BACHELOR

Lattice-Boltzmann D3Q19 on CUDA

Reijers, S.A.

Award date:
2012

Lattice-Boltzmann D3Q19 on CUDA

by Sten Reijers

Nr: R-1807-S

1 Summary

This report evaluates the internship of Sten Reijers at the WDY group. In this period I converted a large part of the current lattice-Boltzmann code (D3Q19) into a GPU/CUDA code. One of the most important things is to organize the memory in a good structure. The best structure to organize memory is structure.p[p][$\vec{r}$] where $\vec{r}$ is a lattice site and p is the population number. Using this structure the memory bandwidth reached was $100.28GB/s$ on average on a system size of $64 * 64 * 64$. The theoretical memory bandwidth of the used graphical card was $144GB/s$. This result is a lot better compared with the existing structure on the CPU code (structure[$\vec{r}$].p[p]), which did only $11.80GB/s$ on the same systemsize. This last structure is not able to exploit the caching done by the graphical hardware. In my experience, using the new CUDA code, you can save up to 40% on hardware costs while doing the same amount work in the same amount of time.

Contents

2 Introduction

In the past decade a new class of mesoscopic methods based on minimal lattice formulations of Boltzmann's kinetic equation has captured significant interest world wide. It is a good alternative to methods based on the discretized Navier-Stokes equation. The lattice-Boltzmann method enables scientists to simulate complex fluid behavior like phase transitions, anomalous viscosity, aging behavior and a lot more. Currently prof. Federico Toschi, chair holder of Computational Physics of Multi-scale Transport Phenomena in the department of Physics at Eindhoven University of Technology, is in charge of the programming project for the lattice-Boltzmann D3Q19 model. The current code is programmed in such a way that it will parallelize it's calculations among a cluster of CPUs using the openMPI library.

This report looks into the possibility of combining openMPI and CUDA into one code. CUDA stands for Compute Unified Device Architecture and is developed by Nvidia as a parallel computing architecture for graphics processing. GPUs (graphical processing units) are usually optimized for floating point operations, especially the NVidia Telsa cores. Theoretically the lattice-Boltzmann code should run faster, which can be used to increase system sizes or do more calculations in shorter time frames.

One of the key questions answered by this report is a way to organize memory inside the GPU. The current datastructure used in the CPU code, structure[$\vec{r}$].p[p] where $\vec{r}$ is a lattice site and p the population on that site, is not suficed for CUDA. A good part of the report also gives a crash course in CUDA programming, new programmers can directly start on the new available CUDA code after reading this report.

3 Theory

3.1 Approach

Foams, gels, slurries, colloidal glasses or other related complex fluids are systems that do not fall within any of the three basic states of matter: gas, liquid or solid. These systems live on a moving border between them. Foams are typically a mixture of gas and liquids, whose properties can change dramatically with the changing proportion of the two; wet foams can flow almost like a liquid, whereas dry foams may conform to regular patterns exhibiting solid like behavior. The behavior of both foams is vitally dependent on surface tension, namely, the interactions that control the physics at the interface between different phases. As a result, they exhibit a number of distinctive features such as: long-time relaxation, anomalous viscosity and aging behavior.

In the study of the dynamics of these substances, one equation rises: The Navier-Stokes equation. In equation 1 the Navier-Stokes equation is given for incompressible fluids $\boldsymbol{\nabla} \cdot \vec{v} = 0$.

$$\frac{\partial \vec{v}}{\partial t} + (\vec{v} \cdot \boldsymbol{\nabla})\vec{v} = -\frac{1}{\rho}\boldsymbol{\nabla} p + \nu \boldsymbol{\nabla}^2 \vec{v} + \vec{g} \tag{1}$$

The solution to the Navier-Stokes equation is a vector field $\vec{v}$. This vector field gives the velocity of the fluid at every point in space $\vec{r}$ at every time t. In almost all real cases the Navier-Stokes equation is nonlinear and an analytical solution for a specific problem is hard, if not impossible, to find. Numerical simulations are needed to simulate the complex behavior of the fluids. The discretization techniques used to break down the Navier-Stokes equation rely on Eulerian and Lagrangian discretization techniques. In Eulerian methods, the physical observables are attached to a fixed grid and monitored as they change in time at each grid location. The Lagrangian method uses a different approach, it "goes with the flow", the degrees of freedom are attached to the moving fields and most notably to the critical regions of the flow where the most abrupt changes take place (usually the interfaces). Both methods have their advantages and disadvantages. Lagrangian methods do not waste degrees of freedom on uninteresting regions of the flow, however when the changes to the field are too abrupt the engine is forced to do ad hoc readjustments, which may eventually fail and lead to false results. On the other side, the Eulerian methods need a very high resolution around the interface to obtain the correct results. This is very costly in terms of calculations.

In the past decade, a new class of mesoscopic methods based on minimal lattice formulations of Boltzmann's kinetic equation has captured significant interest as an efficient alternative to continuum methods based on the discretization of the Navier-Stokes equations for non-ideal fluids. A popular mesoscopic technique is the pseudo potential lattice-Boltzmann (LBSC) method, developed over a decade ago by Shan and Chen. The mathematical formulation of this discipline is given in the next section [4].

Complex fluids sets a pressing challenge for computer simulations, since characteristic calculations of disordered fluids can escalate very fast in terms of calculation time. It is therefore critical to optimize any programming code that is used to calculate the fluid. In chapter 3.4 a powerfull toolkit, called CUDA, will be explained and used to optimize existing lattice-Boltzmann code.

3.2 The lattice-Boltzmann method

The lattice-Boltzmann method (LBM) is a computational method used to simulate complex fluid dynamics. Instead of solving the Navier-Stokes equation, the discrete Boltzmann equation is solved to simulate the dynamics. This discrete function is a derivative of the Boltzmann equation which describes a system which is not in its equilibrium state. The Boltzmann equation is given by:

$$\frac{\partial F}{\partial t} + \nu \cdot \boldsymbol{\nabla}_x F = \Theta(F) \tag{2}$$

In this equation $F(t, x, v)$ represents the probability density function of particles in the phase space. $\Theta(F)$ represents the Boltzmann collision operator and acts only on the velocity variables v and is local in (t, x).

To do numerical calculations on a computer, one needs to build a discrete version of the Boltzmann method.

Imagine a gas of particles on a set of discrete points that are spaced at a regular interval to form a lattice. Time is also divided into discrete time intervals. During each time step particles jump to the next lattice site and scatter according to the rules given by the collision operator $\Omega(f_i(\mathbf{x}, t))$. Each lattice site represents a single-particle distribution function, which is equal to the expected number of identical particles in each of the available particle states i. Each particle state i is defined by a velocity, which is of course again limited to a set of discrete velocities. During each time step Δt the particles hop to their i different neighbors. Obeying rules given by the collision operator $\Omega(f_i(\mathbf{x}, t))$, this leads to a new particle-distribution function for every site. In this report the D3Q19 model is used, meaning that the model works in three dimensions and it has 19 discrete velocities for each lattice site (see figure 1). The size of the complete lattice is usually determined by the problem and limited by the amount of available processing power.

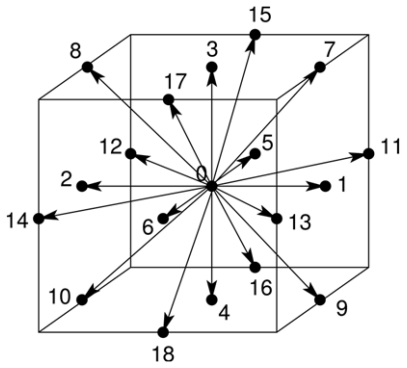

Figure 1: A lattice site with 19 different discrete speed velocities.

All this combined leads to the lattice-Boltzmann equation for a lattice site $\mathbf{x}$:

$$f_i(\mathbf{x} + \mathbf{e_i}, t + \Delta t) = f_i(\mathbf{x}, t) + \Omega(f_i(\mathbf{x}, t)) \tag{3}$$

where $\Omega(f_i(\mathbf{x}, t))$ is the collision operator on site, $i = 0, 1, 2, ..19$ is the scatter direction for a lattice site and Δt is the time interval [5]. The collision function $\Omega(f_i(\mathbf{x}, t))$ has embedded rules to leave the sum of the $f_i's$ unchanged and it conserves total energy and momentum at each site. This means that no particles are lost and when there are no forces energy and momentum should be conserved ($\frac{d\mathbf{p}}{dt} = 0$). The outcome of the collisions is very simply approximated by assuming that the momenta of the interacting particles will be redistributed at some constant rate toward an equilibrium distribution $f_i^{eq}(\mathbf{x}, t)$. This simplification is called the single-time-relaxation approximation. In mathematical terms $\Omega(f_i(\mathbf{x}, t))$ becomes:

$$\Omega(f_i(\mathbf{x}, t)) = \frac{f_i(\mathbf{x}, t) - f_i^{eq}(\mathbf{x}, t)}{\tau} \tag{4}$$

The rate of change toward the equilibrium is $\frac{1}{\tau}$. τ is called the relaxation time and it is closely related to the fluid viscosity.

3.3 Interparticle forces in lattice-Boltzmann

To simulate multiphase fluids, we need long range interactions between the fluid particles. In this report two types of interactions are used:

- Single belt interaction.
- Double belt interaction.

Single belt interaction is the interaction with the nearest neighbor and double belt interaction is interaction with the second neighbor. Forces are typically implemented via an interaction potential which is function of the density $\rho(\vec{r})$. Equation (5) shows the interaction potential used in this report

$$\Psi(\rho(\vec{r},t)) = \Psi_0 exp(\frac{-\rho_0}{\rho(\vec{r},t)}) \tag{5}$$

where Ψ_0 and ρ_0 are constants. The interaction potential can be any function as long as it is bounded and monotonically increasing, another interaction potential is shown in equation (6) (Qian et al. 1995).

$$\Psi(\rho(\vec{r},t)) = \frac{g\rho_0^2\rho(\vec{r},t)^2}{2(\rho_0+\rho(\vec{r},t))^2} \tag{6}$$

The forcing formula for single belt interaction on a lattice position $\vec{r}$ is given in equation (7). The force at lattice site $\vec{r}$ is nothing but a coupling factor G_1 and a sum over multiplications between the two densities normalized by a population factor W_a.

$$\vec{F}(\vec{r},t) = -G_1 \sum_{a=1}^{18} W_{a_1} \Psi(\vec{r},t)\Psi(\vec{r}+\vec{e_a}\Delta t,t)\vec{e_a} \tag{7}$$

The normalization factor W_{a_1} is $\frac{1}{18}$ for $a = [1,2,3,4,5,6]$ and $\frac{1}{36}$ for $a = [7,8,9,10,11,12,13,14,15,16,17,18]$. For double belt the force is given by equation (8).

$$\vec{F}(\vec{r},t) = -G_2 \sum_{a=1}^{92} W_{a_2} \Psi(\vec{r},t)\Psi(\vec{r}+\vec{e_a}\Delta t,t)\vec{e_a} \tag{8}$$

The same equation is true for double belt interaction, the only difference is the coupling factor G_2 and there are 92 neighbors to interact with instead of 18. The normalization factor W_{a_2} holds 93 values now. Eventually these forces are included in the equilibrium distribution $f_i^{eq}(\mathbf{x},t)$.

3.4 Programming with CUDA

3.4.1 Basics of CUDA

In this section the basics of programming with CUDA are explained [1]. CUDA stands for Compute Unified Device Architecture and is developed by Nvidia for graphical processing. It dramatically increases computing performance by harnessing the power of the GPU. The reason behind the discrepancy in floating-point capability between the CPU and the GPU is that the GPU is specialized for compute-intensive, highly parallel computation and therefore designed such that more transistors are devoted to data processing rather than data caching and flow control. More specifically, the GPU is especially well-suited to address problems that can be expressed as data-parallel computations. Because the same program is executed for each data element, there is a lower requirement for sophisticated flow control, and because it is executed on many data elements and has high arithmetic intensity, the memory access latency can be hidden with calculations instead of big data caches. This suits especially well for the lattice-Boltzmann method, since all operations on each lattice are the same! In the example 1 a simple CUDA kernel is programmed.

Listing 1: A simple CUDA kernel

```
#include <stdio.h>
#include <stdlib.h>

__global__ my_test_kernel()
{
  // Index
  int index;
  index = blockIdx.x * blockDim.x + threadIdx.x;
```

```
  // Do work
}

int main(int argc, char *argv[])
{
  int blocks, threads;
  blocks = 1;
  threads = 1;
  my_test_kernel<<<blocks, threads>>>();
  cudaDeviceSynchronize();
  return 0;
}
```

CUDA distinguishes two types of functions: host functions and device functions. A device function always carries a global prefix (*__global__*) before its function declaration. A host function has no prefix and is the same as you would program normal C++. A device function is called a kernel, in this case our kernel is called my_test_kernel. To execute a kernel two important parameters are needed: the number of blocks and threads. These two parameters determine how much work is done parallel: $totalruns = threads * blocks$.

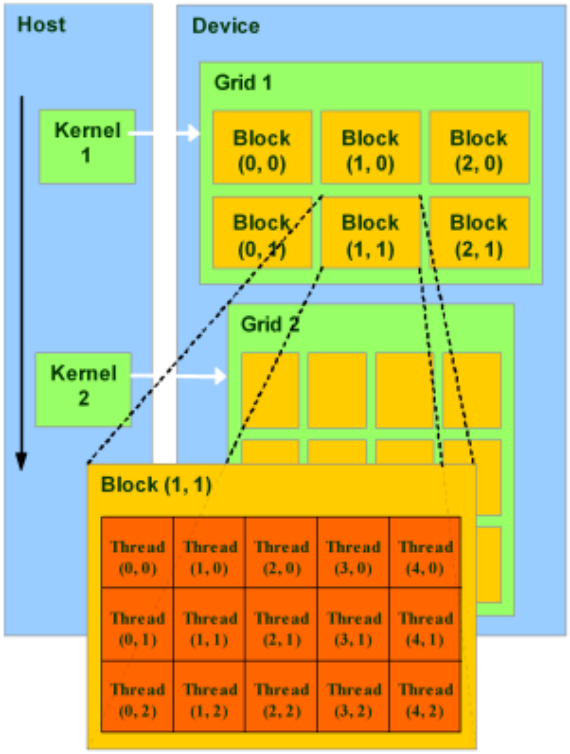

Figure 2: A physical representation on how CUDA executes kernels parallel. A grid has a number of blocks inside; these blocks get divided over the number of multiprocessors available on the GPU. Every block has a number of threads associated.

In figure 2 a physical representation is shown on how threads and blocks are related to a kernel launch in two dimensions. In this figure the block size dim2(3,2) was used with a thread size of dim2(5,3). Overall this means that the kernel was launched $15 * 6 = 90$ times parallel. Don't get ahead of yourself by thinking that this way we can launch an unlimited number of threads per block. The GPU has limits regarding the threadsize and block size, the GPU's available today can run 4 blocks with 1024 threads per block on average. These numbers strongly depend on the GPU type. Physically speaking, the GPU can only run a x number of blocks simultaneously, the rest is queued and run when a block finishes. In example 1, only one block with one thread will be executed.

Inside this kernel there are several global variables defined:

- blockIdx.(x, y, z) The current block number.
- blockDim.(x, y, z) The total number of blocks in this grid.
- threadIdx.(x, y, z) The current thread number inside this block.

These global variables are used to create an unique index for every kernel. This index can be used to do operations on an array. In normal circumstances all host and device functions run asynchronously, that's why *cudaDeviceSynchronize*(); is called after the kernel launch. This causes the host program to stop executing its tasks and waits until all kernels have finished their work. If this function is not called, the host function will return zero and the program would be finished without consulting back to the device functions.

3.4.2 Memory operations in CUDA

Listing 2: A simple memory operation

```
#include <stdio.h>
#include <stdlib.h>

__global__ my_test_kernel(int *device_array)
{
  // Index
  int index;
  index = blockIdx.x * blockDim.x + threadIdx.x;

  // Set value to 1
  device_array[index] = 1;
}

int main(int argc, char *argv[])
{
  // Defines
  int blocks, threads;
  int array_size = 512;
  int *host_array;
  int *device_array;

  // Allocate memory
  host_array = (int *) malloc(array_size*sizeof(int));
  cudaMalloc((void**) &device_array, array_size*sizeof(int));

  // Run operations
  blocks = 1;
  threads = 512;
  my_test_kernel<<<blocks, threads>>>(device_array);
  cudaDeviceSynchronize();

  // Copy result back to host memory
  cudaMemcpy(host_array, device_array, array_size*sizeof(int), cudaMemcpyDeviceToHost);

  // Evaluate the data
  //***

  // End
  return 0;
}
```

In CUDA all memory accessed from within the kernel should reside on the GPU. In example 2 a simple C script is given to set all values of an integer array to the value 1 via CUDA. Two allocations are needed,

one on the host and one on the GPU. All cuda functions return a *cudaError_T* enum, via this enum one can check to see if a function call was successful. These checks are omitted in the example. The kernel is executed using one block and 512 threads per block. The same result can be obtained using 2 blocks and 256 threads.

3.4.3 Using a cudastream

Listing 3: A simple cudaStream

```
#include <stdio.h>
#include <stdlib.h>

__global__ my_test_kernel()
{
  // Index
  int index;
  index = blockIdx.x * blockDim.x + threadIdx.x;
  // Do work
}

int main(int argc, char *argv[])
{
  // Defines
  int blocks, threads;
  cudaStream_t cuda_stream;
  cudaEvent_t cuda_start, cuda_stop;
  float cuda_time;

  // Create stream and events
  cudaStreamCreate(&cuda_stream);
  cudaEventCreate(&cuda_start);
  cudaEventCreate(&cuda_stop);

  // Run operations
  blocks = 1;
  threads = 512;

  // Start timer and the kernel
  cudaEventRecord(cuda_start, cuda_stream);
  my_test_kernel<<<blocks, threads, 0, cuda_stream>>>();

  // Wait for the kernel to finish
  cudaStreamSynchronize(cuda_stream);
  cudaEventRecord(cuda_stop, cuda_stream);

  // Report timing profile
  cudaEventElapsedTime(&cuda_time, cuda_start, cuda_stop);
  printf("Finished in %g ms\n", cuda_time);

  // End
  return 0;
}
```

A good way to organize all your work with CUDA is using a cudastream. A cudastream essentially is a queue list. You can push work on the queue, which gets executed right away. You can check how much work is inside the queue and you can also check execution times.

Example 3 shows how to measure the execution time of a kernel with high precision.

3.4.4 Optimizing a forloop

Listing 4: A for-loop in C

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
    int iwork = 512;
    int jwork = 512;
    int i,j;
    for(i=0;i<iwork;i++)
    {
        for(j=0;j<jwork;j++)
        {
            // do work
        }
    }

  // End
  return 0;
}
```

Listing 5: A for-loop in CUDA

```
#include <stdio.h>
#include <stdlib.h>

__global__ my_test_kernel(int iwork, int jwork)
{
    // Index
    int index, i, j;
    index = blockIdx.x * blockDim.x + threadIdx.x;

    // Recover i,j
    i = index % iwork;
    j = ((index - i) / iwork) % jwork;

    // Do work
}

int main(int argc, char *argv[])
{
    // Defines
    int blocks, threads;
    int iwork = 512;
    int jwork = 512;

    // Run operations
    blocks = 512;
    threads = 512;
    my_test_kernel<<<blocks, threads>>>(iwork, jwork);
    cudaDeviceSynchronize();

    // Evaluate the data
    //***

    // End
    return 0;
}
```

In example 4 and example 5 one can see how to convert a normal for-loop into a CUDA for-loop. In this example the variable i and j are recovered via the global thread index using a modulus.

3.4.5 Additional notes

- Double precision calculations only work on SM_21 architectures or higher.
- Memory calls in CUDA are synchronous, if you want to do asynchronous calls you should use: $cudaMemcpyAsync()$.
- Global variables without any prefix are not globals inside the kernel! To define a global variable you should use $\_\_device\_\_$ before declaration or you pass the pointer to the kernel as argument.

For more information regarding CUDA please consult [3].

3.4.6 lattice-Boltzmann and CUDA

Listing 6: CPU Pseudo code for LBE

```
# Global structure that hold all the pointers
structure cuda_info {
  # Cuda
  CudaStream cuda_stream;

  # Arrays holding data
  array host_f, host_feq;
  array dev_f, dev_feq;
  array host_speeds, host_density, host_forces;
  array dev_speeds, dev_density, dev_forces;

  # More information
  ..
}

# Global structure that hold all the information for all kernels
structure kernel_info {
  # Blocks and threads for every kernel call
  int pbc_threads, pbc_blocks;
  int stream_threads, stream_blocks;
  ..
}

function main()
  # Allocate memory for all pointers in the strucutre
  CudaMalloc(cuda_info);
  Malloc(kernel_info);

  # Set host_f to a initial value
  Set_initial_f(cuda_info);

  # Set all information to kernel_info
  Set_kernel_info(kernel_info);

  # Copy all host pointers to gpu
  CudaMemcpy(cuda_info);

  # Main timeloop
  for(i = 0 ; i < timeSteps ; i++)
    PBC_F_run_gpu(kernel_info, cuda_info);
    Stream_run_gpu(kernel_info, cuda_info);
    CalcMacro_run_gpu(kernel_info, cuda_info);
    PBC_Rho_run_gpu(kernel_info, cuda_info);
    Zero_forces_run_gpu(kernel_info, cuda_info);
    Set_forces_run_gpu(kernel_info, cuda_info);
    if(i%diag)
```

```
      CudaStreamSynchronize(cuda_info->cuda_stream);
      Diagnoses_run_cpu(cuda_info);
    Equilibrium_run_gpu(kernel_info, cuda_info);
    Collision_run_gpu(kernel_info, cuda_info);
    CalcMacro_run_gpu(kernel_info, cuda_info);

  # Wait for all work to be finished
  CudaStreamSynchronize(cuda_info->cuda_stream);

  # Free all memory
  CudaFree(cuda_info);
  Free(cuda_info);
  Free(kernel_info);
```

Listing 7: GPU Pseudo code for LBE

```
# A device function
__global__ function kernel_work(array dev_f)
  # Retrieve index
  index = blockIdx.x * blockDim.x + threadIdx.x;

  # Do work on cuda_info arrays
  dev_f[index] += 1.0;

# A CPU function that calls a device function
function kernel_work_run_gpu()
  kernel_work<<<kernel_info->kernel_work_blocks, kernel_info->kernel_work_threads, 0,
      cuda_info->cuda_stream>>>(cuda_info->dev_f);
```

When coding with CUDA keep in mind that the CPU is always the leading party. The GPU is doing the hard labor, but the CPU is the one who tells the GPU what to do.

In example 6 the CPU part of the code is shown in pseudo-code. Instead of using global variables, each defined by a $\_\_device\_\_$, I defined two structures called $cuda\_info$ and $kernel\_info$. These structures contain information for both the CPU and for the GPU. Pointers that are available for the GPU have a $dev\_$ prefix, pointers that are available for the CPU have a $host\_$ prefix. All GPU functions carry the "$\_run\_gpu$" endfix and they take two arguments: $kernel\_info$ and $cuda\_info$. The advantage of this method is that the code stays readable and organized. The CPU continues to push on work on the cudaStream until it reaches a $CudaStreamSynchronize$ function. This causes the CPU to block until all GPU work is finished. All functions inside the for-loop are defined in the table 1.
In example 7 the GPU part of the code is shown in pseudo-code. The GPU part is compiled using nvcc, the nvidia CUDA compiler. In my code a kernel always has two functions: A device function defined by the $\_\_global\_\_$ prefix and a small CPU part that starts the kernel defined by the "$\_run\_gpu$" endfix.

3.5 Validation between CPU and GPU

One important aspect is the validation between the GPU and CPU version. The CPU version has been put in practice and thus can be considered correct. To validate the code the GPU distribution function was compared after each timestep with the CPU distribution function. If this is done for enough time steps one can be confident that they are the same. The code for the validation is given in example 8. A pop_type is a structure where the information for the distribution function is stored.

Function	Details
$PBC\_F\_run\_gpu$	This function creates periodic boundary conditions at the 6 walls of the cube for the distribution function f.
$Stream\_run\_gpu$	This function moves the 19 particles per lattice site around to the next lattice point. It essentially does a time step Δt
$CalcMacro\_run\_gpu$	This calculates the new density ρ and the new velocities v_x, v_y, v_z.
$PBC\_Rho\_run\_gpu$	This function creates periodic boundary conditions at the 6 walls of the cube for the density function ρ.
$Zero\_forces\_run\_gpu$	This function sets the forcing arrays to zero.
$Set\_forces\_run\_gpu$	This function calculates the forces in the single and double belt.
$Diagnoses\_run\_cpu$	This function does a diagnoses and dumps the density and velocities to the harddrive for the current time step.
$Equilibrium\_run\_gpu$	This function calculates the equilibrium needed for the collision function.
$Collision\_run\_gpu$	Using values calculated by the equilibrium part, this will finialize the time step by doing the collision part.

Table 1: Definitions for the pseudo-code for-loop functions

Listing 8: Validation code for the GPU

```
/**
 * Simple crosscheck for pop_type
 * @var pop_type *first - The first distribution function
 * @var pop_type *second - The second distribution function
 * @info The code runs over i,j,k and not just over the structure index. This way one can
    debug on specific lattice sites.
 * @return boolean
 */
void validate_pop_type(pop_type *first, pop_type *second) {
    int i, j, k, p, idx0;
    for (i = 0; i < NX; i++) {
        for (j = 0; j < NY; j++) {
            for (k = 0; k < NZ; k++) {
                for (p = 0; p < NPOP; p++) {
                    idx0 = IDX(i,j,k);
                    if (first[idx0].p[p] != second[idx0].p[p]) {
                        return false;
                    }
                }
            }
        }
    }
    return true;
}
```

3.6 Mayonnaise with CUDA

The goal of the CUDA code is to simulate "mayonnaise" behavoir. Meaning, eventually we would like to simulate a lot of bubbles that feel a repulsive force for each other. This is the same behavior that you would expect to see from mayonnaise when you zoom into the fat bubbles. For this to work we need to define a few parameters in the code, please refer to table 2 for the parameters. For the code to run properly we need to have an initial condition of the fluid. Equation (9),(10),(11) describe how the density is divided over the whole system at $t = 0$.

$$k_x = \frac{8\pi}{NX}; k_y = \frac{8\pi}{NY}; k_z = \frac{8\pi}{NZ} \tag{9}$$

$$\rho_1[\vec{r}] = A(1 + 0.1sin(k_x x)sin(k_y y)sin(k_z z)) \tag{10}$$

$$\rho_2[\vec{r}] = A(1 + 0.05cos(k_x x)cos(k_y y)cos(k_z z)) \tag{11}$$

where A is the initial amplitude taken as the average of both densities and k_x, k_y, k_z are the wavenumbers.

Definition	Value
Systemsize	$128 * 128 * 128$
Timesteps	1000000
ρ_1 (density)	0.22
ρ_2 (density)	1.00
τ (relaxation time)	1.0
Single belt coupling factor (G_1)	-9.0
Double belt coupling factor (G_2)	8.1
Single component multiphase ρ (density)	0.83

Table 2: List of definitions to define a mayonnaise problem in the current code

4 Results

4.1 Organizing memory

This section will show bandwidth results using different datastructures to store information about the distribution function. Take a lattice of size $NX * NY * NZ$, every lattice point $\vec{r}$ carries 19 populations p. There are two obvious ways to store this information inside an array:

- array[p][$\vec{r}$]
- array[$\vec{r}$][p]

There are some more combinations available when using a structure:

- structure.p[p][$\vec{r}$]
- structure[$\vec{r}$].p[p]

let $\vec{r} = ((z)+\text{NZ}*((y)+\text{NY}*(x)))$ be the unique index in the array for a lattice site on $\vec{r} = (x, y, z)$.

The main code uses a structure, the reason is that inside a structure you can eventually set more data or flags for every lattice point which can be very useful in the future. There are two straightforward ways to store the information inside a structure, which one should be chosen?

To solve this question a simple piece of code was written to do streaming on both kind of structures with different thread and block sizes. When we know how much time it takes to do streaming for the two structures, the associated bandwidth can be calculated with the associated thread- and block size. Two system sizes will be tested ($NX * NY * NZ$):

- $32 * 32 * 32$
- $64 * 64 * 64$

The associated memory bandwidth in bytes per second is estimated using the following formula:

$$Bandwidth = \frac{b * n_{systemsize} * n_{populations} * n_{operations}}{t_{kernel}} \tag{12}$$

where b is the size of a double in bytes and t_{kernel} is the execution time of the kernel in seconds. The fact that the kernel also does calculations was neglected, since $t_{calculations} << t_{memoryoperation}$.

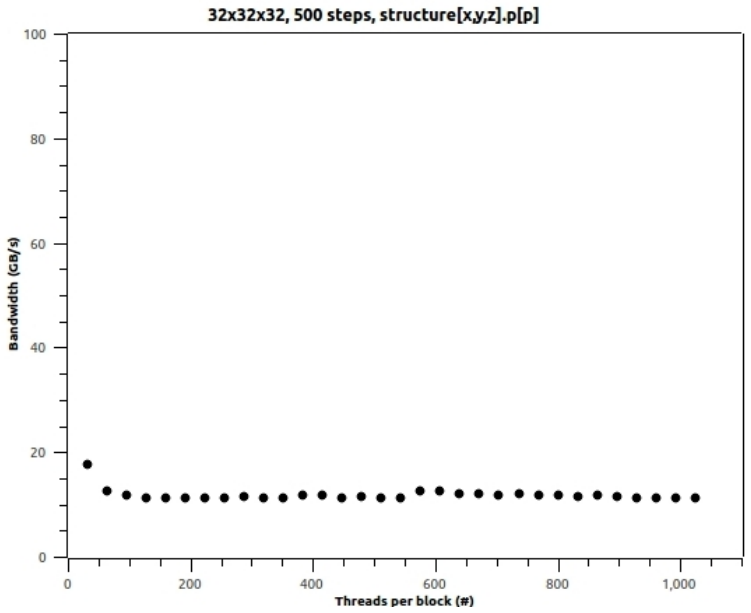

Figure 3: Results for a Nvidia Tesla C2050 / C2070 graphical card. System size was $32*32*32$ and 500 time steps were taken per data point. The used was structure[$\vec{r}$].p[p]. The average bandwidth is (11.79 ± 1.15) GB/s.

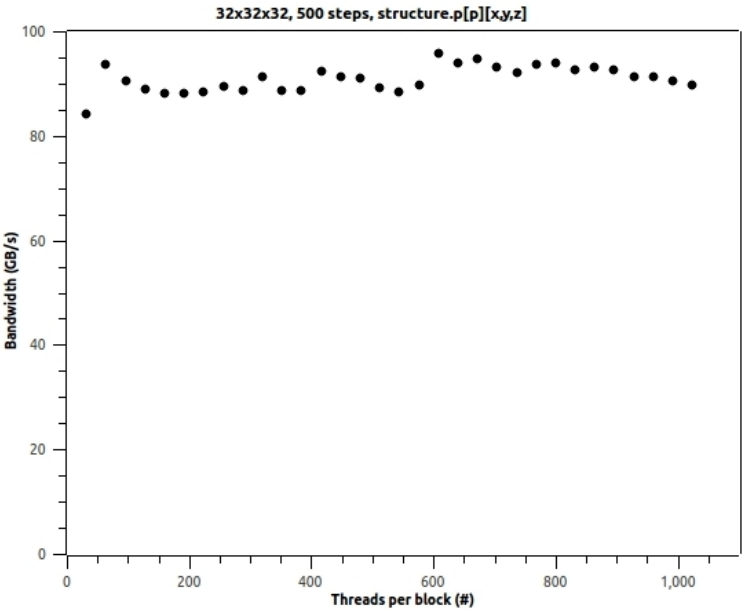

Figure 4: Results for a Nvidia Tesla C2050 / C2070 graphical card. System size was $32*32*32$ and 500 time steps were taken per data point. The used was structure.p[p][$\vec{r}$]. The average bandwidth is (90.89 ± 2.50) GB/s.

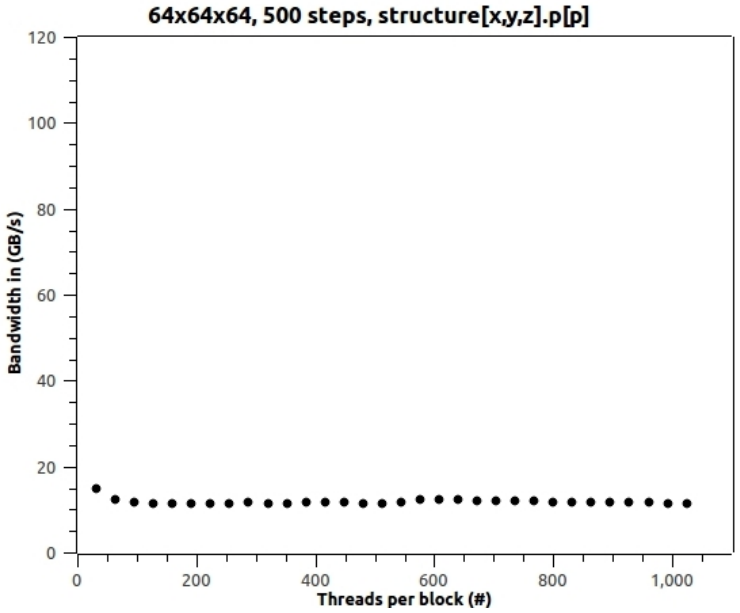

Figure 5: Results for a Nvidia Tesla C2050 / C2070 graphical card. System size was $64*64*64$ and 500 time steps were taken per data point. The used was structure[$\vec{r}$].p[p]. The average bandwidth is (11.80 ± 0.62) GB/s.

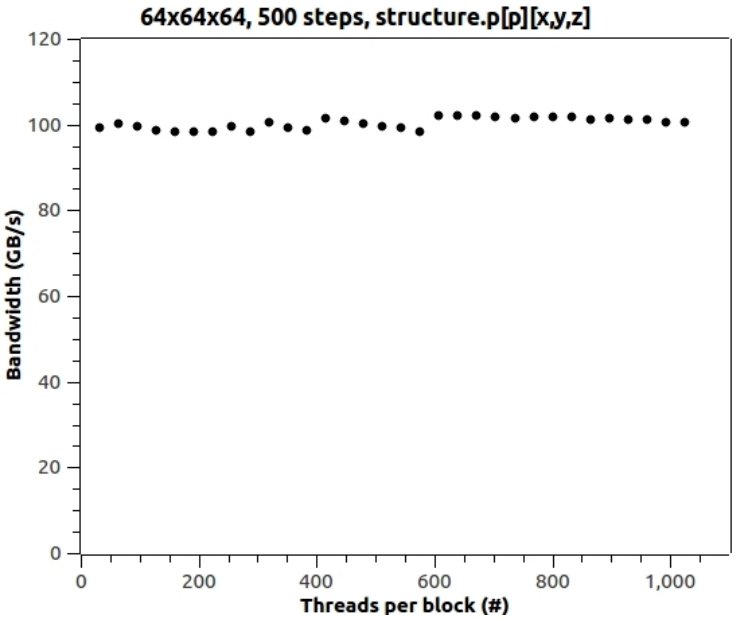

Figure 6: Results for a Nvidia Tesla C2050 / C2070 graphical card. System size was $64*64*64$ and 500 time steps were taken per data point. The used was structure.p[p][$\vec{r}$]. The average bandwidth is (100.28 ± 1.32) GB/s.

In figure 3 and figure 4 the results are plotted using a Nvidia Tesla C2050 / C2070 (see appendix A for specifications) using the system size $32 * 32 * 32$ and 500 time steps. The result strongly suggest to use the structure.p[p][$\vec{r}$] structure. In figure 5 and figure 6 the results are plotted using a Nvidia Tesla C2050 / C2070 using the system size $64 * 64 * 64$ and 500 time steps. This result also strongly suggest to use the structure.p[p][$\vec{r}$] structure.

4.2 Performance GPU vs CPU

This section discusses the performance advantages of using CUDA. In table 3 you can see the results of two different processors and two different graphical cards using a system size of $32 * 32 * 32$. The time per step is the time it takes to do one iteration (see appendix A for specifications). In table 4 you can see the

Device	Time per step (ms)	Price in june 2012 (€) [2]
Intel(R) Xeon(R) CPU E31245 (1 CPU)	76.1 ms	€264,12
Intel(R) Xeon(R) CPU E31245 (4 CPU)	23.5 ms	€264,12
Intel(R) Core(TM) i7 930 (1 CPU)	226.7 ms	€220,00 (last seen price)
Intel(R) Core(TM) i7 930 (4 CPU)	60.0 ms	€220,00 (last seen price)
Nvidia Quadro 600 1GB GDDR3	27.25 ms	€151,13
Nvidia Tesla C2050 / C2070 2.5GB GDDR5	5.83 ms	€2593,81 (last seen price)

Table 3: Performance results on a $32 * 32 * 32$ lattice.

results of two different processors and two different graphical cards using a system size of $64 * 64 * 64$.

Device	Time per step (ms)	Price in june 2012 (€) [2]
Intel(R) Xeon(R) CPU E31245 (1 CPU)	553.3 ms	€264,12
Intel(R) Xeon(R) CPU E31245 (4 CPU)	196.2 ms	€264,12
Intel(R) Core(TM) i7 930 (1 CPU)	1810.0 ms	€220,00 (last seen price)
Intel(R) Core(TM) i7 930 (4 CPU)	463.1 ms	€220,00 (last seen price)
Nvidia Quadro 600 1GB GDDR3	207.1 ms	€151,13
Nvidia Tesla C2050 / C2070 2.5GB GDDR5	37.83 ms	€2593,81 (last seen price)

Table 4: Performance results on a $64 * 64 * 64$ lattice.

4.3 Mayonnaise

In figure 9 the result was plotted for the real example. In this example all features of the CUDA code were turned on.

Listing 9: Settings file for the real example

```
lbe_sx 128.0
lbe_sy 128.0
lbe_sz 128.0
lbe_steps 1000000.0
lbe_diag_nsteps 5000.0
lbe_rho_1 0.22
lbe_rho_2 1.0
scmp_rho_0 0.83
lbe_tau_1 1.0
lbe_tau_2 1.0
scmc_init_slab_rho_x_m 20.0
scmc_init_slab_rho_x_p 40.0
scmp_coupling_g11 -9.0
scmp_double_belt_coupling_g11 8.1
scmp_coupling_g22 -8.0
```

```
scmp_double_belt_coupling_g22 7.1
scmc_coupling_g12 0.405
scmc_droplet_x0 16.0
scmc_droplet_y0 1.0
scmc_droplet_z0 16.0
scmc_droplet_radius 10
scmp_droplet_x0 16.0
scmp_droplet_y0 16.0
scmp_droplet_z0 16.0
scmp_droplet_radius 10
```

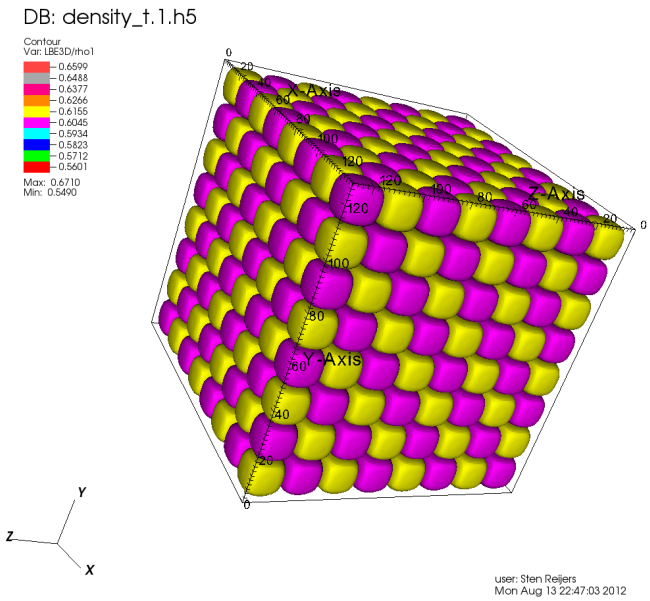

Figure 7: Snapshot of the first time step of a $128 * 128 * 128$ system (Initial condition).

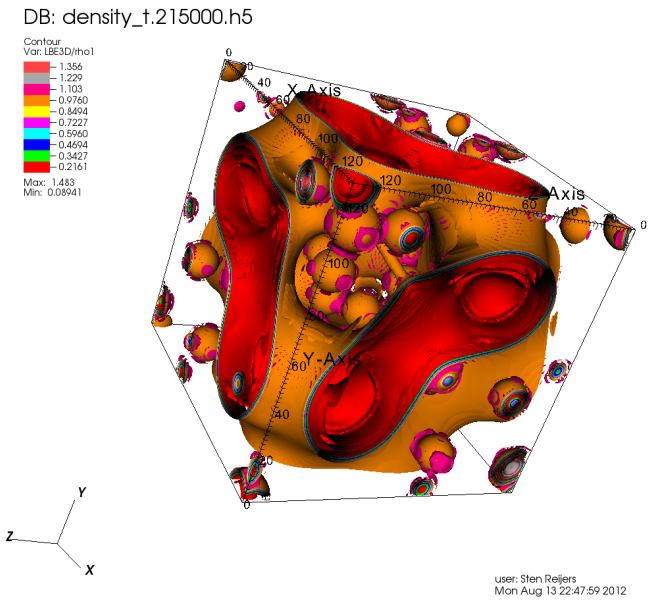

Figure 8: Snapshot after 215000 time steps of a $128 * 128 * 128$ system.

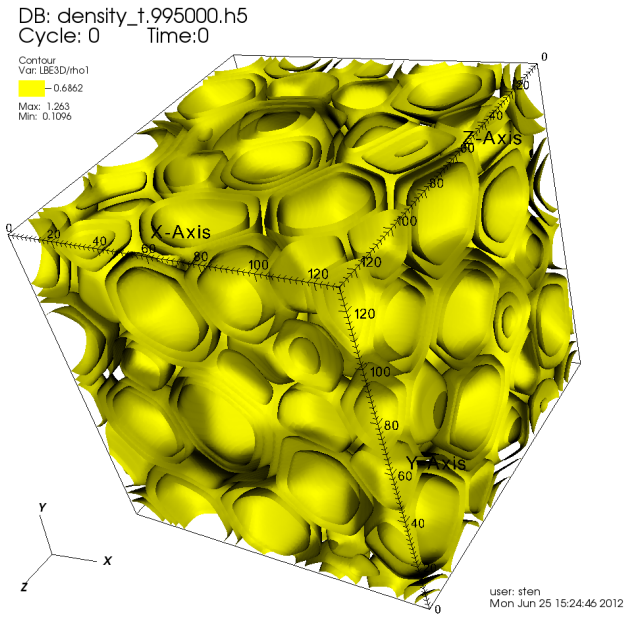

Figure 9: Snapshot after 995000 time steps of a $128 * 128 * 128$ system.

5 Conclusion

5.1 Memory

The best way to organize memory for the distribution function is to use the structure.p[p][$\vec{r}$] structure. This is of course not a coincidence. The structure.p[p][$\vec{r}$] structure has a lot more advantages in terms of caching compared with the structure[$\vec{r}$].p[p] structure. The reason is when the kernel gets executed, all running threads wants to access information for the first population ($p = 0$) (streaming kernel). This causes the hardware caching algorithm to cache a complete in place memory block. When using the structure.p[p][$\vec{r}$] the cache will contain a lot of values for $p = 0$, whereas using the structure[$\vec{r}$].p[p] structure the cache will contain values for $p = 0 : 19$ per $\vec{r}$. Using the last structure, the cache contains a lot of useless values that are not going to be used anyway. Not really efficient! Operations could run even faster when using a $\_\_syncthreads()$; after changing to another population. This causes the kernel to wait untill all threads have done their memory operations on $p = 0$, before moving to $p = 1$. This could give another performance boost, although its doubtfull that it's going to produce a noticable boost.

5.2 Performance

Table 3 and table 4 show the performance results for a $32 * 32 * 32$ and $64 * 64 * 64$ system respectively. The Nvidia Quadro 600 runs the code almost twice as fast as the single core code on Intel(R) Xeon(R) E31245 CPU, but when MPI is enabled the CPU tends to run a little bit faster. When the CUDA code gets executed on a real work horse (Nvidia Tesla C2050 / C2070) it runs five times faster compared to the Intel(R) Xeon(R) CPU E31245 in 4 threads. It's difficult to express the performance in terms of timing, because every CPU or GPU has its own advantages and specifications. It is maybe better to express the performance in terms of money. Comparing the Nvidia Quadro 600 and the Intel(R) Xeon(R) E31245 CPU, you can save approximately 40% for the same execution time. This is just a rough estimation based on just two hardware devices. In any case however, this shows that the GPU is the better choice for the job.

5.3 Mayonnaise

The final stage of my internship was to test the GPU code with a real physics problem, namely simulating mayonnaise. In this section the conclusion on the results are given together with the problems encountered.

In the theory section the initial condition for the density field is given in equation 10 and 11. These initial conditions will, surprisingly, not work. The reason for this is the number π. The number π is too well defined on a computer and the system is left in a high symmetrical state. Meaning that the system is stuck in this state. Using $\pi = 3.14$, the system will break out of this high symmetry immediately. Figure 7 shows the first time step of the system in which the density field is plotted (equation 10 using $\pi = 3.14$) on a $128 * 128 * 128$ cube.

With this problem fixed, the system will try to find a new equilibrium state as fast as possible. The parameters and the implemented lattice Boltzmann model will cause the system to form bubbles. The two interaction potentials set a global rule: The bubbles cannot merge into each other when they have little momentum.

Figure 8 gives an intermediate result at time step 215000. The system is in a chaotic state at this time, bubbles that are formed are expanding rapidly. There is also a large bubble in the middle; this large bubble turns out to be instable at later time steps. Figure 9 shows the result of the simulation after 995000 time steps. The system now has reached a semi-stable state, meaning most of the bubbles have interactions with neighbour bubbles. The forces between these bubbles will still cause the system to change, but not as dramatically as the first 50000 time steps.

Overal this makes the simulation a success, but these results still need to be checked with realtime experiments!

Validation played an important rule throughout my internship. A validation code was written to validate the distribution function after each time step with the CPU version. This code piece is given in example 8. It turns out that this validation code did not work, because there are differences between architecture when calculating with double precision. This causes the result to be slightly different. The validation code was changed to test against an error tolerance of 10^{-4} for 500 time steps. This result passed for: periodic boundaries, streaming, collision, equilibrium and singlebelt interaction.

Evaluating the error margin of 10^{-4}. It was better to try error tolerances of 10^{-52} or 10^{-53}, because this is the error what you expect. One important footnote is that the double-belt and double phase part of the code is not yet validated (when this report was written). Before anybody uses the code I advise to make a test for these functions as well using an error tolerance of 10^{-52} or 10^{-53}.

5.4 Optimizations

In the short term that I did work on the code, I had little time for fine tuning the kernels for better performance. I've put some of the changes that could really optimize the code below:

- The recovery of the variables i, j, k inside every kernel is not really needed. Everything can be calculated from the thread index right away.
- Some kernels do additional memory operations to get data. In most cases this memory access can be replaced by code, which is faster.
- The code should also be optimized for MPI operations together with CUDA.
- Check the code for errors against a tolerance of 10^{-52} or 10^{-53} with the CPU version.

6 Appendix

6.1 Appendix A

Technical Specification	Value
Form Factor	9.75”PCIe x16 form factor
Number Of Tesla GPUs	1
Number Of CUDA Cores	448
Frequency of CUDA Cores	1.15GHz
Double Precision Floating Point Performance (PEAK)	515Gflops
Single Percision Floating Point Performance (PEAK)	1.03Tflops
Total Dedicated Memory	3GB GDDR5
Memory Speed	1.5Ghz
Memory Bandwidth	144GB/s
Power Consumption	238W TDP
System Interface	PCIe x16 Gen2
Thermal Solution	Active Fansink
Display Support	Dual-Link DVI-I

Table 5: Technical specifications of the NVidia Tesla C2050 / C2070.

Technical Specification	Value
Form Factor	2.731” H x 6.6” L Single Slot
Total Dedicated Memory	1 GB DDR3
CUDA Cores	96
Memory Bandwidth	25.6 GB/s
Power Consumption	40W TDP

Table 6: Technical specifications of the NVidia Quadro 600.

References

[1] Massimo Bernaschi. A crash course on c-cuda programming for multi gpu, 2012.

[2] Pricewatch for hardware prices. http://tweakers.net/pricewatch/.

[3] NVidia Joe Stam, 2009. Maximizing GPU efficiency in exterme throughput applications.

[4] S. Succi M. Bernaschi R. Benzi, M.Sbragaglia and S. Chibbaro. Mesoscopic lattice boltzmann modeling of soft-glassy systems: Theory and simulations. 2009.

[5] Gary D. Doolen Shiyi Chen and Kenneth G. Eggert. Lattice-boltzmann fluid dynamics. *Los Alamos Science*, 22, 1994.