary conditions we refer the reader to [14, 16]. Note that, due to different reference values (u_r,ρ_r), the results of $t=360$ are equivalent to that of $t=120$ in [14, 15]. Figure 1 also shows the density schlieren at $t=360$.

The convective terms are discretized by WENO schemes and the viscous terms are discretized using the standard fourth-order central difference scheme. One-sided second-order and third-order schemes are used for boundary and near boundary points, respectively. The time integration method is the third-order TVD Runge–Kutta scheme developed in [5]:

$$\begin{aligned}U^{(1)}&=U^n+\Delta t\,L(U^n)\\ U^{(2)}&=\frac{3}{4}U^n+\frac{1}{4}U^{(1)}+\frac{1}{4}\Delta t\,L(U^{(1)})\\ U^{n+1}&=\frac{1}{3}U^n+\frac{2}{3}U^{(2)}+\frac{2}{3}\Delta t\,L(U^{(2)})\end{aligned} \tag{3}$$

Note that computing the WENO fluxes, due to lculations for acomplex schemes and matrix c characteristic decompostion, are the most time . consuming kernel in the code

3. Domain and Grid Decomposition

The domain is decomposed into two sub-domains along x-direction. Each sub-domain is assigned to a single GPU. This means if a grid of $M_x\times M_y$ is considered for the domain, then each sub-domain is assigned a grid of $N_x\times N_y$ where $N_x=M_x/2$ and $N_y=M_y$ (see Figure 2).

Each GPU has its own dedicated memory. Therefore, for computing the convective and viscous terms for the points near the common boundary of the sub-domains, we require the points from the other sub-domain which is allocated on the other GPU. The usual technique is to define halo points corresponding to the points on the other grid and copy the necessary data from the other GPU to these halo points. This data transfer is the main issue in converting a single-GPU code to a multi-GPU code. Figure 2 shows the addition of one layer of halo points to each sub-domain. However, more halo points are required for the schemes used in this simulation. For a $(2k-1)$th-order WENO scheme, k layers of halo points and for the fourth-order central scheme, 2 layers of halo points are required. Therefore $\max(2,k)$ layers of halo points must be considered for each domain.

4. OpenMP Implementation

The OpenMP library is used to modify a single GPU code to simultaneously utilize both the GPUs. For the details of the single-GPU code the reader is referred to [13]. The following parts of the developed codes compare the single-GPU and double-GPU implementations:

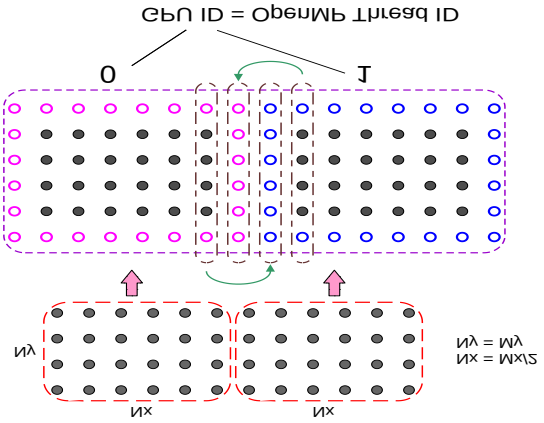

Figure 2 Top: Domain decomposition. Bottom: Addition of one-layer of halo points to each sub-domain. The curved arrows show the data transfer between sub-domains

Single-GPU code:

```
subDomain *d_dm;
…
SomeKernel<<<blockPerGrid,threadPerBlock>>>(d_dm);
…
```

Double-GPU code:

```
const int numDevs = 2;           //number of GPU devices
omp_set_num_threads(numDevs); //create as many CPU threads as those of GPU devices
subDomain *d_dm[numDevs];
…
#pragma omp parallel
{
        int iDev = omp_get_thread_num();
        cudaSetDevice(iDev);
        …
        SomeKernel<<<blockPerGrid,threadPerBlock>>>(d_dm[iDev]);
        …
```

Note that the number of OpenMP threads is set to be the same as the number of GPU devices (see also Figure 2). The variable type "subDomain" is a "struct" which includes the sub domain properties and the address of arrays on a GPU device as given below

```
Struct subDomain{
        int Nx,Ny;     // Grid dimensions
        float *x,*y;   // address (array) of position variables
        float *rho;    // address (array) of density variable
        …
```

Data transfer is required before each stage of the Runge-Kutta time integration. This means three times per time step. The following part of the code shows how each OpenMP thread transfers data from one GPU to the other:

Double-GPU code:

```
size_t transferSize = N_EQN * (Ny+2*numHaloPoints) * sizeof(float); // N_EQN = 4

GPU_SendBuffer <<<bpg[20], threadperblock >>>(U, pdm);

#pragma omp barrier
switch (iDev)
{
case 0:
        cudaMemcpyPeer(omp_dm[0].RRecvBuffer, 0, omp_dm[1].LSendBuffer, 1, transferSize);
        break;
case 1:
        cudaMemcpyPeer(omp_dm[1].LRecvBuffer, 1, omp_dm[0].RSendBuffer, 0, transferSize);
        break;
}
#pragma omp barrier

GPU_RecvBuffer <<<bpg[20], threadperblock >>>(U, pdm);
```

Data transfer between GPUs is done by "cudaMemcpyPeer" function. This is a CUDA built-in function. Here, we see the thread 0 is responsible to copy data from GPU 1 to GPU 0 and simultaneously thread 1 is responsible to copy data from GPU 0 to GPU 1. The function may be called four times for each of the conservative variables ($\rho, \rho u, \rho v, E$). However, it is more efficient to send and receive all the required data by a single call. Therefore, two kernels are added to the code: "GPU_SendBuffer" and "GPU_RecvBuffer". The kernel "GPU_SendBuffer" copies the halo points' values of the four conservative variables a buffer array and then with a single call of "cudaMemcpyPeer", the data is sent from a GPU to the buffer array of the other GPU. Finally, the kernel "GPU_RecvBuffer" copies the transferred data to the corresponding arrays of conservative variables. Figure 3 shows a schematic diagram of the send-receive process on both the GPUs.

The lines before and after the block of "switch" statement ("`#pragma omp barrier`"), prevent the CPU threads from computing the next stage before the other thread finishes its corresponding data transfer task. This is necessary to have updated values in the halo points.

Another important issue is the implementation of boundary conditions. The inlet boundary must be managed only by thread 0 and similarly the outlet boundary must be managed only by thread 1. This is simply accomplished by an if-statement in the code.

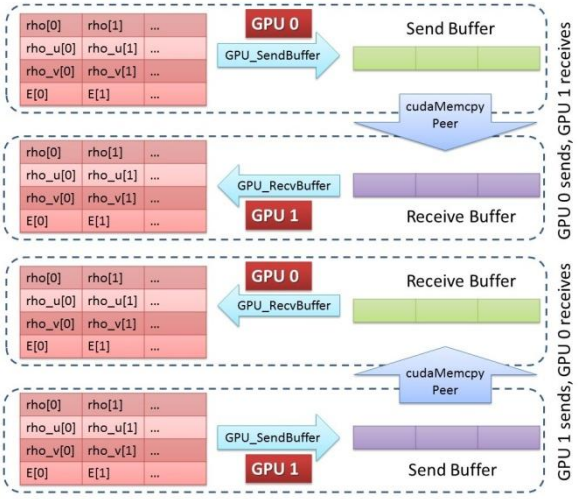

Figure 3 Top: Thread 0, GPU 0 Sends to GPU 1. Bottom: Thread 1, GPU 1 Sends to GPU 0.

5. Numerical Results

First, the simulations are carried out using a uniform grid of 512×128. Specifically, each GPU is assigned a grid of 256×128. In addition, for instance, for the WENO9 scheme, five layers of halo points are added to the grids which results in a grid of 266×138 for each GPU. Also, the number of threads per block is taken to be 256.

Figure 4 shows the density contours for both the single- and double-GPU codes using WENO9 scheme. The figure displays the flow after reaching the fully periodic state. Note that, due to different reference values, the results of t = 360 are equivalent to that of t =

120 in [14, 15]. The obtained results verify the developed double-GPU code.

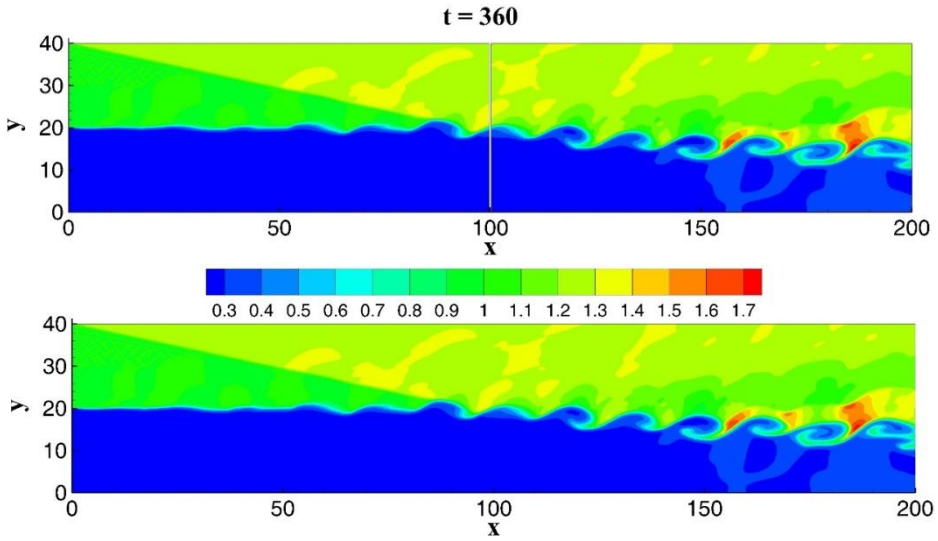

Figure 4 Density contours at $t = 360$. Top: Single-GPU. Bottom: Double-GPU.

Figure 5 compares the density distribution along $x = 150$ for different WENO schemes. The figure also shows a WENO9 solution obtained on a finer grid. The figure demonstrates by increasing the order of the WENO scheme more accurate solutions are obtained especially in the complex region of the flow.

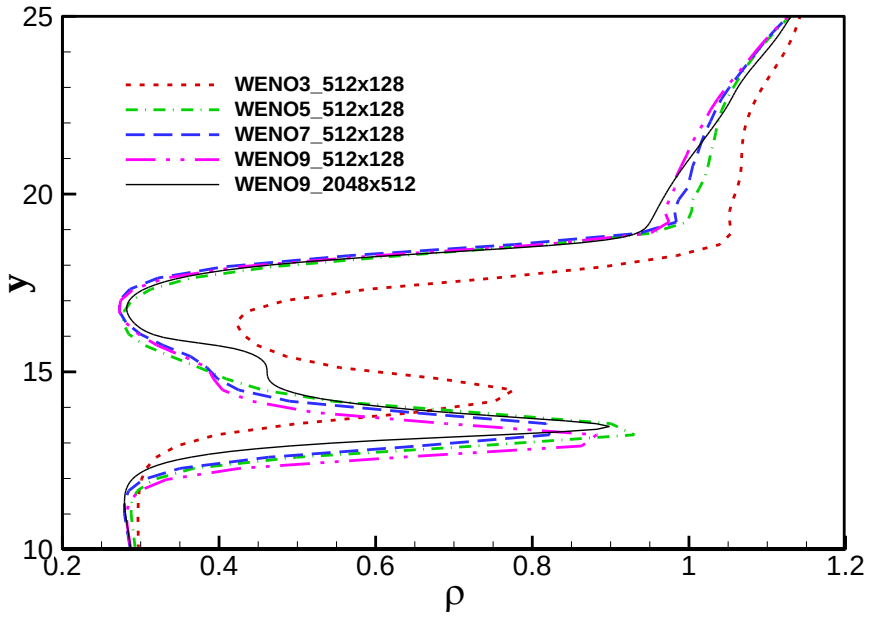

Figure 5 Density distribution along $x = 150$ at $t = 360$.

The PC which runs the simulations, is equipped with two different GPUs: the first GPU is a GeForce GTX 750 Ti and the second one is a GeForce GTX 550 Ti TOP GPU. Table 1 shows the important specifications of the GPUs. Therefore, it is not efficient to assign equal number of nodes to both of them. Since the first GPU has better specifications, more nodes must be assigned to this GPU. By running the code for each GPU separately

(single GPU run), it is found that that the computational performance of the first GPU is two times more than that of the second one. In addition, by running the double-GPU code, it is found that the best performance achieves when 11/16 and 5/16 of the nodes are assigned to the first and second GPUs, respectively. For instance, a grid of 256×256 is decomposed into two grids of 176×256 and 80×256.

Table 1: The GPUs specifications.

GPU	Compute Capability	Clock Speed	Effective Memory Clock Speed	Memory	CUDA Cores	Memory Clock Speed	Maximum Registers per Thread	Max. Shared Memory
GTX 750 Ti	5.0	1020 MHz	5400 MHz	2 GB	640	1350 MHz	255	64 KB
GTX 550 Ti	2.1	900 MHz	4104 MHz	1 GB	192	1026 MHz	63	48 KB

Table 2 compares the runtimes obtained by a single-GPU code and those of the double-GPU code. Note that the runtimes for the single-GPU code belong to the second GPU and are reported from [13], where it was optimized and assessed in very detail on the second GPU. The speed-ups are also reported in the table. For the single-GPU runs, as expected, the runtimes increase as the order of the WENO scheme increases. However, for the double-GPU runs the runtimes of the WENO7 scheme are more than that of the WENO9 scheme, which is peculiar.

Table 2: Comparison of Single- and Double-GPU runtimes.

Grid	Single-GPU time	Double-GPU time	speed-up	Single-GPU time	Double-GPU time	speed-up
		WENO3			WENO5	
256×256	7.98	4.61	1.73	10.01	5.42	1.85
512×512	28.95	15.30	1.89	37.68	18.41	2.05
1024×1024	114.98	55.72	2.06	150.62	68.12	2.21
		WENO7			WENO9	
256×256	11.52	8.91	1.29	13.75	6.31	2.18
512×512	43.91	32.47	1.35	52.69	20.99	2.51
1024×1024	175.85	123.82	1.42	211.27	78.30	2.70

CUDA profiler is a tool, which helps the programmer to analyze each kernel and identify opportunities to optimize the GPU code. In addition to a kernel runtime, an important quantity to assess a kernel is the achieved occupancy. Roughly speaking, the occupancy indicates how much a kernel utilizes the GPU resources. Figure 6 shows the runtimes, the achieved occupancy and the number of registers per thread for WENO kernels on each GPU on a grid of 256×256. The figure shows for the second GPU the runtimes of the WENO schemes increases gradually as expected. For the first GPU the same trend is observed except for the WENO7 scheme, which its runtime is significantly more than the other WENO schemes. The figure also shows, except WENO7, the runtimes of each WENO scheme are nearly equal for both the GPUs, which indicates the computational load is balanced between both the GPUs.

Considering the achieved occupancy for each kernel, we see a meaningful relation between the higher occupancy and less computational runtime. The occupancy of all the WENO schemes on the second GPU are almost the same and are about 30 percent. However, on the first GPU the occupancies are considerably different from each other. The occupancy for the WENO3 and WENO5 schemes are about 25 and for the WENO7 and WENO9 schemes are about 12 and 47 percent, respectively. The low occupancy of the WENO7 scheme on the first GPU is responsible for its high computational runtime (see table 2). The reason for the low occupancy of this scheme (according to CUDA profiler) is the number of registers and the amount of shared memory.

The GPU device has several types of memory [17]: the global, register and shared memory (see table 1). Although the global memory is the main memory of the

GPU, it is the slowest memory. Register memory resides in GPU chip and is the fastest GPU memory. The variables defined inside a kernel are local variables, which reside on global or register memory. The compiler automatically places some of these variables in the register memory space and places the remaining in global memory.

Depending on the available number of registers per thread, the compiler most probably places scalar variables and static small arrays in register memory. Because the complier make this decision, the programmer does not have enough control on the placement of local variables. However, the programmer can limit the number of register variables. Figure 6 also shows the number of registers used by each kernel on both the GPUs. The second GPU uses its maximum registers (63 as seen in table 1). The first GPU uses different number of registers for each kernel. Although the register memory is the fastest memory, using more registers may affect the occupancy and cause lower performance. This is also true for shared memory. As seen in the figure, the WENO9 scheme uses the least number of register, but has the highest occupancy. This convinces us to limit the number of registers in the compiler options and reduce the amount of shared memory by reducing the number of threads per block.

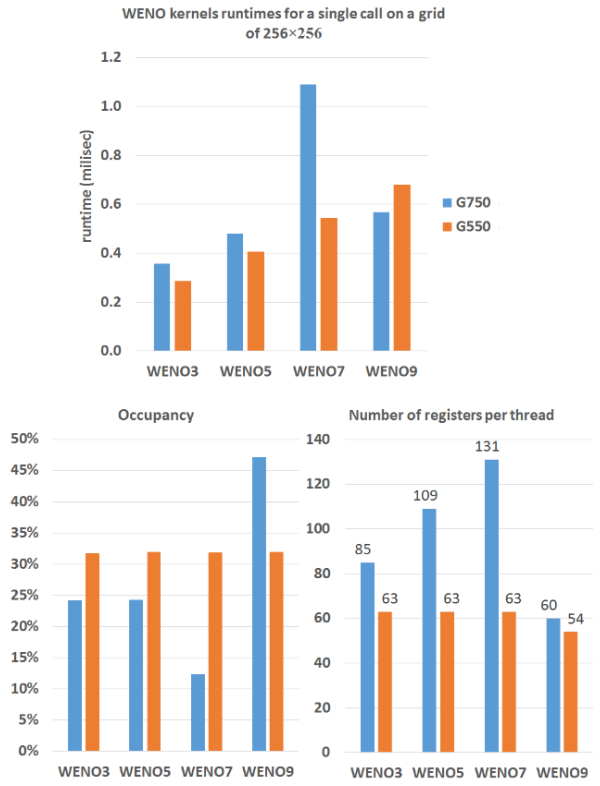

Figure 6 WENO kernels runtimes for a single call for both the GPUs.

Figure 7 shows the results as those of Figure 6 after limiting the number of registers to 63 and reducing the number of threads per block from 256 to 128. Note that using only one of these two modification does not considerably affect the occupancy. The figure shows the occupancies on the first GPU for the WENO3, WENO5 and WENO7 increases and equals to that of the WENO9 scheme and therefore the runtimes decreases for the mentioned schemes. However, since for the first GPU the amount of registers are already limited to 63 because of hardware limitation (see table 1), no significant change is observed for the kernels on this GPU. Note that it is possible to more limit the number of registers and obtain more occupancy, however this means to lose the benefit of the GPU fastest memory and therefore cause performance decrease.

Table 3 shows the runtimes after the modifications mentioned above. The table shows an approximate 2.2, 2.25, 2.35, 2.5 times faster execution runtimes are obtained for the WENO3, WENO5, WENO7 and WENO9, respectively. Note that, since 5/16 of the nodes are assigned the second GPU, one may roughly expect a speed-up of 3.2 (=16/5). However, due to data transfer a bit less speed-ups are obtained.

6. Conclusions

Using OpenMP library, we were able to modify a single-GPU code to a double-GPU one with a little effort. The code was used to accelerate third- to ninth-order WENO schemes. The main issue was the data transfer for the points near the common boundary of the sub-domains between the two GPUs. This was carried out by defining halo points for numerical grids and by using CUDA built-in function “cudaMemcpyPeer”. Furthermore, the halo node values of all the four conservative variables are copied to a buffer to reduce the time for data transfer between GPUs. Another issue was the implementation of boundary conditions, which was simply accomplished by an if-statement in the code. Due to different GPU specifications, the numerical grid was decomposed into two unequal grids. Using CUDA profiler, we were able to detect that the number of registers and the amount of shared memory caused WENO7 scheme low performance. By limiting the number of registers per thread and reducing the number of threads per block, the occupancy of WENO3, WENO5 and WENO7 kernels increased and reached to that of the WENO9 kernel. This indicated that for heterogeneous GPUs, an optimized code for a specific GPU might not be optimum for other GPUs and the performance improvement had to be done simultaneously for all the GPUs. The results also indicated, speed-ups of about 2.25 with respect to the single-GPU runs were obtained which were acceptable considering that the ideal speed-up is 3.2 and data transfer is a slow process.

7. Acknowledgments

The author would like to acknowledge the financial support of University of Tehran and Iran's National Elites Foundation for this research under grant number 01/1/28745.

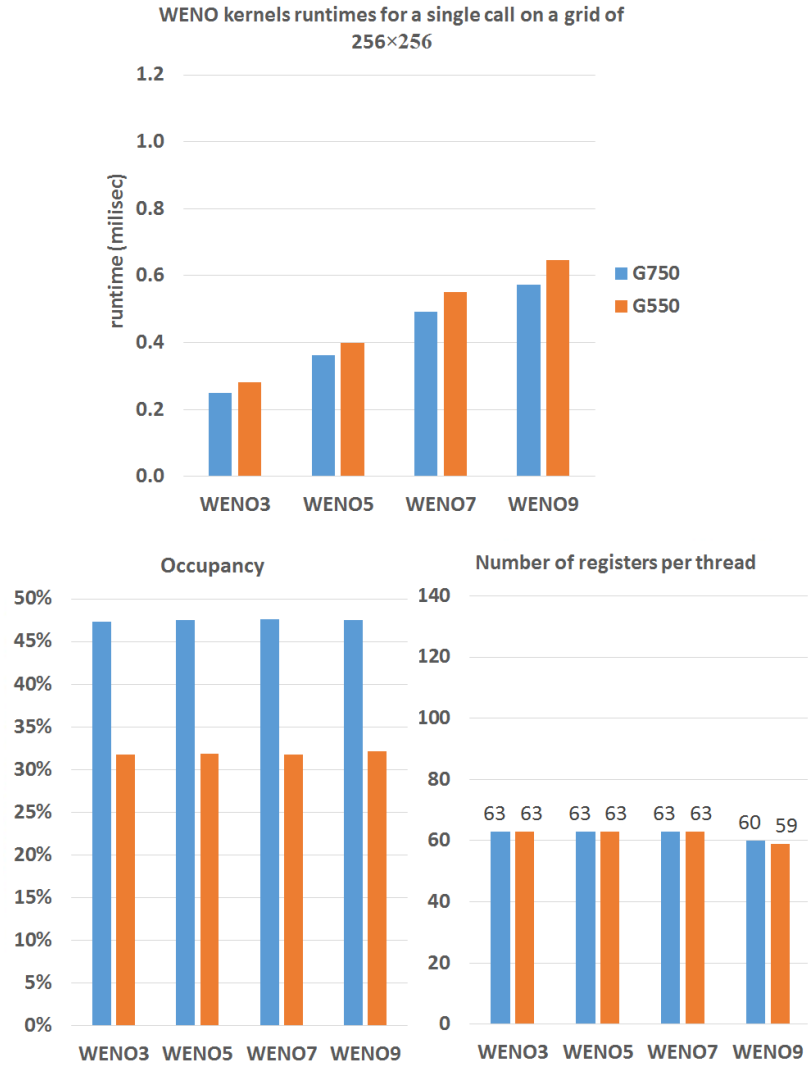

Figure 7 WENO kernels runtimes for a single call for both the GPUs after limiting the number of registers and reducing the amount of shared memory by reducing the number of threads per block from 256 to 128

Table 3: Comparison of Single- and Double-GPU runtimes after limiting the number of registers and reducing the amount of shared memory by reducing the number of threads per block from 256 to 128.

Grid	Single-GPU time	Double-GPU time	speed-up	Single-GPU time	Double-GPU time	speed-up
		WENO3			WENO5	
256×256	7.98	4.13	1.93	10.01	4.92	2.03
512×512	28.95	13.37	2.17	37.68	16.23	2.32
1024×1024	114.98	47.94	2.40	150.62	58.95	2.56
		WENO7			WENO9	
256×256	11.52	5.61	2.05	13.75	6.20	2.2
512×512	43.91	19.47	2.26	52.69	20.87	2.52
1024×1024	175.85	71.54	2.46	211.27	77.65	2.72

8. References

[1] H. P. Le, J. L. Cambier, L. K. Cole, GPU-based flow simulation with detailed chemical kinetics, *Computer Physics Communications,* Vol. 184, No. 3, pp. 596-606, 2013.

[2] A. Khajeh-Saeed, J. Blair Perot, Direct numerical simulation of turbulence using GPU accelerated supercomputers, *Journal of Computational Physics,* Vol. 235, pp. 241-257, 2013.

[3] B. Tutkun, F. O. Edis, A GPU application for high-order compact finite difference scheme, *Computers and Fluids,* Vol. 55, pp. 29-35, 2012.

[4] J. A. Ekaterinaris, High-order accurate, low numerical diffusion methods for aerodynamics, *Progress in Aerospace Sciences,* Vol. 41, No. 3-4, pp. 192-300, 2005.

[5] G. S. Jiang, C. W. Shu, Efficient implementation of weighted ENO schemes, *Journal of Computational Physics*, Vol. 126, No. 1, pp. 202-228, 1996.

[6] X. D. Liu, S. Osher, T. Chan, Weighted Essentially Non-oscillatory Schemes, *Journal of Computational Physics,* Vol. 115, No. 1, pp. 200-212, 1994.

[7] V. Esfahanian, K. Hejranfar, H. M. Darian, Implementation of high-order compact finite-difference method to parabolized Navier-Stokes schemes, *International Journal for Numerical Methods in Fluids*, Vol. 58, No. 6, pp. 659-685, 2008.

[8] K. Heiranfar, V. Esfahanian, H. M. Darian, On the use of high-order accurate solutions of PNS schemes as basic flows for stability analysis of hypersonic axisymmetric flows, *Journal of Fluids Engineering, Transactions of the ASME,* Vol. 129, No. 10, pp. 1328-1338, 2007.

[9] S. K. Lele, Compact finite difference schemes with spectral-like resolution, *Journal of Computational Physics,* Vol. 103, No. 1, pp. 16-42, 1992.

[10] H. Mahmoodi Darian, V. Esfahanian, K. Hejranfar, A shock-detecting sensor for filtering of high-order compact finite difference schemes, *Journal of Computational Physics,* Vol. 230, No. 3, pp. 494-514, 2011.

[11] A. S. Antoniou, K. I. Karantasis, E. D. Polychronopoulos, J. A. Ekaterinaris, Acceleration of a finite-difference WENO scheme for large-scale simulations on many-core architectures, in *Proceeding of*.

[12] V. Esfahanian, H. M. Darian, S. M. Iman Gohari, Assessment of WENO schemes for numerical simulation of some hyperbolic equations using GPU, *Computers and Fluids,* Vol. 80, No. 1, pp. 260-268, 2013.

[13] H. M. Darian, V. Esfahanian, Assessment of WENO schemes for multi-dimensional Euler equations using GPU, *International Journal for Numerical Methods in Fluids,* Vol. 76, No. 12, pp. 961-981, 2014.

[14] S. C. Lo, G. A. Blaisdell, A. S. Lyrintzis, High-order shock capturing schemes for turbulence calculations, *International Journal for Numerical Methods in Fluids,* Vol. 62, No. 5, pp. 473-498, 2010.

[15] H. C. Yee, N. D. Sandham, M. J. Djomehri, Low-Dissipative High-Order Shock-Capturing Methods Using Characteristic-Based Filters, *Journal of Computational Physics,* Vol. 150, No. 1, pp. 199-238, 1999.

[16] M. Khoshab, A. A. Dehghan, V. Esfahanian, H. M. Darian, Numerical assessment of a shock-detecting sensor for low dissipative high-order simulation of shock-vortex interactions, *International Journal for Numerical Methods in Fluids,* Vol. 77, No. 1, pp. 18-42, 2015.

[17] N. Corporation, 2010, *NVIDIA CUDA C Programming Guide*,