Computers & Fluids 80 (2013) 102–115

Contents lists available at SciVerse ScienceDirect

Computers & Fluids

journal homepage: www.elsevier.com/locate/compfluid

Coalesced computations of the incompressible Navier–Stokes equations over an airfoil using graphics processing units

S.M. Iman Gohari *, Vahid Esfahanian, Hamed Moqtaderi

School of Mechanical Engineering, University of Tehran, Iran
Vehicle, Fuel and Environment Research Institute, University of Tehran, Iran

A R T I C L E   I N F O

Article history:
Received 30 September 2011
Received in revised form 10 April 2012
Accepted 17 April 2012
Available online 5 May 2012

Keywords:
GPU
Coalesced pattern
Finite differencing time domain
CUDA
Airfoil

A B S T R A C T

This paper presents a Graphics Processing Unit (GPU) based implementation of the Finite Differencing Time Domain (FDTD) methods, for solving unsteady incompressible viscous flow over an airfoil using the Stream function-Vorticity formulation for a structured grid. For the large-scale simulations, FDTD methods can be computationally expensive and require considerable amount of time to solve on traditional CPUs. On the contrary, modern GPGPUs such GTX 480 are designed to accelerate lots of independent calculations due to advantage of their highly parallel architecture. In present work, the main purpose is to show a new configuration for leveraging GPU processing power for the computationally expensive simulations based on explicit FDTD method and CUDA language. Our proposed work improves the GPU FDTD results by increasing the global memory coalescence with the same amount of occupancy, resulting in an increase in maximum output performance. In addition, this study introduces a more coalesced pattern of data loading which reduces the global memory requests. Although both GPU based programs are over 28 times faster than a sequential CPU based version, Implementation of our proposed work showed up to 44% decrease in execution time comparing to the naive GPU method.

1. Introduction

Graphics Processing Units (GPUs) are specifically designed to be extremely fast for processing large graphics data sets for graphical tasks. However, in recent years due to higher computational power of GPUs than PC-based CPUs by more than one order of magnitude, the use of GPU in non-graphical computations has been grown significantly. Therefore, major GPU vendors have been targeting the high performance computing market by introducing GPU hardware implementations. Software toolkits such as Compute Unified Device Architecture (CUDA), released by NVIDIA in early 2007 [7], provide a conveniently developed platform abstracting the GPU and allowing easy access to its underlying stream computing architecture. NVIDIA GPU consists of several so-called Streaming Multiprocessors (SMs). Each SM drives several processor cores in a Single Instruction Multiple Data (SIMD) fashion. Every SM has some register memory available, as well as on-chip shared memory. This architecture allows efficient data synchronization and data sharing among threads in the same thread block [11]. It is very important to note that on-chip memories are limited in size because of reduction in manufacturing cost. Furthermore, an incremental raise in the usage of shared memory can arise in a concrete decrease in the number of threads that can be concurrently executed and thus significantly reducing the parallelism level [11]. There is also an off-chip global memory with the size of the device memory which can be accessed by all threads with higher latency.

To achieve maximum performance, the primary concern often is managing global memory latency. This is carried out by generating enough threads to keep SMs occupied while many threads are waiting on global memory accesses. Global memory request and access patterns are two effective concerns for the DRAM efficiency [15,4]. Because each data transferring is done by memory request, any redundancy in memory requests can make long-latency. Moreover, coalesced memory access can reduce multiple memory accesses from threads to a specific memory region into a fewer memory access. Hence, the memory access pattern is also an important for achieving high GPU performance due to limitations of on-chip memories and high latency of global memory. More exhaustive elucidation of the NVIDIA GPUs can be found in [7,5,12].

There are several schemes presented in the literature to accelerate calculations on GPUs including using structure of arrays instead of array of structures [4], using of shared memory [21,11], using of split/reversed scheme in global memory [24] and register packing [28]. Although it is possible to develop the special procedure for the certain method or specific calculation [29,28], it is preferred

* Corresponding author at: School of Mechanical Engineering, University of Tehran, Iran.
E-mail address: iman.gohari@ut.ac.ir (S.M. Iman Gohari).

0045-7930/$ - see front matter © 2012 Elsevier Ltd. All rights reserved.
http://dx.doi.org/10.1016/j.compfluid.2012.04.022

to have a robust algorithm with the discerning balance between complexity of computer programming and the rate of performance improvement. In some studies, alleviating the pressure on global memory bandwidth involves additional usage of register and shared memories, which in turn can limit the number of simultaneously executing threads and hence can reduce the SM's occupancy [4]. The improvement idea of present work is to reduce the global memory requests and registers which causes a higher coalesced pattern for the memory bandwidth.

In the current study, an optimized scheme is utilized to achieve a better memory address pattern in GPU programming with the concept of Cooperative Thread Array (CTA [16]) in collaboration with shared memory for the finite difference method. The fundamental concept of present method is to modify the CTA configuration with the association of shared memory. It will reduce redundant global memory requests and cause uniform streaming in the global memory without changing the program structure. Implementation of present work consists of two cardinal modules. The first one is to make more coalesced memory access and request patterns by modifying the CTA configurations and the later one is to use the on-chip shared memory. It is expected that present method will reduce number of GPU global memory transactions due to higher coalesced pattern and consequently lower runtime will be achievable. Moreover, by reducing the redundant global memory requests with the same amount of occupancy, it is expected that the present work has lower limit on GPU memory bandwidth and hence higher memory throughput will be attainable.

To implement present GPU optimization, unsteady flow solver based on Stream function-Vorticity formulation and finite difference method is developed to simulate unsteady incompressible turbulent flow over an airfoil. However, flow simulation in analyzing process is more time consuming than the grid generation procedure, in the design or optimization purposes, grid generation takes considerable amount of time. As a result, both grid generation process and flow simulation is calculated by GPU. The comparisons are based on CPU/GPU solver runtime and performance. Both CPU and GPU version of solvers are developed through CUDA enabled C/C++ language. In the present work, all the comparisons for solvers are evaluated with the same set of parameters i.e. number of time steps, flow time, etc. for two different CTA configurations. In addition, to have a fair comparison between two CUDA and C++ compilers, similar compiler settings are used. The GPU performance and speedup ratio for different grid sizes are calculated and the ability of present GPU parallelizing work is investigated and discussed.

2. Numerical implementation

The incompressible Navier–Stokes equations are solved numerically with the Stream function-Vorticity formulation for a structured grid. It is clear that common CFD simulation needs appropriate computational grid. Therefore, grid generation is one of the most important parts of CFD simulations. In addition, some CFD simulations need grid adaptation and refinement. This procedure needs grid regeneration and sometimes takes more time than the flow simulation itself. As [22] showed, to have a highly accelerated flow solver, it is crucial to construct the computational grid through the GPU. In the present study, both flow simulation and grid generation is done through the GPU. The Stream function-Vorticity formulation needs an orthogonal grid points with the desirable grid spacing to capture high velocity gradient in the boundary layer. [23] showed the iterative procedure for constructing the computational grid points which is used in the present work. The numerical simulation of the unsteady incompressible Navier–Stokes equations for the laminar/turbulent flow around arbitrarily shaped two-dimensional airfoils is also considered. This solution is based on the technique of numerical generation of a curvilinear coordinate system which has coordinate lines coincident with the airfoil contour regardless of its shape. The explicit simulation utilizes the Stream function-Vorticity formulation with the direct satisfaction of No-slip condition [27] on the airfoil surface.

2.1. Grid generation formulation and algorithm

Consider general Cartesian coordinates x, y indicate grid points in the physical space where in the airfoil and airflow exist. As showed in Fig. 1, a "C-type" grid is a conformal mapping between physical space and computational one as ξ, η for $0 \leqslant \xi \leqslant \xi_{max}$ and $0 \leqslant \eta \leqslant \eta_{max}$. The boundary $\xi = 0$ is mapped into the grid line moving forward from the outer boundary to the trailing edge. Because $\xi = \xi_{max}$ line is placed on $\xi = 0$ line in the physical space, the periodic boundary is considered on these grid lines. The boundary $\eta = 0$ is mapped into the inner boundary (the airfoil surface) with $\xi = 0$ at the trailing edge and ξ increasing clockwise around the airfoil. The boundary $\eta = \eta_{max}$ is mapped into the outer boundary in the same manner.

Consider $\xi = \xi(x, y)$ and $\eta = \eta(x, y)$ define the mapping from the physical space to the computational space. The mapping functions are required to satisfy the Poisson equations as follow:

$$\begin{aligned}\nabla^2 \xi &= P,\\ \nabla^2 \eta &= Q.\end{aligned} \tag{1}$$

To obtain the Poisson equation in physical space, the following relations are helpful in transforming equations between physical and computational spaces:

$$\begin{aligned}\xi_x &= \frac{y_\eta}{J}, \quad \xi_y = -\frac{x_\eta}{J},\\ \eta_x &= -\frac{y_\xi}{J}, \quad \eta_y = \frac{x_\xi}{J}.\end{aligned} \tag{2}$$

where $J = x_\xi y_\eta - y_\xi x_\eta$. By using Eq. (2) in Eq. (1), the Poisson equations in physical domain are obtained as follow:

$$\begin{aligned}\alpha x_{\xi\xi} - 2\beta y_{\xi\eta} + \gamma x_{\eta\eta} &= -J^2(Px_\xi + Qx_\eta),\\ \alpha y_{\xi\xi} - 2\beta y_{\xi\eta} + \gamma y_{\eta\eta} &= -J^2(Py_\xi + Qy_\eta).\end{aligned} \tag{3}$$

in which $\alpha = x_\eta^2 + y_\eta^2$, $\beta = x_\eta x_\xi + y_\eta y_\xi$ and $\gamma = x_\xi^2 + y_\xi^2$. Solving Eq. (3) with inhomogeneous term of P and Q, make the appropriate grid be generated. To accomplish this, these terms are defined as follow:

$$\begin{aligned}P(\xi,\eta) &= p(\xi)e^{-a\eta} + r(\xi)e^{-b(\eta_{max}-\eta)},\\ Q(\xi,\eta) &= q(\xi)e^{-c\eta} + s(\xi)e^{-d(\eta_{max}-\eta)}.\end{aligned} \tag{4}$$

where a, b, c and d are adjustable positive parameters. Imposing desirable grid spacing and angle at the body surface are done by the use of the parameter $p(\xi)$, $r(\xi)$, $q(\xi)$ and $s(\xi)$. For this purpose, consider the following equations on the inner (or outer) boundary which impose the grid spacing and angle at the body surface [23]:

$$\begin{aligned}x_\eta &= s_\eta \frac{x_\xi cos(\theta) - y_\xi sin(\theta)}{\sqrt{x_\xi^2 + y_\xi^2}},\\ y_\eta &= s_\eta \frac{-y_\xi cos(\theta) + x_\xi sin(\theta)}{\sqrt{x_\xi^2 + y_\xi^2}}.\end{aligned} \tag{5}$$

in which s_η is the desirable grid spacing, and θ is desirable angle at the boundary. By imposing the desirable grid spacing and angle on the boundaries, formulation of $p(\xi)$, $r(\xi)$, $q(\xi)$ and $s(\xi)$ is achievable [23]. To evaluate the value of $p(\xi)$, $r(\xi)$, $q(\xi)$ and $s(\xi)$, it is necessary to have all the first and second derivatives of η and ξ. The derivative of ξ can be simply achieved by central finite difference approximation. The derivative of η in interior grid points is also derived by central finite difference approximation. Eq. (5) is used for the first

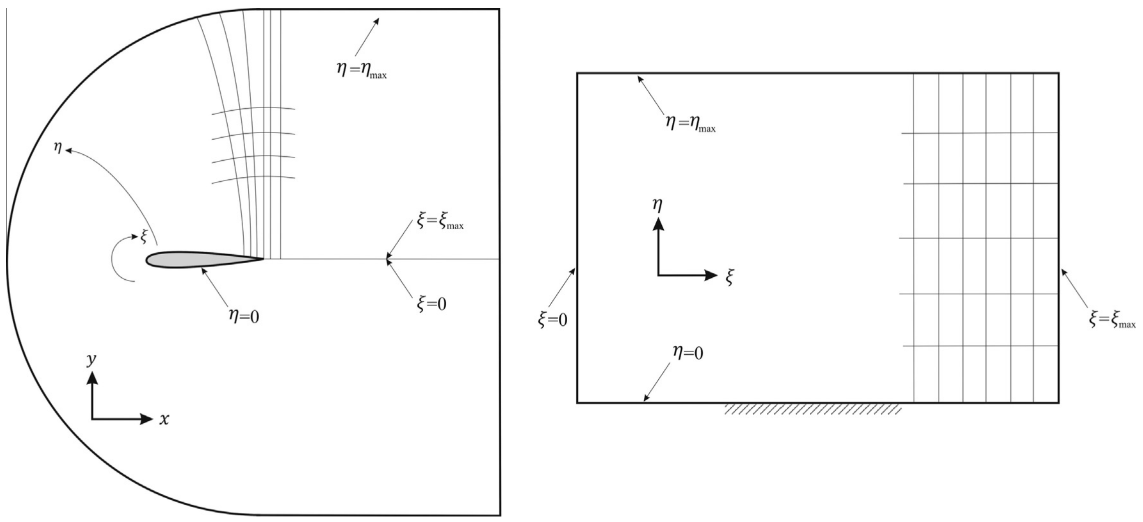


Fig. 1. Typical C type grid over an airfoil: physical domain (left), computational domain (right).

derivative of η on the boundary and the second derivative of η is computed by using field values as follow:

$$x_{\eta\eta}|_{\eta=0} = \frac{-7x|_{\eta=0} + 8x|_{\eta=1} - x|_{\eta=2}}{2} - 3x_{\eta}|_{\eta=0},$$
$$x_{\eta\eta}|_{\eta=\eta_{max}} = \frac{-7x|_{\eta=\eta_{max}} + 8x|_{\eta=\eta_{max-1}} - x|_{\eta=\eta_{max-2}}}{2} + 3x_{\eta}|_{\eta=\eta_{max}}.$$

With these finite-difference approximations, the discrete equations can be obtained from Eq. (3).

2.2. Stream function-Vorticity formulation and algorithm

Governing equations of unsteady two-dimensional viscous flow are presented in Eq. (6).

$$\begin{aligned}&\nabla^2\psi = -\omega,\\ &\omega_t + \tfrac{\partial(u\omega)}{\partial x} + \tfrac{\partial(v\omega)}{\partial y} = \nabla^2(v\omega).\end{aligned} \tag{6}$$

where ψ and ω are Stream function and Vorticity, respectively. By considering the transforming equations, the governing equations can be obtained in a general curvilinear coordinate as depicted in Eq. (7).

$$\begin{aligned}&\frac{\alpha\psi_{\xi\xi} - 2\beta\psi_{\xi\eta} + \gamma\psi_{\eta\eta}}{J^2} + P\psi_{\xi} + Q\psi_{\eta} = -\omega,\\ &\omega_t + \frac{\psi_{\eta}\omega_{\xi} - \psi_{\xi}\omega_{\eta}}{J} =\\ &\frac{\alpha(v\omega)_{\xi\xi} - 2\beta(v\omega)_{\xi\eta} + \gamma(v\omega)_{\eta\eta}}{J^2} + P(v\omega)_{\xi} + Q(v\omega)_{\eta}.\end{aligned} \tag{7}$$

where all the coefficients are defined in previous section. With the finite-difference approximation, fundamental equations are discretized in the curvilinear coordinate. The numerical code includes options for using 1st or 2nd order upwind scheme for the convective terms and basic 2nd order central scheme for the diffusion terms and the explicit time discretization for time marching. No slip condition at the airfoil surface is directly satisfied by following equation [27]:

$$\omega_{wall} = \frac{2\gamma(\psi_{i,1} - \psi_{i,0})}{J^2}$$

where $\psi_{i,0}$ and $\psi_{i,1}$ are the Stream function at, and one point above the wall respectively. The closure of mean flow equations is computed by the algebraic Balwin–Lomax turbulence method which is very popular for the aerodynamic applications [32,34]. Hence, non-dimensional kinematic viscosity, υ, is define as below:

$$v_{Laminar} = 1/Re,$$
$$v_{turbulence} = v_{Laminar} + v_t.$$

where the procedure of obtaining of v_t is presented in [32]. Pressure coefficient on the airfoil surface is obtained by the line integral method. If the primitive variables of the Navier–Stokes equations are evaluated on the body surface, the time derivative of velocity and the inertia terms vanish yielding:

$$\nabla P = \nabla^2(vV).$$

Using a vector identity to eliminate the Laplace of the velocity, dotting above equation with dr (an arc length differential along the body), and transforming equations provide:

$$dP = \frac{1}{J}(\beta(v\omega)_{\xi} - \gamma(v\omega)_{\eta})d\xi,$$

and integrating among the body surface yields:

$$C_p^*(\xi) = P(\xi) - P_{T.E.}(\xi) = \int_{\xi_{T.E.}}^{\xi} \frac{(\beta(v\omega)_{\xi} - \gamma(v\omega)_{\eta})}{J} d\xi'. \tag{8}$$

By substituting Eq. (8) in stress vector for the viscous filed and integrating along body surface, Lift and Drag coefficient can be obtained.

Discrete equations (for both flow solver and grid generation with finite-difference method) represent a system of non-linear equations that can be solved by matrix inversion or point to point iterative methods. Gauss–Seidel and Jacobi algorithms are two well known explicit iterative techniques for solving this system of equations, but over relaxed Jacobi algorithm is used for the following reasons: (1) Although Gauss–Seidel is the faster algorithm for CPUs, due to its dependencies on two new calculated values, it needs appropriate technique such as Red–Black method [40,43] for GPU programming. Consequently, by using over relaxed Jacobi algorithm low barrier synchronization is needed; (2) [3] showed that Jacobi algorithm on the GPU has higher convergence rate than the Gauss–Seidel; (3) due to the same complexity patterns of the GPU/CPU solvers, the comparison is based on the equal number of iterations for the specified convergence criteria [41]. Consequently, convergence acceleration is not target of present work and Jacobi algorithm is appropriate technique.

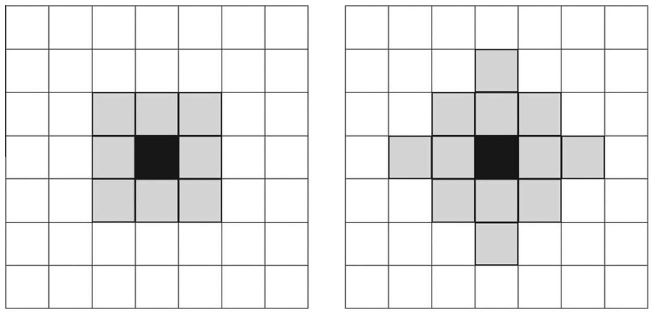


Fig. 2. Typical memory halo for the 2D problems.

3. Memory halo

Navier–Stokes equations using finite-difference time-domain approximation always access the values of neighboring nodes to decide how to correct the value of each node. The neighbor's data in which the node must access are known as the "memory halo". Fig. 2 shows two usual memory halo patterns for the two dimensional problems. The size of memory halo for the equations has paramount effect on how well the equation can be split across GPU [8].

An equation with the large memory halos will demand more nodes to be shared in each step than an equation with smaller. Consequently for solving a problem which contains large memory halos, the domain decomposition for the multiple GPU programming and CTA configuration in single GPU programming are very critical. In this paper we discuss optimized procedure for the CTA configuration for a structured grid which improves the memory halo access.

4. GPU implementation

The GPU enabled solver begins the calculation on the CPU which carries out the data inputs and the algebraic grid construction over an airfoil. All the data transferred to the GPU, where both grid generation and unsteady flow simulation take place through the GPU kernels. During the iterative simulation procedure, CPU and GPU communicate for storing the results at the certain physical flow time. This communication worsens the computational cost and decreases the total speedup. As a result, solver runtime used for calculating the speedup does not contain the time of storing section. The flowchart of calculation is presented in Fig. 13.

Although Finite Difference Method (FDM) through other numerical procedures i.e. finite volume method or finite element method is still one of highly accurate methods, it has high computational cost and some stability limitations [10,17,36]. For the present Stream function-Vorticity formulation of Navier–Stokes calculation with the finite difference method, the discrete equations contain at least eight memory halos for the each node. On the structured grids, where all neighboring nodes with consecutive indices (Fig. 3) encircle any internal node (with respect to boundary nodes having two or three nodes), the calculation is straightforward and with some considerations quite efficient [4,17].

The global memory bandwidth is used more efficiently if concurrent memory accesses by threads are coalesced into a single transaction [25]. It means that when the data segment transferred in one transaction, there should be as larger as possible number of threads using the segment. For the finite element method or even finite volume method with the unstructured grids, it is very complicated to find a coalesced pattern for the global memory access [25,29].

Another memory region where in threads have access, is the texture memory based on 2D space. [18] showed that the optimal memory region for the finite difference equations on the Fermi and Tesla architecture is different. Texture memory can be used for the two dimensional problems on the Tesla architecture with the advantage of two dimensional spatial locality and achieving high performance [30]. But for the Fermi architecture along with the high performance of L1/L2 cache, global memory shows better access pattern [8]. More information about the type of GPU memory can be found in [11,7].

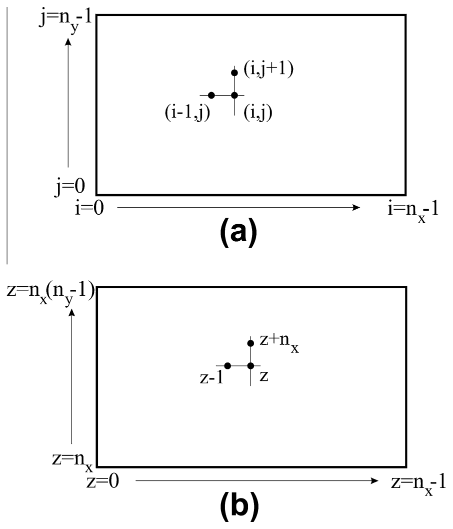


Fig. 3. Structured grid: (a) two variables indixing and (b) one variable indixing.

Undoubtedly, each procedure which improves the GPU computational power will have a significant effect on reducing overall runtime. This optimization is different in single and multiple GPU programming. The most important factor in multiple GPU programming optimization which can highly affect the output performance, is domain decomposition. [19] showed a simple effective procedure for domain decomposition which improved the GPU performance. But in single GPU programming in which there is no decomposition and all the computations are done through a single processor, one of the most effective parameters in output performance is CTA configuration. Cooperative Thread Array (CTA) is an array of simultaneous threads that cooperate and execute to compute a desire result. In the GPU programming model, the thread block is the CTA. CTA configuration i.e. size, shape, dimension are declared by programmer and has major effect on GPU performance and runtime. Although CTA configuration is a fundamental concept in GPU programming science, but [41,19,42,38] showed the dependency of GPU computational power to CTA configuration. In addition [42], proposed a complex pattern for optimizing GPU memory bandwidth in accordance with decoupling of CTA configuration from data size/shape. Although there are general comments about CTA configuration in literature, in present study we completely elaborated its effect on GPU efficiency factors. We precisely clarified that how the GPU efficiency factors i.e. GPU memory request and access patterns, Global memory throughputs and occupancy, will change by modification of CTA configuration. In the presence of significantly larger number of cores on the GPU in the contrast to CPU, in order to fully saturate the available computational resources, 1D CTA configuration is considered for this study. Meaning that each node on the two-dimensional plane mapped on the GPU, is assigned to a unique thread corresponding to the processing of node data.

Present strategy of using 1D CTA (illustrated in Figs. 4–6) shows higher speedup and performance than the regular 2D one. In addition, to increase the memory throughput with high expression of parallelism, the number of threads on each block has been set as a multiple of 32, correlating to the GPU warp size. It is clear

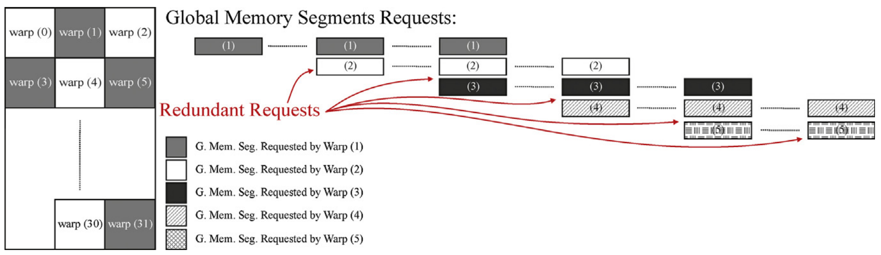

warp (0)
warp (1)
warp (2)
warp (3)
warp (4)
warp (5)
warp (30)
warp (31)
Global Memory Segments Requests:
Redundant Requests
G. Mem. Seg. Requested by Warp (1)
G. Mem. Seg. Requested by Warp (2)
G. Mem. Seg. Requested by Warp (3)
G. Mem. Seg. Requested by Warp (4)
G. Mem. Seg. Requested by Warp (5)

Fig. 4. Global memory requests for regular CTA configuration.

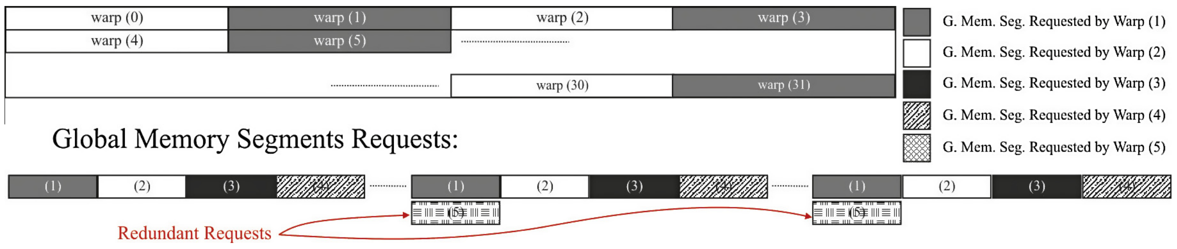

warp (0)
warp (1)
warp (2)
warp (3)
warp (4)
warp (5)
warp (30)
warp (31)
Global Memory Segments Requests:
Redundant Requests
G. Mem. Seg. Requested by Warp (1)
G. Mem. Seg. Requested by Warp (2)
G. Mem. Seg. Requested by Warp (3)
G. Mem. Seg. Requested by Warp (4)
G. Mem. Seg. Requested by Warp (5)

Fig. 5. Global memory requests for present CTA configuration.

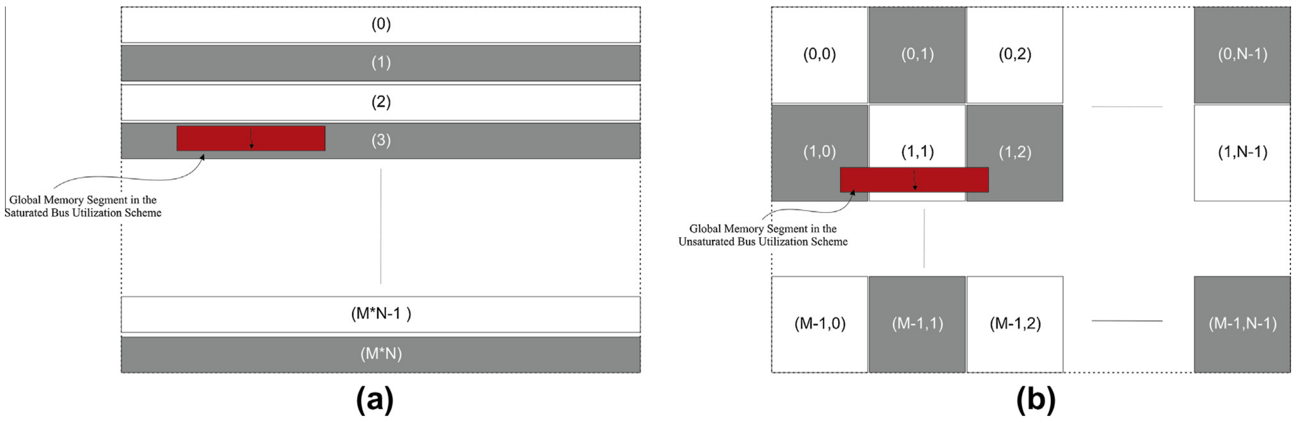

(0)
(1)
(2)
(3)
(M*N-1)
(M*N)
Global Memory Segment in the Saturated Bus Utilization Scheme
(a)
(0,0)
(0,1)
(0,2)
(0,N-1)
(1,0)
(1,1)
(1,2)
(1,N-1)
(M-1,0)
(M-1,1)
(M-1,2)
(M-1,N-1)
Global Memory Segment in the Unsaturated Bus Utilization Scheme
(b)

Fig. 6. CTA configuration: one-dimensional (a), two-dimensional (b).

because the warp is the abbreviated GPU unit which can be scheduled at once on each SM of the GPU. Present work proposes 1D configuration for the CTA execution in conjunction with shared memory which assists in making higher coalesced memory access pattern for the two or even three dimensional problems having a structured grid. Along with the necessary consideration, two variable indexing for a structured grid is convertible to the single one, as illustrated in Fig. 3. For the multiple GPU programming, [6,20,19] proposed highly efficient domain decomposition yielding to high performance calculation. But for intra-GPU calculation, present work with the following consideration shows higher performance:

- *Higher memory request coalescing*: Transferring data segments from global memory of graphics card is carried out by the global memory requests. Threads within the same warp have a coalesced memory requests for loading specific memory region when they can load the data segments by fewer requests [15]. To have highly coalesced memory access pattern, it is required to have as few as possible redundancies in memory requests generated by each thread to eliminate the expense of associative search and alleviate the schedule of each SM [15,4].
  Present GPU work has great reduction in global memory requests. Previously it was clarified that for the finite difference method, at least eight neighboring data are needed for correcting each node value. For the two-dimensional CTA configuration, as illustrated in Fig. 4, each warp requests the upper, respective and lower data segment from the global memory. But because in each warp, the upper data segment is the respective data segment for the upper warp, the respective data segment is the upper data segment for the lower warp and finally the lower data segment is the respective data segment for the lower warp, there is great redundant memory requests (Fig. 4). On the other hand, as illustrated in Fig. 5, for the one-dimensional CTA configuration because of extension of block width, there will be fewer redundant requests for loading the needed data segments. For example, Figs. 5 and 4 show 75% reduction in memory requests only for first five warps. Consequently it is expected that present method will reduce total number of global memory requests for loading data.
- *Higher memory access coalescing*: the concept of coalescence of memory access pattern was discussed previously. It was indicated that the memory bus utilization will be improved if parallel memory accesses from CTA threads could be coalesced into fewer wide memory accesses and they all access a contiguous memory region. In the modern GPU architecture

(Fermi), the size of segments with the use of L1 cache is usually greater than two dimensional CTA size. As illustrated in Fig. 6, it is highly credible for the two dimensional CTA that there may be a transferred segment containing data which is required for no thread. But in 1D CTA configuration, with same size of CTA, larger number of threads in each CTA will use transferred data yielding to more saturated bus utilization and coalescence. Both presented coalescing concerns are based on this fact that in contrast to texture memory, global memory is a linear space [15,7]. Consequently, all the conclusions of present work are valid only for the global memory.

- *Less register memory usage*: register memory is one of scratch-pad memories in GPU which is almost the fastest one. Some efforts were done based on reduction in register memory usage [28,11]. According to the [7], if register memory of CTAs overflows, local memory will be used instead which is a slow off chip memory and cause a great surging in runtime. In the present work due to dichotomization of thread indices, the associated register memory will halve meaning the less register memory usage. Consequently, it is expected that the register memory will not be the limiting factor of present output performance.

The first module of present work is presented till now. The cardinal purpose of this module is to achieve higher coalesced calculation by trapping memory halo in each data segment and reducing global memory requests. This module demonstrates the significant results of having such an optimized scheme in Fermi architecture, based on the experimental results of our unsteady turbulent flow solver consisting of intensive arithmetic calculations. The second module of present method is using shared memory. The shared memory, which is about hundred times faster than the global one, along with the necessary memory fence and intrinsic functions are used to store the remaining memory halos and nodal residuals for controlling the convergence. Because for both storing memory halos and nodal residuals each thread uses different shared memory bank, this method is bank conflict free in each GPU compute capability.

Although all the numerical simulations in present study are limited to 2D problems, the optimization concept of this study is not limited to problem dimensionality. Because the basic version of flow solver has been developed with Stream function-Vorticity formulation, it was not extendable for 3D simulations. It is the reason that 2D flow simulation is considered. Fig. 18 shows the procedure of using present work for 3D problems. It is clear that by using GPU structuring 2 instead of 1, the output performance will increase in accordance to previous considerations. Along with CTA configuration, it is required to change the data structure in present strategy. Fig. 7 shows the data structure changing for 2 and 3D problems in present work. By changing the CTA configuration and using the proposed data structure (in Fig. 7), numerical implementation is achievable. In addition, [19] showed that in some 3D simulations, large number of grid points is used along the stream-wise and normal (to the wall) directions while the number of points in the span-wise direction is smaller. As a result, [19] used 2D CTA configuration for 3D problems and all the points in each span-wise line are mapped to the single thread corresponding to calculate the node data. Consequently, the present study for 2D CTA configuration can easily be extended to 3D simulations.

5. NVIDIA GTX 480 graphics card and compilers

The NVIDIA GeForce GTX 480 card, used in this paper, is based on the GF 100 Fermi architecture. Its computing power (estimated up to 1330 GFLOPS for the single precision arithmetic) relies on the

$$u[i][j] \rightarrow u[i + n_i j] \quad in\,2D\ problems$$
$$u[i][j][k] \rightarrow u[i + n_i j + n_i n_j k] \quad in\,3D\ problems$$

Fig. 7. Data structure for 2D and 3D problems.

capability if performing simultaneous data processing in accordance with the Single Instruction Multiple Thread (SIMT) fashion. The basic computational unit is the Streaming Processor (SP), a fully pipelined processor with the two Arithmetic Logic Units (ALUs) and a single Floating Point Unit (FPU). A Streaming Multiprocessor (SM) contains 32 SPs (CUDA cores), configurable L1 cache and shared memory, register memory and instruction cache blocks (Fig. 8). The numerical code development on the GTX 480 used in present work is based on CUDA 3.2 [7], which is an extended version of the *C* language. CUDA allows programmer to define a new class of function, called *kernels*. Whenever a *kernel* is called, it will be executed *N* different CUDA threads concurrently as opposed to only once like regular *C* functions. Many thread blocks assembled together to form a grid of blocks. From the hardware point of view, each SM may simultaneously execute up to 1024 threads or 32 warps i.e. groups of 32 parallel threads. Each block is associated to each SM and its shared memory (48 or 16 KB configurable). The later one along with the available 32 768 32-bit registers per SM, define and limit the number of execution thread block of each SM. At last, all the threads (from all blocks) have access to the same global memory and the read-only constant and the texture memory spaces which are persistent across kernel launches [5]. Briefly the specifications of the utilized GPU on which codes are executed are presented in Table 1. The personal computer used for the comparison in this study is a regular Intel based PC equipped with an Intel Core 2 Duo E7400 CPU [13].

Two compilers were used in present study are Visual Studio 2008 Compiler (C++) and NVCC (CUDA). Visual Studio Compiler is a portable compiler running on Windows platform, and can produce output for many types of processors. Visual Studio Compiler is not only a native compiler but also it can cross-compile any program, producing executable files for a different system from the one used by itself. Visual Studio Compiler is used for C/C++ languages in which there is direct access to the computer memory. In many applications, C/C++ languages have been used for developing low-level systems software and applications in where high-performance or control over resource usage are critical. However, great consideration is required to ensure that memory is accessed correctly to avoid corrupting other data structures. Table 2 shows the C++ compiler settings which are used in present study. Because time comparison is one of the concerns is present study, *Maximum Speed* Option for the C++ optimization is used to obtain the lowest runtime.

For GPU applications which consist of a mixture of conventional C++ code plus GPU functions, it is required to use NVCC or CUDA compiler [39]. The NVCC compiler separates the device functions from the CPU code, compiles the GPU functions using proprietary NVIDIA compilers/assemblers and compiles the CPU code using general purpose C/C++ compiler. This complication is available on the CPU platform and afterwards adds the compiled GPU functions as load images in the CPU object file. In the linking stage, specific CUDA runtime libraries is required for two purposes: (1) for supporting remote SIMD procedure calling and (2) for providing explicit GPU manipulation such as allocation of GPU memory buffers and CPU-GPU data transfer. In present study, primary options of NVCC were used for *Runtime API* mode. It is required to set the GPU architecture option as implemented GPU compute capability to have right access to GPU computing features.

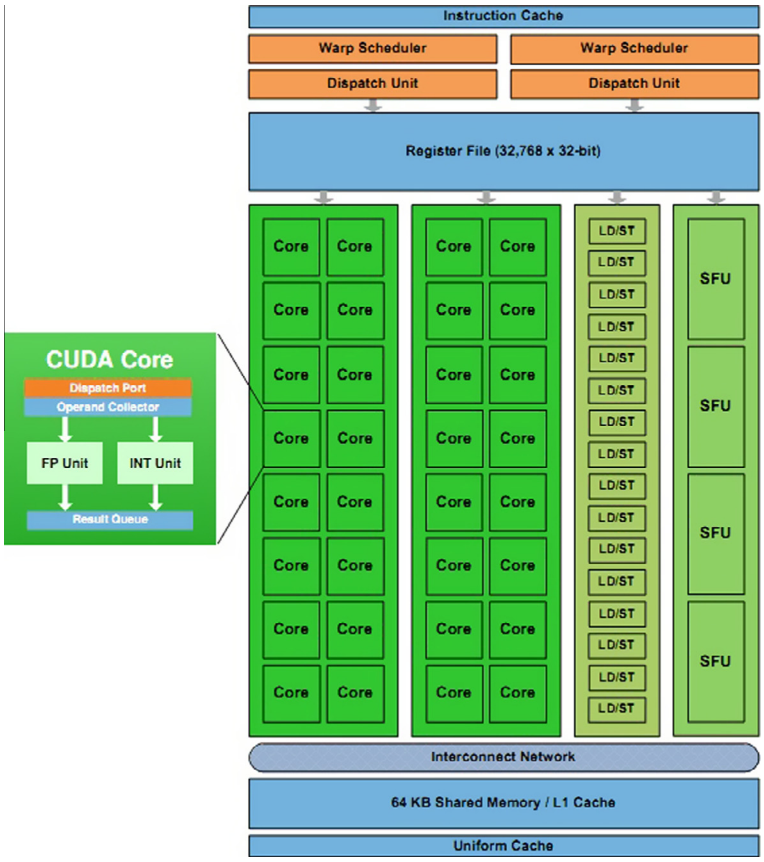


Fig. 8. Fermi Streaming Multiprocessor (SM) [5].

Table 1
Specifications of GPU hardware [14].

Features	GTX 480 GPU
Number of cores	480
Memory size (MB)	1585
Clock speed (MHz)	1401
Memory bandwidth (GB/s)	177.4

Table 2
Visual studio compiler options.

Setting	Option
Solution configuration	Release mode
Optimization	Maximum speed (/O$_2$)
Solution platform	64-Bit
OpenMP	Disabled

6. Results

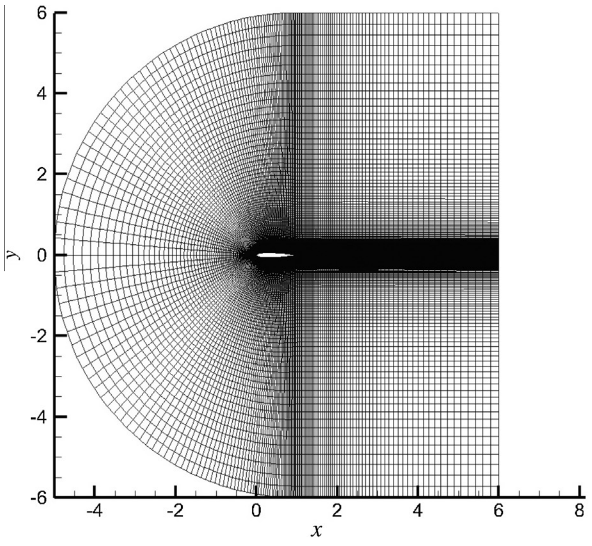
6.1. Computational implementation

This section is devoted to discussion of: (1) numerical accuracy; (2) present work methodology; and (3) test cases and results.

In recent years highly accurate numerical investigations are performed on GPUs [19,10,20]. But these highly accurate calculations are not precious without consideration of machine accuracy. The method numerical accuracy should be proportional to the round off error or machine accuracy. For example [19] showed accelerated computations of 7th and 9th order WENO scheme for the large scale simulations i.e. on the average 10^3 grid points on the GPU. Consequently, the numerical error of this work is approximately 10^{-20} needing at least double precision arithmetic for the calculations. But [19] used single precision arithmetic merely to show high amount of speedup, because GPU calculation of single precision arithmetic is nearly an order of magnitude faster than double precision. This is clear that this study needs further consideration. In present work, 2nd order finite-difference method is used for the various grid sizes. By considering the method accuracy and the largest grid size of 1024×1024, the numerical error is about 10^{-6} yielding to use the single precision arithmetic. It is strictly important for the GPU calculations to have consistent scheme of using numerical methods and arithmetic operation accuracy.

It is mentioned before that equations with the larger memory halos must access more neighboring values and are generally slower which distorts the output performance. In the grid generation process, the computation pressure is on the procedure of imposing grid spacing and orthogonality. Because these calculations have no memory halo, the computations is quiet “Aligned” and efficient on the GPU. Consequently, it is expected that present method of GPU execution has a little improvement on the grid generation process.

Fig. 9 shows GPU structured grid over NACA0012 airfoil with the grid size of 256×256. On the other hand, flow simulation equations have larger number of memory halo accesses. It is due to this fact that at least eight neighboring data are required for computing Stream function and Vorticity of each node. GPU results of present solver are in good agreement with similarity solution of Blasius for the Incompressible laminar flow over a flat plat as illustrated in Fig. 10. The rise of skin friction at trailing edge of the flat



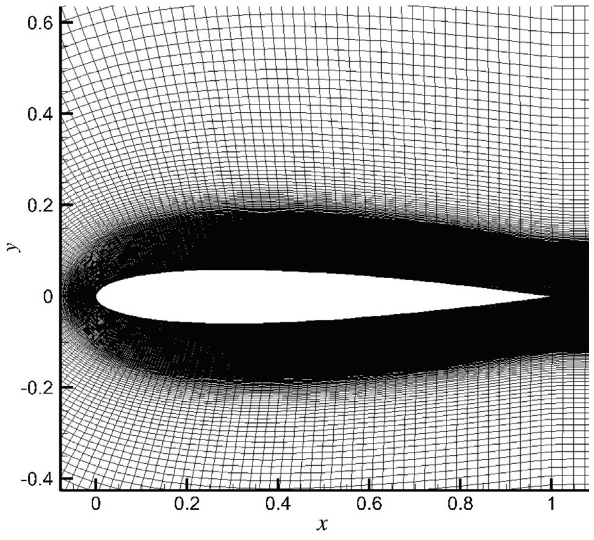


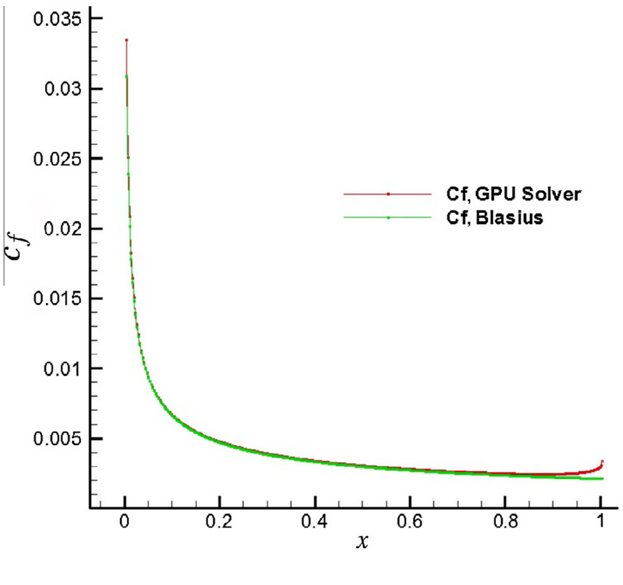
Fig. 9. GPU structured grid over NACA0012 airfoil: over view (left), close view (right).

plate is explained by triple-deck theory [9]. For the next, numerical results of laminar/turbulent flow over an airfoil is investigated. Fig. 11 shows the skin friction and pressure coefficient over NACA0012 airfoil for the Reynolds number of 5000. The separated flow in the turbulent wake region behind the NACA0012 airfoil in the different physical flow time is presented in Fig. 12a–d.

6.2. Performance analysis

Both numerical grid generation and flow solver are used to evaluate the performance of current GPU implementation. In order to evaluate the performance of presented GPU method, the experiments using a variety of combinations among simulation parameters and experimental platform deployment are conducted. It is clear that the first idea of using GPUs in non-graphical purpose was to reduce the runtime and achieve speedup [7]. Consequently, runtime and speedup were unanimously considered as the comparison criteria in almost all the GPU studies and investigations have been presented in literature [1,2,19,35,37]. Although runtime is a tangible comparison criterion needed to be presented in each study, for the following reasons they would not be a scientific factor for the comparison of the GPU implementation: (1) to compare the runtime of two programs, they have to be written in the same complexity patterns. In some cases implementing the CPU algorithm on the GPU will not be efficient or even possible; (2) runtime of each calculation method i.e. finite difference, finite elements, LBM, etc. fundamentally differs from others and is not comparable; and (3) each numerical algorithm i.e. explicit or implicit, 2nd or 4th order, etc. in specific method has a discrepant runtime (4) in the multiple GPU programming which has been grown in recent years [8,20] because domain decomposition and inter-GPU communications have a crucial impact on performance, the runtime and speedup would not show a justifiable expression of GPU implementation.

By considering these reasons in one hand and existing Compute Visual Profiler [33] on the other hand, the GPU performance analyzing is required to see the GPU implementation through a scientific conclusion. This analysis includes the evaluation of memory



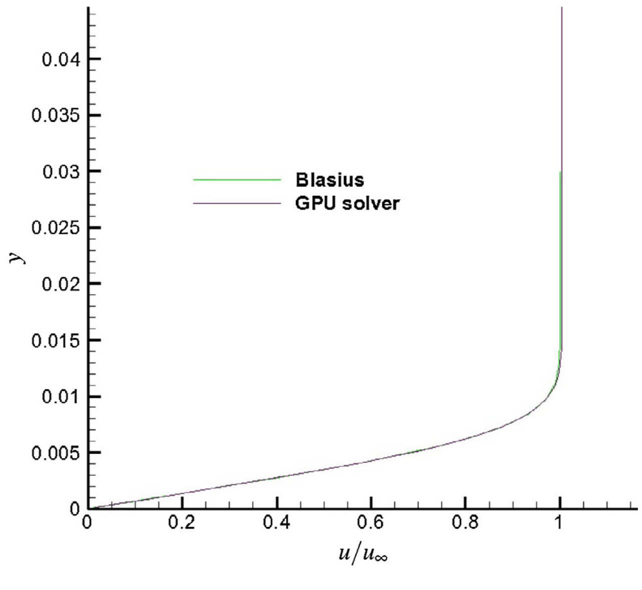


Fig. 10. Laminar flow over a flat plate: skin friction (left), velocity profile (right), *Re* = 100000.

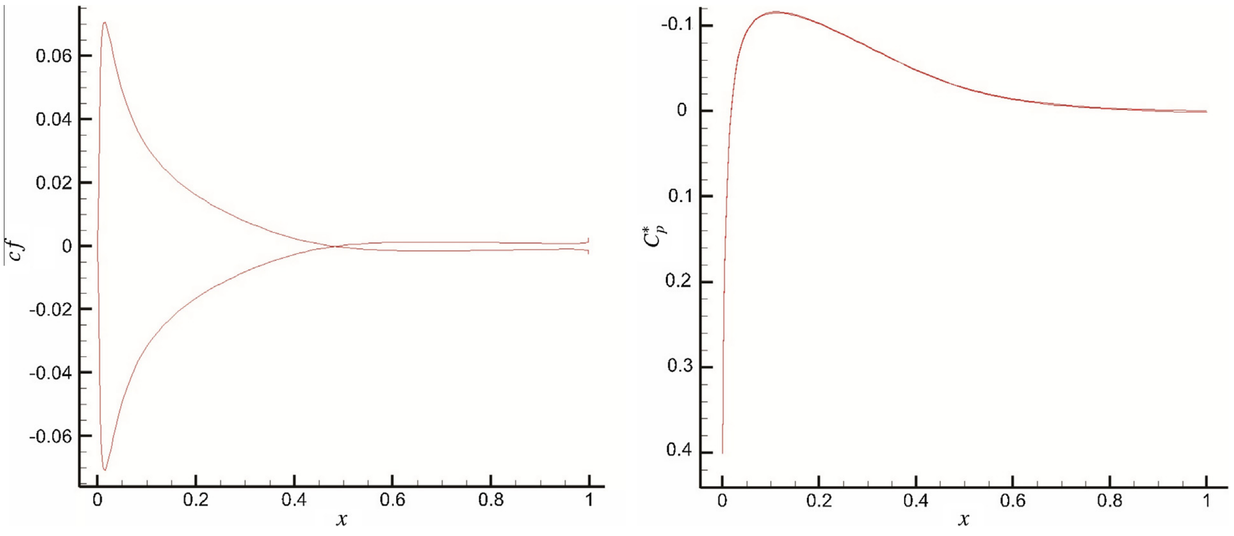


Fig. 11. Flow over NACA0012: skin friction (left) and pressure coefficient (right), *Re* = 5000, *AoA* = 0.

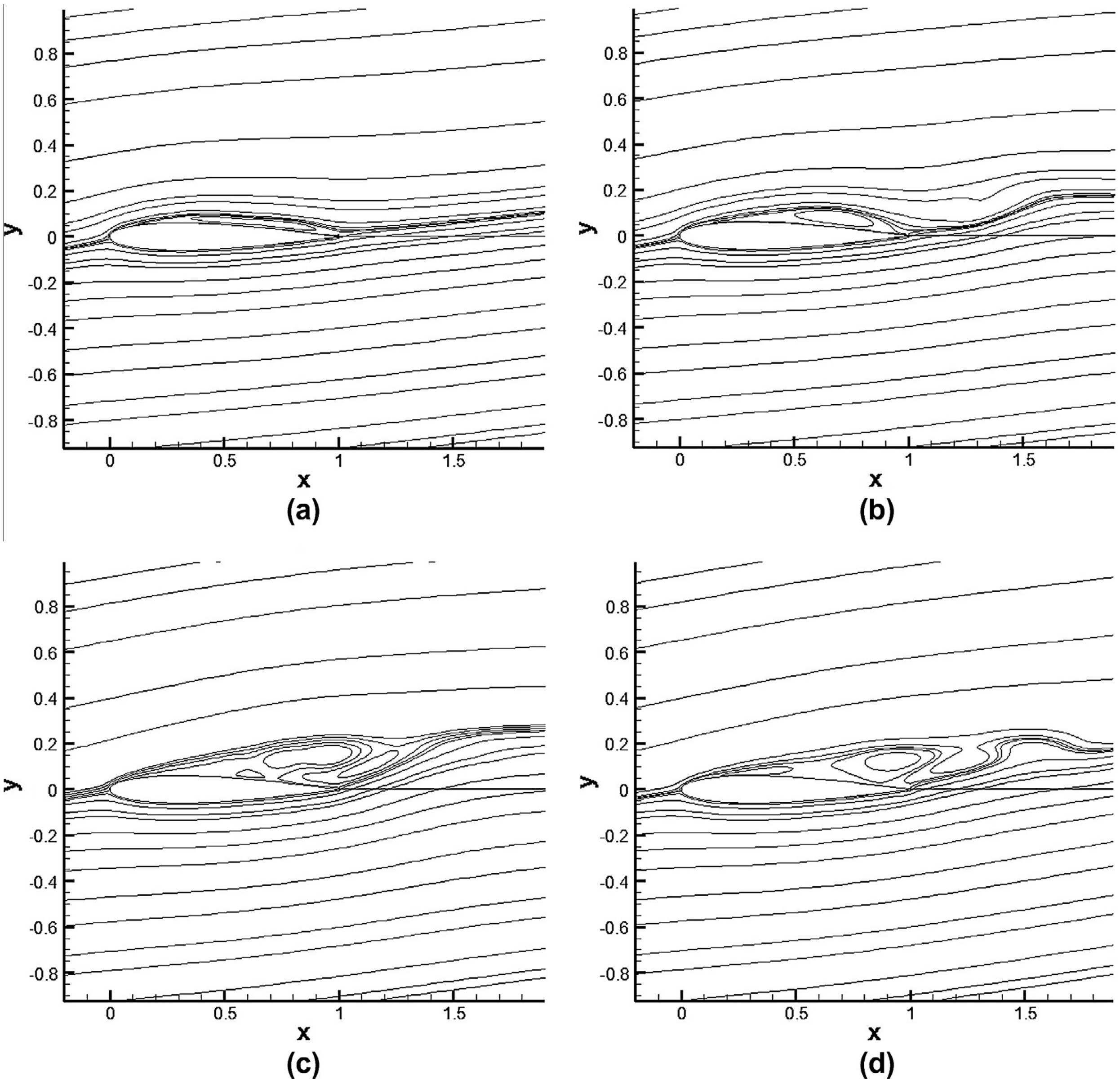


Fig. 12. Stream function contour, *Re* = 100,000 *AoA* = 8: (a) *t* = 2.5 s. (b) *t* = 5 s. (c) *t* = 7.5 s. (d) *t* = 10 s.

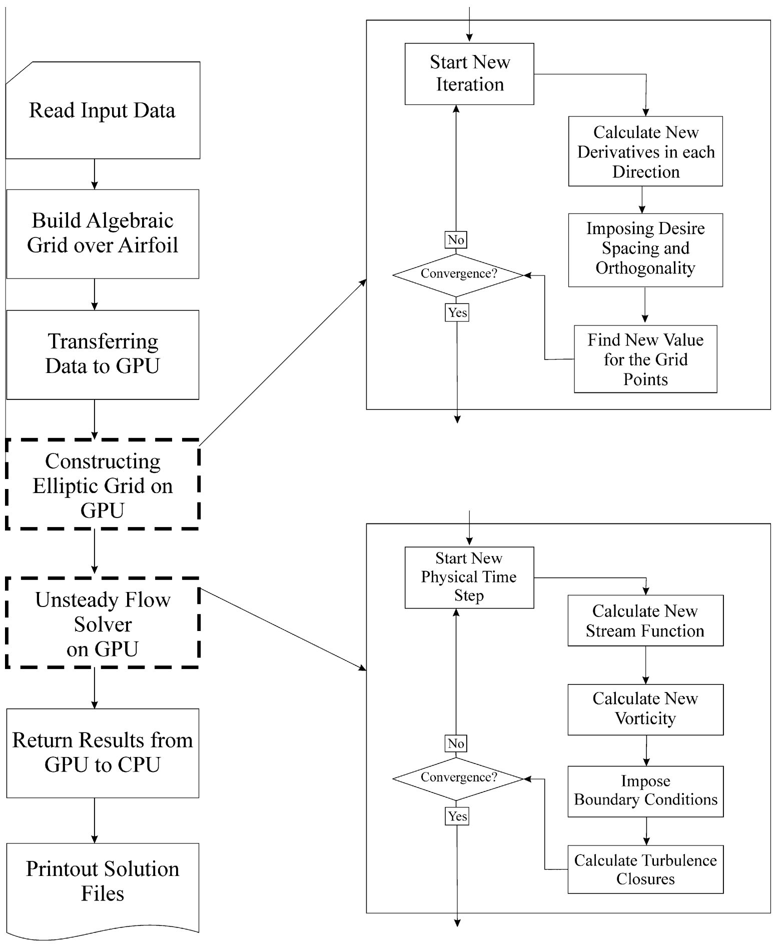

Read Input Data
Build Algebraic Grid over Airfoil
Transferring Data to GPU
Constructing Elliptic Grid on GPU
Unsteady Flow Solver on GPU
Return Results from GPU to CPU
Printout Solution Files
Start New Iteration
Calculate New Derivatives in each Direction
Imposing Desire Spacing and Orthogonality
Find New Value for the Grid Points
Convergence?
No
Yes
Start New Physical Time Step
Calculate New Stream Function
Calculate New Vorticity
Impose Boundary Conditions
Calculate Turbulence Closures
Convergence?
No
Yes

Fig. 13. Flowchart of the GPU-enabled programs.

Table 3
Comparison between GPU and CPU grid constructors runtime (time is measured in s).

Grid size	GPU time	CPU time	Speedup ratio
128 × 256	113.60	7720.25	67.95
256 × 256	200.00	13561.25	67.80
256 × 512	370.96	27757.42	74.83
512 × 512	724.76	91364.54	126.06
512 × 1024	1413.82	213113.21	150.73

Table 4
Comparison between GPU and CPU flow solvers runtime (time is measured in s).

Grid size	Present GPU time	Regular GPU time	CPU time
256 × 256	173.02	221.15	6241.85
256 × 512	302.18	434.07	12565.19
512 × 512	597.44	863.11	24976.37
512 × 1024	1150.32	2297.50	48788.64
1024 × 1024	2311.44	3267.00	83567.66

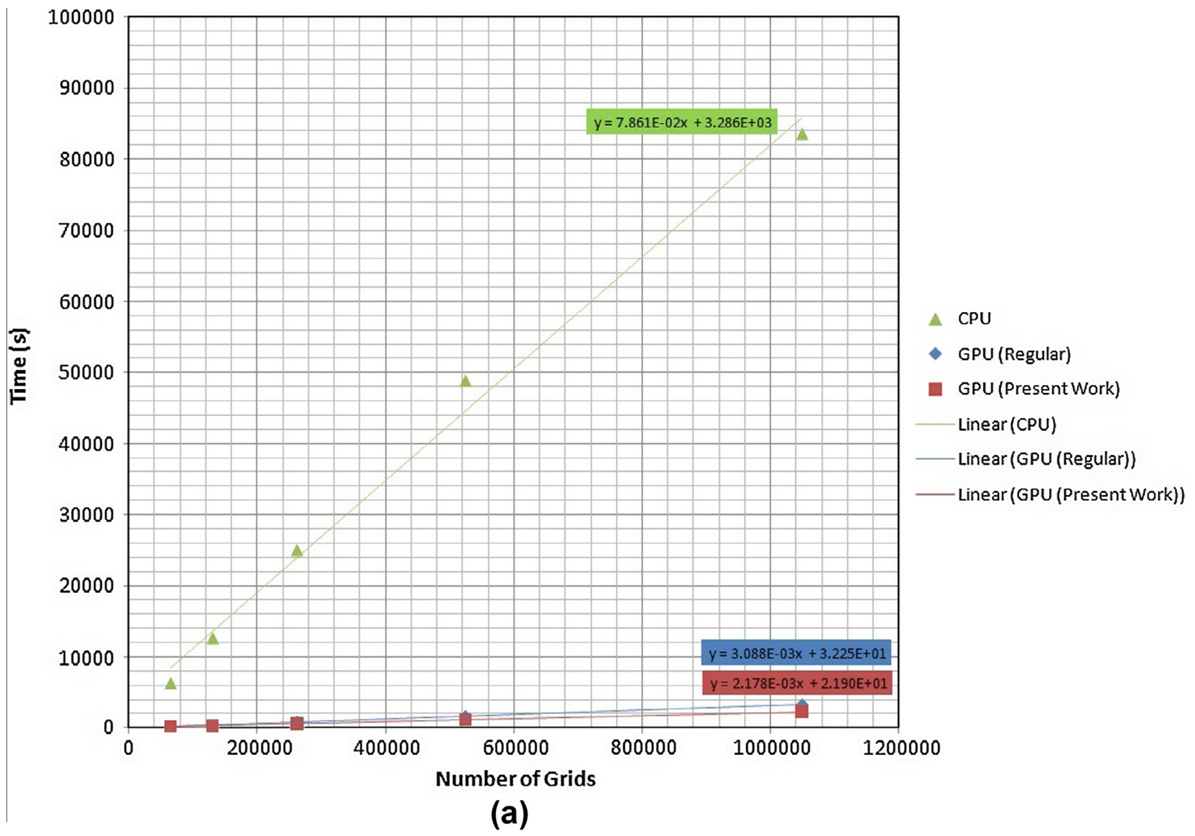
Table 5
Speedup of present and regular GPU work compared to CPU.

Grid size	Present GPU work	Reqular GPU work	Improvement(%)
256 × 256	36.07	28.24	28
256 × 512	41.55	28.93	44
512 × 512	41.83	28.95	44
512 × 1024	42.41	29.57	43
1024 × 1024	36.15	25.58	42

throughput, number of bank conflicts and occupancy [4]. As a result, we present results in term of execution time and their respective speedups in association with the GPU performance analyzing. Table 3 shows the runtime of both version of the grid generation programs and the achieved speedups. The succeeded speedup (Fig. 15) in this case is about 100× on the average for the several grid sizes. Another important point is greater time vs. grid size slope of CPU calculations. Fig. 15 shows that CPU time slope is one hundred fifty times greater than GPU. Consequently, simple GPU parallelization can be effectively useful as shown by massively generated grid in present study.

As it was mentioned previously, different CTA configurations are implemented for the GPU based flow solver. Tables 4 and 5 show the runtime and speedups of present GPU work, regular GPU and the sequential CPU computations. Present GPU work results in providing approximately 40% improvement for the runtime and speedup in almost all the grid sizes. The details of measured speedup and runtime are illustrated in Fig. 14–17.

The Compute Visual Profiler is used to evaluate performance analysis of GPU-based codes. In the grid generation process, because no shared memory is used in GPU programming, there is no bank-conflict. Moreover, as explicated before, there is no bank-conflict in the flow solver program. In addition because the shared memory of GTX 480 can be configured, shared memory is set as 16 KB and L1 cache is set as 48 KB. It was figured out that this configuration will make the performance a bit better. Using of Compute Visual Profiler, it was demonstrated that GPU version of grid generation program has overall memory throughput of 98.79 GB/s. By comparing the theoretical bandwidth of 177.4 GB/s it can be



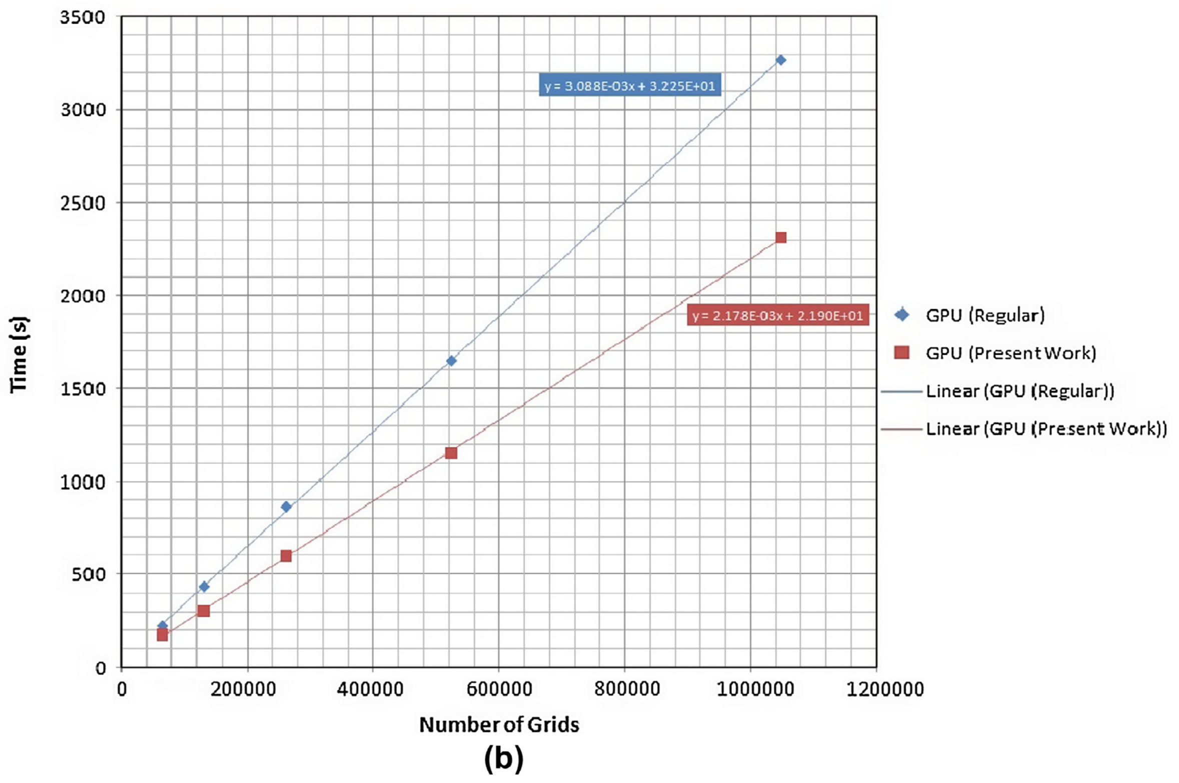


Fig. 14. Execution time of present work and regular GPU method compared to: (a) sequential CPU execution and (b) each other (Flow Simulation).

inferred that the present GPU grid generation program achieved 55.6% of its theoretical value.

Additional parameter which can be evaluated by Compute Visual Profiler is number of global memory requests. As mentioned before, it is expected that present configuration has fewer redundant memory requests. It was observed that for the grid size of 1024 × 1024, the total number of global memory requests has 40% reduction on average. As a result, present work alleviates the

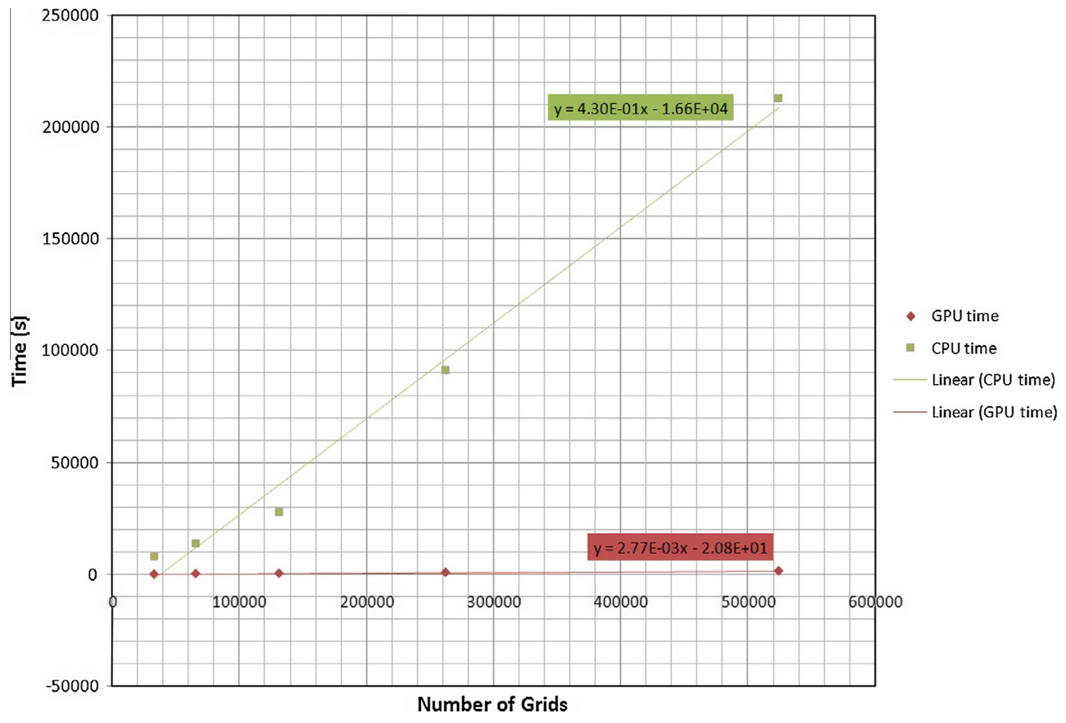


Fig. 15. Execution time of regular GPU method compared to sequential CPU execution (grid construction).

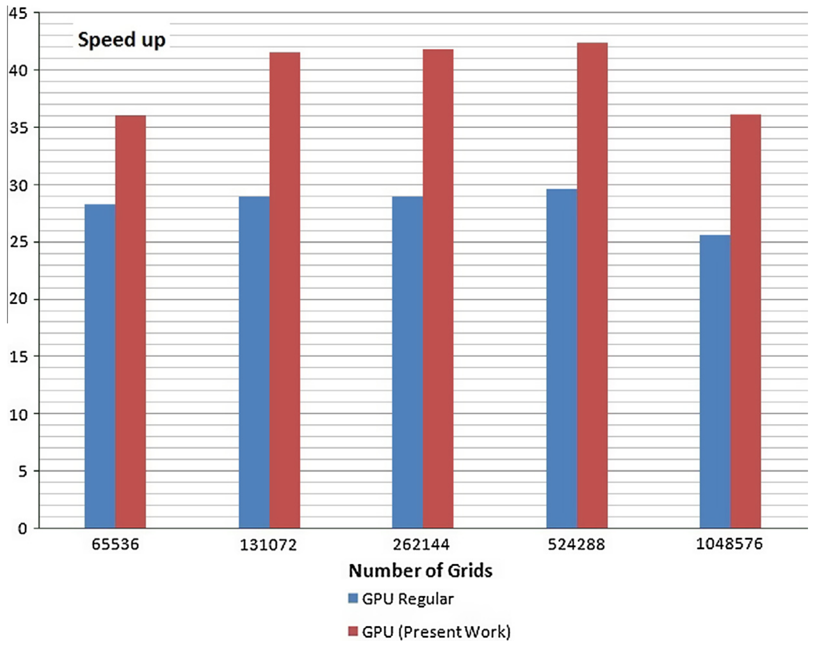


Fig. 16. Speedup of present work and regular GPU method compared to CPU execution (flow simulation).

pressure on global memory bandwidth without additional usage of register memory and hence generates higher memory throughput as expected before. For the GPU based flow solver it is measured that the regular GPU solver has overall memory throughput of 105.21 GB/s which is almost 59.30% of theoretical value. By implementing the new approach of GPU execution, overall memory throughput of 135.17 GB/s achieved which is 76.23% of theoretical value. Not only does present method improve the total memory throughput by 17%, but also it has lower runtime.

Another part of performance analyzing is Single Multiprocessors (SMs) occupancy which is defined as the ratio of the number of active warps per SM to the maximum number of active warps. Occupancy of each GPU kernel is specific because a SM can only schedule one kernel at a time [26]. Moreover, running multiple warps allows each SM to hide global memory latency by switching out stalled warps for warps which are ready to execute instructions. It is common for each numerical procedure that innovative implementation improves one aspect of computing i.e. runtime or throughput, but may worsen another. It was shown before that as throughput and runtime points of view, present work improves the GPU computations. To have complete performance analysis, it is important to compare the occupancy which is the last

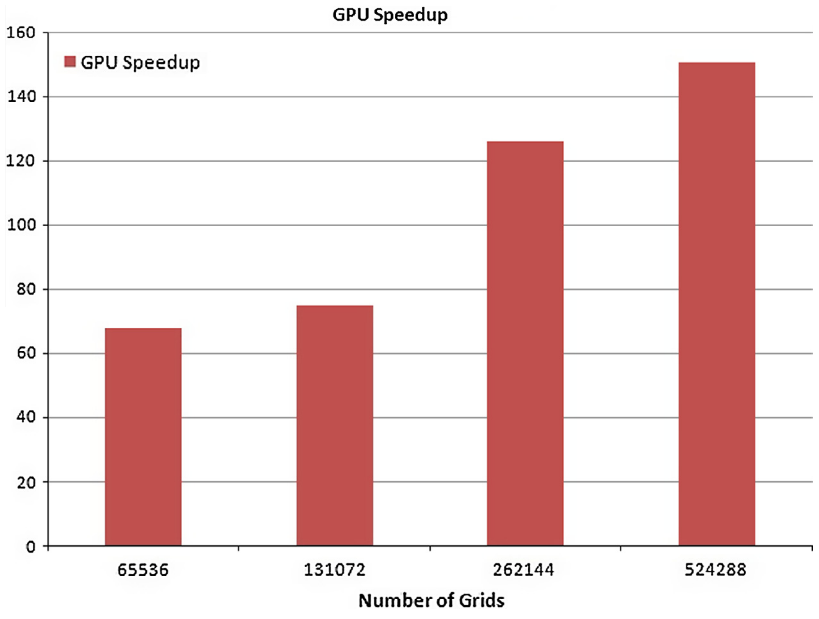


Fig. 17. Speedup of regular GPU method compared to CPU execution (grid construction).

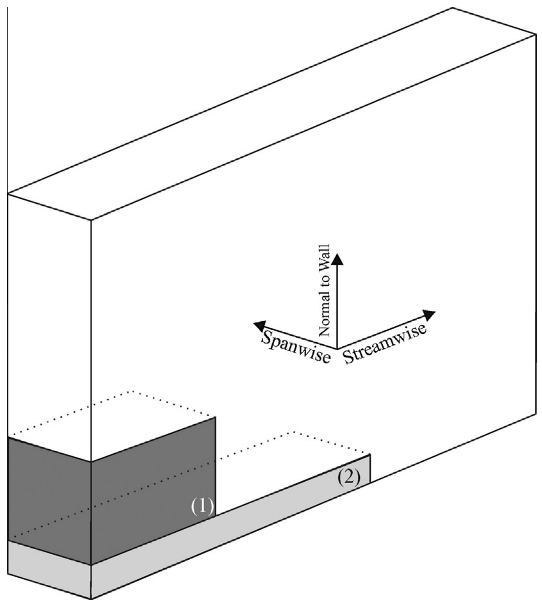


Fig. 18. CTA configuration for 3D simulation.

performance factor. Note that the GPU occupancy is the ratio of the activated threads to the maximum possible active threads in GPU which is considered as appropriate metric of implementation efficiency. It is notable that present GPU method has the same occupancy as regular one. It is clear that for throughput performance and runtime aspects, the present GPU method promotes the GPU computations with the constant 66.67% occupancy.

7. Conclusions and future lines

We have investigated two dimensional unsteady flow solver for the incompressible Navier–Stokes equations based on Stream function-Vorticity formulation for graphics processing units. Both grid construction and flow simulation have been taken placed through GPU kernels with the Finite Differencing Time Domain (FDTD) method. Two methods for implementing finite difference computations are presented with the concept of Cooperative Thread Array (CTA) configuration. We have shown that the one dimensional configuration of GPU CTAs can reliably provide higher coalesced memory access pattern. In addition, one dimensional CTA configuration has fewer redundant global memory requests in contrast to two dimensional one. It was shown that for the grid size of 1024×1024, the present method needs only 60% of naive method requests for data loading. Consequently, the present method has lower limit on GPU memory bandwidth. Although both GPU versions has over $28\times$ speedup over than a sequential CPU version, the more coalesced pattern of the present configuration showed up to 44% decrease in execution time comparing to the naive GPU method. In addition, the guidelines of using present work in three dimensional problems were also explained.

It was depicted that comprehensive GPU analysis is required to see the GPU implementations through a completely scientific conclusion. Hence, careful GPU analyses including Global memory throughput, occupancy and bank conflict were carried out. Compute Visual Profiler is used to analyze the GPU performance. It was figured out the present work has 17% increase in global memory throughput with the same amount of occupancy. It was also clarified that the present work has no bank conflict with the lower usage of register memory. As a result, from scratchpad memory point of view, present method has optimized scheme. It was also shown that however modern GPGPUs have opened the door to intensive and fast data-parallel arithmetic, in any GPU study it is required to consider the consistency between machine accuracy and numerical implementation.

References

[1] Hagen TR, Lie KA, Natvig JR. Solving the Euler equations on graphics processing units. Comput Sci 2006;3994(2):220–7.
[2] Elsen E, LeGresley P, Darve E. Large calculation of the flow over a hypersonic vehicle using a GPU. J Comput Phys 2008;227(24):10148–61.
[3] Jespersen DC. Acceleration of a CFD Code with a GPU. Sci Program Explor Lang Express Medium Massive On-Chip Parallelism 2010;18(3-4).
[4] Ryoo S, Rodrigues CI, Baghsorkhi SS, Stone SS, Kirk DB, Hwu W. Optimization principles and application performance evaluation of multithreaded gpu using

cuda. In: PPoPP '08: proceeding of the 13th ACM SIGPLAN symposium on principles and practice of parallel programming; 2008.
[5] NVIDIA Corporation. Nvidia's next generation cuda compute architecture: Fermi; 2009.
[6] Nere A, Hashmi A, Lipasti M. Proïňling heterogeneous multi-GPU systems to accelerate cortically inspired learning algorithms. In: IPDPS '2011: proceeding of IEEE international parallel and distributed processing symposium; 2011.
[7] NVIDIA Corporation. NVIDIA CUDA™ Programming Guide, Version 3.2; 2010.
[8] Playne DP, Hawick KA. Comparison of GPU architectures for asynchronous communication with finite-differencing applications. Concurr Comput: Practice Exper 2012;24(1):73–83.
[9] Messiter AF. Boundary layer flow near the trailing edge of a flat plate. SIAM J Appl Math 1970;18(1):241–57.
[10] Adams S, Payne J, Boppana R. Finite difference time domain (FDTD) simulations using graphics processors. In: DoD high performance computing modernization program users group conference (HPCMP-UGC); 2007.
[11] Moazeni M, Bui A, Sarrafzadeh M. A memory optimization technique for software-managed scratchpad memory in GPUs. In: SASP '09: proceeding of IEEE 7th symposium on application specific processors; 2009.
[12] Kirk D. NVIDIA CUDA software and GPU parallel computing architecture. In: ISMM '09: proceedings of the 6th international symposium on memory management; 2007.
[13] Intel. Intel Core 2 Duo Processor E7400 Product Details [Online]. <http://ark.intel.com/products/36500/Intel-Core2-Duo-Processor-E7400-(3M-Cache-2_80-GHz-1066-MHz-FSB), 2012>.
[14] NVIDIA Corporation. NVIDIA GeForce GTX 480/470/465 GPU Datasheet; 2010.
[15] Bakhoda A, Yuan GL, Fung WWL, Wong H, Aamodt TM. Analyzing CUDA workloads using a detailed GPU simulator. In: ISPASS '09: proceeding of IEEE symposium on performance analysis of systems and software; 2009.
[16] Lindholm E, Nickolls J, Oberman S, Montrym J. NVIDIA Tesla: a unified graphics and computing architecture. IEEE Micro 2008;28(2):39–55.
[17] Micikevicius P. 3D finite difference computation on GPUs using CUDA. In: Proceedings of 2nd workshop on general purpose processing on graphics processing units; 2009.
[18] Leist A, Playne DP, Hawick KA. Exploiting graphical processing units for data-parallel scientific applications. Concurr Comput: Practice Exper 2009;21(18): 2400–37.
[19] Antoniou AS, Karantasis KI, Polychronopoulos ED, Ekaterinaris JA. Acceleration of a finite-difference WENO scheme for large-scale simulations on many-core architectures. In: Proceedings of 48th AIAA aerospace sciences meeting including the new horizons forum and aerospace exposition; 2010.
[20] MichÃl'a D, Komatitsch D. Accelerating a three-dimensional finite-difference wave propagation code using GPU graphics cards. Geophys J Int 2010;182(1):389–402.
[21] TÃűlke J. Implementation of a lattice Boltzmann kernel using the compute unified device architecture developed by NVIDIA. Comput Visual Sci 2009;13(1):1–11.
[22] Wang P, Abel T, Kaehler R. Adaptive mesh fluid simulation on GPU. New Astron 2010;15(7):581–9.
[23] Sorenson LR. A computer program to generate two-dimensional grids about airfoils and other shapes by the use of Poisson's equation. NASA Tech Memorandum 1980;81:198.
[24] Obrecht C, Kuznik F, Tourancheau B, Roux JJ. A new approach to the lattice Boltzmann method for graphics processing units. Comput Math Appl 2010;61:3628–38.
[25] Asouti VG, Trompoukis XS, Kampolis IC, Giannakoglou KC. Unsteady CFD computations using vertex–centered finite volumes for unstructured grids on graphics processing units. Int J Numer Methods Fluids 2011;67(2):232–46.
[26] Bailey P, Myre J, Walsh SDC, Lilja DJ, Saar MO. Accelerating lattice boltzmann fluid flow simulations using graphics processors. In: Proceedings of the international conference on parallel processing (ICPP); 2009.
[27] GueÌÃvremont G. Finite element vorticity-based methods for the solution of the incompressible and compressible Navier–Stokes equations, Ph.D. Thesis, Concordia University; 1993.
[28] Davidson A, Owens JD. Register packing for cyclic reduction: a case study. In: Proceedings of the 4th workshop on general purpose processing on graphics processing units; 2011.
[29] KlÃűckner A, Warburton T, Bridge J, Hesthaven JS. Nodal discontinuous Galerkin methods on graphics processors. J Comput Phys 2009;228(21): 7863–82.
[30] Sanders J, Kandrot E. CUDA by example: an introduction to general-purpose GPU programming. Addison-Wesley; 2010. ISBN 978-0131387683.
[32] Christensen HF. A vorticity–stream function formulation for turbulent airfoil flows, Ph.D. Thesis, Technical University of Denmark; 1993.
[33] NVIDIA Corporation. COMPUTE VISUAL PROFILER, User Guide; 2010.
[34] Kim S, Alonso JJ, Jameson A. Multi-element high-lift configuration design optimization using viscous continuous adjoint method. J Aircraft 2004;41(5): 1082–97.
[35] Eller PR, Cheng C, Albert DG, Liu L. Performance improvement of the 2-D finite difference time domain acoustic wave simulation using multiple GPUs. In: Presented in the 27th army science conference; 2010.
[36] Chan TF. Stability analysis of finite difference schemes for the advection–diffusion equation. SIAM J Numer Anal 1984;21(2):272–84.
[37] Sosnick M, Hsu W. Efficient finite difference-based sound synthesis using GPUs. In: Proceedings of SMC conference; 2010.
[38] Kirk DB, Hwu W. Programming massively parallel processors: a hands-on approach. Morgan Kaufman-Elsevier; 2010. ISBN: 978-0-12-381472-2.
[39] NVIDIA Corporation. The CUDA compiler driver NVCC; 2007.
[40] Liu J, Ma Z, Li S, Zhao Y. A GPU accelerated red-black SOR algorithm for computational fluid dynamics problems. Adv Mater Res 2011;320:335–40.
[41] Gohari SMI, Esfahanian V, Moqtaderi H, Mahmoodi D. Coalesced simulations of the incompressible Navier–Stokes equations over airfoil using GPU. In: GPU technology conference. California, USA; 2012.
[42] Bauer M, Cook H, Khailany B. CudaDMA: optimizing GPU memory bandwidth via warp specialization. In: Proceedings of 2011 international conference for high performance computing, networking, storage and analysis; 2011.
[43] Vanka SP, Shinn AF, Sahu KC. Computational fluid dynamics using graphics processing units: challenges and opportunities. In: Proceedings of the ASME international mechanical engineering congress and exposition; 2011.