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

ARTICLE INFO

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

ABSTRACT

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:

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

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:

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

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]:

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

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