

BAB V

KESIMPULAN DAN SARAN

Berdasarkan visualisasi gelombang tsunami yang telah dilakukan menggunakan CPU dan GPU NVdia CUDA dapat ditarik beberapa kesimpulan dan saran sebagai berikut:

5.1. Kesimpulan

1. Kode program dari Dr. Graham Pullan yang memvisualisasi metode lattice Boltzmann dapat digunakan untuk melakukan visualisasi tsunami dan dapat berjalan dengan baik pada CPU dan GPU.
2. Dengan pemrograman paralel menggunakan GPU NVdia Cuda dapat mempercepat proses komputasi pada visualisasi tsunami, dan variasi ukuran citra yang digunakan sebagai peta dalam visualisasi tsunami ini sangat berpengaruh terhadap kecepatan proses visualisasi tsunami, semakin besar ukuran citra yang digunakan semakin berkurang kecepatan pemrosesannya.
3. Berdasarkan hasil perhitungan kecepatan proses visualisasi menggunakan NVdia Cuda pada laptop lebih cepat rata-rata 5.31 dari pada menggunakan CPU, dengan menggunakan desktop. Lebih cepat

rata-rata 5.016 dari CPU. Sedangkan desktop lebih cepat 1.822 kali dari laptop pada simulasi CUDA GPU.

5.2. Saran

1. Penelitian ini dapat dikembangkan dengan mengunakan data asli dari kejadian tsunami.
2. Visualisi tsunami dapat dikembangkan ke 3D.
3. Penelitian selanjutnya dapat dikembangkan dengan menambah metode lain, dan membandingkan dengan metode lattice Boltzmann.

DAFTAR PUSTAKA

- Arifin, S. (2005). Strategi untuk mengurangi kerusakan lingkungan yang diakibatkan oleh gempa dan gelombang tsunami. *Jurnal Arsitektur "Atrium"*, 28-33.
- Balevic, A. (2009). Parallel Variable-Length Encoding on GPGPUs. *HPPC 2009 – the 3rd Workshop on Highly Parallel Processing on a Chip*, 19.
- Bernaschi, M., & dkk. (2009). *A flexible high-performance Lattice Boltzmann GPU code for the simulations of fluid flows in complex geometries, CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE*,. Retrieved from Wiley InterScience: www.interscience.wiley.com
- Gohari, S. M., & Ghadyani, M. (2012). Effects of GPU Structuring on Accelerated Schemes of Lattice Boltzmann and Classical CFD for Flow over a Flat Plate, . *Journal of Mathematics and System Science* 2.
- Huda, I., & dkk. (2009). kondisi vegetasi dan kerang geloina pasca tsunami dalam kawasan ekosistem mangrove pesisir barat kabupaten aceh besar,. *Torani (Jurnal Ilmu Kelautan dan Perikanan) Volume 19*, 82 – 89.
- Januszewski, M. (2012). Sailfish: Lattice Boltzmann Fluid Simulations with GPUs and Python, Institute of Physics University of Silesia in Katowice, Poland, Google. *GTC 2012*.

- Kirk D. & Hwu, W. (2010). *Programing Massively Parallel Processors*. Morgan Kaufmann Publishers.
- Nazaruddin, A., & Pranowo. (2013). Model 2D visualisasi tsunami aceh dengan metode Lattice Boltzmann. *Proceeding Sentika* (p. 240). Yogyakarta: Universitas Atma Jaya .
- Nur, A. M. (2010). gempa bumi, tsunami dan mitigasinya. *jurnal geografi volume 7 no. 1*.
- Ramya, V., & Palaniappan, B. (2011). An automated tsunami alert system,. *international journal of embedded system application (IJESA)*, Volume 1 no. 2.
- Revell, A. (2013). GPU Implementation of Lattice Boltzmann Method with Immersed Boundary: observations and results,. *The Oxford e-Research Centre Many-Core Seminar Series*.
- Thurey, N. (2003). A single-phase free-surface Lattice-Boltzmann Method. M.Phil. . *thesis, FRIEDRICH-ALEXANDER-UNIVERSITAT ERLANGEN-NURNBERG*.
- Thurey, N., & dkk. (2006). Animation of open water phenomena with couple shallow water and free surface simulations. *Eurographics/ACM SIGGRAPH symposium on computer animation*.
- Ward, S. N. (2000). Landslide Tsunami. *Journal of Geophysical Research*, Volume 1 no 8.

Webb, C. J. (2010). *computing 3d finite difference scheme for acoustics*. University of Edinburgh.

Zakia, Z. (2004). *Gempa dan Tsunami Getarkan Aceh*. Retrieved Juli 19, 2013, from <http://nationalgeographic.co.id/berita/2012/12/26-desember-2004-gempa-dan-tsunami-getarkan>



LAMPIRAN

1. Hasil Perhitungan Waktu Visualisasi

1. 1. Visualisasi menggunakan CPU

Jenis peta	Iterasi		
	500	1000	2000
	Waktu / m	Waktu / m	Waktu / m
Aceh	0.638	1.256	2.489
Jawa	0.632	1.272	2.562
Jepang	0.638	1.231	2.459

Jenis peta	Iterasi		
	500	1000	2000
	Waktu / m	Waktu / m	Waktu / m
Aceh	3.225	6.589	13.329
Jawa	3.264	6.567	13.301
Jepang	3.211	6.543	13.190

Jenis peta	Iterasi		
	500	1000	2000
	Waktu / m	Waktu / m	Waktu / m
Aceh	7.164	14.645	29.654
Jawa	7.015	14.431	29.218
Jepang	6.816	13.900	28.220

1. 2. Visualisasi menggunakan GPU

Jenis peta	Iterasi		
	500	1000	2000
	Waktu / m	Waktu / m	Waktu / m
Aceh	0.178	0.340	0.657
Jawa	0.177	0.335	0.652
Jepang	0.179	0.335	0.651

Jenis peta	Iterasi		
	500	1000	2000
	Waktu / m	Waktu / m	Waktu / m
Aceh	0.620	1.141	2.181
Jawa	0.552	1.072	2.111
Jepang	0.544	1.061	2.107

Jenis peta	Iterasi		
	500	1000	2000
	Waktu / m	Waktu / m	Waktu / m
Aceh	1.253	2.355	4.550
Jawa	1.149	2.241	4.428
Jepang	1.146	2.236	4.421

2. Source Code Matlab

Source code konversi citra warna ke citra biner.(matlab)

```
clear all;
I=imread('a.png');
Ir=double(I(:,:,1));
[imax,jmax]=size(Ir);
A=double(Ir);
B=double(Ir);
for i=1:imax
for j=1:jmax
if Ir(i,j)>=255
A(i,j)=0;
else
A(i,j)=255;
end end end
Ir=flipdim(Ir,1);
surf(Ir);shading interp;
imshow (A);
Source code konversi ke bentuk .dat
```

```
clc;
clear all;
```

```

close all;
I=imread('a.jpg');
figure,imshow(I);
BW=double(I);
save('map400.dat','BW','-ASCII');

```

3. Source Code GPU Tsunami

```

// Crude 2D Lattice Boltzmann Demo program
// CUDA version
// Graham Pullan - Oct 2008
//
// This is a 9 velocity set method:
// Distribution functions are stored as "f" arrays
// Think of these as the number of particles moving in these directions:
//
//      f6  f2  f5
//      \  |  /
//       \ | /
//      f3---f1
//       / | \
//      /  |  \
//     f7  f4  f8
//
// and f0 for the rest (zero) velocity
//
// //////////////////////////////////////
//
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <conio.h>
#include <time.h>
#include <sys/stat.h>
#include <string.h>
#include "GL/glew.h"
#include "GL/glut.h"
#include "GL/glu.h"
#include "GL/gl.h"
#include <cutil.h>
#include <cuda_runtime_api.h>
#include <cuda_gl_interop.h>

#define TILE_I 16
#define TILE_J 8
#define I2D(ni,i,j) (((ni)*(j)) + i)

// OpenGL pixel buffer object and texture //

```



```

                                float *f7_data, float *f8_data, float
*plot_data, float *h_surf_data,int ni, int nj);

__global__ void apply_Periodic_BC_kernel (int ni, int nj, int pitch,
                                float *f2_data, float *f4_data, float
*f5_data,
                                float *f6_data, float *f7_data, float
*f8_data);

__global__ void apply_BCs_kernel (int ni, int nj, int pitch, float vxin,
float hout,
                                float faceq2, float faceq3,
                                float *f0_data, float *f1_data, float
*f2_data,
                                float *f3_data, float *f4_data, float
*f5_data,
                                float *f6_data, float *f7_data, float
*f8_data, int* solid_data);

__global__ void get_rgba_kernel (int pitch, int ncol, float minvar, float
maxvar,
                                float *plot_data,
                                unsigned int *plot_rgba_data,
                                unsigned int *cmap_rgba_data,
                                int *solid_data);

//
// CUDA kernel C wrappers
//
void stream(void);
void collide(void);
void apply_Periodic_BC(void);
void apply_BCs(void);
void get_rgba(void);
float h_max(float *h_surf_data,int pitch,int ni, int nj);
float h_min(float *h_surf_data,int pitch, int ni, int nj);

//float h_max(float *h_surf_data,int ni, int nj);
//float h_min(float *h_surf_data, int ni, int nj);

unsigned int get_col(float min, float max, float val);

////////////////////////////////////
////
////////////////////////////////////
////

int main(int argc, char **argv)
{
    int totpoints,i,j, A[400][400];
    float rcol,gcol,bcol;

```

```
FILE *fp_col;
cudaChannelFormatDesc desc;

// The following parameters are usually read from a file, but
// hard code them for the demo:
ni=512;
nj=400;
hout=1.0;
tau=0.51;
    gr=0.5;
    iter=0;
// End of parameter list

// Write parameters to screen
printf ("ni = %d\n", ni);
printf ("nj = %d\n", nj);
printf ("vxin = %f\n", vxin);
printf ("hout = %f\n", hout);
printf ("tau = %f\n", tau);
    printf ("gr = %f\n", gr);

totpoints=ni*nj;

// allocate memory on host

f0 = (float *)malloc(ni*nj*sizeof(float));
f1 = (float *)malloc(ni*nj*sizeof(float));
f2 = (float *)malloc(ni*nj*sizeof(float));
f3 = (float *)malloc(ni*nj*sizeof(float));
f4 = (float *)malloc(ni*nj*sizeof(float));
f5 = (float *)malloc(ni*nj*sizeof(float));
f6 = (float *)malloc(ni*nj*sizeof(float));
f7 = (float *)malloc(ni*nj*sizeof(float));
f8 = (float *)malloc(ni*nj*sizeof(float));
    h_surf = (float *)malloc(ni*nj*sizeof(float));

plot = (float *)malloc(ni*nj*sizeof(float));

solid = (int *)malloc(ni*nj*sizeof(int));

plot_rgba = (unsigned int *)malloc(ni*nj*sizeof(unsigned int));

//
// allocate memory on device
//
CUDA_SAFE_CALL(cudaMallocPitch((void **)&f0_data, &pitch,
    sizeof(float)*ni, nj));
CUDA_SAFE_CALL(cudaMallocPitch((void **)&f1_data, &pitch,
    sizeof(float)*ni, nj));
CUDA_SAFE_CALL(cudaMallocPitch((void **)&f2_data, &pitch,
    sizeof(float)*ni, nj));
```

```

CUDA_SAFE_CALL(cudaMallocPitch((void **)&f3_data, &pitch,
                               sizeof(float)*ni, nj));
CUDA_SAFE_CALL(cudaMallocPitch((void **)&f4_data, &pitch,
                               sizeof(float)*ni, nj));
CUDA_SAFE_CALL(cudaMallocPitch((void **)&f5_data, &pitch,
                               sizeof(float)*ni, nj));
CUDA_SAFE_CALL(cudaMallocPitch((void **)&f6_data, &pitch,
                               sizeof(float)*ni, nj));
CUDA_SAFE_CALL(cudaMallocPitch((void **)&f7_data, &pitch,
                               sizeof(float)*ni, nj));
CUDA_SAFE_CALL(cudaMallocPitch((void **)&f8_data, &pitch,
                               sizeof(float)*ni, nj));
CUDA_SAFE_CALL(cudaMallocPitch((void **)&plot_data, &pitch,
                               sizeof(float)*ni, nj));
    CUDA_SAFE_CALL(cudaMallocPitch((void **)&h_surf_data, &pitch,
                                   sizeof(float)*ni, nj));
    CUDA_SAFE_CALL(cudaMallocPitch((void **)&solid_data, &pitch,
                                   sizeof(int)*ni, nj));

desc = cudaCreateChannelDesc<float>();
CUDA_SAFE_CALL(cudaMallocArray(&f1_array, &desc, ni, nj));
CUDA_SAFE_CALL(cudaMallocArray(&f2_array, &desc, ni, nj));
CUDA_SAFE_CALL(cudaMallocArray(&f3_array, &desc, ni, nj));
CUDA_SAFE_CALL(cudaMallocArray(&f4_array, &desc, ni, nj));
CUDA_SAFE_CALL(cudaMallocArray(&f5_array, &desc, ni, nj));
CUDA_SAFE_CALL(cudaMallocArray(&f6_array, &desc, ni, nj));
CUDA_SAFE_CALL(cudaMallocArray(&f7_array, &desc, ni, nj));
CUDA_SAFE_CALL(cudaMallocArray(&f8_array, &desc, ni, nj));

//
// Some factors used in equilibrium f's
//
faceq1 = 4.f/9.f;
faceq2 = 1.f/9.f;
faceq3 = 1.f/36.f;

    for ( j=0; j<nj; j++) {
        for ( i=0; i<ni; i++) {
            int i0 = I2D(ni,i,j);
            h_surf[i0] =1.0+0.2*expf(-0.050*((1.0*(i+40.0)-
150.0)*(1.0*(i+40.0)-150.0)+(1.0*(j-120.0)-150.0)*(1.0*(j-120.0)-150.0)));

        }
    }
//
// Initialise f's
//
    for (i=0; i<totpoints; i++) {
        f0[i] = h_surf[i]*(1.0 - 5.0*gr*h_surf[i]/6-
2.0*vxin*vxin/3.0);
        f1[i] = h_surf[i]*(gr*h_surf[i]/6.0 + vxin/3.0 + vxin*vxin/2.0 -
vxin*vxin/6.0);

```

```

        f2[i] = h_surf[i]*(gr*h_surf[i]/6.0 + 0.0/3.0 + 0.0*0.0/2.0 -
vxin*vxin/6.0);
        f3[i] = h_surf[i]*(gr*h_surf[i]/6.0 - vxin/3.0 + vxin*vxin/2.0 -
vxin*vxin/6.0);
        f4[i] = h_surf[i]*(gr*h_surf[i]/6.0 - 0.0/3.0 + 0.0*0.0/2.0 -
vxin*vxin/6.0);
        f5[i] = h_surf[i]*(gr*h_surf[i]/6.0+( vxin + 0.0)/3.0 + ( vxin
+ 0.0)*( vxin + 0.0)/2.0 - vxin*vxin/6.0)/4.0;
        f6[i] = h_surf[i]*(gr*h_surf[i]/6.0+(-vxin + 0.0)/3.0 + (-vxin +
0.0)*(-vxin + 0.0)/2.0 - vxin*vxin/6.0)/4.0;
        f7[i] = h_surf[i]*(gr*h_surf[i]/6.0+(-vxin - 0.0)/3.0 + (-vxin -
0.0)*(-vxin - 0.0)/2.0 - vxin*vxin/6.0)/4.0;
        f8[i] = h_surf[i]*(gr*h_surf[i]/6.0+( vxin - 0.0)/3.0 + ( vxin -
0.0)*( vxin - 0.0)/2.0 - vxin*vxin/6.0)/4.0;
        plot[i] = h_surf[i];
        solid[i] = 1;
    }
    hmin=1.0;
    hmax=1.2;

    fp_col = fopen("aceh400.dat","r");
    if (fp_col==NULL) {
        printf("Error: can't open file peta \n");
        return 1;
    }
    // allocate memory for colourmap (stored as a linear array of int's)
    fscanf (fp_col, "%d", &ncol);
    fscanf (fp_col, "%d", &nrow);

    peta_rgba = (unsigned int *)malloc(ncol*sizeof(unsigned int));
    peta_rgba = (unsigned int *)malloc(nrow*sizeof(unsigned int));
    CUDA_SAFE_CALL(cudaMalloc((void **)&peta_rgba_data,
                                sizeof(unsigned int)*ncol));
    CUDA_SAFE_CALL(cudaMalloc((void **)&peta_rgba_data,
                                sizeof(unsigned int)*nrow));

    // read colourmap and store as int's
    for (i=0;i<ncol;i++){
        for (j=0;j<nrow;j++){
            fscanf(fp_col, "%f", &rcol);
            A[i][j]=rcol;
            if(rcol == 1){
                i0=I2D(ni,i,j);
                solid[i0]=1;
            }
            else{
                i0=I2D(ni,i,j);
                solid[i0]=0;
            }
        }
    }

```

```

}

fclose(fp_col);
printf(" Image [i][j] = [%d][%d] \n", i,j);
printf(" ncol = %d \n", ncol);
printf(" nrow = %d \n", nrow);
printf(" rcol = %2.f \n", rcol);
//
// Read in colourmap data for OpenGL display
//
fp_col = fopen("cmap.dat","r");
if (fp_col==NULL) {
    printf("Error: can't open cmap.dat \n");
    return 1;
}

fscanf (fp_col, "%d", &ncol);
cmap_rgba = (unsigned int *)malloc(ncol*sizeof(unsigned int));
CUDA_SAFE_CALL(cudaMalloc((void **)&cmap_rgba_data, sizeof(unsigned
int)*ncol));

for (i=0;i<ncol;i++){
    fscanf(fp_col, "%f%f%f", &rcol, &gcol, &bcol);
    cmap_rgba[i]=((int)(255.0f) << 24) | // convert colourmap to int
        ((int)(bcol * 255.0f) << 16) |
        ((int)(gcol * 255.0f) << 8) |
        ((int)(rcol * 255.0f) << 0);
}
fclose(fp_col);

printf("ncol = %d \n", ncol);
printf("rcol = %f%f%f \n", rcol, gcol, bcol);

//
// Transfer initial data to device
//
CUDA_SAFE_CALL(cudaMemcpy2D((void *)f0_data, pitch, (void *)f0,
    sizeof(float)*ni,sizeof(float)*ni, nj,
    cudaMemcpyHostToDevice));
CUDA_SAFE_CALL(cudaMemcpy2D((void *)f1_data, pitch, (void *)f1,
    sizeof(float)*ni,sizeof(float)*ni, nj,
    cudaMemcpyHostToDevice));
CUDA_SAFE_CALL(cudaMemcpy2D((void *)f2_data, pitch, (void *)f2,
    sizeof(float)*ni,sizeof(float)*ni, nj,
    cudaMemcpyHostToDevice));
CUDA_SAFE_CALL(cudaMemcpy2D((void *)f3_data, pitch, (void *)f3,
    sizeof(float)*ni,sizeof(float)*ni, nj,
    cudaMemcpyHostToDevice));
CUDA_SAFE_CALL(cudaMemcpy2D((void *)f4_data, pitch, (void *)f4,
    sizeof(float)*ni,sizeof(float)*ni, nj,
    cudaMemcpyHostToDevice));
CUDA_SAFE_CALL(cudaMemcpy2D((void *)f5_data, pitch, (void *)f5,

```

```

        sizeof(float)*ni, sizeof(float)*ni, nj,
        cudaMemcpyHostToDevice));
CUDA_SAFE_CALL(cudaMemcpy2D((void *)f6_data, pitch, (void *)f6,
        sizeof(float)*ni, sizeof(float)*ni, nj,
        cudaMemcpyHostToDevice));
CUDA_SAFE_CALL(cudaMemcpy2D((void *)f7_data, pitch, (void *)f7,
        sizeof(float)*ni, sizeof(float)*ni, nj,
        cudaMemcpyHostToDevice));
CUDA_SAFE_CALL(cudaMemcpy2D((void *)f8_data, pitch, (void *)f8,
        sizeof(float)*ni, sizeof(float)*ni, nj,
        cudaMemcpyHostToDevice));
CUDA_SAFE_CALL(cudaMemcpy2D((void *)plot_data, pitch, (void *)plot,
        sizeof(float)*ni, sizeof(float)*ni, nj,
        cudaMemcpyHostToDevice));
CUDA_SAFE_CALL(cudaMemcpy2D((void *)h_surf_data, pitch, (void
*)h_surf,
        sizeof(float)*ni, sizeof(float)*ni, nj,
        cudaMemcpyHostToDevice));
CUDA_SAFE_CALL(cudaMemcpy2D((void *)solid_data, pitch, (void *)solid,
        sizeof(int)*ni, sizeof(int)*ni, nj,
        cudaMemcpyHostToDevice));
CUDA_SAFE_CALL(cudaMemcpy((void *)cmap_rgba_data,
        (void *)cmap_rgba, sizeof(unsigned int)*ncol,
        cudaMemcpyHostToDevice));

//
// Initialise OpenGL display - use glut
//
glutInit(&argc, argv);
glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB);
glutInitWindowSize(ni, nj); // Window of ni x nj
pixels
glutInitWindowPosition(100, 100); // Window position
glutCreateWindow("CUDA 2D LB"); // Window title

printf("Loading extensions: %s\n", glewGetErrorString(glewInit()));
if(!glewIsSupported(
        "GL_VERSION_2_0 "
        "GL_ARB_pixel_buffer_object "
        "GL_EXT_framebuffer_object "
        ")){
    fprintf(stderr, "ERROR: Support for necessary OpenGL extensions
missing.");
    fflush(stderr);
    return 1;
}

// Set up view
glClearColor(0.0, 0.0, 0.0, 0.0);
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glOrtho(0, ni, 0, nj, -200.0, 200.0);

```

```

// Create texture and bind to gl_Tex
glEnable(GL_TEXTURE_2D);
glGenTextures(1, &gl_Tex);           // Generate 2D texture
glBindTexture(GL_TEXTURE_2D, gl_Tex); // bind to gl_Tex
// texture properties:
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA8, ni, nj, 0,
             GL_RGBA, GL_UNSIGNED_BYTE, NULL);

// Create pixel buffer object and bind to gl_PBO
glGenBuffers(1, &gl_PBO);
glBindBuffer(GL_PIXEL_UNPACK_BUFFER_ARB, gl_PBO);
glBufferData(GL_PIXEL_UNPACK_BUFFER_ARB, pitch*nj, NULL,
GL_STREAM_COPY);
CUDA_SAFE_CALL( cudaGLRegisterBufferObject(gl_PBO) );
printf("Buffer created.\n");

printf("Starting GLUT main loop...\n");
glutDisplayFunc(display);
glutReshapeFunc(resize);
glutIdleFunc(display);
//glutMouseFunc(mouse);
// glutMotionFunc(mouse_motion);

glutMainLoop();
finalise();
return 0;
}

void finalise()
{
    cudaFreeHost( f0 );
    cudaFreeHost( f1 );
    cudaFreeHost( f2 );
    cudaFreeHost( f3 );
    cudaFreeHost( f4 );
    cudaFreeHost( f5 );
    cudaFreeHost( f6 );
    cudaFreeHost( f7 );
    cudaFreeHost( f8 );
    cudaFreeHost( h_surf );
    cudaFreeHost( plot );
    cudaFreeHost( solid );
    cudaFreeHost( plot_rgba );

    cudaFree( f0_data );
    cudaFree( f1_data );
    cudaFree( f2_data );

```

```

cudaFree( f3_data );
cudaFree( f4_data );
cudaFree( f5_data );
cudaFree( f6_data );
cudaFree( f7_data );
cudaFree( f8_data );
cudaFree( plot_data );
cudaFree( h_surf_data );
cudaFree( solid_data );
cudaFree( peta_rgba_data );
}

__global__ void stream_kernel (int pitch, float *f1_data, float *f2_data,
                             float *f3_data, float *f4_data, float
*f5_data,
                             float *f6_data, float *f7_data, float *f8_data)
// CUDA kernel
{
    int i, j, i2d;

    i = blockIdx.x*TILE_I + threadIdx.x;
    j = blockIdx.y*TILE_J + threadIdx.y;

    i2d = i + j*pitch/sizeof(float);

    // look up the adjacent f's needed for streaming using textures
    // i.e. gather from textures, write to device memory: f1_data, etc
    f1_data[i2d] = tex2D(f1_tex, (float) (i-1) , (float) j);
    f2_data[i2d] = tex2D(f2_tex, (float) i , (float) (j-1));
    f3_data[i2d] = tex2D(f3_tex, (float) (i+1) , (float) j);
    f4_data[i2d] = tex2D(f4_tex, (float) i , (float) (j+1));
    f5_data[i2d] = tex2D(f5_tex, (float) (i-1) , (float) (j-1));
    f6_data[i2d] = tex2D(f6_tex, (float) (i+1) , (float) (j-1));
    f7_data[i2d] = tex2D(f7_tex, (float) (i+1) , (float) (j+1));
    f8_data[i2d] = tex2D(f8_tex, (float) (i-1) , (float) (j+1));
}

void stream(void)
// C wrapper
{
    // Device-to-device mem-copies to transfer data from linear memory
    (f1_data)
    // to CUDA format memory (f1_array) so we can use these in textures
    CUDA_SAFE_CALL(cudaMemcpy2DToArray(f1_array, 0, 0, (void *)f1_data,
pitch,
                                sizeof(float)*ni, nj,
                                cudaMemcpyDeviceToDevice));
    CUDA_SAFE_CALL(cudaMemcpy2DToArray(f2_array, 0, 0, (void *)f2_data,
pitch,

```

```

        sizeof(float)*ni, nj,
        cudaMemcpyDeviceToDevice));
    CUDA_SAFE_CALL(cudaMemcpy2DToArray(f3_array, 0, 0, (void *)f3_data,
pitch,
        sizeof(float)*ni, nj,
        cudaMemcpyDeviceToDevice));
    CUDA_SAFE_CALL(cudaMemcpy2DToArray(f4_array, 0, 0, (void *)f4_data,
pitch,
        sizeof(float)*ni, nj,
        cudaMemcpyDeviceToDevice));
    CUDA_SAFE_CALL(cudaMemcpy2DToArray(f5_array, 0, 0, (void *)f5_data,
pitch,
        sizeof(float)*ni, nj,
        cudaMemcpyDeviceToDevice));
    CUDA_SAFE_CALL(cudaMemcpy2DToArray(f6_array, 0, 0, (void *)f6_data,
pitch,
        sizeof(float)*ni, nj,
        cudaMemcpyDeviceToDevice));
    CUDA_SAFE_CALL(cudaMemcpy2DToArray(f7_array, 0, 0, (void *)f7_data,
pitch,
        sizeof(float)*ni, nj,
        cudaMemcpyDeviceToDevice));
    CUDA_SAFE_CALL(cudaMemcpy2DToArray(f8_array, 0, 0, (void *)f8_data,
pitch,
        sizeof(float)*ni, nj,
        cudaMemcpyDeviceToDevice));

    // Tell CUDA that we want to use f1_array etc as textures. Also
    // define what type of interpolation we want (nearest point)
    f1_tex.filterMode = cudaFilterModePoint;
    CUDA_SAFE_CALL(cudaBindTextureToArray(f1_tex, f1_array));

    f2_tex.filterMode = cudaFilterModePoint;
    CUDA_SAFE_CALL(cudaBindTextureToArray(f2_tex, f2_array));

    f3_tex.filterMode = cudaFilterModePoint;
    CUDA_SAFE_CALL(cudaBindTextureToArray(f3_tex, f3_array));

    f4_tex.filterMode = cudaFilterModePoint;
    CUDA_SAFE_CALL(cudaBindTextureToArray(f4_tex, f4_array));

    f5_tex.filterMode = cudaFilterModePoint;
    CUDA_SAFE_CALL(cudaBindTextureToArray(f5_tex, f5_array));

    f6_tex.filterMode = cudaFilterModePoint;
    CUDA_SAFE_CALL(cudaBindTextureToArray(f6_tex, f6_array));

    f7_tex.filterMode = cudaFilterModePoint;
    CUDA_SAFE_CALL(cudaBindTextureToArray(f7_tex, f7_array));

    f8_tex.filterMode = cudaFilterModePoint;

```

```

CUDA_SAFE_CALL(cudaBindTextureToArray(f8_tex, f8_array));

dim3 grid = dim3(ni/TILE_I, nj/TILE_J);
dim3 block = dim3(TILE_I, TILE_J);

stream_kernel<<<grid, block>>>(pitch, f1_data, f2_data, f3_data,
f4_data,
                                f5_data, f6_data, f7_data, f8_data);

CUT_CHECK_ERROR("stream failed.");

CUDA_SAFE_CALL(cudaUnbindTexture(f1_tex));
CUDA_SAFE_CALL(cudaUnbindTexture(f2_tex));
CUDA_SAFE_CALL(cudaUnbindTexture(f3_tex));
CUDA_SAFE_CALL(cudaUnbindTexture(f4_tex));
CUDA_SAFE_CALL(cudaUnbindTexture(f5_tex));
CUDA_SAFE_CALL(cudaUnbindTexture(f6_tex));
CUDA_SAFE_CALL(cudaUnbindTexture(f7_tex));
CUDA_SAFE_CALL(cudaUnbindTexture(f8_tex));
}

__global__ void collide_kernel (int pitch, float gr, float tau, float
faceq1, float faceq2, float faceq3,
                                float *f0_data, float *f1_data, float
*f2_data,
                                float *f3_data, float *f4_data, float
*f5_data, float *f6_data,
                                float *f7_data, float *f8_data, float
*plot_data, float *h_surf_data, int ni, int nj)
// CUDA kernel
{
    int i, j, i2d;
    float h, vx, vy, v_sq_term, rtau, rtau1;
    float f0now, f1now, f2now, f3now, f4now, f5now, f6now, f7now, f8now;
    float f0eq, f1eq, f2eq, f3eq, f4eq, f5eq, f6eq, f7eq, f8eq;
    float hmax = -1.0e6;
        float hmin = 1.0e6; gr=0.5;

    i = blockIdx.x*TILE_I + threadIdx.x;
    j = blockIdx.y*TILE_J + threadIdx.y;

    i2d = i + j*pitch/sizeof(float);

    rtau = 1.f/tau;
    rtau1 = 1.f - rtau;

    // Read all f's and store in registers
    f0now = f0_data[i2d];
    f1now = f1_data[i2d];
    f2now = f2_data[i2d];
    f3now = f3_data[i2d];

```

```

f4now = f4_data[i2d];
f5now = f5_data[i2d];
f6now = f6_data[i2d];
f7now = f7_data[i2d];
f8now = f8_data[i2d];

// Macroscopic flow props:
h = f0now + f1now + f2now + f3now + f4now + f5now + f6now + f7now +
f8now;
h_surf_data[i2d]=h;
vx = (f1now - f3now + f5now - f6now - f7now + f8now)/h;
vy = (f2now - f4now + f5now + f6now - f7now - f8now)/h;

// Set plotting variable to velocity magnitude
plot_data[i2d] = h;
// Calculate equilibrium f's
v_sq_term = (vx*vx + vy*vy);
f0eq = h*(1.0 - 5.0*gr*h/6.0-2.0*v_sq_term/3.0);
f1eq = h*(gr*h/6.0+vx/3.0 + vx*vx/2.0 - v_sq_term/6.0);
f2eq = h*(gr*h/6.0+vy/3.0 + vy*vy/2.0 - v_sq_term/6.0);
f3eq = h*(gr*h/6.0-vx/3.0 + vx*vx/2.0 - v_sq_term/6.0);
f4eq = h*(gr*h/6.0-vy/3.0 + vy*vy/2.0 - v_sq_term/6.0);
f5eq = h*(gr*h/24.0+( vx + vy)/12.0 + ( vx + vy)*( vx + vy)/8.0 -
v_sq_term/24.0);
f6eq = h*(gr*h/24.0+(-vx + vy)/12.0 + (-vx + vy)*(-vx + vy)/8.0 -
v_sq_term/24.0);
f7eq = h*(gr*h/24.0+(-vx - vy)/12.0 + (-vx - vy)*(-vx - vy)/8.0 -
v_sq_term/24.0);
f8eq = h*(gr*h/24.0+( vx - vy)/12.0 + ( vx - vy)*( vx - vy)/8.0 -
v_sq_term/24.0);

// Do collisions
f0_data[i2d] = rtau1 * f0now + rtau * f0eq;
f1_data[i2d] = rtau1 * f1now + rtau * f1eq;
f2_data[i2d] = rtau1 * f2now + rtau * f2eq;
f3_data[i2d] = rtau1 * f3now + rtau * f3eq;
f4_data[i2d] = rtau1 * f4now + rtau * f4eq;
f5_data[i2d] = rtau1 * f5now + rtau * f5eq;
f6_data[i2d] = rtau1 * f6now + rtau * f6eq;
f7_data[i2d] = rtau1 * f7now + rtau * f7eq;
f8_data[i2d] = rtau1 * f8now + rtau * f8eq;

hmax=h_max(h_surf_data,pitch,ni,nj);
hmin=h_min(h_surf_data,pitch,ni,nj);

}

void collide(void)

// C wrapper

{

```

```

dim3 grid = dim3(ni/TILE_I, nj/TILE_J);
dim3 block = dim3(TILE_I, TILE_J);

collide_kernel<<<grid, block>>>(pitch,gr, tau, faceq1, faceq2, faceq3,
                                f0_data, f1_data, f2_data, f3_data,
f4_data,
                                f5_data, f6_data, f7_data, f8_data,
plot_data, h_surf_data,ni,nj);

CUT_CHECK_ERROR("collide failed.");

}

__global__ void apply_BC_kernel (int ni, int nj, int pitch, float vxin,
float hout,
                                float faceq2, float faceq3,
                                float *f0_data, float *f1_data, float
*f2_data,
                                float *f3_data, float *f4_data, float
*f5_data,
                                float *f6_data, float *f7_data, float
*f8_data,
                                int* solid_data)
// CUDA kernel all BC's apart from periodic boundaries:
{
    int i, j, i2d, i2d2;
    float f1old, f2old, f3old, f4old, f5old, f6old, f7old, f8old;

    i = blockIdx.x*TILE_I + threadIdx.x;
    j = blockIdx.y*TILE_J + threadIdx.y;

    i2d = i + j*pitch/sizeof(float);

    // Solid BC: "bounce-back"
    if (solid_data[i2d] == 0) {
        f1old = f1_data[i2d];
        f2old = f2_data[i2d];
        f3old = f3_data[i2d];
        f4old = f4_data[i2d];
        f5old = f5_data[i2d];
        f6old = f6_data[i2d];
        f7old = f7_data[i2d];
        f8old = f8_data[i2d];

        f1_data[i2d] = f3old;
        f2_data[i2d] = f4old;
        f3_data[i2d] = f1old;
        f4_data[i2d] = f2old;
        f5_data[i2d] = f7old;

```

```

f6_data[i2d] = f8old;
f7_data[i2d] = f5old;
f8_data[i2d] = f6old;
}

// Exit BC - very crude
// left side
if (i==0){
    i2d2 = i2d + 1;
    f1_data[i2d] = f1_data[i2d2];
    f5_data[i2d] = f5_data[i2d2];
    f8_data[i2d] = f8_data[i2d2];
}

// right side;
if (i == (ni-1)) {
    i2d2 = i2d - 1;
    f3_data[i2d] = f3_data[i2d2];
    f6_data[i2d] = f6_data[i2d2];
    f7_data[i2d] = f7_data[i2d2];

    // bottom side
    if (j==0){
        i2d = I2D(ni,i,0);
        i2d2 = I2D(ni,i,1);
        f2_data[i2d] = f2_data[i2d2];
        f5_data[i2d] = f5_data[i2d2];
        f6_data[i2d] = f6_data[i2d2];
    }

    // top side
    if (j==(nj-1)){
        i2d = I2D(ni,i,nj-1);
        i2d2 = I2D(ni,i,nj-2);
        f4_data[i2d] = f4_data[i2d2];
        f7_data[i2d] = f7_data[i2d2];
        f8_data[i2d] = f8_data[i2d2];
    }
}

}

void apply_BCs(void)

// C wrapper

{
    dim3 grid = dim3(ni/TILE_I, nj/TILE_J);
    dim3 block = dim3(TILE_I, TILE_J);

    apply_BCs_kernel<<<grid, block>>>(ni, nj, pitch, vxin, hout,
    faceq2,faceq3,

    f0_data, f1_data, f2_data,
    f3_data, f4_data, f5_data,

```

```

solid_data);

    CUT_CHECK_ERROR("apply_BCs failed.");
}

__global__ void apply_Periodic_BC_kernel (int ni, int nj, int pitch,
                                         float *f2_data, float *f4_data, float
*f5_data,
                                         float *f6_data, float *f7_data, float
*f8_data)
// CUDA kernel
{
    int i, j, i2d, i2d2;

    i = blockIdx.x*TILE_I + threadIdx.x;
    j = blockIdx.y*TILE_J + threadIdx.y;

    i2d = i + j*pitch/sizeof(float);

    if (j == 0 ) {
        i2d2 = i + (nj-1)*pitch/sizeof(float);
        f2_data[i2d] = f2_data[i2d2];
        f5_data[i2d] = f5_data[i2d2];
        f6_data[i2d] = f6_data[i2d2];
    }
    if (j == (nj-1)) {
        i2d2 = i;
        f4_data[i2d] = f4_data[i2d2];
        f7_data[i2d] = f7_data[i2d2];
        f8_data[i2d] = f8_data[i2d2];
    }
}

// C wrapper
void apply_Periodic_BC(void)
{
    dim3 grid = dim3(ni/TILE_I, nj/TILE_J);
    dim3 block = dim3(TILE_I, TILE_J);

    apply_Periodic_BC_kernel<<<grid, block>>>(ni, nj, pitch,
                                              f2_data, f4_data, f5_data,
                                              f6_data, f7_data, f8_data);

    CUT_CHECK_ERROR("apply_Periodic_BC failed.");
}

```

```

__global__ void get_rgba_kernel (int pitch, int ncol, float minvar, float
maxvar,
                                float *plot_data,
                                unsigned int *plot_rgba_data,
                                unsigned int *cmap_rgba_data,
                                int *solid_data)

// CUDA kernel to fill plot_rgba_data array for plotting

{
    int i, j, i2d, icol;
    float frac;

    i = blockIdx.x*TILE_I + threadIdx.x;
    j = blockIdx.y*TILE_J + threadIdx.y;

    i2d = i + j*pitch/sizeof(float);

    frac = (plot_data[i2d]-minvar)/(maxvar-minvar);
    icol = (int)(frac * (float)ncol);
    plot_rgba_data[i2d] = solid_data[i2d] * cmap_rgba_data[icol];
}

void get_rgba(void)
// C wrapper
{
    dim3 grid = dim3(ni/TILE_I, nj/TILE_J);
    dim3 block = dim3(TILE_I, TILE_J);

    get_rgba_kernel<<<grid, block>>>(pitch, ncol, minvar, maxvar,
                                      plot_data, plot_rgba_data, cmap_rgba_data,
                                      solid_data);

    CUT_CHECK_ERROR("get_rgba failed.");
}

void display(void)
// This function is called automatically, over and over again, by GLUT
{
    int s;
    // Set upper and lower limits for plotting
    minvar=hmin;
    maxvar=hmax;

    // Do one Lattice Boltzmann step: stream, BC, collide:
    stream();
    apply_Periodic_BC();
    apply_BCs();
}

```

```

collide();

// For plotting, map the plot_rgba_data array to the
// gl_PBO pixel buffer
CUDA_SAFE_CALL(cudaGLMapBufferObject((void*)&plot_rgba_data, gl_PBO));

// Fill the plot_rgba_data array (and the pixel buffer)
get_rgba();
CUDA_SAFE_CALL(cudaGLUnmapBufferObject(gl_PBO));

// Copy the pixel buffer to the texture, ready to display
glTexSubImage2D(GL_TEXTURE_2D,0,0,0,ni,nj,GL_RGBA,GL_UNSIGNED_BYTE,0);

// Render one quad to the screen and colour it using our texture
// i.e. plot our plotvar data to the screen
glClear(GL_COLOR_BUFFER_BIT);
glBegin(GL_QUADS);
glTexCoord2f (0.0, 0.0);
glVertex3f (0.0, 0.0, 0.0);
glTexCoord2f (1.0, 0.0);
glVertex3f (ni, 0.0, 0.0);
glTexCoord2f (1.0, 1.0);
glVertex3f (ni, nj, 0.0);
glTexCoord2f (0.0, 1.0);
glVertex3f (0.0, nj, 0.0);
glEnd();
glFlush();
glutSwapBuffers();

    t = clock();
    s = t/60;

    iter+=1;
    if (iter%1==0)
    {
        printf(" iterasi = %4d  ; t= %4d  ; %4d.s\n", iter, t,
s);
    }

    if (iter==100)
    {
        system("PAUSE");
        exit(0);
    }

}

__device__ float h_max(float *h_surf_data,int pitch,int ni,int nj)
{

```

```

    int totpoints,i,j,i2d;
    float hmax =-1.0e6;

    i = blockIdx.x*TILE_I + threadIdx.x;
    j = blockIdx.y*TILE_J + threadIdx.y;

    i2d = i + j*pitch/sizeof(float);
    totpoints=ni*nj;
    if (i2d<totpoints-1){
        if (h_surf_data[i2d] > hmax){
            hmax=h_surf_data[i2d];}
    }
    return hmax;
}

__device__ float h_min(float *h_surf_data,int pitch,int ni,int nj)
{
    int i,j,i2d, totpoints;
    float hmin =1.0e6;
    i = blockIdx.x*TILE_I + threadIdx.x;
    j = blockIdx.y*TILE_J + threadIdx.y;

    i2d = i + j*pitch/sizeof(float);
    totpoints=ni*nj;
    if (i2d<totpoints-1){
        if (h_surf_data[i] < hmin)
        {
            hmin=h_surf_data[i];}
    }
    return hmin;
}

void resize(int w, int h)

// GLUT resize callback to allow us to change the window size
{
    width = w;
    height = h;
    glViewport (0, 0, w, h);
    glMatrixMode (GL_PROJECTION);
    glLoadIdentity ();
    glOrtho (0., ni, 0., nj, -200.,200.);
    glMatrixMode (GL_MODELVIEW);
    glLoadIdentity ();
}

```