ENGINEERING APPLICATIONS OF COMPUTATIONAL FLUID MECHANICS
2023, VOL. 17, NO. 1, 2210196
https://doi.org/10.1080/19942060.2023.2210196

Computationally efficient GPU based NS solver for two dimensional high-speed inviscid and viscous compressible flows

Muhammad Naveed Akhtar[a], Kamran Rasheed Qureshi[b], Muhammad Hanif Durad[a], Anila Usman[a], Syed Muhammad Mohsin[c,d], Band Shahab[e] and Amirhosein Mosavi[f,g]

[a]Department of Computer and Information Science, Pakistan Institute of Engineering and Applied Sciences (PIEAS), Islamabad, Pakistan; [b]Department of Mechanical Engineering (DME), Pakistan Institute of Engineering and Applied Sciences (PIEAS), Islamabad, Pakistan; [c]Department of Computer Science, COMSATS University Islamabad, Islamabad, Pakistan; [d]College of Intellectual Novitiates (COIN), Virtual University of Pakistan, Lahore, Pakistan; [e]Future Technology Research Center, Yunlin University, Yunlin, Taiwan; [f]University of Public Service, Budapest, Hungary; [g]John von Neumann Faculty of Informatics, Budapest, Hungary

ABSTRACT
In this study, we proposed a novel GPU-based solution for modelling two-dimensional inviscid and viscous compressible supersonic/hypersonic flows. Texture and surface pointers are used to access GPU memory locations. For effective and efficient use of surface pointers, we grouped multiple 2D arrays referenced and indexed by a single 3D surface pointer. To enable the proposed solver for double-precision calculations, two consecutive 32-bit memory locations were grouped to maintain the efficiency of surface pointers while taking advantage/accuracy of 64-bit calculations. Resolving data and computation dependencies for parallel applications is another complex task that is the focus of this study. Computation dependencies have been solved by using multiple mutually synchronized GPU kernels and executing them sequentially using the GPU default stream to ensure that all relevant data is available to the threads or computed before they actually use it. Consequently, there is no intra-core data dependency in our proposed approach, while inter-core data dependency is successfully solved by stringing multiple kernels together. Using NVIDIA GTX 660 GPUs, we achieved 20x speedup compared to traditional Core i5® computers. This speedup is the result of the Surface Pointer's GPU capabilities for double precision computations. The simulation results are also consistent with the experimental and numerical results of this study.

ARTICLE HISTORY
Received 5 December 2022
Accepted 22 April 2023

KEYWORDS
Computational mathematics; computational complexity; Navier-Stokes equations; graphic processing unit; shock wave; surface pointers

1. Introduction

Computational fluid dynamics (CFD) is the study of gas or fluid flow through software modelling of the underlying physics. The Navier-Stokes (NS) equation is the basic equation for CFD, which describes the relationship between pressure, velocity, density, and temperature for fluids in motion (Chandar et al., 2013). Another important factor for fluids moving at high velocity is the occurrence of shock waves in the flow field (Hoffmann & Chiang, 2000). A shock wave is generated when the fluid, gas, or plasma is flowing faster than the speed of sound. When a shock wave is generated, an almost discontinuous change in the temperature, density, and pressure of the fluid is observed. When modelling and simulating fluid flow at higher velocities, it is important to consider such parameters (Hoffmann & Chiang, 2000).

As the geometry on CFD becomes more complex, the size of the computational problem increases significantly (Soukov, 2021; Wang et al., 2021). In addition, capturing complex flow features further increases the computational complexity by requiring more grid points to be generated around the domain. These features include shock waves, vortex structures, interactions between the boundary layer and flow separation, etc. Chen et al. (2007), Ladeinde and Nearon (1997), and Nielsen (2004). In addition, the underlying numerical methods require a large number of iterations to converge. CFD Simulations on a modern multi-core computer system can take hours or days (Zhai & Chen, 2003). One way to speed up this process is to perform computations in parallel, which requires a highly scalable parallel computing platform such as a graphics processing unit (GPU).

GPUs are the most compact and massively parallel hardware. Modern GPUs are equipped with thousands of cores operating in parallel and very high

CONTACT Muhammad Naveed Akhtar naveed@pieas.edu.pk; Band Shahab shahab@yuntech.edu.tw; Amirhosein Mosavi amir.mosavi@bgk.uni-obuda.hu

© 2023 The Author(s). Published by Informa UK Limited, trading as Taylor & Francis Group.
This is an Open Access article distributed under the terms of the Creative Commons Attribution License (http://creativecommons.org/licenses/by/4.0/), which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited. The terms on which this article has been published allow the posting of the Accepted Manuscript in a repository by the author(s) or with their consent.

memory bandwidth (Lai et al., 2020). GPUs are based on the streaming model of processing and are capable of parallelizing computationally intensive tasks (Glaskowsky, 2009; Weiskopf, 2007). The floating point performance and memory bandwidth of GPUs are several times higher than a conventional CPU (Cuda, 2015). Existing applications that require the computational power of GPUs need to be optimized and rewritten for GPUs using CUDA or OpenCL, etc. Cuda (2015) and Cuda (2020).

Many studies have been conducted to accelerate CFD simulations with GPUs, but most of them considered only incompressible flows. For example, Chandar et al. (2013) presented a GPU-based incompressible NS-solver for motion over fixed grids. Wang et al. (2020) proposed memory access patterns for higher order CFD stencil computation. Wang et al. (2014) solved the incompressible 3D NS equation using a combination of CPU and GPU. Thibault and Senocak (2009) simulated incompressible flows of GPU using the NS equation and CUDA. Griebel and Zaspel (2010) simulated an incompressible 3D two-phase flow on multiple GPUs. Tölke and Krafczyk (2008) presented a TeraFLOPs calculation for GPUs, but for the less computationally intensive Lattice Boltzmann method. Zuo and Chen (2010) accelerated the simulation of the NS equation along with the transport equation for airflow in HVAC systems with GPU. Goddeke et al. (2009) have proposed GPU acceleration for FEM NS solver. Rogers and Kwak (1990) used an updraft difference scheme to solve the NS equation accurately in time. Z.H. Ma. et al. have simulated compressible but non-viscus flows on GPU using the meshless method (Ma et al., 2014). Dequan Xu et al. also simulated compressible but non-viscous super-sonic flows on multi-GPU platform (Xu et al., 2021). Zhengyu Tian et al. has used GPUs for simulating supersonic flows on hybrid grids (Tian et al., 2020) and established the accuracy of GPU computation due to its double precision. Other related work could also be found in Adeeb and Ha (2022), Kun and Xiaowen (2022), Kale et al. (2022), Kale, Sharma et al. (2022), Lai et al. (2020), Shao et al. (2022), Wei et al. (2020), and Weng et al. (2021).

To the best of our knowledge, there is no work reporting results from GPU-based NS solvers which consider compressible, viscus and for supersonic/hypersonic flows at the same time. Moreover, there are no reports on using the surface pointer capability of GPUs for such simulations. In this work, we have presented a GPU-based solver for viscus, compressible supersonic and hypersonic flows and its simulation results. We have implemented the modified Runge-Kutta method (Damevin & Hoffmann, 2001) together with the Harten-Yee upwind total variation diminishing (HY-TVD) scheme (Yee, 1989) for shock wave detection on GPU by exploiting the power of surface pointers in an innovative and efficient way. To the best of our knowledge, this is the very first implementation of high-speed, viscous, and compressible flows on GPUs using their surface pointer capability. Main contributions of this study are given below.

(1) We grouped multiple 2D arrays referenced and indexed by a single 3D surface pointer to leverage the efficiency of surface pointers on GPU.
(2) We proposed 3D arrays of size X×Y×N, where X and Y are the grid dimensions in a problem and N is the number of 2D arrays packed in the collection, to allow easier and more effective access to GPU memory space.
(3) For double precision calculations, 2 consecutive 32-bit memory locations have been grouped to preserve the efficiency of surface pointers while taking advantage/accuracy of 64-bit calculations.
(4) Multiple mutually synchronized GPU kernels are implemented to solve the problem of data and computation dependencies in parallel applications/computations.

The rest of the paper is organized as follows. Section 2 describes the governing equations for simulation and explains numerical method in detail. Implementation details of our proposed GPU-based NS solver are given in Section 3. Test cases for performance evaluation are detailed in Section 4 and results are discussed in Section 5. Finally, Section 6 concludes this study.

2. Mathematical modelling

2.1. Governing equations

In the present study, the two-dimensional NS equation (Hoffmann & Chiang, 2000), which contains continuity, momentum and energy equations, is solved to simulate viscous compressible flows with high velocity. NS Equation can be expressed in the strong conservation form as shown in Equation (1) and described in Equations (2) and (3).

$$\frac{\partial Q}{\partial t} + \frac{\partial E}{\partial x} + \frac{\partial F}{\partial y} = \frac{\partial E_v}{\partial x} + \frac{\partial F_v}{\partial y} \tag{1}$$

Where;

$$Q = \begin{bmatrix} \rho \\ \rho u \\ \rho v \\ \rho e_t \end{bmatrix}, \quad E = \begin{bmatrix} \rho u \\ \rho u^2 + p \\ \rho u v \\ (\rho e_t + p)\, u \end{bmatrix},$$

$$F = \begin{bmatrix} \rho v \\ \rho v u \\ \rho v^2 + p \\ (\rho e_t + p)\, v \end{bmatrix} \tag{2}$$

and,

$$E_v = \begin{bmatrix} 0 \\ \tau_{xx} \\ \tau_{xy} \\ u\tau_{xx} + v\tau_{xy} - q_x \end{bmatrix}, \quad F_v = \begin{bmatrix} 0 \\ \tau_{yx} \\ \tau_{yy} \\ u\tau_{yx} + v\tau_{yy} - q_y \end{bmatrix} \tag{3}$$

Here Q is the vector of conservative variables, and E, F show inviscid fluxes, and E_v, F_v are viscous fluxes. All other variables denote their conventional meanings. For inviscid flow simulation, the viscous fluxes in the right side of Equation (1) become zero. The above Equations (1) –(3) are transformed from Cartesian (x, y) to curvilinear coordinates (ξ, η) and can be expressed in flux vector form as shown in Equation (4) and described in Equations (5) and (6) in the following.

$$\frac{\partial \bar{\mathrm{Q}}}{\partial t} + \frac{\partial \bar{\mathrm{E}}}{\partial \xi} + \frac{\partial \bar{\mathrm{F}}}{\partial \eta} = \frac{\partial \bar{\mathrm{E}}_v}{\partial \xi} + \frac{\partial \bar{\mathrm{F}}_v}{\partial \eta} \tag{4}$$

Where;

$$\begin{aligned} \bar{\mathrm{Q}} &= \frac{\mathrm{Q}}{J} \\ \bar{E} &= \frac{1}{J}\left(\xi_t \mathrm{Q} + \xi_x \mathrm{E} + \xi_y \mathrm{F}\right) \\ \bar{F} &= \frac{1}{J}\left(\eta_t \mathrm{Q} + \eta_x \mathrm{E} + \eta_y \mathrm{F}\right) \end{aligned} \tag{5}$$

and,

$$\begin{aligned} \bar{E}_v &= \frac{1}{J}\left(\xi_x \mathrm{E}_v + \xi_y \mathrm{F}_v\right) \\ \bar{F}_v &= \frac{1}{J}\left(\eta_x \mathrm{E}_v + \eta_y \mathrm{F}_v\right) \end{aligned} \tag{6}$$

Where J denotes the Jacobian of transformation from (x, y) to (ξ, η) coordinates. Further details of the transformation can be found in Hoffmann and Chiang (2000). In the present study, air is assumed to be a perfect gas, as shown in Equation (7). Therefore;

$$\rho e_t = \frac{1}{2}\rho\left(u^2 + v^2\right) + p/(\gamma - 1) \tag{7}$$

Where, for air, $\gamma = 1.4$ The absolute viscosity of the air is computed by Sutherland's law, described in Equation (8).

$$\mu_\infty = \frac{1.4586 \times 10^{-6} T_\infty^{1.5}}{T_\infty + 110.4} \tag{8}$$

2.2. Numerical method

Equations described in previous section are hereby solved by using fourth order modified Runge-Kutta scheme (Damevin & Hoffmann, 2001), as shown in Equation (9).

$$\begin{aligned} \bar{Q}^1_{i,j} &= \bar{Q}^n_{i,j} - \frac{1}{4}\Delta\tau\, L\bar{Q}^n_{i,j} \\ \bar{Q}^2_{i,j} &= \bar{Q}^n_{i,j} - \frac{1}{3}\Delta\tau\, L\bar{Q}^1_{i,j} \\ \bar{Q}^3_{i,j} &= \bar{Q}^n_{i,j} - \frac{1}{2}\Delta\tau\, L\bar{Q}^2_{i,j} \\ \bar{Q}^{n+1}_{i,j} &= \bar{Q}^n_{i,j} - \Delta\tau\, L\bar{Q}^3_{i,j} \end{aligned} \tag{9}$$

Where $L\overline{Q}$ operator consists of convective part and viscous part and has been discretized by 2nd order central difference approximation. Equation (9) is computationally more efficient than the classical Runge-Kutta method, since it is not necessary to store the previous three steps in order to compute the n+1 step. Since this method is a straight-order method, undesirable oscillations may occur in the vicinity of shock waves or other flow gradients. However, these oscillations can be suppressed by adding an artificial damping mechanism. In Yee (1989), the Harten-Yee's formulation, total variation diminishing (TVD), was used as an artificial damping term. There are two different ways to introduce the damping mechanism into the scheme. One way is to add it at each step of the modified Runge-Kutta scheme. The second approach is to add it in the post-processor phase after computing $n+1$ steps. In this study, the latter approach was used because it requires less computational effort and time. The post-processor step with the dissipation part of the TVD scheme can be expressed as shown in Equation (10).

$$\begin{aligned} \bar{Q}^{n+1}_{i,j} = \bar{Q}^{n+1}_{i,j} &- \frac{1}{2}\frac{\Delta\tau}{\Delta\xi}\left[(X_A)^n_{i+1/2,j}(\Phi_\xi)^n_{i+1/2,j}\right] \\ &+ \frac{1}{2}\frac{\Delta\tau}{\Delta\xi}\left[(X_A)^n_{i-1/2,j}(\Phi_\xi)^n_{i-1/2,j}\right] \\ &- \frac{1}{2}\frac{\Delta\tau}{\Delta\eta}\left[(X_B)^n_{i,j+1/2}(\Phi_\eta)^n_{i,j+1/2}\right] \\ &+ \frac{1}{2}\frac{\Delta\tau}{\Delta\eta}\left[(X_B)^n_{i,j-1/2}(\Phi_\eta)^n_{i,j-1/2}\right] \end{aligned} \tag{10}$$

The formulation of Harten-Yee's upwinded TVD scheme in generalized coordinates, as per (Yee, 1989), is shown in following Equations (11) and (12).

$$\begin{aligned} (\Phi_\xi)^n_{i+1/2,j} = \sigma\left[(\alpha_\xi)_{i+1/2,j}\right]\left[(G_\xi)_{i+1,j} + (G_\xi)_{i,j}\right] \\ - \psi\left[(\alpha_\xi)_{i+1/2,j} - (\beta_\xi)_{i+1/2,j}\right](\delta_\xi)_{i+1/2,j} \\ (\Phi_\xi)^n_{i-1/2,j} = \sigma\left[(\alpha_\xi)_{i-1/2,j}\right]\left[(G_\xi)_{i,j} + (G_\xi)_{i-1,j}\right] \end{aligned}$$

$$
\begin{aligned}
&\quad - \psi\left[\left(\alpha_{\xi}\right)_{i-1/2,j} - \left(\beta_{\xi}\right)_{i-1/2,j}\right](\delta_{\xi})_{i-1/2,j} \\
(\Phi_{\eta})^{n}_{i,j+1/2} &= \sigma\left[\left(\alpha_{\eta}\right)_{i,j+1/2}\right]\left[\left(G_{\eta}\right)_{i,j+1} + \left(G_{\eta}\right)_{i,j}\right] \\
&\quad - \psi\left[\left(\alpha_{\eta}\right)_{i,j+1/2} - \left(\beta_{\eta}\right)_{i,j+1/2}\right](\delta_{\eta})_{i,j+1/2} \\
(\Phi_{\eta})^{n}_{i,j-1/2} &= \sigma\left[\left(\alpha_{\eta}\right)_{i,j-1/2}\right]\left[\left(G_{\eta}\right)_{i,j} + \left(G_{\eta}\right)_{i,j-1}\right] \\
&\quad - \psi\left[\left(\alpha_{\eta}\right)_{i,j-1/2} - \left(\beta_{\eta}\right)_{i,j-1/2}\right](\delta_{\eta})_{i,j-1/2}
\end{aligned} \tag{11}
$$

$$
\begin{aligned}
\sigma\left(\alpha_{\xi}\right) &= \frac{1}{2}\psi\left(\alpha_{\xi}\right) + \frac{\Delta t}{\Delta \xi}\left(\alpha_{\xi}\right)^{2} \\
\sigma\left(\alpha_{\eta}\right) &= \frac{1}{2}\psi\left(\alpha_{\eta}\right) + \frac{\Delta t}{\Delta \eta}\left(\alpha_{\eta}\right)^{2}
\end{aligned} \tag{12}
$$

The function $\psi(y)$ is the entropy correction function and is expressed in Equation (13). Further, mathematical descriptions of $(\delta_{\xi})_{i+1/2,j}$, $(\delta_{\eta})_{i,j+1/2}$, $(\beta_{\xi})_{i+\frac{1}{2},j}$ and $(\beta_{\eta})_{i,j+1/2}$ are given in Equations (14) and (15).

$$
\psi(y) = \begin{cases} |y| & \text{if } |y| > \varepsilon \\ \dfrac{y^{2} + \varepsilon^{2}}{2\varepsilon} & \text{if } |y| \le \varepsilon \end{cases} \tag{13}
$$

$$
\begin{aligned}
(\delta_{\xi})_{i+1/2,j} &= (X_{A})^{-1}_{i+\frac{1}{2},j}\left(J^{-1}_{i+\frac{1}{2},j}\right)\left(Q_{i+1,j} - Q_{i,j}\right) \\
(\delta_{\eta})_{i,j+1/2} &= (X_{B})^{-1}_{i,j+\frac{1}{2}}\left(J^{-1}_{i,j+1/2}\right)\left(Q_{i,j+1} - Q_{i,j}\right)
\end{aligned} \tag{14}
$$

$$
\begin{aligned}
&\left(\beta_{\xi}\right)_{i+\frac{1}{2},j} \\
&\quad = \sigma\left[\left(\alpha_{\xi}\right)_{i+\frac{1}{2},j}\right] \\
&\quad\quad \times \begin{cases} 0, & \text{for } (\delta_{\xi})_{i+\frac{1}{2},j} = 0 \\ \dfrac{\left(G_{\xi}\right)_{i+1,j} - \left(G_{\xi}\right)_{i,j}}{\left(\delta_{\xi}\right)_{i+\frac{1}{2},j}}, & \text{for } \left(\delta_{\xi}\right)_{i+\frac{1}{2},j} \neq 0 \end{cases} \\
&\left(\beta_{\eta}\right)_{i,j+1/2} \\
&\quad = \sigma\left[\left(\alpha_{\eta}\right)_{i,j+1/2}\right] \\
&\quad\quad \times \begin{cases} 0, & \text{for } (\delta_{\eta})_{i,j+1/2} = 0 \\ \dfrac{\left(G_{\eta}\right)_{i,j+1} - \left(G_{\eta}\right)_{i,j}}{\left(\delta_{\eta}\right)_{i,j+1/2}}, & \text{for } \left(\delta_{\eta}\right)_{i,j+1/2} \neq 0 \end{cases}
\end{aligned} \tag{15}
$$

Here G is a limiter which has been computed using Equation (16).

$$
G_{i+1/2,j} = \min\bmod \begin{bmatrix} 2\delta_{i-1/2,j}, \quad 2\delta_{i+1/2,j} \\ 2\alpha_{i+3/2,j} \\ \dfrac{1}{2}\left(\delta_{i-1/2,j} + \delta_{i+3/2,j}\right) \end{bmatrix} \tag{16}
$$

3. Implementation details of proposed GPU-based NS solver

As described earlier, GPUs enable massively parallel implementation of the numerical solution to the Navier-Stokes equation. In this section, we explain our implementation in this regard. A simplified sequence of operations to solve the equations under Section 2 is detailed in Algorithm 1. About 30 two-dimensional arrays of the same size are needed to accommodate all variables. When multiple blocks are modelled simultaneously with these equations, the number of arrays increases even further. All these arrays had to be accommodated in the GPU's global memory and are updated accordingly. The fastest way to access memory locations on the GPU is through textures and surface pointers when used intelligently (Cuda, 2014). Typically, a surface pointer binds to a single CUDA array, so we need about 30 surface pointers to manipulate all of these arrays. According to Table 13 of Cuda (2014), there is an upper limit of 8 surface pointers to global memory for GPUs with processing power 2.x and 16 for GPUs with higher processing power. Clearly, this number is not sufficient to directly exploit the efficiency of surface pointers. If we still want to take advantage of the performance and efficiency of surface pointers, the only way is to use some kind of coordinate transformation technique and show multiple variable arrays with few surface pointers.

In this study, multiple 2D arrays are grouped to be referenced by a single 3D surface pointer and indexed accordingly. This technique is explained in Figures 1 and 2. All 2D arrays needed for the problem are packed into a single 3D array and a surface pointer references this common block. The size of the 3D array is given by X×Y×N, where X and Y are the grid dimensions in a problem and N is the number of 2D arrays packed in the collection. CUDA Surfaces use a 32-bit computing architecture by default. To make it suitable for double-precision computations, 2 consecutive 32-bit memory locations have been grouped to preserve the efficiency of surface pointers while taking advantage/accuracy of 64-bit computations.

Resolving data and computation dependencies is another complex task for parallel applications. These dependencies have been resolved by using multiple GPU kernels and executing them sequentially using the GPU's default stream. The reason for this is to ensure that all relevant data is available to the threads or computed before they actually use it. Also, a grid point of the geometry and all related computations for that point have been assigned to a single thread. With this approach, no thread has to wait for data computed by a neighbouring thread. In other words, it can be safely said that there is no

Algorithm 1 Simplified sequence of operations

1: $\{\xi_x, \xi_y, \eta_x, \eta_y\} \longleftarrow \{InputFile\}$
2: $GPU \longleftarrow \{\xi_x, \xi_y, \eta_x, \eta_y\}$
3: $\{P, T, U, V, \rho\} \longleftarrow$ Initialize variables on GPU( )
4: **while** Iteration $\leq$ maxiterations **do**
5: $\quad \Delta t_l \leftarrow$ Local time step $(\xi_x, \xi_y, \eta_x, \eta_y)$
6: $\quad Q^{1\ldots4} \longleftarrow$ IndVar to Q (P, T, U, V, ρ)
7: $\quad Tx^{1\ldots4} \longleftarrow$ Calculate TVD(x) $(\xi_x, \xi_y, \eta_x, \eta_y)$
8: $\quad Ty^{1\ldots4} \longleftarrow$ Calculate TVD(y) $(\xi_x, \xi_y, \eta_x, \eta_y)$
9: $\quad Qn^{1\ldots4} \longleftarrow$ RK4 Stage 1 $(\alpha, \beta, \Delta t_l, P, T, U, V, \rho, Q^{1\ldots4})$
10: $\quad \{P, T, U, V, \rho\} \longleftarrow$ Q to IndVar ($Qn^{1\ldots4}$)
11: $\quad \{P, T, U, V, \rho\} \longleftarrow$ Boundary Conditions (P, T, U, V, ρ)
12: $\quad Qn^{1\ldots4} \longleftarrow$ RK4 Stage 2 $(\alpha, \beta, \Delta t_l, P, T, U, V, \rho, Q^{1\ldots4})$
13: $\quad \{P, T, U, V, \rho\} \longleftarrow$ Q to IndVar ($Qn^{1\ldots4}$)
14: $\quad \{P, T, U, V, \rho\} \longleftarrow$ Boundary Conditions (P, T, U, V, ρ)
15: $\quad Qn^{1\ldots4} \longleftarrow$ RK4 Stage 3 $(\alpha, \beta, \Delta t_l, P, T, U, V, \rho, Q^{1\ldots4})$
16: $\quad \{P, T, U, V, \rho\} \longleftarrow$ Q to IndVar ($Qn^{1\ldots4}$)
17: $\quad \{P, T, U, V, \rho\} \longleftarrow$ Boundary Conditions (P, T, U, V, ρ)
18: $\quad Qn^{1\ldots4} \longleftarrow$ RK4 Stage 4 $(\alpha, \beta, \Delta t_l, P, T, U, V, \rho, Q^{1\ldots4})$
19: $\quad \{P, T, U, V, \rho\} \longleftarrow$ Q to IndVar ($Qn^{1\ldots4}$)
20: $\quad \{P, T, U, V, \rho\} \longleftarrow$ Boundary Conditions (P, T, U, V, ρ)
21: $\quad$ Apply TVD ($Qn^{1\ldots4}, Tx^{1\ldots4}, Ty^{1\ldots4}$)
22: $\quad \{P, T, U, V, \rho\} \longleftarrow$ Q to IndVar ($Qn^{1\ldots4}$)
23: $\quad \{P, T, U, V, \rho\} \longleftarrow$ Boundary Conditions (P, T, U, V, ρ)
24: **end while**
25: $CPU \longleftarrow \{P, T, U, V, \rho\}$

intra-core data dependency in our proposed approach, while inter-core data dependency is successfully solved by queuing multiple kernels in order of computation to the default stream. The kernels from the standard stream are started in the order of the program and the next kernel starts its operation only when the previous one finishes. In this way, synchronization between kernels is achieved. Also, where necessary, two separate sets of variables were used for source and destination to avoid conflicts when reading and writing memory.

In total, 9 GPU kernels were used to compute different parts of the problem respectively, which are summarized in Table 1. All of these kernels are enqueued into the standard GPU stream, as shown in Figure 3. A 2D block distribution was used to decompose the problem. We used the NVIDIA C compiler (nvcc) from CUDA toolkit 6.5 to compile all the code. The host operating system was Fedora 20 with a memory of 4 GB. All programs run on an NVIDAI Geforce GTX 660 GPU.

```
surface<void, cudaSurfaceType3D> SOLTNs;

#define SOL_WRITE(var, value, I, J)                                  \
        surf3Dwrite((double) (value), SOLTNs,                        \
        (I) * sizeof(double), J, var,cudaBoundaryModeClamp)

__device__ double inline READ_SURFACE(int I, int J, int var)
{
    double value;
    surf3Dread(&value, SOLTNs, I * sizeof(double), J,
               var, cudaBoundaryModeClamp);
    return value;
}

//Variable Definition
//All 30 Variables are defined in similar fashion
#define iP 1
#define iT 2
#define iU 3

//Definition for reading variable from Surface Array
//All 30 Variables are defined in similar fashion
#define P(I,J) READ_SURFACE(I,J,1)
#define T(I,J) READ_SURFACE(I,J,2)
#define U(I,J) READ_SURFACE(I,J,3)
```

Figure 1. Setting up memory read/write through surface pointers.

```
//Example for Writing a variable to Surface Array
SOL_WRITE(iP, value, I, J);
SOL_WRITE(iT, value, I, J);
SOL_WRITE(iU, value, I, J);

//Example for Reading a variable from Surface Array
value = P(I,J);
value = T(I,J);
value = U(I,J);
```

Figure 2. Memory read/write example through surface pointers.

Table 1. List of kernels and their utilization in stated cases.

Kernel	Purpose	Case-1	Case-2	Case-3	Case-4
Kernel-1	RK4 routine (Viscous term)	✓	✓	–	✓
Kernel-1	RK4 routine (No Viscous term)	–	–	✓	–
Kernel-2	TVD(x) Calculation	✓	✓	✓	✓
Kernel-3	TVD(y) Calculation	✓	✓	✓	✓
Kernel-4	Q to IndVar	✓	✓	✓	✓
Kernel-5	Apply Boundry Conditions	✓	✓	✓	✓
Kernel-6	Local Δt calculation	✓	✓	✓	✓
Kernel-7	Apply TVD calculations	✓	✓	✓	✓
Kernel-8	IndVar to Q	✓	✓	✓	✓
Kernel-9	Communicatin on Interface	–	✓	–	–

4. Performance evaluation

The proposed scheme for accelerating the simulation of inviscid and viscous flows was considered for different geometries. The obtained results were also compared with existing experimental and numerical results (see

```
kernel_INIT<<<dimGrid, dimBlock>>>();
int K=0;
while(K < max_iterations){
    K = K+1;
    kernel_DelT<<<dimGrid, dimBlock>>>();
    kernel_IndVar_to_Q<<<dimGrid, dimBlock>>>();
    kernel_TVDx<<<dimGrid, dimBlock>>>();
    kernel_TVDy<<<dimGrid, dimBlock>>>();
    kernel_RK4_STAGE<<<dimGrid, dimBlock>>>(1/8.0, 4.0); //RK4 Stage 1
    kernel_Q_to_IndVar<<<dimGrid, dimBlock>>>();
    kernel_ApplyBC<<<dimGrid, dimBlock>>>(); //Boundary Conditions
    kernel_RK4_STAGE<<<dimGrid, dimBlock>>>(1/6.0, 3.0); //RK4 Stage 2
    kernel_Q_to_IndVar<<<dimGrid, dimBlock>>>();
    kernel_ApplyBC<<<dimGrid, dimBlock>>>(); //Boundary Conditions
    kernel_RK4_STAGE<<<dimGrid, dimBlock>>>(1/4.0, 2.0); //RK4 Stage 3
    kernel_Q_to_IndVar<<<dimGrid, dimBlock>>>();
    kernel_ApplyBC<<<dimGrid, dimBlock>>>(); //Boundary Conditions
    kernel_RK4_STAGE<<<dimGrid, dimBlock>>>(1/2.0, 1.0); //RK4 Stage 4
    kernel_Q_to_IndVar<<<dimGrid, dimBlock>>>();
    kernel_ApplyBC<<<dimGrid, dimBlock>>>(); //Boundary Conditions
    kernel_ApplyTVD<<<dimGrid, dimBlock>>>(1/2.0);
    kernel_Q_to_IndVar<<<dimGrid, dimBlock>>>();
    kernel_ApplyBC<<<dimGrid, dimBlock>>>(); //Boundary Conditions
    HANDLE_ERROR( cudaGetLastError() );
    HANDLE_ERROR( cudaDeviceSynchronize() );
}
HANDLE_ERROR( cudaMemcpy3D(&copyParamsOut) );
```

Figure 3. Example sequence of kernel execution for case-2 mentioned in Table 1.

Section 5). The details of the test cases used on GPU are given below.

4.1. Case-1: 5^0 wedge with cylindrical leading edge

Steady state, inviscid simulation around a two dimensional wedge having cylindrical leading edge was performed with conditions given in Table 2, found in Prabhu et al. (1989). The grid generated around wedge along with boundary conditions is shown in Figure 4(a) and it is composed of 162×101 points.

4.2. Case-2: compression corner

The simulation of a steady-state viscous hypersonic flow over a 7.5^0 compression corner (Simeonides et al., 1994) was performed under the conditions given in Table 2. The numerical simulation around the compression corner is described by the leading edge shock, the corner oblique shock, the interaction of the shock boundary layer, and the recirculation region at the corner. To capture these features, a 110×55 point mesh was generated around the compression corner. The resolution of the mesh near the wall was kept high to capture the flow gradients and recirculation region, as shown in Figure 4(b). Flow characteristics were extrapolated at the outflow and at the top of the region. Noslip and adiabatic boundary conditions were established at the wall.

4.3. Case-3: 2D half cylinder

A two-dimensional, steady-state, viscous hypersonic flow over a half-cylinder (Prabhu et al., 1989) is simulated with the parameters listed in Table 2. The numerical simulation of the hypersonic flow around such a blunt body is described by a strong detached bow shock and a subsonic flow in the stagnation region. Figure 4(c) shows a grid with 55×85 points around the half-cylinder. To calculate the heat flux at the stagnation point, the spacing of the first grid point was set to 1.0×10^{-6} m. The solution was started under the free flow conditions. At the outflow, the flow variables were extrapolated. At the wall, a slip-free boundary condition was applied.

4.4. Case-4: backward facing step

For the conditions listed in Table 2, a steady-state viscous high-speed simulation was performed at the backward 2D stage (Smith, 1967). For this configuration, two mesh blocks were generated, shown in Figure 4(d). The first

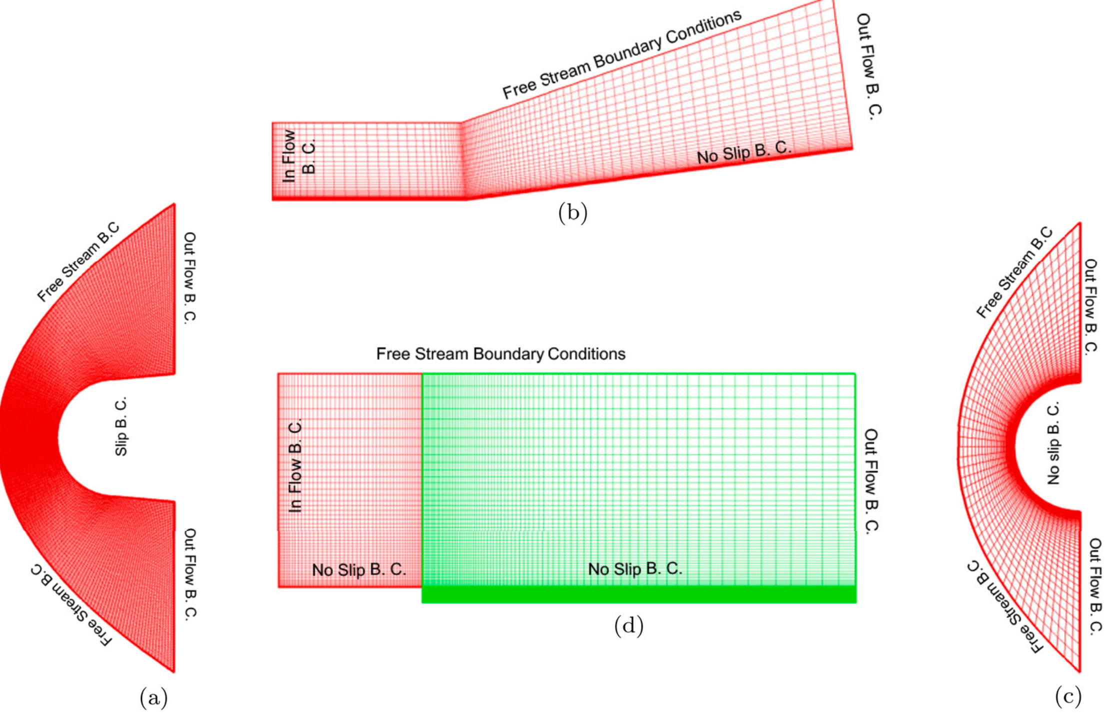

Figure 4. Mesh structure for all 4 problems. (a) Mesh for 5^0 wedge, (b) Mesh for compression corner, (c) Mesh for half cylinder and (d) Mesh for backward facing step.

Table 2. Parameter values for performance test cases.

Description	Symbol	Case-1	Case-2	Case-3	Case-4
Mach number	M_∞	15.0	6.0	16.34	5.0
Free stream density (kg/m^3)	ρ_∞	0.002	0.04142	–	–
Free stream temperature (K)	T_∞	295	57.3	52	–
Free stream pressure (N/m^2)	p_∞	–	–	82.95	–
Wall Temperature (*K*)	T_w	–	–	294.4	–
Total Temperature (*R*)	T_o	–	–	–	680°
Specific gas constant (J/kg · K)	R	287	287	287	287
Specific heat ratio	γ	1.4	1.4	1.4	1.4
Reynolds number (/m)	Re_L	–	0.8×10^6	3.94×10^6	3.94×10^6
Corner deflection angle	θ_c	–	7.5°	–	–
Cylinder radius (m)	r	–	–	0.038	–
Length forward plate (m)	L_1	–	–	–	0.1016
Length aft plate (m)	L_2	–	–	–	0.3048
Step height (m)	H	–	–	–	0.010922

block consists of 52×52 grid points, while the second block has 81×97 grid points. The supersonic flow field on the backward stage is characterized by an expansion wave at the corner, a separation region, a reattachment, and a re-compression wave. Free flow conditions were specified at the beginning of the solution. The flow characteristics were extrapolated at the outflow and at the top of the two blocks. Slip-free and adiabatic boundary conditions were specified at both walls.

All cases described above use the GPU kernel execution order defined in Algorithm 1. The only exception is the backward steps case, where 2 blocks are actually used. In this case, steps 5 through 23 of Algorithm 1 are repeated twice (i.e. once for each block). Another important difference is that the 5^0 wedge geometry is solved using the Euler equation, which does not require calculations of the viscous term. This change is explained in Algorithms 2 and 3 (One stage of the RK4 method). In all cases, Algorithm 2 is used for the calculations in all stages of the RK4 method, while the 5^0 wedge geometry uses Algorithm 3 for all stages of the RK4 method. The value of α used for the modified RK4 method is 1/8, 1/6, 1/4, 1/2 and the value of β is 4, 3, 2, 1 for all 4 stages (Damevin & Hoffmann, 2001).

Algorithm 2 Simplified one Stage of RK4 Method (viscous term)

1: $\delta\xi, \delta\eta \leftarrow$ CONSTANTS.
2: $\alpha, \beta, \Delta t_l, P, T, U, V, \rho, Q^{1\ldots4} \leftarrow$ INPUT.
3: $V^{2\ldots4} \leftarrow$ Viscous (U, V, T).
4: $E^{1\ldots4} \leftarrow$ Flux E (P, U, V, T, ρ).
5: $F^{1\ldots4} \leftarrow$ Flux F (P, U, V, T, ρ).
6: $\xi_c \leftarrow \alpha \times \frac{\Delta t_l}{\delta\xi}$
7: $\eta_c \leftarrow \alpha \times \frac{\Delta t_l}{\delta\eta}$
8: $Qn^1 \leftarrow Q^1 - \xi_c\Delta E^1 - \eta_c\Delta F^1$
9: $Qn^{2\ldots4} \leftarrow Q^{2\ldots4} - \xi_c\Delta E^{2\ldots4} - \eta_c\Delta F^{2\ldots4} + \frac{V^{2\ldots4}}{\beta}$
10: OUTPUT $\leftarrow Qn^{1\ldots4}$

Algorithm 3 Simplified one Stage of RK4 Method (5^0 wedge)

1: $\delta\xi, \delta\eta \leftarrow$ CONSTANTS.
2: $\alpha, \Delta t_l, P, T, U, V, \rho, Q^{1\ldots4} \leftarrow$ INPUT.
3: $E^{1\ldots4} \leftarrow$ Flux E (P, U, V, T, ρ).
4: $F^{1\ldots4} \leftarrow$ Flux F (P, U, V, T, ρ).
5: $\xi_c \leftarrow \alpha \times \frac{\Delta t_l}{\delta\xi}$
6: $\eta_c \leftarrow \alpha \times \frac{\Delta t_l}{\delta\eta}$
7: $Qn^{1\ldots4} \leftarrow Q^{1\ldots4} - \xi_c\Delta E^{1\ldots4} - \eta_c\Delta F^{1\ldots4}$
8: OUTPUT $\leftarrow Qn^{1\ldots4}$

5. Results and discussion

As mentioned earlier, a total of 4 different case geometries were simulated using our proposed NS solver on GPU. The structured mesh for each of the geometries is shown in Figure 4. The geometries named compression corner, 5^0 wedge and half cylinder were modelled as single block mesh, while the backward step was modelled with 2 blocks due to the presence of corner in the domain. The simulation results for the density profile and mach number for each case are shown in Figures 5(a,b), 6(a,b), 7(a,b), and 8(a,b). The simulated results for these geometries were also compared with the available experimental or numerical results and good agreement was found, as shown in Figures 9(a,b) and 10(a–d). The time analysis and comparison of all cases on GPU and CPU for 10 iterations is shown in Figure 11. The individual time taken for each kernel is shown in Figure 12. The time comparison for 10k iterations is shown in Figure 13 and the speedup achieved in each case is given in Figure 14.

Figure 11 compares the GPU and CPU time required for 10 iterations, considering all 4 geometries. The GPU total time is the sum of kernel time and memory copy time. The memory copy time is negligible because the memory copy is performed only once before the start of the computation and then the results are copied back to CPU after the completion of the iterations. In all 4 cases, significantly less time was required on the GPU than on CPU. The geometry for the 5^0 wedge contains

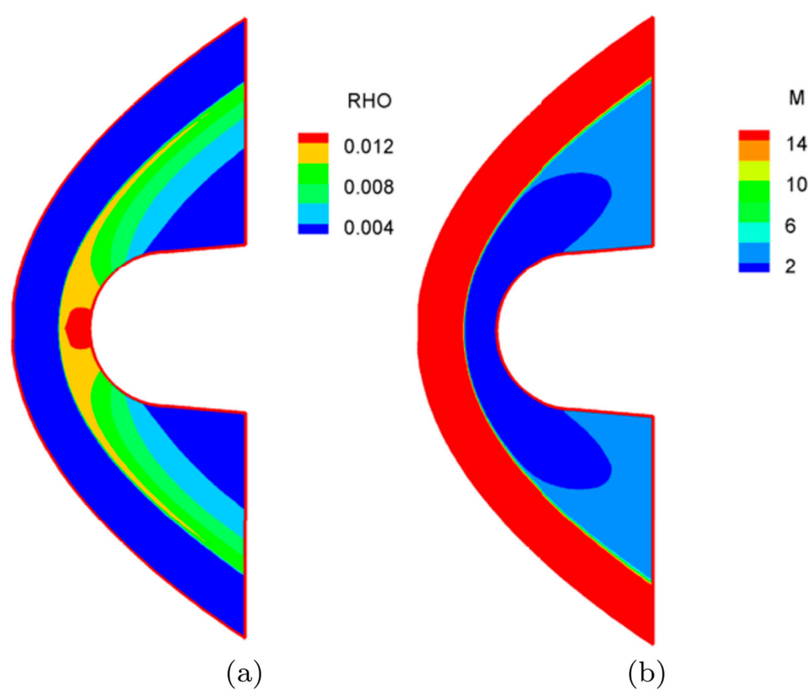

Figure 5. Simulation results for 5^0 wedge geometry. (a) Density profile and (b) Mach no. profile.

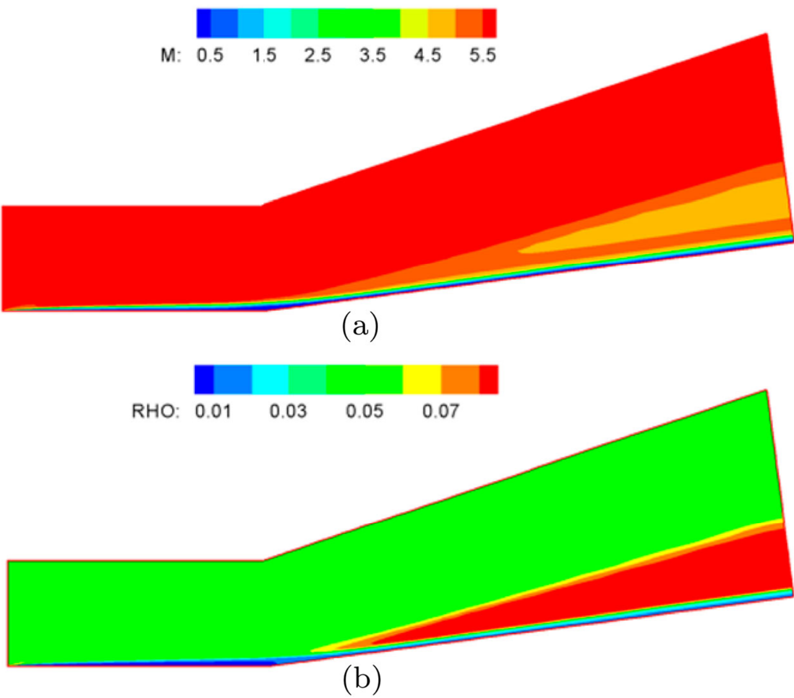

Figure 6. Simulation results for compression corner. (a) Density profile and (b) Mach no. profile.

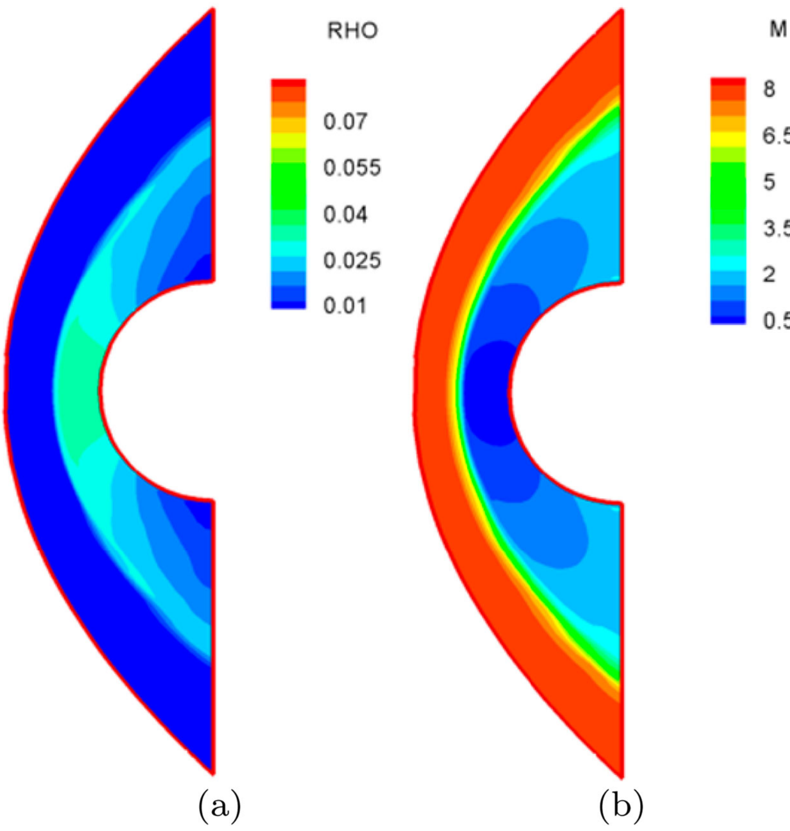

Figure 7. Simulation results for 2D half cylinder. (a) Density profile and (b) Mach no. profile.

the largest number of points to compute, but it too takes less time than the 2D half cylinder which has a smaller number of points to compute. This is because there are no viscous terms that require additional computation time. This difference in time is evident in the next Figure 12.

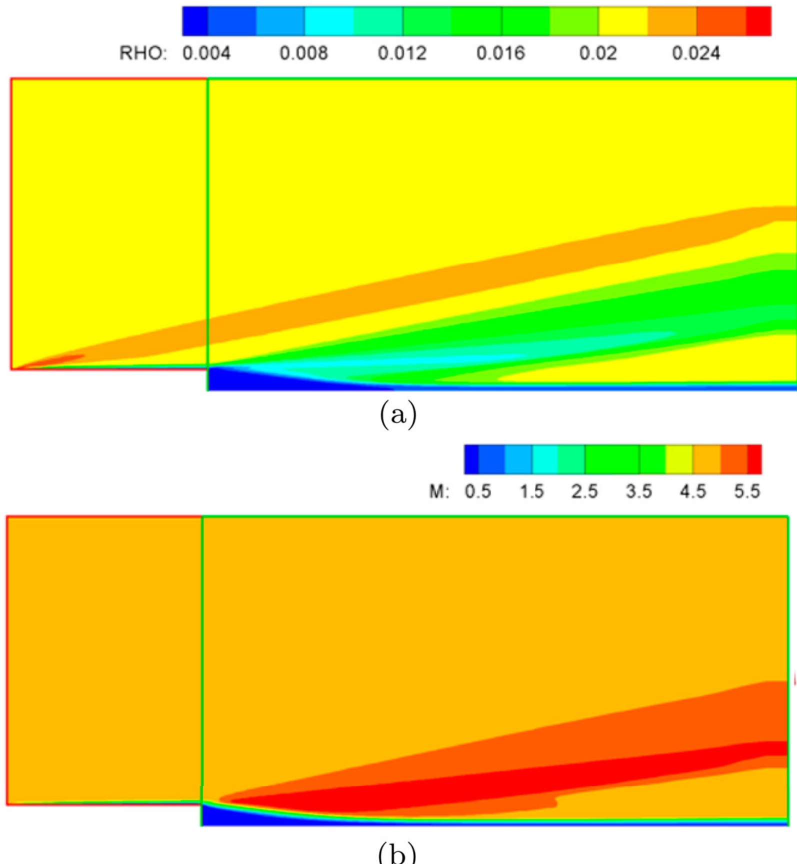

Figure 8. Simulation results for backward facing step. (a) Density profile and (b) Mach no. profile.

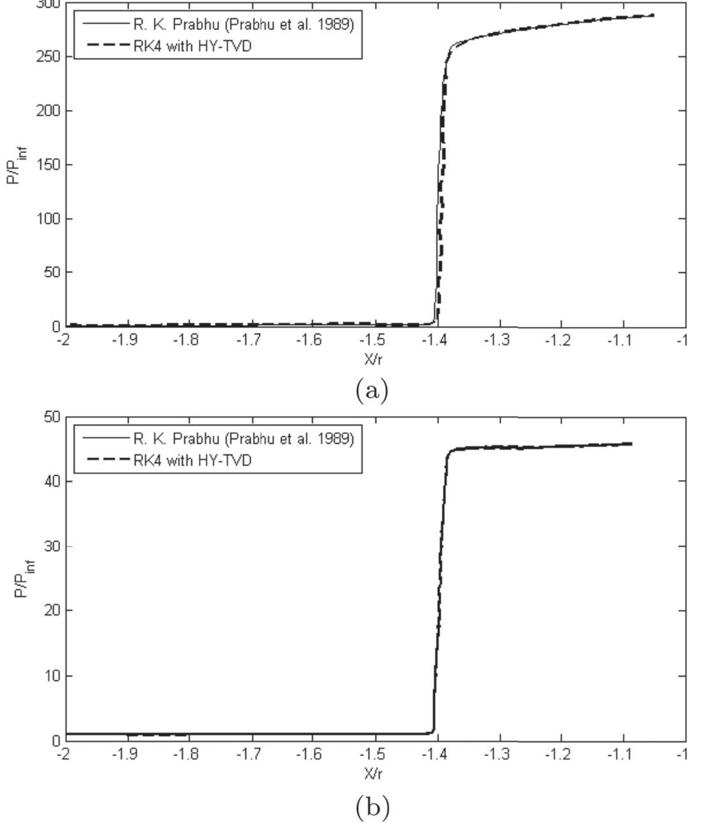

Figure 9. Comparison of RK4 + HY-TVD with existing numerical and experimental results for 5^0 wedge. (a) Comparison of pressure distribution along stagnation line for 5^0 wedge and (b) Comparison of temperature distribution along stagnation line for 5^0 wedge.

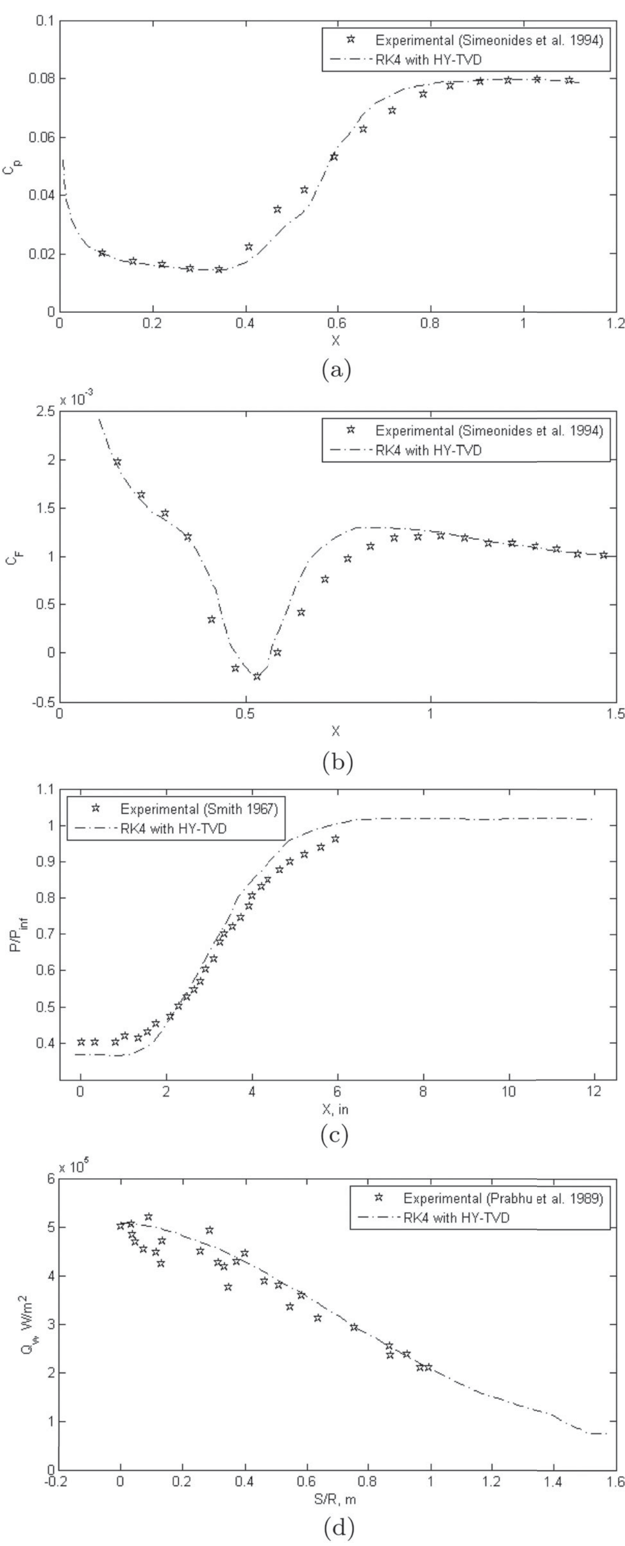

Figure 10. Comparison of RK4 + HY-TVD with existing numerical and experimental results for cases 2 to 4. (a) Pressure coefficient distribution along compression, (b) Skin fraction coefficient distribution along compression corner, (c) Pressure distribution along the wall of 2D backward facing step and (d) Heat flux distribution along 2D half cylinder.

Figure 12 shows the time taken by each kernel computation for each geometry. The first noticeable point is that most of the time is spent on kernel-1, kernel-2, and kernel-3 calculations. All other kernels and memory copies account for less than 10% of the total computation time in all cases. Kernel-1 is the main routine that

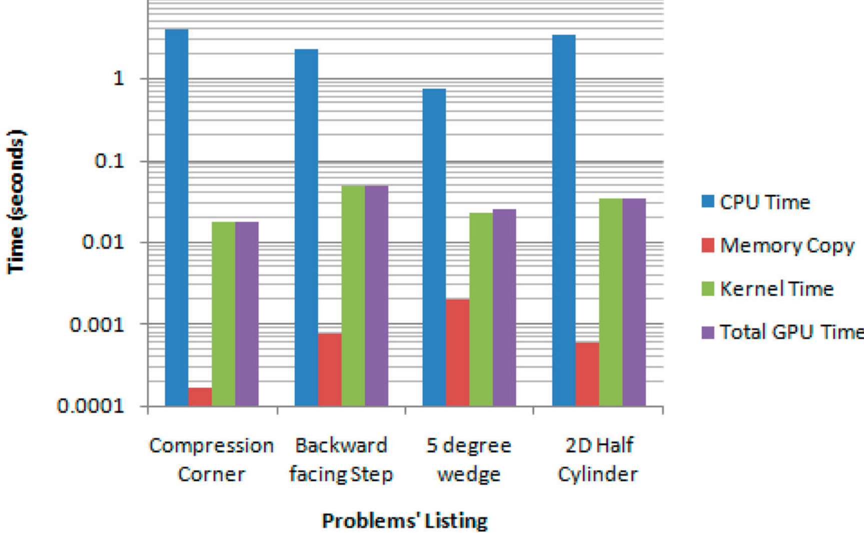

Figure 11. CPU and GPU time comparison for all 4 cases.

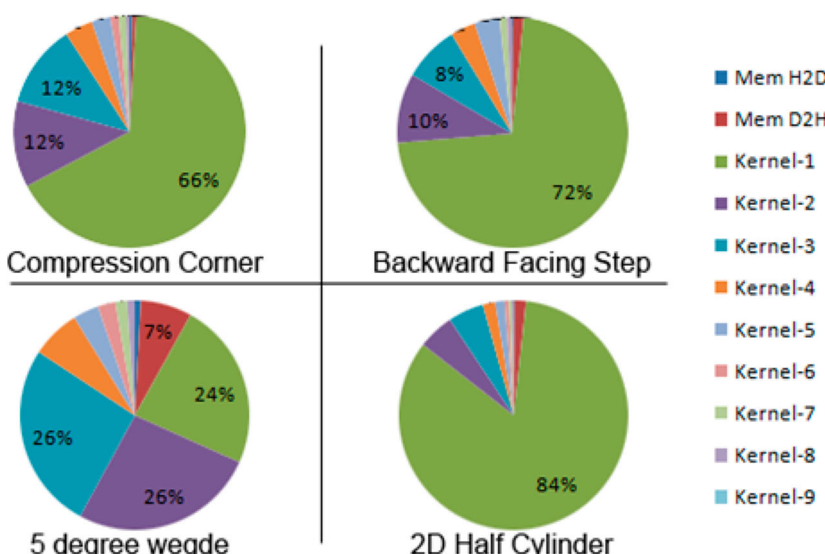

Figure 12. Time taken by individual kernel.

computes all 4 steps of RK4 and the associated variables. Kernel-2 and Kernel-3 are responsible for calculating the total variation diminishing (TVD) variables (TVD(x) and TVD(y)). The calculations performed by all other kernels are explained in Table 1. Another noticeable point is that the computation time trend is similar for all cases, but not for the 5^0 wedge geometry. This is due to the complex and time-consuming calculations for viscous terms, which are not considered in this case. This can be easily found out if you compare the two Algorithms 2 and 3. Algorithm 3 lacks the calculations for viscous terms, which are not required for the 5^0 wedge geometry because it is simulated with the Euler equation.

Figure 13 shows the time required by CPU and GPU for all cases where it takes thousands of iterations to converge the problem to the solution. It is obvious that the problems solved on the GPU are really fast. Backstepping geometry is a comparatively computationally intensive problem and takes about 450 seconds on CPU. Solving the same problem on a GPU takes less than 50 seconds, which is clearly better than solving on CPU. The speedup achieved on the GPU is shown in Figure 14. The maximum speedup achieved on the GPU is about 23, for a compressed corner geometry. The speedup is nearly constant for each number of iterations.

6. Conclusion and future work

This study presents an efficient GPU based solver for compressible, high speed flows along with simulations

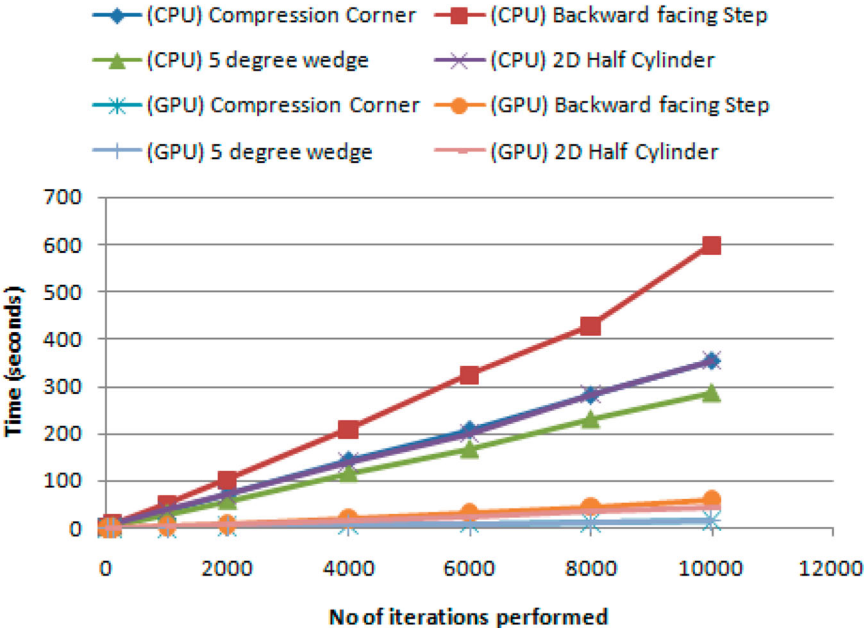

Figure 13. Time vs. increasing number of iterations.

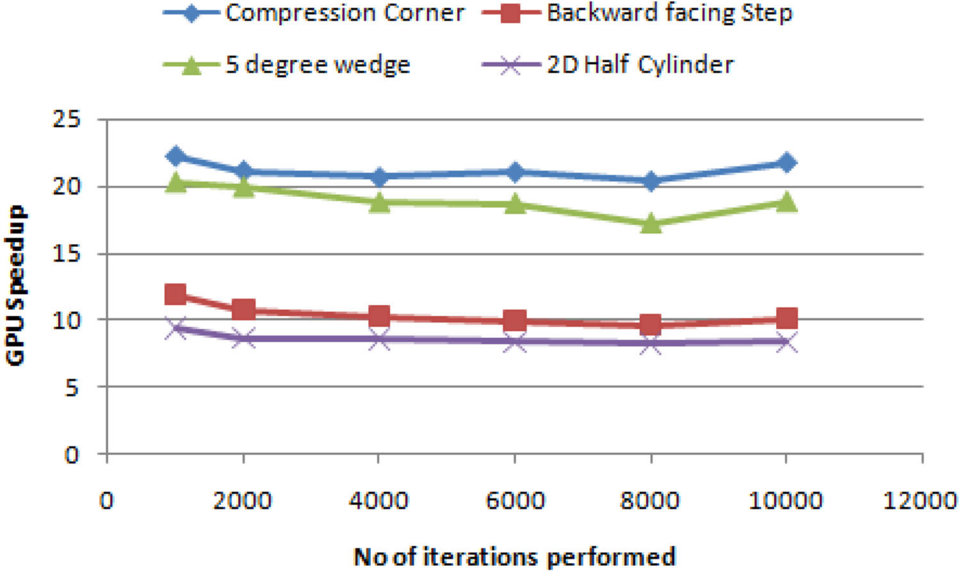

Figure 14. Speedup vs. increase in number of iteration on GPU.

results obtained for four different 2D structured geometries using this solver. Modified RK4 method along with Harten-Yee upwind TVD scheme for shockwave capturing has been implemented for solving the governing equations. The most powerful, compact and massively scalable parallel platform (GPU) was used for these simulations for the very first time. Surface pointers capability of GPU was also exploited to speedup double precision computations. For these problems, results show a speedup between 8 to 22 and more than 20 on an average GPU than that of standard core i5® computing machines. It is evident if CFD codes and software use GPU for simulation of problems, the computation speed can be enhanced.

In the future, this research could be extended to 3D solvers. Moreover, the concept could be extended to implicit solvers. A thorough investigation might be needed to determine the computational cost of these types of solvers. Another research direction could be to study the behaviour of solvers for transient and fuzzy systems.

Disclosure statement

No potential conflict of interest was reported by the author(s).

References

Adeeb, E., & Ha, H. (2022). Computational analysis of naturally oscillating tandem square and circular bluff bodies: A GPU based immersed boundary–lattice Boltzmann approach. *Engineering Applications of Computational Fluid Mechanics*, *16*(1), 995–1017. https://doi.org/10.1080/19942060.2022.2060309

Chandar, D. D., Sitaraman, J., & Mavriplis, D. J. (2013). A GPU-based incompressible Navier–Stokes solver on moving overset grids. *International Journal of Computational Fluid Dynamics*, *27*(6-7), 268–282. https://doi.org/10.1080/10618562.2013.829915

Chen, Q., Zhang, Z., & Zuo, W. (2007). Computational fluid dynamics for indoor environment modeling: Past, present and future. In Proceedings xxv congresso della trasmissione del calore uit.6th International Conference on Indoor Air Quality, Ventilation and Energy Conservation in Buildings: Sustainable Built Environment, IAQVEC 2007. https://research.polyu.edu.hk/en/publications/computational-fluid-dynamics-for-indoor-environment-modeling-past.

Cuda, C. (2014). Programming guide 6.5.

Cuda, C. (2015). Programming guide 7.5.

Cuda, C. (2020). Programming guide 11.0.

Damevin, H. M., & Hoffmann, K. (2001). Development of a modified Runge-Kutta scheme with TVD limiters for ideal three-dimensional magnetogasdynamics. In 32nd AIAA plasmadynamics and lasers conference (p. 2739). https://doi.org/10.2514/6.2001-2739

Glaskowsky, P. N. (2009). *NVIDIA's Fermi: The first complete GPU computing architecture*. White paper 18.

Goddeke, D., Buijssen, S. H., Wobker, H., & Turek, S. (2009). GPU acceleration of an unmodified parallel finite element Navier-Stokes solver. In International conference on high performance computing & simulation (pp. 12–21). https://doi.org/10.1109/HPCSIM.2009.5191718

Griebel, M., & Zaspel, P. (2010). A multi-GPU accelerated solver for the three-dimensional two-phase incompressible Navier-Stokes equations. *Computer Science-Research and Development*, *25*(1), 65–73. https://doi.org/10.1007/s00450-010-0111-7

Hoffmann, K. A., & Chiang, S. T. (2000). *Computational fluid dynamics volume II*. Engineering Education System.

Kale, B. S., Bhole, K. S., Dhongadi, H., Oak, S., Deshmukh, P., Oza, A., & Ramesh, R. (2022). Effect of polygonal surfaces on development of viscous fingering in lifting plate Hele-Shaw cell. *International Journal on Interactive Design and Manufacturing (IJIDeM)*, 1–8. https://doi.org/10.1007/s12008-022-01030-9

Kale, B. S., Bhole, K. S., & Sharma, C. (2022). Effect of anisotropies in formation of viscous fingering in lifting plate Hele-Shaw cell. *Advances in Materials and Processing Technologies*, *8*(4), 3780–3793. https://doi.org/10.1080/2374068X.2021.2013679

Kun, Y., & Xiaowen, S. (2022). Progress and prospects of multi-speed lattice Boltzmann method. *Acta Aerodynamica Sinica*, *40*(3), 23–45. http://dx.chinadoi.cn/10.7638/kqdlxxb-2021.0348

Ladeinde, F., & Nearon, M. D. (1997). CFD applications in the HVAC&R industry. *ASHRAE Journal*, *39*(1), 44–48. http://newsite.ttctech.com/Docs/insted/ASHRAE_Journal-v39n1.CFDApps.HVAC.R.pdf

Lai, J., Tian, Z., Yu, H., & Li, H. (2020). Numerical investigation of supersonic transverse jet interaction on CPU/GPU system. *Journal of the Brazilian Society of Mechanical Sciences and Engineering*, *42*(2), 1–13. https://doi.org/10.1007/s40430-019-2160-6

Lai, J., Yu, H., Tian, Z., & Li, H. (2020). Hybrid MPI and CUDA parallelization for CFD applications on multi-GPU HPC clusters. *Scientific Programming*, *202*. https://doi.org/10.1155/2020/8862123.

Ma, Z., Wang, H., & Pu, S. (2014). GPU computing of compressible flow problems by a meshless method with space-filling curves. *Journal of Computational Physics*, *263*, 113–135. https://doi.org/10.1016/j.jcp.2014.01.023

Nielsen, P. V. (2004). Computational fluid dynamics and room air movement. *Indoor Air*, *14*(Supplement 7), 134–143. https://doi.org/10.1111/ina.2004.14.issue-s7

Prabhu, R., Stewart, J., & Thareja, R (1989). A Navier-Stokes solver for high speed equilibrium flows and application to blunt bodies. In *27th aerospace sciences meeting* (p. 668). https://doi.org/10.2514/6.1989-668

Rogers, S. E., & Kwak, D. (1990). Upwind differencing scheme for the time-accurate incompressible Navier-Stokes equations. *AIAA Journal*, *28*(2), 253–262. https://doi.org/10.2514/3.10382

Shao, X., Santasmasas, M. C., Xue, X., Niu, J., Davidson, L., Revell, A. J., & Yao, H. D. (2022). Near-wall modeling of forests for atmosphere boundary layers using lattice Boltzmann method on GPU. *Engineering Applications of Computational Fluid Mechanics*, *16*(1), 2142–2155. https://doi.org/10.1080/19942060.2022.2132420

Simeonides, G., Haase, W., & Manna, M. (1994). Experimental, analytical, and computational methods applied to hypersonic compression ramp flows. *AIAA Journal*, *32*(2), 301–310. https://doi.org/10.2514/3.11985

Smith, H. E. (1967). *The flow field and heat transfer downstream of a rearward facing step in supersonic flow* [Tech. Rep.]. Aerospace Research Labs Wright-Patterson AFB OH.

Soukov, S. (2021). Heterogeneous parallel algorithm for compressible flow simulations on adaptive mixed meshes. In *Russian supercomputing days* (pp. 102–113). Springer. https://link.springer.com/chapter/10.1007/978-3-030-92864-3_8.

Thibault, J., & Senocak, I. (2009). CUDA implementation of a Navier-Stokes solver on multi-GPU desktop platforms for incompressible flows. In *47th AIAA aerospace sciences meeting including the new horizons forum and aerospace exposition* (p. 758). https://doi.org/10.2514/6.2009-758.

Tian, Z., Lai, J., Yang, F., & Li, H (2020). GPU-accelerated computations for supersonic flow modeling on hybrid grids. In *5th international conference on mechanical, control and computer engineering (ICMCCE)* (pp. 1391–1397). IEEE. https://doi.org/10.1109/ICMCCE51767.2020.00305.

Tölke, J., & Krafczyk, M. (2008). TeraFLOP computing on a desktop PC with GPUs for 3D CFD. *International Journal of Computational Fluid Dynamics*, *22*(7), 443–456. https://doi.org/10.1080/10618560802238275

Wang, S., Li, Z., & Che, Y. (2020). Memory access optimization of high-order CFD stencil computations on GPU. In *International conference on parallel and distributed computing: Applications and technologies* (pp. 43–56). Springer. https://link.springer.com/chapter/10.1007/978-3-030-69244-5_4.

Wang, Y., Baboulin, M., Rupp, K., Le Maitre, O., & Fraigneau, Y. (2014). Solving 3D incompressible Navier-Stokes equations

on hybrid CPU/GPU systems. In *High performance computing symposium (hpc'14)*. https://inria.hal.science/hal-01205305/

Wang, Y., Yan, X., & Zhang, J. (2021). Research on GPU parallel algorithm for direct numerical solution of two-dimensional compressible flows. *The Journal of Supercomputing*, *77*(10), 10921–10941. https://doi.org/10.1007/s11227-021-03704-9

Wei, F., Jin, L., Liu, J., Ding, F., & Zheng, X. (2020). GPU acceleration of a 2D compressible Euler solver on CUDA-based block-structured cartesian meshes. *Journal of the Brazilian Society of Mechanical Sciences and Engineering*, *42*(5), 1–12. https://doi.org/10.1007/s40430-020-02290-w

Weiskopf, D. (2007). *Gpu-based interactive visualization techniques*. Springer.

Weng, Y., Zhang, X., Guo, X., Zhang, X., Lu, Y., & Liu, Y. (2021). Effects of mesh loop modes on performance of unstructured finite volume GPU simulations. *Advances in Aerodynamics*, *3*(1), 1–23. https://doi.org/10.1186/s42774-020-00055-6

Xu, D., Luo, S., Song, J., Liu, J., & Cao, W. (2021). Direct numerical simulations of supersonic compression-expansion slope with a multi-GPU parallel algorithm. *Acta Astronautica*, *179*, 20–32. https://doi.org/10.1016/j.actaastro.2020.10.047

Yee, H. C. (1989). A class of high-resolution explicit and implicit shock-capturing methods.

Zhai, Z., & Chen, Q. Y. (2003). Solution characters of iterative coupling between energy simulation and CFD programs. *Energy and Buildings*, *35*(5), 493–505. https://doi.org/10.1016/S0378-7788(02)00156-1

Zuo, W., & Chen, Q. (2010). Simulations of air distributions in buildings by FFD on GPU. *Hvac&R Research*, *16*(6), 785–798. https://doi.org/10.1080/10789669.2010.10390934