CS267 Final Project: Eulearian Fluid Simulation for Computer Graphics in CUDA

Woody Ki Fung Chow

University of California, Berkeley
woody.kfchow@berkeley.edu

I. Introduction

In this final project, I created a high performance Eulerian fluid simulation kernel for Computer Graphics in CUDA.

If one is by any chance interested in the rendering of the fluid data generated by the simulation kernel, he can visit my write up page (http://woodychow.com/berkeley/cs283/fluid) for the class Advanced Computer Graphics.

II. Motivation

At the time of 2013, most real time applications, namely video games, simulate fluids by particle based system with limited number of particles because of its robust performance. Eulerian fluid simulation is another approach that is capable of realistic simulation, yet rarely used in real time applications because of its slow performance. The bottleneck of Eulerian fluid simulation is solving discretized 3D Poisson equations, which is a huge matrix at each time step.

III. Project Context

The goal of this project is to create a high performance Eulerian fluid simulation kernel, fused with a robust parallel sparse matrix solver using CUDA. The data structure underlying Eulerian fluid simulation is Structured Grid. Optimization techniques discussed in class should be used to improve the code.

IV. Theory

The simulation method follows [Rob08] closely. Incompressible Navier-Stokes equations are used to model the fluid. It can be divided into the following four parts.

IV.1 Advection

Semi-Lagrangian Advection is used, which is, updating quantities of each grid cell by tracking a particle centered at the grid cell back a time step. Runge-Kutta method is used to find out the origin of the particles. In our case of smoke, velocity, smoke concentration and temperature are advected.

IV.2 Body Force

Body force on the fluid includes gravitational force in most cases, and buoyancy in our case of smoke. Body force is applied to each grid cell individually at each time step.

IV.3 Diffusion

Diffusion can be modeled using a Laplacian term in PDE as described in [Rob08]. In short, finite differences of smoke and concentration are taken with the neighbouring cells are added to the cell adjusted by one arbitrary parameter for each respectively.

IV.4 Pressure Solve

Solve for pressure for each grid cell such that the result velocity field satisfies the incompressibility condition of the incompressible flow Navier-Stokes equations inside the fluid.

This involves solving a matrix since the pressure of each cell is implicitly defined and depends on the neighbouring pressure values, which happens to be a discretized 3D Poisson equation in the case of Eulerian simulation. The matrix is gigantic since the dimension is number of grids by the number of grids, which is width * height * depth by width * height * depth. Luckily, the matrix happens to be symmetric positive-definite, which one can take advantage of.

V. Iterative Solver

The core part of the simulation kernel is the matrix solver. Since the matrix to be solved happens to be symmetric positive-definite, Jacobi and Conjugate Gradient method can be use to solve the matrix iteratively [BHM00].

VI. Implementation Details

VI.1 Optimization Techniques

VI.1.1 Exploit GPU Architecture

As CUDA needs many threads to hide latency, the number of threads casted is maximized. For example, one thread is casted for each grid cell for per cell operations such as advection, body force and pressure update.

VI.1.2 Using Vendor Libraries when Appropriate

cuBlas library is used heavily in the simulation kernel, routines such as axpy, dot, amax are called in the iterative solver. Also, copy routine of cuBlas is used throughout the simulation kernel.

Reduce function of the Thrust library is used to calculate the mean of the residual vector in the iterative solver.

VI.1.3 Compact Representation of the Matrix

Entries are resolved in runtime from the voxelized scene geometry. By doing so, memory space for 4 * width * height * depth variables are saved assuming only half of the symmetric matrix is stored. This is especially beneficial in high resolution simulation.

Moreover, by using a tailor-made "ApplyA" function that exploits the repetitive pattern in each row in the matrix of discretized 3D Poisson equation, matrix vector multiplication in the kernel is at least 5 times faster than using dgemm routine of cuSparse. This is surprising yet understandable because cuSparse should be tuned for general symmetric matrices but not one specific type of symmetric matrix.

VI.2 Unexpected Behavior of CUDA

I have experienced strange behavior of CUDA constantly. I suspect that Nvidia GPU driver is the cause of them. Yet, the driver running on the test platform is the newest available. I am unable to eliminate the below unexpected behavior at the moment.

VI.2.1 Compiled code ran perfectly before, yet CUDA kernel would not launch after reboot

The common cause for this kind of strange behavior is usually memory allocation error. However, every return code of cudaMalloc call is checked, and none of them returned an error code. Mysteriously, the code would run perfectly again sometimes by recompiling the code or running it using Nvidia Visual Profiler, which is provided in the CUDA SDK.

VII. Results

The program is tested on a GeForce GTX 660 Ti with 2GB ram. The 600 series are known to perform really bad on double precision operations. The result confirms it as well.

	Jacobi Float	Jacobi Double	CG Float	CG Double
Time to 1e-4(s)	18.5s	39.7s	0.85s	1.42s
Iteration to 1e-4(s)	6151	10835	277	341
Time Per Iteration	3.0ms	3.6ms	3.1ms	4.2ms

Table 1: Performance of iterative solvers on a 128x128x128 Grid

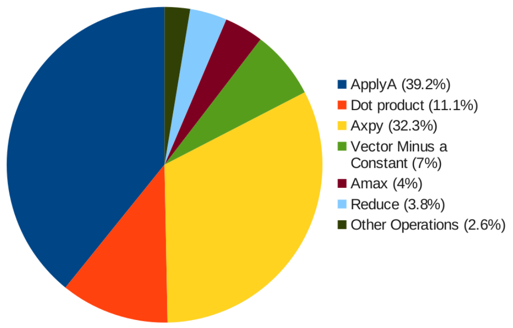

Figure 1: Percentage Time Spent in Different CUDA Kernels

The program stays in highly optimized functions and vendor libraries for most of the time. Moreover, according to Nvidia Visual Profiler, GPU utilization rate of the code is above 80%. That implies that the program is well-tuned.

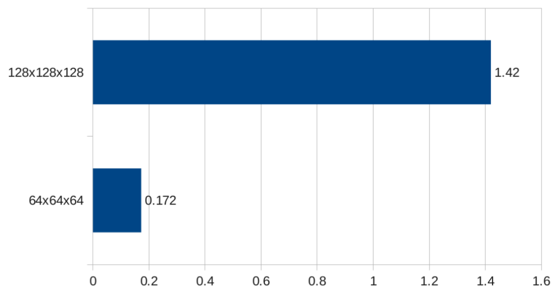

Figure 2: Time to 1e-4(s) for different resolution

Time for higher resolution is not available because the GPU does not have enough memory space. Yet, the relationship of time taken between 64x64x64 and 128x128x128 is linear. 128x128x128 has 8 times more grid cells than 64x64x64, and the time taken for 128x128x128 is about 8 times greater.

VIII. Conclusion and Future Work

The iterative solver runs very quickly thanks to GPGPU technology. The entire program achieved a GPU utilization rate of 80% and above. However, the convergence rate of both numerical solvers are unsatisfactory.

The simulation kernel would perform much faster if Multigrid method is implemented as a direct solver or a preconditioner of Conjugate Gradient method. Multigrid preconditioned conjugate gradient method can takes less than 20 iterations for the residual error to go under 1e-04 according to [MST10].

The best time per frame achieved in this project for 128x128x128 is 0.85s. I believe with Multigrid method implemented and a more powerful GPU, real time Eulerian fluid simulation can be achieved. I look forward to seeing Eulerian fluid simulation being used in real time applications.

References

[BHM00] Briggs W., Henson V., McCormick S.: A Multigrid Tutorial. Second Edition. Siam (2000).

[Bri08] Bridson R.: Fluid Simulation for Comptuer Graphics. AK Peters (2008).

[MST10] McAdams A., Sifakis E., Teran J.: A parallel multigrid Poisson solver for simulation on large grids. In Proceedings of Eurographics/ ACM SIGGRAPH Symposium on Computer Animation (2010), M. Otaduy and Z. Popovic (Eds.).