3D Fluid Simulation of Rayleigh-Taylor Instability on GPU

Xiaoyi Xie[1],* Yu Guo[2], and Xinwei Li[3]
[1]*Department of Physics, New York University*
[2]*Courant Institute of Mathematical Sciences, New York University and*
[3]*Courant Institute of Mathematical Sciences, New York University*

This is the final project paper for HPC Fall 2012 course in New York University. In this paper, we describe an implemention of compressible inviscid fluid solvers on Graphics Processing Units using NVIDIA's CUDA. Using the method of lines approach with the third order Runge-Kutta time integration scheme, piecewise linear reconstruction, and a Harten-Lax-van Leer Riemann solver, we achieve an overall speedup of aproximately 10 times faster execution on a laptop graphics card as compared to a single core on the host computer.

Keywords: Hydrodynamics, HLL Rieman Solver, Euler Equation, Rayleigh-Taylor Instability

I. INTRODUCTION

In natural world, there are lots of phenomon could be described using physical system model. The aim of system model is to obtain in mathematical form a description of the dynamical behavior of a system in terms of some physically significant variables. As the nature of the system changes, the system variables change. Conservation of physical quantities is a fundamental physical principle that is often used to derive models in the natural sciences. In this paper, we put our focus on an interesting hydrodynamics phenomon–Rayleigh Taylor Instability. It's what happens when you put a dense liquid on top of a less-dense liquid. Plumes of liquid erupt and curl, making beautiful, ever-shifting patterns. The exact patterns depend on the relative density of both liquids, the purtabation given on the interface, and the time they're given to mix together. The dynamics of this process could be described by Euler equations(assuming both fluid are inviscid). The equations represent conservation of mass(continuity), momemtum, and energy. In differential form, Euler equations could be expressed as:

$$\frac{\partial \rho}{\partial t} + \nabla \cdot \rho \vec{u} = 0 \tag{1}$$

$$\frac{\partial \rho \vec{u}}{\partial t} + \nabla \cdot (\vec{u} \bigoplus (\rho \vec{u})) + \nabla (p + \Phi) = 0 \tag{2}$$

$$\frac{\partial E}{\partial t} + \nabla \cdot (\vec{u}(E + p)) = 0 \tag{3}$$

$$\tag{4}$$

Here, ρ denotes the density, $\vec{u} = (u, v, w)$ the velocity vector, p the pressure, Φ the gravitational potential, and E the total energy(kinetic plus internal energy) given by $E = \rho(u^2 + v^2 + w^2)/2 + p/(\gamma - 1)$. In all computations we use $\gamma = 1.4$(which means both fluid are comparable to ideal gas). Euler equations can also be expressed in vector and conservation form like follows.

$$\begin{bmatrix} \rho \\ \rho u \\ \rho v \\ \rho w \\ E \end{bmatrix}_t + \begin{bmatrix} \rho u \\ \rho u^2 + p \\ \rho uv \\ \rho uw \\ u(E+P) \end{bmatrix}_x + \begin{bmatrix} \rho v \\ \rho uv \\ \rho v^2 + p \\ \rho vw \\ v(E+P) \end{bmatrix}_y + \begin{bmatrix} \rho w \\ \rho uw \\ \rho vw \\ \rho w^2 + p \\ w(E+p) \end{bmatrix}_z = \begin{bmatrix} 0 \\ 0 \\ 0 \\ g\rho \\ g\rho w \end{bmatrix} \tag{5}$$

The Euler equations are one particular example of a large class of equations called *hyperbolic systems of conservative laws*, which can be written on the form.

$$U_t + F(U)_x + G(U)_y + H(U)_z = S(U) \tag{6}$$

This class of PDEs exhibits very singular behaviour and admits various kinds of discontinuous and nolinear waves, such as shocks, rarefactions, phase boundaries, fluid and material interfaces, etc.

II. NUMERICAL METHODS

To solve the Euler equations in two and three dimensions, we will make use of an approximate riemann solver, explicitly HLL (Harten - Lax Van Leer) riemann solver. First, to numerically solve the Equ(6), the time dependent evolution can be expressed in the semi-discrete form

$$\frac{dU_{ijk}}{dt} = L(U) \tag{7}$$

$$L(U) = -\frac{F_{i+1/2,jk} - F_{i-1/2,jk}}{\Delta x} - \frac{G_{i,j+1/2,k} - G_{i,j-1/2,k}}{\Delta y} \tag{8}$$

$$-\frac{H_{ij,k+1/2} - H_{ij,k-1/2}}{\Delta z} + S_{ijk} \tag{9}$$

where $F_{i\pm1/2,jk}$,$G_{i,j\pm1/2,k}$,and $H_{i,j,k\pm1/2}$ are the fluxes at the cell interface in x, y and z direction respectively. The time integration can be done using the first-order forward Euler method,

$$U^{n+1} = U^n + \Delta t L(U^n) \tag{10}$$

where U^n is the conserved variable at $t = t^n$ and U^{n+1} is the conserved variable after advancing one time step.

To obtain the fluxes at the cell interface, we can solve the so-called Riemann problem. Given the variables at cell i, j, k and $i+1, j, k$, we can calculate the x direction flux at the interface, $x = x_{i+1/2,j,k}$. The exact solution of this problem can be solved numerically. But it is very expensive because iterations are involved. Fortunately, we can use an approximate Riemann solver. There are different approximate Riemann solvers. Among them, HLL Riemann solver is an example of efficient approximate Riemann solvers. Given the left state

* xx315@nyu.edu

U^L and right state U^R, the HLL flux can be written as,

$$F^{HLL} = \frac{\alpha^+ F^L + \alpha^- F^R - \alpha^+ \alpha^- (U^R - U^L)}{\alpha^+ + \alpha^-} \tag{11}$$

where α^+ and α^- are related to the minimal and maximum eigenvalues of the Jacobians of the left and right states in the form.

$$\alpha^{\pm} = MAX\{0, \pm\lambda^{\pm}(U^L), \pm\lambda^{\pm}(U^R)\}. \tag{12}$$

Here, the minimal and maximum eigenvalues $\lambda^{\pm}$ are given by,

$$\lambda^{\pm} = v \pm c_s \tag{13}$$

where $c_s = \sqrt{\gamma P/\rho}$ is the sound speed. The HLL flux formula can be used to calculate the flux at the cell interface. For example, to obtain $F_{i+1/2,jk}$, substitute "L" and "R" by i and $i+1$ in Eq(11).
The time step Δt needs to satisfy the Courant-Fridrich-Lewy condition. Thus the following condition must be satisfied,

$$\Delta t < \Delta x / MAX(\alpha^{\pm}) \tag{14}$$

III. HIGH ORDER SCHEME

It is very straightforward to extend the first-order method we discussed in the above section to high order.
The time integration can now be done with a high-order Runge-Kutta method. A very popular third-order Runge-Kutta scheme in computational gas dynamics was designed by Shu&Osher. The method combines the first-order forward Euler steps with prediction& correction. The method is as following

$$U^{(1)} = U^n + \Delta t L(U^n) \tag{15}$$

$$U^{(2)} = \frac{3}{4}U^n + \frac{1}{4}U^{(1)} + \frac{1}{4}\Delta t L(U^{(1)}) \tag{16}$$

$$U^{n+1} = \frac{1}{3}U^n + \frac{2}{3}U^{(2)} + \frac{2}{3}\Delta t L(U^{(2)}) \tag{17}$$

$$\tag{18}$$

where $L(U)$ is the right hand side of Eq(7). U^{n+1} is the final value after advancing one time step from U^n.

To achieve high order in space, the left and right states at the cell interface must be reconstructed using high-order interpolation methods instead of using the values at the cell center. A simple high-order reconstruction method is the piecewise linear method(PLM) with a generalized minmod slope limiter. To obtain pressure, density and velocity of the left and right states at the cell interface $i+1/2$, we need the states at $i-1, i, i+1$, and $i+2$. Note that two ghost cells are needed at the boundaries. Given c_{i-1}, c_i, c_{i+1} the left-biased interface value of the left state reads,

$$c^L_{i+1/2} = c_i + 0.5 minmod(\theta(c_i - c_{i-1}), 0.5(c_{i+1} - c_{i-1}), \theta(c_{i+1} - c_i))$$

where c denotes pressure, density or velocity, $1 \le \theta \le 2$, and the minmod function reads,

$$minmod(x, y, z) = \frac{1}{4}|sgn(x)+sgn(y)|(sgn(x)+sgn(z))min(|x|, |y|, |z|) \tag{19}$$

here the sgn function returns the sign of the number. This becomes the more diffusive normal minmod limiter when $\theta = 1$, and becomes the monotonized central-difference limiter when $\theta = 2$. We usually use $\theta = 1.5$. To obtain the right state at $i+1/2$, variable c_i, c_{i+1}, c_{i+2} are used in the right-biased reconstruction with an expression similar to Eq(19). More, specifically, the right state at interface $i+1/2$ reads,

$$c^R_{i+1/2} = c_{i+1} - 0.5 minmod(\theta(c_{i+1} - c_i), 0.5(c_{i+2} - c_i), \theta(c_{i+2} - c_{i+1})) \tag{20}$$

IV. GPU IMPLEMENTATION

Graphics Processing Units (GPUs) are specialized for math intensive highly parallel computation. Studies involving fluid dynamics benefits a lot by using spatial and temporal adaptive mesh refinement. Our results show that computational fluid algorithm is easy to achieve parallel computing. Our computation is based on Geforce GT650M graphic card which could be found installed on many current laptop computers. NVIDIA CUDA allow programmers to define kernels which can be executed in parallel by many threads on GPU. Threads are organized into 1D,2D or 3D thread blocks while blocks are organized into 1D or 2D grids. Each thread can access its thread and block indices by two built-in variables threadIdx and blockIdx. As discussed above, a GPU has an array of SMs. Geforece GT650 has two SM. Each SM is composed of 192 SP. In CUDA, each thread is executed on a single SP while a block is decomposed into several wraps which are executed on single SM. CUDA exposes the hardware memory hierarchy by allowing threads to access data from multiple memory spaces. All threads have access to the same global memory. Each thread block has a shared memory visible to all threads of the block and with the same set of registers. There are also two additional read-only memories accessible by all threads: the constant and texture memory.

The shared memory is much smaller than global memory, typically 32 KB, but it is on-chip so it has very high register-level bandwidth. A typical programming pattern utilizing this fact is to stage data from global memory into shared memory, process the data there and then write the results back to global memory.

V. MAPPING FLUID SOLVER TO GPU

Usually fluid solver will use spatial grids to simulate actual spatial physical domain. In computation, that means to malloc a memory block to store physical elements data in grid form. In order to easily tell different grid element, we will store data sequentially according to its space grid index number(i, j, k). In our case, each element in space have five physical variable ρ, v_x, v_y, v_z, p. Its value are stored in memory identified by number $i*s_x+j*s_y+k*s_z+l, s_x = 5*Y*Z, s_y = 5*Z, s_z = 5$. i, j, k correspond to element's grid index. Mapping fluid solver to GPU, is building linking between the thread's ID and element's grid index, making each launched thread handle the computation of one identical grid element. We build the connection between thread ID and grid index number easily by the following method. If we need to handle X*Y*Z elements. We could launch X*Y*Z threads correspondingly.

```
threadID = blockDim.x * blockIdx.x + threadIdx.x;
i = threadID/(Y*Z);
j = threadID/Z - i*Y;
k = threadID - (threadID/Z)*Z;
```

Nowadays, the global memory of GPU is typically larger 512MB or 1GB. The size of global memory will constrain the memory size a program could malloc on device. In order to achieve maximum speed and efficiency. We malloc and free every memory block on device. This method is fast, but could only handle medium grid size (maximum 96x96x288 with 1GB global memory) In order to compute larger grid size, one way is to map host pageable memory to device. Since a grid size of 96x96x288 is sufficient for our simulation. We'll focus on the speedup aspect of our simulation. We launch several kernels to handle ourschemes step by step. In the following, we list our pseudo-code:

allocate memory for conserved variables, physical variables on GPU
call kernels for flux computation,
call kernels for state eigen value computation, find maximum sound speed on CPU, call kernels for L(U) computation,
call kernels for time update $U_a dv = U + \Delta t L(U)$,
copy $U_a dv$ to U
update dt info on CPU
if (t ¡ tmax) repeat above steps

The kernel code for the flux computation is listed below. Note that we make use of parrallel reduction to get maximum sound speed in our code.

```
  #define BLOCK_SIZE 256

  ...
  int i, j, k, l,N;
  float AlphaPlus,AlphaMinus,max;
  float SoundSpeedL, SoundSpeedR;
  int Sx = 5*(Y)*(Z), Sy = 5*(Z), Sz = 5;
  int threadID = blockDim.x*blockIdx.x + threadIdx.x;
  if(threadID < NThreads){
    i = threadID/(Y*Z);
    j = threadID/(Z) - i*(Y);
    k = threadID - (threadID/(Z))*(Z);
    N = Sx*i + Sy*j + Sz*k;
    __shared__ float local_max[BLOCK_SIZE];
    /***** x from 0 to X-1 **********/
    AlphaPlus = 0.;
    AlphaMinus = 0.;
    max = 0.;
    SoundSpeedL = sqrtf (GAMMA * physL[N+4] / physL[N+0]);
    SoundSpeedR = sqrtf (GAMMA * physR[N+4] / physR[N+0]);
    if (AlphaPlus  <  physL[N+1] + SoundSpeedL )
        AlphaPlus  =  physL[N+1] + SoundSpeedL;
    if (AlphaMinus < -physL[N+1] + SoundSpeedL )
        AlphaMinus = -physL[N+1] + SoundSpeedL;
    if (AlphaPlus  <  physR[N+1] + SoundSpeedR )
        AlphaPlus  =  physR[N+1] + SoundSpeedR;
    if (AlphaMinus < -physR[N+1] + SoundSpeedR )
        AlphaMinus = -physR[N+1] + SoundSpeedR;
    for (l=0; l<5; l++) {
    N = Sx*i+Sy*j+Sz*k+l;
    F_mid[N] = (AlphaPlus*FL[N]+AlphaMinus*FR[N] \
     -AlphaMinus*AlphaPlus*(UR[N]-UL[N]))/(AlphaPlus+AlphaMinus);
    }

    if (max < AlphaPlus) max = AlphaPlus;
    if (max < AlphaMinus) max = AlphaMinus;
    local_max[threadIdx.x] = max;
    __syncthreads();
    volatile float *c = local_max;
    float temp;
    if(blockIdx.x < gridDim.x - 1){
      for(unsigned int s=blockDim.x/2; s>32; s>>=1){
if(threadIdx.x < s){
  temp = c[threadIdx.x];
  c[threadIdx.x] = temp = (temp < c[threadIdx.x + s])? \
  c[threadIdx.x + s] : temp;
}
__syncthreads();
      }
      if(threadIdx.x < 32)
{
  temp = c[threadIdx.x];
  c[threadIdx.x] = temp = (temp < c[threadIdx.x + 32])? \
                          c[threadIdx.x + 32] : temp;
  temp = c[threadIdx.x];
  c[threadIdx.x] = temp = (temp < c[threadIdx.x + 16])? \
                          c[threadIdx.x + 16] : temp;
  temp = c[threadIdx.x];
  c[threadIdx.x] = temp = (temp < c[threadIdx.x + 8])? \
                          c[threadIdx.x + 8] : temp;
  temp = c[threadIdx.x];
  c[threadIdx.x] = temp = (temp < c[threadIdx.x + 4])? \
                          c[threadIdx.x + 4] : temp;
  temp = c[threadIdx.x];
  c[threadIdx.x] = temp = (temp < c[threadIdx.x + 2])? \
                          c[threadIdx.x + 2] : temp;
  temp = c[threadIdx.x];
  c[threadIdx.x] = temp = (temp < c[threadIdx.x + 1])? \
                          c[threadIdx.x + 1] : temp;
}
      __syncthreads();
      if(threadIdx.x == 0 ){
global_max[blockIdx.x] = local_max[threadIdx.x];
      }
    } else {
int s = 0;
for(s=(NThreads - (gridDim.x-1)*blockDim.x)/2;s>0; s>>=1){
  if(threadIdx.x < s) {
    temp = c[threadIdx.x];
    c[threadIdx.x] = temp = (temp < c[threadIdx.x + s])? \
      c[threadIdx.x + s] : temp;
  }
  __syncthreads();
}
if(threadIdx.x == 0){
  temp = c[0];
  c[0] = temp = (temp < c[NThreads - (gridDim.x - 1)*blockDim.x - 1]) 1
    c[NThreads - (gridDim.x - 1)*blockDim.x -1]: temp;
  global_max[blockIdx.x] = local_max[0];
}
    }
  }
}
```

VI. EMPIRICAL RESULTS

All the problems we try to do will be run on a Geforce GT650M card. we show the technical specifications of Geforce GT650M in Table I. The corresponding CPU comparison cases are run on an quad-core i7(typically one core is responble for the calculation of sequential code).

A. 2D Rayleigh-Taylor Instability

We make use of the follow set up to simulate 2D Rayleigh-Taylor.

```
Domain
   2D: -0.25  x  0.25, -0.75  z  0.75
   3D: Same as in 2D, but with the y domain included:
   -0.25  y  0.25 (the z direction is still the
   direction of gravity.)
Boundary conditions
   2D: Periodic in x, reflecting in z
   3D: Same as in 2D, but with periodic y boundaries
Equation of state
```

TABLE I.
Technical specifications of NVIDIA's Geforce GT650 graphics card.

Compute Capability:	3.0
Total amount of global memory:	1 GBytes
2Multiprocessors x 192CUDA Cores/MP:	384 CUDA Cores
GPU Clock rate:	0.90 GHz
Memory Clock rate:	2508.00 Mhz
Memory Bus width:	128-bit
Total amount of constant memory:	65536 bytes
Total amount of shared memory per block:	49152 bytes
Total number of registers available per block:	65536
Warp size:	32
Maximum number of threads per multiprocessor	2048
Maximum number of threads per block:	1024
Maximum sizes of each dimension of a block:	1024 x 1024 x 64
Maximum sizes of each dimension of a grid:	2147483647 x 65535 x 65535
Maximum memory pitch:	2147483647 bytes
Texture alignment:	512 bytes

```
   Adiabatic with  = 1.4
   Initial density
    = 1 ifor y  0.0 and  = 2 for y > 0.0
Initial pressure
   The pressure is initialized to give hydrostatic
   equilibrium: P = 2.5 - gy.

Initial velocity
   2D: For the single mode perturbation, we perturb
   the y velocity with A [1+cos(2x/Lx)] [1+cos(2y/Ly)]/4
   with Lx and Ly being the size of the x and y domains
   respectively, and A = 0.01. For the random perturbation,
   the y velocity is given a random value between -A/2 and
   3D: Same as in 2D, but for the single mode perturbation,
   vy is A [1+cos(2x/Lx)] [1+cos(2y/Ly)] [1+cos(2z/Lz)]/8
   with Lz being the size of the z domain.
```

By setting our grid dimemsion to 32x2x96, 64x2x192, 128x2x384, we get the following density contour of 2D Rayleigh-Taylor Instability result using our sequential code.

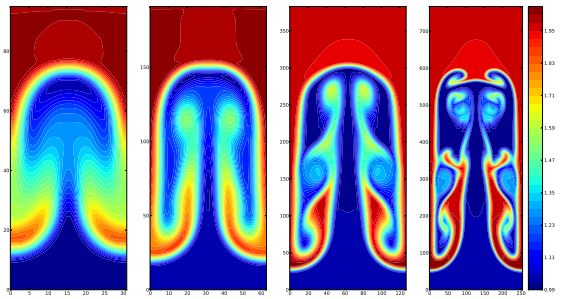

FIG. 1. (Color online) The density contour of 2D Rayleigh-Taylor Instability for single mode perturbation. The up fluid density is 2, the down fluid density is 1. The fluid elememt is subject to gravitational force in vertical direction.

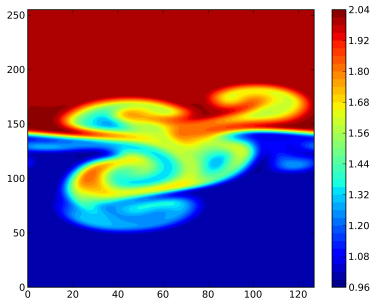

FIG. 2. (Color online) The density contour of 2D Rayleigh-Taylor Instability for multimode perturbation.

B. 3D Rayleigh-Taylor Instability

By setting our grid dimemsion to 32x32x96, 64x64x192, 128x128x384, we could get more accurate density contour of Rayleigh-Taylor Instability in 3 Dimension. We also make use of visit to visualize our 3d result.

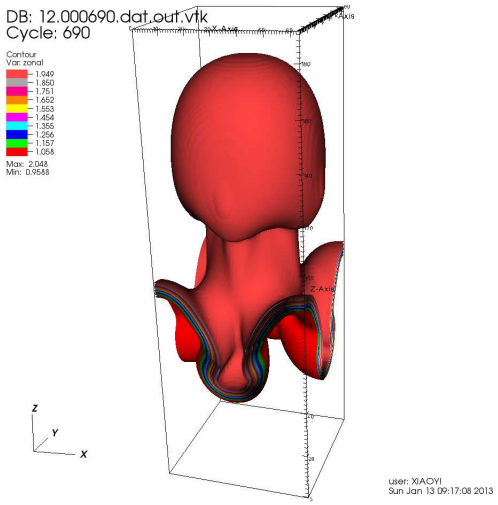

FIG. 3. (Color online) The density constour of 3D Rayleigh-Taylor Instability. The cell dimemsion is 64x64x192. Simulation time is 12.000690.

C. GPU parallel code test

In the following, we present several numerical tests to access the computational efficiency of our GPU implemention. To this end, we compare runtimes on NVIDIA GeForce GT650M with runtimes on 2.3 GHz Intel Core i7. To ensure a fair comparison, we have used the same design choices for the GPU implementations, trying to retain a one-to-one correspondence of statements in the CPU and GPU computational kernels.

Another important question is accuracy. The numerical method considered in the paper is stable and the accuracy.

Our tests indicate that the difference between single precision(GPU) and double precision(CPU) results are of order 10^{-6}.

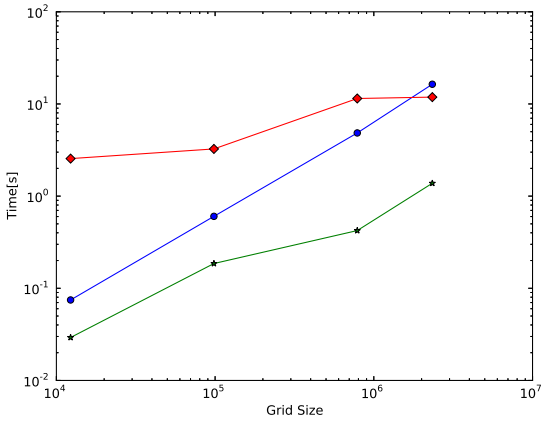

FIG. 4. Running time of a single timestep for CPU code (sphere) and GPU code (asterisk) for 3D Rayleigh-Taylor instability with grid sizes N = 16x16x48, 32x32x96, 64x64x192, 96x96x288. The diamonds show the ratio of the CPU and GPU running time.

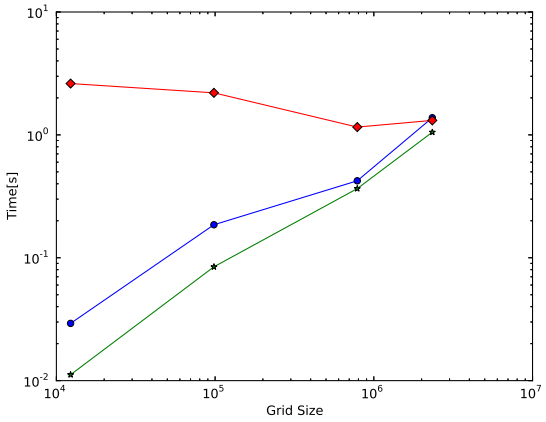

FIG. 5. Running time of a single timestep for direct mapping gpu code(sphere) and GPU code making use of shared memory (asterisk) for 3D Rayleigh-Taylor instability with grid sizes N = 16x16x48, 32x32x96, 64x64x192, 96x96x288. The diamonds show the ratio of the running time.

First, we directly map our sequential scheme to GPU. Malloc global memory on device for each calculation term (conserved variable, flux, physical variable). Then, we make use of shared memory on device, instead of getting elements from global memory, do the calculation, and write back to global memory. We first load data from global memory to shared memory, do the calculation with data elements in the shared memory then write back to global memory. Since shared memory has less access time. The runtime will get further speedup. Fig(4) compares the runtime between sequential code and direct mapping GPU code. Fig(5 compares the runtime between direct mapping GPU code and GPU code making use of shared memory.

Since there are several kernels in the parrallel code, it would be better to have a clear idea about the runtime and efficiency of each kernel. Profiling information will aid our program optimization. In the following, we list the profiling information of kernels generated by NVIDIA command line profiler (nvprof). by comparing the gpu profile result, we know that making use of shared memory in our scheme will get two to three times speed up.

TABLE II. profiler for kernels using global memory.

Time(%)	Time	Calls	Avg	Name
14.41	27.26ms	19	1.43ms	h_Ucalc
10.68	20.20ms	3	6.73ms	h_StateX
8.86	16.75ms	12	1.40ms	h_Ucalcinv
6.26	11.83ms	3	3.94ms	h_FluxMidX
6.17	11.66ms	3	3.89ms	h_FluxMidZ
6.13	11.60ms	3	3.87ms	h_FluxMidY
5.99	11.34ms	3	3.78ms	h_FluxCalcPX
5.99	11.33ms	3	3.78ms	h_FluxCalcPY
5.86	11.09ms	3	3.70ms	h_FluxCalcPZ
5.68	10.75ms	3	3.58ms	h_StateZ
5.67	10.72ms	3	3.57ms	h_StateY
4.43	8.38ms	1	8.38ms	h_U_update3
4.19	7.92ms	1	7.92ms	h_U_update2
3.57	6.75ms	1	6.75ms	h_U_update1
1.62	3.07ms	3	1.02ms	h_BoundaryZ
1.32	2.49ms	3	829.21us	h_Cal_Source
1.26	2.38ms	3	792.90us	h_BoundaryY
1.23	2.33ms	3	777.89us	h_BoundaryX
0.64	1.22ms	1	1.22ms	init_water_oil_3d
0.05	86.91us	9	9.66us	[CUDA memcpy DtoH]

[1] Peng Wang, Tom Abel, Ralf Kaehler, *Adaptive mesh fluid simulations on GPU*.

[2] Trond Runar Hagen..etc, *Solving the Euler Equations on Graphics Processing Units*.

TABLE III. kernel profiling result for parrallel code using shared memory.

Time(%)	Time	Calls	Avg	Name
11.57	8.00ms	3	2.67ms	h_StateX
10.49	7.26ms	3	2.42ms	h_FluxMidY
9.62	6.66ms	19	350.32us	h_Ucalc
9.57	6.62ms	3	2.21ms	h_FluxMidX
9.04	6.26ms	3	2.09ms	h_FluxMidZ
6.83	4.72ms	12	393.62us	h_Ucalcinv
6.29	4.35ms	3	1.45ms	h_StateY
6.20	4.29ms	3	1.43ms	h_StateZ
5.78	4.00ms	1	4.00ms	h_U_update3
4.90	3.39ms	1	3.39ms	h_U_update2
3.88	2.68ms	1	2.68ms	h_U_update1
3.04	2.10ms	6	350.52us	h_FluxCalcPY_N
3.00	2.08ms	6	346.41us	h_FluxCalcPX_N
2.93	2.02ms	6	337.31us	h_FluxCalcPZ_N
1.70	1.17ms	3	390.93us	h_BoundaryZ
1.51	1.04ms	3	347.22us	h_Cal_Source
1.47	1.02ms	3	338.77us	h_BoundaryY
1.36	941.61us	3	313.87us	h_BoundaryX
0.73	506.34us	1	506.34us	init_water_oil_3d
0.08	57.54us	9	6.39us	[CUDA memcpy DtoH]