Computational Fluid Dynamic on GPU

Ting Xingdong[1], Sen Li [1]

1.Supercomputing Center, Chinese Academy of Sciences

1.Introduction

Over the past few years, GPUs (graphics processing units) have seen a tremendous increase in performance. In order to develop a scalable, high-accuracy CFD program and test GPU performance in this area, we have ported three two-dimensional CFD codes to the CUDA and OpenCL platforms. The cases include codes of incompressible Navier-Stokes (N-S) equations, Euler and compressible N-S equation. One of the cases (airfoil, as shown in Figure 1) has now been used as a component in Hoam-OpenCFD project to simulate flow turbulence on the airplane wing.

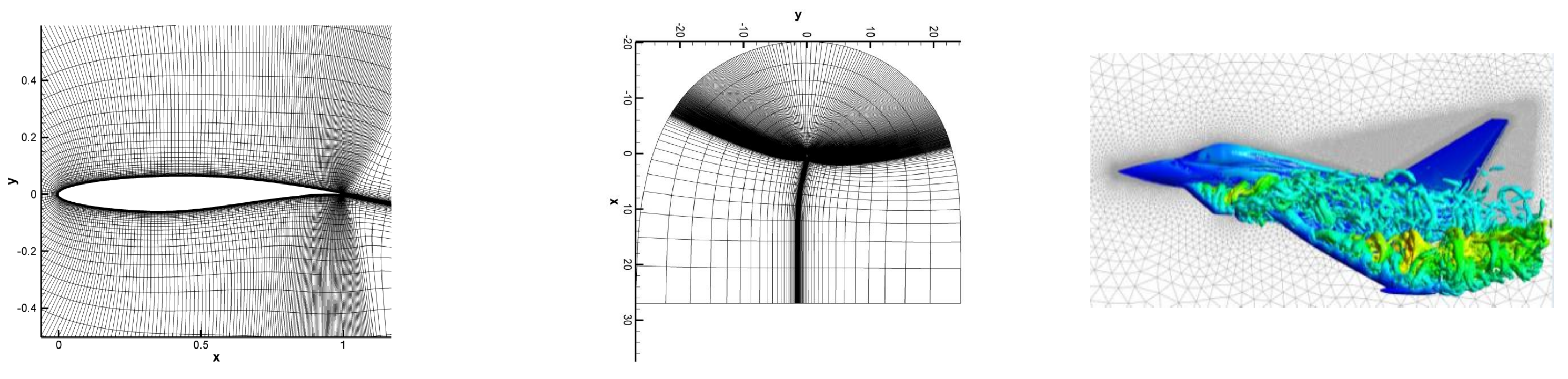

Figure1: airfoil and its original grid and new grid

In the following parts, we give detailed descriptions of the problems, followed by their CUDA implementation together with code optimization strategies. Next we show our experiment results. We demonstrate that CUDA can accelerate around 16 to 63× speed up in most of our cases.

2.Problem Description

The code we ported to the GPU included 2D cave flow, 2D Riemann problem, and 2D flow over a RAE2882 airfoil. We used finite differences to solve all the problems. The following table summarizes details of problem description and numerical methods.

Test	Descriptions	Equations	Numerical method	Grid resolution
1	2D cave flow	Incompressible N-S	3rd order quick method;laminar	96*96-1024*1024
2	2D riemann	Euler	2rd order NND and 1st order Euler	512*512-1024*1024
3	2D flow over a RAE2882 airfoil	Compressible N-S	5th order upwind scheme for inviscousterm;6th order central scheme for viscous term;3rd Runge-Kutta;B-L model;1st for boundary 2d for sub-boundary	369*65

3.CUDA Implementation of CFD Code

In all cases, the CPU is responsible for the initialization of the data, reading initial and grid files and coping the data into a space of the same size on device memory of GPU. The GPU takes over the most expensive computing until the loop is over. Finally, the CPU handles the post-processing. The entire procedure can be divided into the following 4 steps:

1.Decomposition and thread mapping. For structured grid applications, a natural way is to split the domain into smaller parts which can fit into the thread block. Each thread computes the updated variables of one element within each of the smaller parts in a straightforward way.

2.Memory Access. The algorithm dictates the complexity of the stencil. Consider a straightforward 2D 5-point stencil. To update the element *i*, *j*, thread *i*, *j* needs to read its four neighbours. This means that data of each point will be loaded at least four times. Obviously, it is too expensive to let each thread directly access data from external DRAM which has about 500 cycles latency. An efficient method is to copy data within one block from DRAM into shared memory, which serves as cache where threads read directly to reduce the cost.

3. Shared Memory Padding. Because threads on the block boundary have only three neighbours in the same thread block, shared memory must be padded with ghost cells before computing

4.Execution. Each thread is identified with a unique ID in the whole grid, and the kernel executed across these parallel threads in the grid.

4.Optimization Techniques

We used several techniques to optimize our codes. First, as we described in the CUDA Implementation section, we used shared memory to store ghost cells in order to reduce global memory access. Second, we use texture memory, a read-only memory space with 2D rotationally-invariant caching, to minimize the penalty of dealing with ghost cells. We use texture when performing complex stencil differencing. Third, when a loop within a kernel incurred, we unroll the loop to reduce divergence. We also experimented with different block and grid sizes to see which is the best for our program.

5.Performance Results

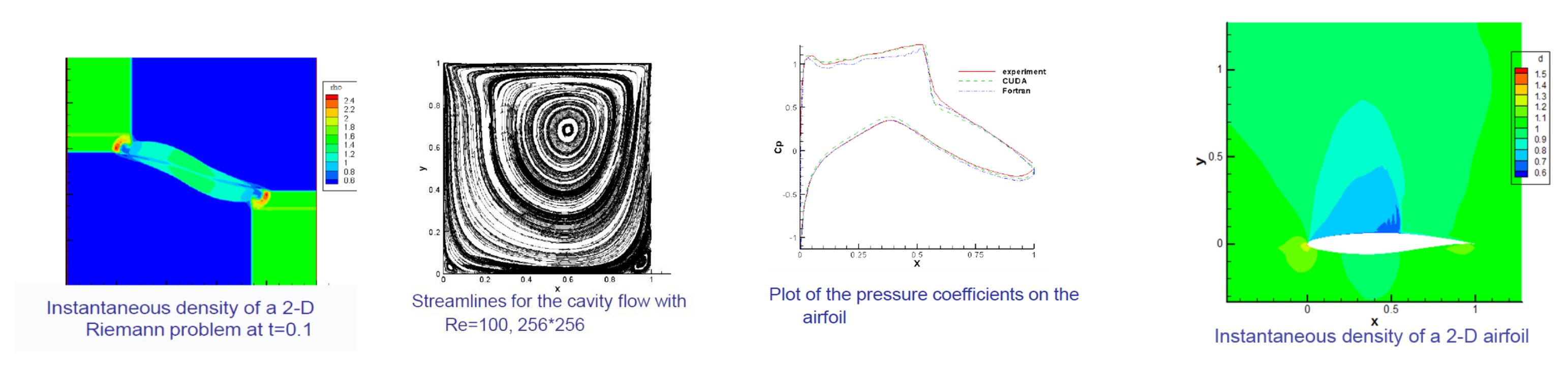

Figure 2: experiment results of cavity flow, Riemann problem and 2-D airfoil problem

The figures above are respectively the streamlines for cavity flow, instantaneous density of a 2-D Riemann problem at t=0.1 and plot of the pressure coefficients on the airfoil. The following table illustrates their corresponding speed:

Test	Grid size	GPU time	AMD CPU time	GPU speedups
2D Riemann Problem	512*512	4.67s	148.42s	31.7
2D Cavity flow	1024*1024	24.3s	1543s	63.5
2D airfoil	1024*128	6m	100m	16

We also tried the OpenCL platform to measure performance on the cavity flow problem, with the following results:

scale size	serial	openmp(4 cores)	OpenCL(Nvidia)	OpenCL(ATI)	CUDA
512 * 512	13.9844	9.406	2.975	7.588	1.34
1024 * 1024	70.83	52.2614	9.213	24.785	4.81
2048 * 2048	281.4307	217.9449	32.651	91.246	17.54
4096 * 4096	3823.8038	1515.912	124.742	XX	66.91
5120 * 5120	6843.9784	2378.9607	196.625	XX	108.3499

6.Conclusion

By using graphic hardware, we have demonstrated that a substantial performance gain can be achieved on some of the CFD codes. Based on these results, we can expect that GPU will be a powerful tool in the future scientific computing.

7.Reference

Tingxing Dong,Xinliang Li, Sen Li, Xuebin Chi *Acceleration of Computational Fluid Dynamic Codes on GPU*,
Tingxing Dong (2010), *Simulation of Transonic Flow over Airfoil on GPU*, SCCAS