Benchmarking the Lattice Boltzmann model implemented using NumPy and PyCUDA

Jens Bang & Jacob Salomonsen

bachelor@snej.dk or jiekebo@gmail.com

Supervisor - Professor Brian Vinter

Bachelor Project

Department of Computer Science

University of Copenhagen

June 10, 2011

Abstract (*Jens Bang*)

The Lattice Boltzmann Model is widely used for various simulations of movement of liquids and gases. This paper will show that it is possible to achieve high gains in execution speed by combining ease of implementation of Python with parallel processing, by implementing the Lattice Boltzmann Model in Python, using the PyCUDA extension to access the highly parallel CUDA architecture. We will compare run times between the PyCUDA version and a version utilizing the more conventional, and not parallel, NumPy extension, to show the speed increases.

Contents

1 Introduction (*Jens Bang*)

This paper is a bachelor thesis written by bachelor degree students at The Computer Science Department of The University of Copenhagen. As such the intended readers are anyone with an interest in the subject matter, as well as a basic understanding of computer science and programming in Python. The object of the thesis is to examine a possible gain in execution speed, when using CUDA to implement the Lattice Boltzmann Model. It is not the object of the thesis to fully explain the Lattice Boltzmann Model, so while a reasonable understanding of physics is preferrable, it is not necessary.
When we started working on the project, the plan was to implement the Lattice Boltzmann Model in two version, both using CUDA. One version in Python, and one version in C/C++. The object was to compare CUDA implementations in Python and C/C++, to see if the ease of implementation in Python would carry with it a loss of execution speed, as compared to the execution speed of the more cumbersome implementation time of C/C++.
During the initial work on the project we realized that writing CUDA code in Python is done by simply embedding CUDA C code in the Python code. This CUDA C code will then be passed on to precisely the same compiler as the CUDA code in the C/C++ version. This would essentially mean that comparing the run times of the two implementations, would only reveal the differences between the run times of the initializing code, and would reveal nothing about the run times of the CUDA code itself.
Since this would be a comparison of normal C/C++ code run times to normal Python code run times, with no CUDA involved, a case that has already been extensively documented, and since we wanted to investigate a possible increase/decrease in execution speed when implementing CUDA code in Python, as opposed to implementing the same CUDA code in C/C++, going further would clearly not accomplish anything new. For this reason we, in agreement with out advisor, changed the scope of the thesis to investigate possible speed increases or decreses, when implementing the Lattice Boltzmann Model in Python using PyCUDA, as opposed to implementing the same model in Python using NumPy.
The two Python implementations of the Lattice Boltzmann Model are both based on a MatLab implementation of the model, which will be introduced in the Matlab implementation analysis section (subsection 4.1) of this thesis.

2 Abbreviations

ALU	=	Arithmetic logic unit
CPU	=	Central processing unit
CUDA	=	Compute Unified Device Architecture
D2Q9	=	Two dimensional, 9 directional vectors per lattice-point
D3Q19	=	Three dimensional, 19 directional vectors per lattice-point
GPU	=	Graphics processing unit
LBM	=	Lattice Boltzmann Method
PTX	=	Parallel Thread Execution
SIMD	=	Single instruction, multiple data
SIMT	=	Single Instruction Multiple Threads

3 Theory

3.1 The Lattice Boltzmann Model (*Jens Bang*)

The Lattice Boltzmann Model (LBM) is a simplified version of the Boltzmann equation, which simulates the behaviour of fluid flows. The simplification consists of limiting the particles to only occupy certain points in space (vertices in a lattice), and to only travel along specified directional vectors, with constant speed.
In this way the Lattice Boltzmann Model (LBM) describes simple fluids (gases and liquids), i.e. it ignores thermal effects and tracers. The LBM simulates movements in fluids by looking at particles found at points in a lattice at discrete time-steps. There are different versions of the LBM using either 2-dimensional or 3-dimensional lattices.
Each point in the lattice has a set of state-variables attached, describing the state of the particle found at that point. Each point also has a set of fixed directions, along which the particle can travel. In a 2-dimensional lattice there are normally 9 directions, while in a 3-dimensional lattice you see either 15 or 19 directions. These 3 setups are normally referred to as D2Q9, D3Q15 and D3Q19.

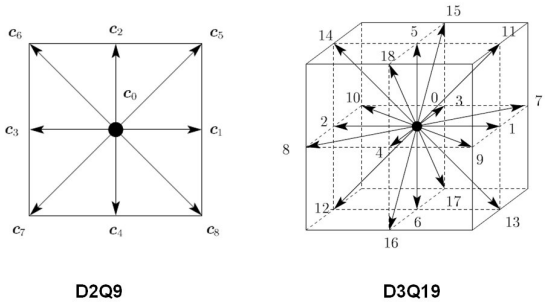


Figure 1: LBM directions (from [12])

The LBM divides the flow of gases or fluids into small, discrete time-steps. For each time-step the algorithm traverses all lattice points and for each point calculates the velocity and direction the particle travels in, as well as handling any collisions between particles. The algorithm also handle situations where particles hit an obstacle, either an individual object or the boundary of the field of simulation (i.e. if simulating flow in a tube), and is therefore thrown back in the direction from which it came, the so called bounce-back effect.

3.1.1 Using the LBM

Since the LBM is especially well suited to simulate flows around even very complex geometric structures, it has a wide variety of practical uses. Among these are:

- Simulating pore-scale processes in porous media

- Simulating Wind turbines
- Simulating the water flow around bridge pillars

3.2 Parallel processing (*Jens Bang*)

Parallel computing is the execution of multiple processes on more than one CPU or processor core. Years ago computers with multiple CPUs or processor cores were rare, and most computers only had one CPU with one processor core. This meant that they had to simulate parallel processing, by using concurrency, also known as time slicing. Nowadays computers with multiple CPUs, or at least one multi-core CPU, are commonplace, which gives us true parallel processing, where multiple tasks are physically run at the same time, each on their own CPU or processor core, or even each on their own separate computer.
Parallel processing can take on different guises: Sometimes the processes run the same code on different data, sometimes the processes run completely different code sets. An example of the latter is a program that uses one thread to receive a video stream from an IP camera, while another thread displays the received video images on the computer's screen. A well known example of the former, on a grand scale, is the SETI@Home program, where the immense amounts of data collected by SETI is broken up into small packets, which are then handed over to different private PCs to be processed individually by running the exact same code on each computer, only on different parts of the data set.
Sometimes parallel processing is performed by processes each working on their own discrete data set, as seen in SETI@Home, while at other times the processes work on a larger, shared data set. The important thing is not if they perform the same task or different tasks, nor is it important if the processes work on the same larger data set or on smaller, different data sets. The important thing in parallel processing is that the processes physically run at the same time on different CPUs or processor cores.

3.3 CPU vs. GPU (*Jens Bang*)

When comparing a CPU with a GPU, it is clear that different goals for processor functionality produce quite different designs. While the CPU has to be good for solving *any* problem, parallel workload or not, the GPU is designed with a specific task in mind, a task with a highly parallel workload.

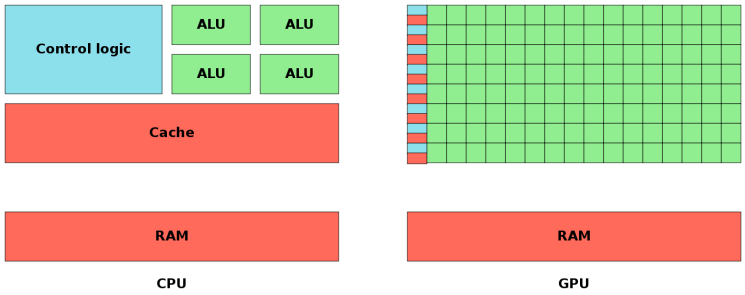


Figure 2: Visualization of the amount of a CPU which is devoted to data processing as compared to a GPU (inspired by [1] figure 1-2 p. 3).

Today's CPUs very often have more than one core, normally 2 or 4, which does let the CPU achieve a certain degree of parallelism. But because of the very often non-parallel nature of their workload, the CPU comes with a very sophisticated control logic, which is put in place to minimize the latency stemming from trying to parallelise a non-parallel workload. Today's CPUs also come with a big on-chip cache to minimize latency stemming from having to wait for the comparatively slow reads from general memory. All this results in a large part of the CPU being used for other tasks than the actual data processing.
Todays GPUs take advantage of the fact that graphics processing tasks are highly parallel, this is also known as data level parallelism, because the parallelism is built in on the data level so to speak. This fact helps the GPU to cut down on the control logic as compared to CPUs, since many of the non-parallel task problems seen in CPUs are simply not present in GPU tasks, and so the control logic to handle these problems are not needed in the GPU. This results in a maximized throughput of all processes in the GPU, since more of the GPU's computing power can be used for actual data processing.
The reason that data level parallelism does not need much control logic, is because the same set of instructions has to be performed on large parts of the data set at the same time, so the GPU can load a large block of data and then run the same instruction on each bite (this can be one byte or several, depending on the instruction) of data in unison. This form of parallelism is what Flynn calls SIMD (Single instruction, multiple data), and was used as early as the CM-1 from Thinking Machines Corporation.
The form of parallelism seen in nVidia's CUDA enabled GPUs is what nVidia call SIMT (Single Instruction Multiple Threads). The difference between original SIMD and SIMT is that under SIMT while each thread does run the same code, each individual thread doesn't have to execute exactly the same instruction at the same time. Threads can diverge, at which point the GPU's control logic will start cycling through sets of threads that agree on where in the executing code they are, issuing the next instruction to each set of threads in order [2, p. 81-82].

3.4 CUDA (*Jacob Salomonsen*)

In the beginning, GPUs had only very limited purposes. Mostly they were about generating real-time graphics for games, but also in a few cases for production and scientific applications. This has changed since the proliferation of the pixel shader. A pixel shader basically produces a color for every point on a screen, by taking into account the (x,y) position of the pixel, the light settings of a scene, the material properties of objects in the scene and so on. Early researchers found that this ability to perform computations on each pixel could be harnessed to other uses. Since pixel shaders are completely controlled by the programmer, the possibility of simply giving a GPU data as input, instead of a scene to render, was within reach. This would mark the beginning of the general purpose GPU.
The general purpose GPU did however lack a platform for developers to build upon, since learning to program pixel shaders required previous knowledge about either OpenGL or DirectX. Also even when knowing these frameworks, the model would lack the perspective of general purpose computing and instead be focused on generating graphics, where the general purpose computing aspect would be achieved by "cheating" the GPU into treating data as if it was a scene to be rasterized.
Enter CUDA. CUDA is nVidia's way of introducing general purpose GPU programming to a wider audience. CUDA utilizes the parallel nature of the GPU, allowing a low end computer to process data in a different, and some times more efficient, way.
The CUDA API exposes a set of tags, which allows a programmer to easily specify what methods, in otherwise ordinary C code, should be executed on the GPU. The way CUDA makes it possible is by using a preprocessor which parses the source code and makes different files for the individual architectures. Thus the normal C code goes to whichever compiler the user chooses, and the parallel section of the code goes to a compiler specifically designed to generate PTX-code. PTX-code is essentially assembly which is not specific to any GPU. The PTX-code is then translated into assembly code specifically targeted at the GPU [13].
This effectively separates the tedium of writing assembly code directly aimed at a specific GPU-architecture, or writing pixel shaders via graphics APIs.

3.4.1 CUDA programming model

The CUDA programming model efficiently supports the steps needed to be taken, in order to make a process parallel. The model is arranged so a decomposition of the problem area is naturally ingrained in the programming procedure, by having the programmer split the problem into parts. Since the GPU is capable of launching many threads, it is necessary to have control of what thread is launched where. Therefore nVidia has structured execution of code in a way, that both allows an easy intuition of the GPU-architecture, as well as supporting the steps in making a process parallel.
The code to be executed on the GPU, *the kernel*, launches a grid which contains blocks of threads. A kernel can only launch one grid at a time. Blocks can be arranged in up to two dimensions, and the threads within the blocks can be arranged in up to

three dimensions. Threads within blocks can cooperate via shared memory, but cannot cooperate with threads from another block. It is therefore pivotal for the performance of a parallel process that a domain analysis has been performed, and the memory intensive parts of the problem are contained within one block.
Blocks do not only have the role of sharing memory among threads. They also serve as a synchronization point for the threads within themselves. A special command within CUDA will make sure that all threads within a block have completed, before further execution is permitted.
To support this structure, CUDA provides built-in facilities for indexing blocks and threads, listed in Table 1. This allows the programmer to write code specifically for a single thread within a grid of many blocks, together containing millions of threads. The possible layouts of the grid structure can be seen in Figure 3.

Keyword	Dimensions	Usage
threadIdx	x,y,z	Index of the thread within the block
blockIdx	x,y	Index of the block
blockDim	x,y,z	Number of threads in each dimension
gridDim	x,y	Number of blocks i each dimension

Table 1: Indexing facilities available in the CUDA api

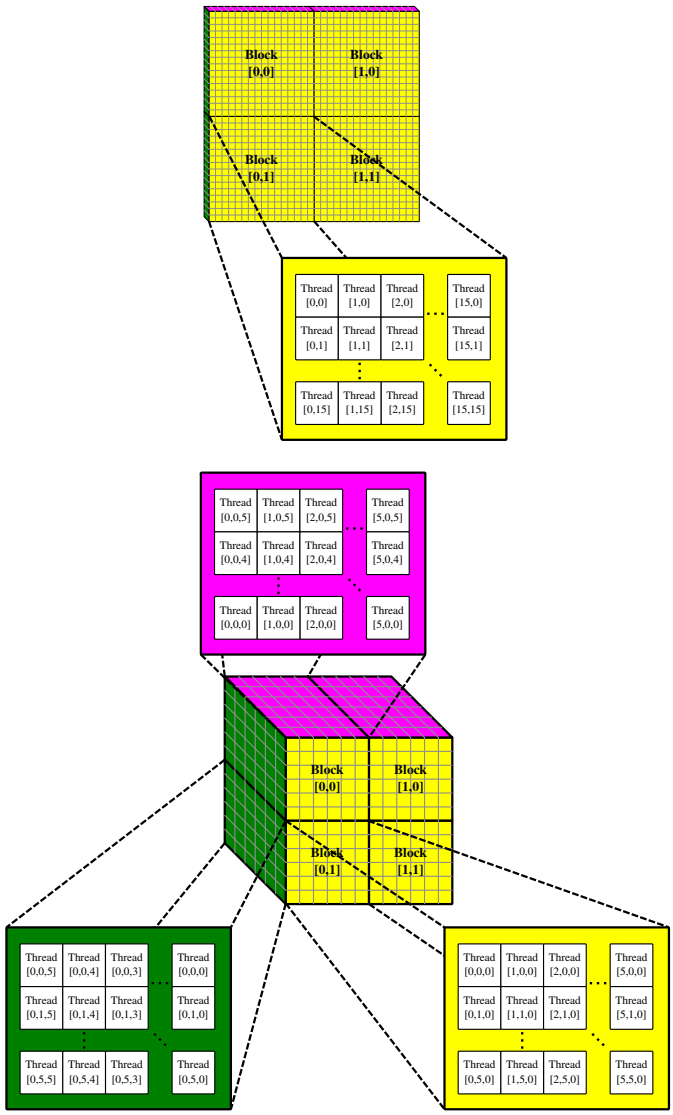


Figure 3: Top: Two dimensional block layout. **Bottom:** Three dimensional block layout. When threads increase in a block the layout is under the forced maximum amount of threads per block, here assumed to be 256 threads (inspired by [14] figure 5.2 p. 71).

When designing CUDA-programs it is important to split the problem up into parts that map directly into this block structure, so each thread has it's own specific and simple task.

3.4.2 CUDA data model

The process of working with CUDA consists of letting the CPU instruct the GPU what to do. This includes the job of memory handling between the host (CPU) and the device (GPU). The CPU is therefore responsible for copying necessary data to the GPU for processing. This work-flow is illustrated in Figure 4

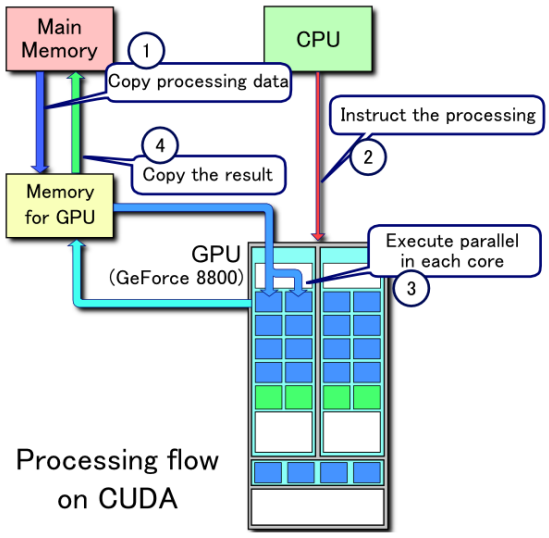


Figure 4: CUDA process model (from Wikipedia article, author: Tosaka)

CUDA exposes more than just one memory model to the programmer, each with a different purpose and performance. Figure 5 shows the different types of memory. The different memory types are listed in Table 2.

Name	Speed	Access	Scope
Global	200-300 clock cycles	Read/write	All threads + host
Texture	$<$ 100 clock cycles	Read	All threads + host
Shared	1-2 clock cycles	Read/write	All threads in a block

Table 2: A subset of the memory types the CUDA api exposes [3]

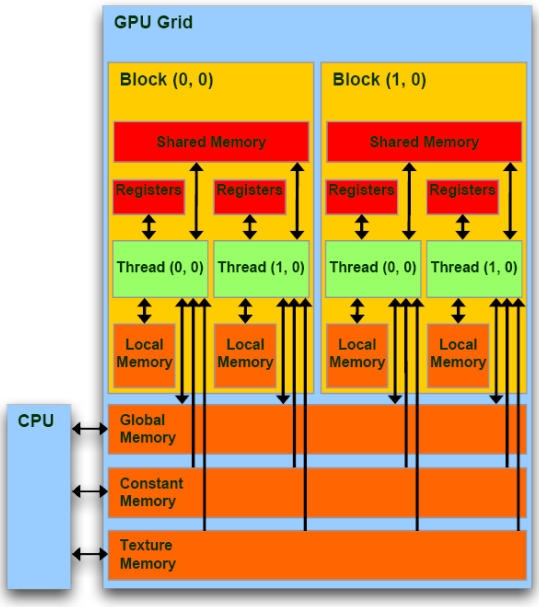


Figure 5: CUDA(GPU) memory model (from [6])

4 Implementation (*Jacob Salomonsen*)

4.1 Matlab implementation analysis

The NumPy (Listing 3) and PyCuda (Listing 4) implementations originates in a Matlab script (Listing 2) (from [4]). The implementation has been made on the D2Q9 version, but the D3Q19 is just an extension of this with an extra dimension. Since the project relies on a black-box implementation scheme, it was necessary to first perform an analysis of the supplied script. The first step taken in the analysis was to map what each of the key variables' purposes were. The result can be seen in Table 3

Variable	Type	Purpose
`F`	$nx \times ny \times 9$ (double)	eight directional matrices and one stationary state matrix
`FEQ`	$nx \times ny \times 9$ (double)	equilibrium state of the nine matrices
`BOUND`	$nx \times ny$ (binary)	boundary defined by 1s (b)
`CI`	8×1 (integer)	linear index of directional matrices
`ON`	$b \times 1$ (integer)	linear index of obstacle nodes
`TO_REFLECT`	$b \times 8$ (integer)	linear index of nodes to save to BOUNCEBACK over all nine matrices
`REFLECTED`	$b \times 8$ (integer)	linear index of nodes to recover from BOUNCEBACK over all nine matrices
`BOUNCEBACK`	$b \times 8$ (integer)	densities bouncing back
`DENSITY`	$nx \times ny$ (double)	summation of all nine matrices
`UX`	$nx \times ny$ (double)	summation of all densities with non zero x-component
`UY`	$nx \times ny$ (double)	summation of all densities with non zero y-component
`U_SQU`	$nx \times ny$ (double)	sum of UX squared and UY squared
`U_C2`	$nx \times ny$ (double)	sum of UX and UY
`U_C4`	$nx \times ny$ (double)	negative sum of UX and UY
`U_C6`	$nx \times ny$ (double)	negative of U_C2
`U_C8`	$nx \times ny$ (double)	negative of U_C4

Table 3: Variables and their purpose in the Matlab version of D2Q9 LBM

The main and most important variable in the script is the `F`-variable. From Table 3 we see that `F` is a $nx \times ny \times 9$ matrix. This means that it has the size of the simulated field, and nine states of the the simulated fields. In thread with left of Figure 1, this variable stores eight directional states and one stationary state.
The secondary result of the variable analysis was a clearer understanding of the proce-

dures taking place upon running the Matlab script. The script can be divided into four separate parts: **Propagation**, **Density**, **Equilibrium**, **Bounce-back**.

Propagation Propagation is where node densities are spread outward to affect neighbouring nodes. In the Matlab code (Listing 2) this happens in line 44-55. Lets examine line 45 and see how this is done

```
>> F(:,:,1)=F([nx 1:nx-1],:,1);
```

On the right hand side a new matrix is created from `F`, by simply re-indexing the original array. Basically if the simulated field is $nx = 10$ wide the statement `[nx 1:nx-1]` creates the following

```
10    1    2    3    4    5    6    7    8    9
```

Essentially propagating the data in an eastward direction, by wrapping the end of the matrix around and shifting the rest forward, corresponding to vector c1 in Figure 1. The rest of the expression says to go along all elements of the second axis, and only operate on matrix one out of nine. One thing to keep in mind when analysing the Matlab code is that Matlab uses one-based indexing, as opposed to most programming languages (e.g. Python) which use zero-based indexing.
The rest of the propagation part of the script is simply an extension of the above, with the only difference being the direction of the propagation, by shifting the indices appropriate to the matrix in question.

Density The first task done in the density part (line 57-61 in Listing 2) of the Matlab script is the saving of densities bouncing back from obstacles. The way this is done is a little involved so it deserves some attention.
On line 21 of Listing 2 the variable `CI` is generated, and as stated in Table 3 the variable is the linear index of the first element of the 1–8 matrices (matrix 9 is not used for bounce-back). This means that if the simulated field is 10×10 the indices would be

```
0   100   200   300   400   500   600   700
```

Combined with the variable `ON`, which contains the linear index for obstacle nodes, this can be used to calculate the linear position of obstacle nodes, in all of the 1–8 matrices. The indices of obstacles in matrix 1–8 is saved in the variable `TO_REFLECT` and then used to access all positions within the `F`-variable where there exists an obstacle, bearing in mind that this is the same position per obstacle for each 1–8 matrix, only with different linear indices.
The density is easily calculated by use of the sum command, which sums the matrices up along the third axis, i.e. the density matrices 1–9 are reduced to one.
The `UX` and `UY` variables are created by summing only the entries in the density matrices, where the x- or y-component respectively are non-zero. The meaning of x- and y-component is illustrated in Figure 6.

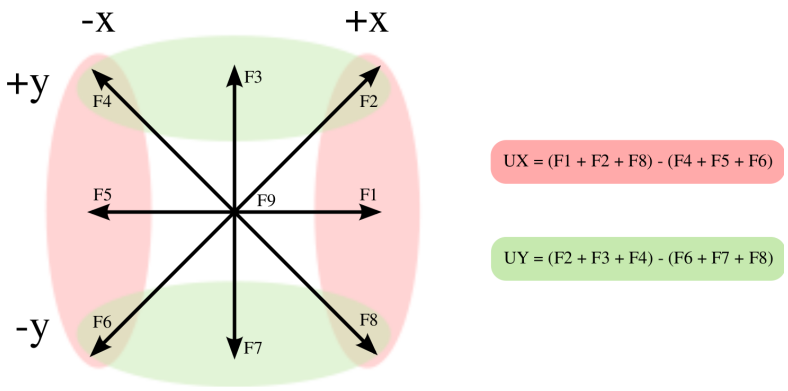


Figure 6: The `UX` and `UY` variables are created

Equilibrium The equilibrium part that runs from 63–88 of Listing 2 increases the inlet pressure as to create a constant flow in the simulation. This part also uses the result from the density part to calculate some constants used in calculation of the equilibrium. The equilibrium is calculated for each density matrix 1–9, the first one being the stationary state as seen below.

```
FEQ(:,:,9)=t1*DENSITY.*(1-U_SQU/(2*c_squ));
```

Here the special '.' operator is used, signifying that it is an element-by-element operation, not an operation on the full matrix.
Subsequently the `FEQ`-variable is added to the `F`-variable

Bounce-back The final part bounce-back is done just like at our first encounter with bounce-back, in the density part where it is saved, only this time the directions have been inverted. The operation is on line 91 of Listing 2. The `REFLECTED` variable is just like `TO_REFLECT`, with the difference, for example, being that density matrix 1 has been swapped with density matrix 5, which is in the opposite westward direction compared to the original direction. Taking this example, looking at Figure 6, F1 is the original direction and the inverse direction is F5. This pattern of inverting is applied for all density matrices, effectively sending density in the opposite direction of the obstacle.

4.2 NumPy

NumPy is a scientific computing package for Python, which allow for multidimensional arrays, and routines for fast operations on these arrays. NumPy was used in implementing the CPU-version of the D2Q9-LBM, seen in Listing 3. NumPy has the disadvantage that it is limited in performance. It does not natively support distributed architectures [9] which severely limits the potential for scientific computing using this package. Despite this the usefulness of this package is that it is easy to do prototype implementations.

At the core of the NumPy package is the *ndarray*. The *ndarray* is a statically allocated data structure, which after creation cannot change in size. If an attempt to change the size of an existing array is made, the existing array is simply overwritten. To allocate an array fitting the data layout (discussed in part in subsection 4.1 and subsubsection 4.4.1), the following command is used:

```
>>> F = numpy.zeros((9,nx,ny), dtype=float)
```

This allocates a three dimensional array consisting of zeros. Here the first dimension spans 9 indices, and the other two dimensions span the given width and height of the simulated field. The `dtype` parameter decides what data-type is to be used, in this case a float of no specification.
As an alternative an array can be allocated with ones instead of zeros, which is useful for generating the scenery in the simulation field. This is simply done by the command

```
>>> BOUNDi = numpy.ones(BOUND.shape, dtype=float)
```

Arrays have certain attributes attached to them, which can be accessed by the dot operator. In the command above it can be seen that one of these attributes, `.shape`, is accessed to create a new array with the same shape as another array. Here the meaning of the command is to create an array consisting of ones called `BOUNDi` in the shape of `BOUND`, i.e. with the same dimensions as `BOUND`.
When dealing with data in an euclidean space it is important to remember that the array class in NumPy is column major. Therefore methods for example used for plotting data will expect the first dimension of a two-dimensional array to correspond with the y-axis of a normal plot.
NumPy supports summation operations so as to be able to reduce the nine directional matrices. This is done by the use of the command

```
>>> DENSITY = numpy.add.reduce(F)
```

Which produces the result of adding all the matrices along the first dimension, i.e. the nine separate directional matrices. This could of course have been done even if the first dimension was not the directional matrices but instead the x-axis of the simulated field. Then an argument would have to be given to the `add.reduce` command to tell it along which axis to sum.

```
>>> DENSITY = numpy.add.reduce(F, axis=2)
```

To expand a matrix by a new dimension, which is needed for the boundary calculations. Since the boundary matrix is only two-dimensional it is necessary to expand upon it to be able to impose the boundary on the directional matrices. This is done by the command

```
>>> BOUND[numpy.newaxis,:,:]
```

4.3 PyCUDA

PyCUDA is an extension for the Python language, that allows for full access to the functionality available to CUDA C. Since Python is a scripting language, which means it is not mainly compiled but interpreted, the programming process is leveraged. The work-flow of PyCUDA is shown in Figure 7. One notable advantage over CUDA C is automatic resource control. Another advantage is the tight coupling with NumPy, which is especially good for the main purpose of this project. The testing procedure will gain more reliability by keeping the execution on the same platform.

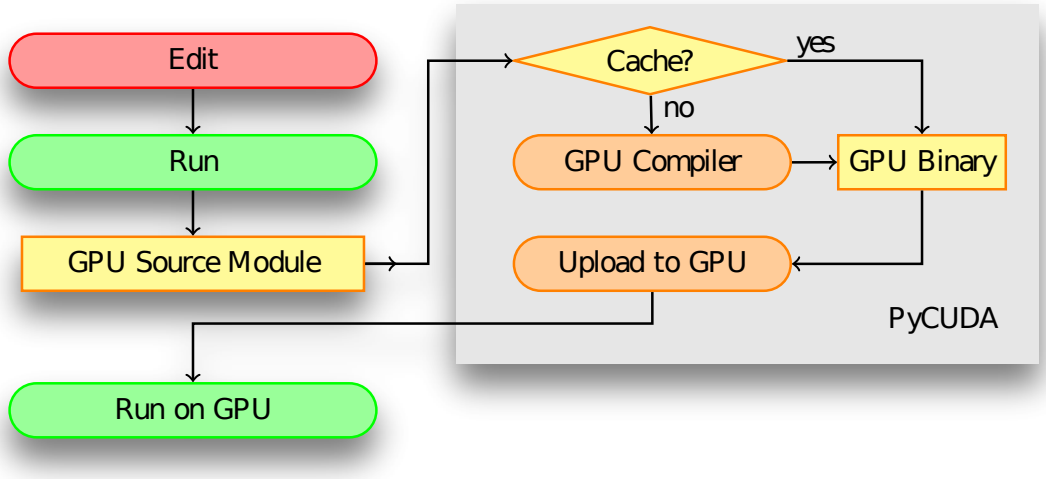


Figure 7: PyCUDA work-flow (from [8])

PyCUDA allows a programmer trained in the use of CUDA C, to easily start programming. Basically PyCUDA allows for CUDA C code to be embedded into the Python script. This is afforded by the `SourceModule` command, which is used to create CUDA kernels. A kernel is defined as a string like this

```
mod = SourceModule("""
__global__ void multiply_them(float *dest, float *a, float *b)
{
  const int i = threadIdx.x;
  dest[i] = a[i] * b[i];
}
""")
```

This is just ordinary CUDA C, embedded in the Python code. To get the handle for the kernel you call `mod.get_function`:

```
multiply_them = mod.get_function("multiply_them")
```

Now `multiply_them` can be called like any normal Python function and the kernel will run with the arguments supplied in the call. When passing parameters like constants, it is easy to string replace them into the kernel before it gets turned into a CUDA kernel. An example of this is shown in Listing 1

```
1  constant = 10
2
3  mult_string = """
4  __global__ void multiply_them(float *dest, float *a, float *b) {
5      const int i = threadIdx.x;
6      dest[i] = a[i] * b[i] * %(CONSTANT)s;
7  }
8  """
9
10 mult_string = mult_string % {
11      CONSTANT : constant
12 }
13
14 mod = SourceModule(mult_string)
```

Listing 1: An example of a kernel implemented in PyCUDA, here showing how constants are supplied to the kernel by string replacement

Here `%(CONSTANT)s` is the label denoting the location for replacement of the string with the defined constant, `constant`, containing the value 10. This is done right after the kernel string by assigning a value to the label `'CONSTANT'`. As usual the string is sent as the argument to the `SourceModule` and the kernel is ready to go.
As with CUDA C it is also necessary to load data onto the GPU by first allocating space and then copying to the GPU-memory. PyCUDA allows for allocating memory on the GPU using the `driver.mem_alloc` command, with argument for the size of the memory to allocate. This command will allocate global memory on the GPU, which is the memory both the CPU and GPU can read and write to. Copying to the GPU is performed with the `device.memcpy_htod`. Copying back from the GPU is afforded by the command `driver.memcpy_dtoh`.
When creating variables in NumPy, meant to be transferred to the GPU by PyCUDA, it is important to specify a data-type the GPU will accept. For example to specify what kind of float type a new array should be created with, the command from the NumPy section can be augmented with the following statement

```
>>> F = numpy.zeros((9,nx,ny), dtype=float).astype(numpy.float32)
```

The `astype` ensures data is allocated as 32-bit float, which is accepted by the GPU used for testing.
PyCUDA allows for allocation to different memory types on the GPU as seen in Figure 5. We have already covered global memory. What remains is one of the CPU read/write GPU read memories, Texture-memory. Texture-memory is a special kind of memory which exhibits a spatial characteristic. This means that data most suited for this memory is either two or three dimensional. This memory is relatively fast compared with the global memory, but the drawback is that seen from the GPU it is read only. The

allocation and usage is reminiscent of the CUDA C method. As with CUDA C a texture is defined in the kernel, which is contained within PyCUDA's `SourceModule`

```
texture<float, 2> tex;
```

Now as with CUDA C the texture reference is tied to the texture's 'name' in the kernel

```
texture_func = mod.get_function("kernel_handle")
texref = mod.get_texref("tex")
```

The reference has been set up, and it is possible to load data into the texture memory. Loading a matrix into the texture is done like this

```
>>> pycuda.driver.matrix_to_texref(a, texref, order="C")
```

Here `a` is used for the kernel call, and so exist on the GPU-side. The last parameter decides in what order the dimensions are found: If the value is 'C' then `tex2D(x,y)` is going to fetch `matrix(y,x)`, otherwise if the value is 'F' then `tex2D(x,y)` is going to fetch `matrix(x,y)` [7].
A very clever feature in PyCUDA is the GPUArray class. The GPUArray behaves like the *ndarray*, but performs its computations on the GPU instead of one CPU-core. To create an empty GPUArray, `a`, using an *ndarray*, `x`, as a template the following command can be used

```
a = gpuarray.empty_like(x)
```

To copy an existing *ndarray* to the GPU using the GPUArray class, the following command can be used.

```
a_gpu = gpuarray.to_gpu(numpy.array(5, dtype=numpy.float32))
```

Hence the memory allocation and copying to memory is mostly hidden behind an abstraction layer and `a_gpu` can be used as an argument for a kernel call immediately. To return the array to the host the following command can be used.

```
a_cpu = a_gpu.get()
```

This will return `a_gpu` to a normal *ndarray*, `a_cpu`. What is special about the GPUArray is that it can perform basic linear algebra on the GPU without creating kernels for this. This adds to the convenience of easy memory allocation, making it an attractive alternative. It is limited though. It is not compatible with custom element-wise operations needed in the program in question, therefore it was opted out from the PyCUDA implementation.

4.4 Design choices

In this section the choices made during the implementation of the D2Q9 LBM will be described. The implementation originates in the Matlab code seen in Listing 2.
To be able to perform a proper comparison between the CPU– and GPU–version of the model, a NumPy version has been created, also originates in the Matlab code and with inspiration from [5]. This allows for the same benchmarking technique within the Python code, using the `timeit` class.

4.4.1 Data layout

The data layout was found, by the analysis of the Matlab script, to consist of a two-dimensional matrix, repeated nine times for the D2Q9 LBM. The D2Q9 LBM maintains densities for eight directional states plus one stationary state per point in the simulated field. If the dimension of the simulated field is $nx \cdot ny$ then, to be able to contain this, the data structure must be able to contain $9 \cdot nx \cdot ny$. For both the NumPy and PyCUDA version the matrix index has been chosen as the first dimension instead of the third as it is in the Matlab script. This is because it is easier to handle linearisation and will fit the usage of some commands like `add.reduce`. Also since Python uses zero based indexing for arrays, index 0 has been chosen to correspond with the stationary state matrix, which has index 9 in the Matlab script.

4.4.2 NumPy

Since Python (and NumPy) is reminiscent of Matlab, there are no significant differences between the Matlab and NumPy versions. The largest difference lies in the method for wrapping an array and the way the boundary is imposed on the directional matrices. The resulting code can be viewed in Listing 3
Wrapping and shifting is done in lines 72–103 of Listing 3 and relies on matrix slicing. The slicing is used to move parts of the array around by assigning a shifted slice to the original matrix, while wrapping the end around to the beginning of the array. This, of course, requires careful consideration so as to avoid writing already shifted data which would lead to undesired results. This problem is easily solved by having a temporary store for the matrix, and shifting the temporary matrix, thereafter assigning it back to the original matrix:

```
F[1,:,0]      = T[1,:,-1]
F[1,:,1:]     = T[1,:,:-1]
```

The first assignment accesses a negative index in the temporary matrix, which in NumPy wraps around to the end of the matrix. This is assigned to the first element in the matrix. Subsequently, elements zero to next to last (`:-1`) are assigned to elements one to the end (`1:`). The mechanics of this procedure is illustrated in Figure 8

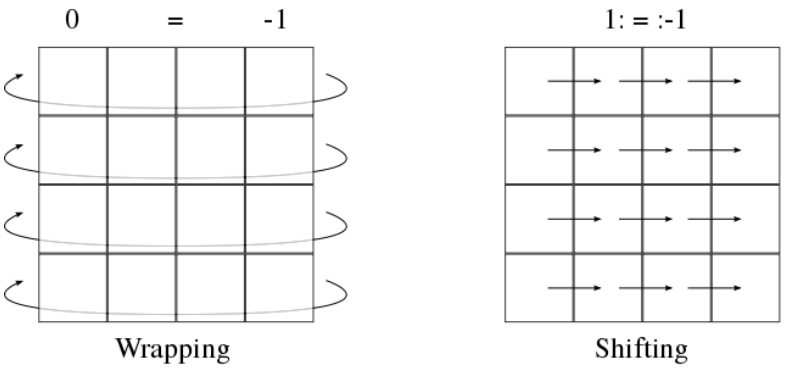


Figure 8: Wrapping and shifting mechanism in one direction for NumPy

The boundary bounce-back calculations are done by using two matrices, one normal and one inverted matrix, the latter containing zeros for the locations of boundaries and ones for the locations of free space. The normal matrix is used for selecting only the nodes in the boundary so as to save their propagational state for later use in the bounce-back section. The inverted boundary matrix is then multiplied onto the eight directional matrices which results in an erasure of all other boundaries than the boundary nodes' stationary matrix. Using the saved state from the boundary nodes, velocities are then transferred back in an inverted pattern, as described in the subsection 4.1.

4.4.3 PyCUDA

The implementation of the PyCUDA version takes origin in the analysis of the Matlab-script. The analysis yielded a clear picture of the procedures inherent in the program. This is key knowledge in attempting to parallelize an algorithm. The resulting code can be viewed in Listing 4.
The knowledge gained from the analysis showed that the procedures in the Matlab-script exhibited characteristics necessary for a program to be parallelized. The problem is very easily decomposed to much smaller sub problems. The object was then to think of a proper way to arrange data and thread execution. In the end it was decided to arrange each lattice node, so that this would be processed by a single GPU-thread. This means that each thread is a single unit in the lattice, taking care of the same data-location in all nine matrices.
The main challenge in arranging data is that it is important that each thread is treated as a single non-dependent unit, which, on execution, knows where it is and what data it is supposed to operate on. To help, CUDA supplies some indexing constants for use in the kernel, as stipulated in the CUDA section (Table 1).
Since CUDA C does not have a construct like NumPy's *ndarray*, it is necessary to linearise data access. Normally this would simply consist of assigning a maximum width of the array, and then let access to an element higher than that width wrap around, at the same time increasing the second dimension as seen in Figure 9.

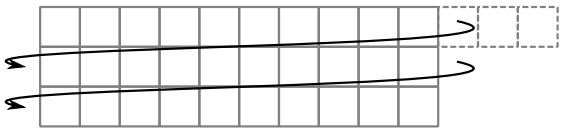


Figure 9: A linear array wrapped to fit with the underlying data-structure (a 2D array)

This however is complicated by the fact that we do not traverse the array accessing each element in sequence. Instead we access the array from certain points within the CUDA block structure. This mean that each thread has its own specific data-point in the array. Thus we need to linearise the array with the tools given to us in the CUDA programming model.
This is done by clever use of the block index, block dimension and thread index. Imagine we have a linear array we need to spread out over a series of CUDA blocks. The kernel has been launched with a pre-calculated block size, and the number of blocks have been calculated to fit with the data in question. If we take a practical example and say our blocks are one-dimensional, and they as a maximum can contain 10 threads. Your array contains 35 elements, and we want to be able to index this within the CUDA-kernel. The linear index mapping from thread location in the CUDA kernel to an array index is then calculated like this

```
idx = threadIdx.x + blockIdx.x * blockDim.x;
```

In Figure 10, it can be seen how this works for two thread locations within the CUDA-kernel. The block dimension is used together with the block index, to determine what block the thread is in, then we only have to add the thread index within the found block and we have the linear index. This technique is used throughout the code in implementing the GPU-version of the D2Q9 LBM, only extended to two dimensions.

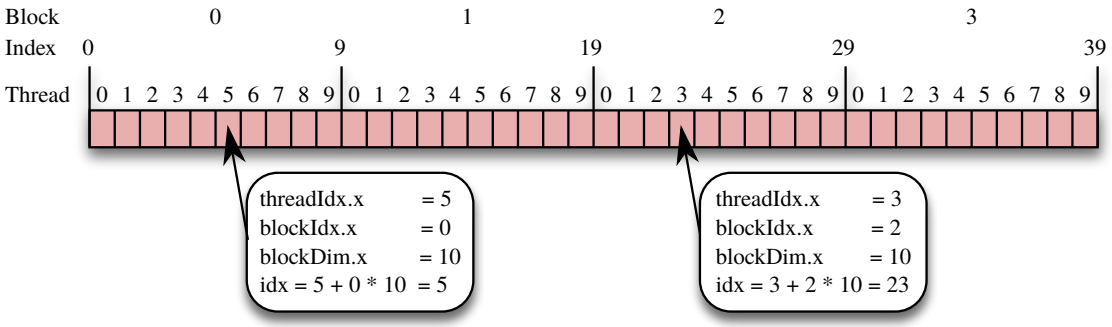


Figure 10: Calculating the linear index of a data point in an array for the CUDA thread to access

Initially the project focused on implementing the D3Q19 LBM so some work went into arranging data to fit this model. As blocks are limited as to how many threads can run

within them, the main issue which must be resolved when making an algorithm parallel arises. For the D3Q19 case the main issue was to fit a three dimensional problem into a two-dimensional space. A block can contain three dimensions, but this will minimize the amount of threads available to run in each dimension. Also, the third dimension will still be limited to the maximum allowable threads per block, as seen in the bottom of Figure 3. The solution is to linearise the third dimension into the grid layout, where a block can have two dimensions corresponding to the simulated field's size, and then be repeated as a series of blocks in the grid corresponding to the third dimension.
This however raises another problem. What if the simulated field is larger than the allowable size of the block? When the simulated field crosses the maximum block size in either dimension, more blocks are needed to support this increase. Since the grid can only be two-dimensional there exists a difficult problem in linearising the blocks from three to two dimensions. This is illustrated in Figure 11

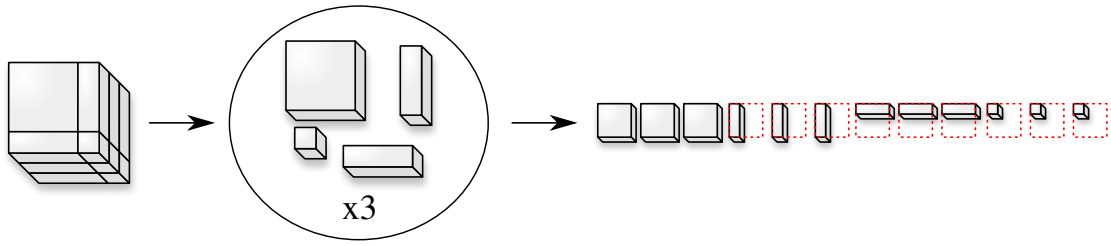


Figure 11: The case of linearising the three dimensional simulated space into the grid's two dimensions (here only one dimension is used from the grid). The blocks are chosen to be two-dimensional also. Note that the grey boxes signify the simulated field cut into block sized bits, i.e. the block size is a constant as illustrated by the dotted red squares. Also note that this is just one of the matrices which is linearised, the other 18 directions remain.

The two-dimensional case D2Q9 is however much simpler. Here it is just a matter of extending the one dimensional case illustrated in Figure 10

```
int x     = threadIdx.x + blockIdx.x * blockDim.x;
int y     = threadIdx.y + blockIdx.y * blockDim.y;
int nx    = %(WIDTH)s;
int cur   = x + y * nx;
```

This successfully emulates a two-dimensional array by using the block and thread index in the y-direction as well. The trick is that when the y-coordinate increases, the index has to follow the principle of the linearisation. Here when the y-coordinate increases it really means it has surpassed a certain amount of horizontal strips in the array, where the length of these strips are added to the index (illustrated in Figure 9). So to clarify the code above, the linear index of the x-coordinate is calculated as before, as well as the y-index, but we cannot access the array with these two indices, since the array is one dimensional. Therefore the y-dimension must be extended along a linear axis. This is

simply done by increasing the index by *nx*, which is the size of the x-axis of the simulated field, for every y-coordinate.
We are still missing one thing before we are ready with our indexing-scheme. We need to access nine separate matrices also by linearising their starting position, much like in the original Matlab-script's variable `CI`. As described in subsubsection 4.4.1, index 0 has been chosen to contain the stationary matrix instead of 9 as in the Matlab-script. In this way all directional matrices can keep their numbering consistent with the Matlab-script since Matlab uses one-based indexing and Python uses zero-based indexing.
To get the starting position of the other matrices, it is simply a matter of finding the size of the matrix, and adding to the index already found above

```
int nx    = %(WIDTH)s;
int ny    = %(HEIGHT)s;
int size  = nx * ny;
```

So when wanting to access the first matrix $1 \times$ `size` is added to the index `cur`, the second matrix $2 \times$ `size`, and so on. Now that the indexing problem is solved, the method can be used throughout the program since the same data-structure as described in subsubsection 4.4.1 will be used for all kernels.
When starting a kernel, CUDA C needs to know the dimension of the blocks and how many blocks to launch in the grid. PyCUDA is of course no exception. The number of blocks to launch is calculated after this simple formula

```
dim         = 16
blockDimX   = min(nx,dim)
blockDimY   = min(ny,dim)
gridDimX    = (nx+dim-1)/dim
gridDimY    = (ny+dim-1)/dim
```

Here it is assumed that a block can maximally contain 256 threads (16×16 threads). This might not be the case depending on the system, but it is selected as a lowest common denominator, and can at any time be changed to fit the system. First the block dimension is calculated. Since the simulated field can be smaller than the block size it is necessary to check for this. Once this is done the grid dimension can be calculated. This part relies on a trick commonly used in integer division [14, p. 65]. The result is the number of blocks which are able to contain the requested number of nodes. These parameters are given as arguments upon a kernel launch like this

```
prop(F_gpu, T_gpu, block=(blockDimX,blockDimY,1), grid=(gridDimX,gridDimY))
```

In case the simulated field is not a multiple of the block size (in this case 16×16), a conditional is inserted before each kernel's calculations. This is there to stop threads from running when there is no data for them to process. The condition is formulated below

```
if(x < nx && y < ny)
```

Here `x` and `y` is the current thread's location in the entire grid, while `nx` and `ny` is the real size of the simulated field.

Propagation kernel The propagation kernel implementation can be seen on lines 103–143 of Listing 4. In the Matlab-script, propagation relied on index shifting and recreation of arrays from this shifted index. In the NumPy code, propagation relied on array slicing and shifting by reassigning sliced parts to other indices than the original matrix. We do not have the luxury of these operations in CUDA C, firstly because they are expensive, secondly because they do not fit with the parallel paradigm. We must instead think as a single unit in the matrix. What is it that is really happening when these re-indexing or shifting actions occur. Breaking it down we see that one node in the matrix actually accepts densities of the nearest neighbours around it, as seen in Figure 12.
As mentioned before, threads must be aware of their data's position in the overall data space, by inferring it from their position in the block/grid layout. This is already done above. What is missing is an indexing of the nearest neighbours, and the wrapping which occurs in the original code. Since we already have the x- and y-indices it is easy to get the nearest neighbours of a node, it is just a matter of adding and/or subtracting one to the x- and/or y-index, still keeping in mind that the y-index needs to be further linearised afterwards.
Wrapping the matrices is a bit more complex. The thread has no concept of the simulated field as is, therefore it is necessary that the thread knows what to do, if it's on the edge of the field and about to be propagated in a direction outside the field.
When a thread tries to access a space which is beyond the border of the simulated field, the index should wrap around to the other side of the field. This is usually done by using integer division. By calculating the index modulus (%) total number of nodes we get this wrapping behaviour. This is done by the formula

```
p + b % b
```

Where `p` is the position in the index and `b` is the width of the matrix. If the matrix is 10 nodes wide for example and we want to propagate the last node in an eastward direction, we would have to wrap around and assign the value in the last node, to the first node in the array. What happens is, if we access element 11 we get

```
(10 + 10) % 10 = 0
```

Keeping in mind that arrays are using zero-based indices, this is exactly what we want to do and it will work for individual nodes independently. They will know if they should wrap around depending on their location, and the direction of the wrapping in question. As another example when we want to propagate westward, we would have to access element -1, i.e. wrap around to the end of the simulated field. By using the same formula we get

```
(-1 + 10) % 10 = 9
```

the last index in the array. Although this is a viable solution, it is very inefficient to use integer division in CUDA C, therefore it should be avoided altogether ([1] section 5.1.1 p. 45).
A better way to solve this problem relies on conditionals. This is done by simply letting the thread check if the node is on the edge of the simulated field by comparing the current node's position with the maximum amount of nodes, or with zero if it is propagating in the other direction. For brevity the ternary operator is used, which has the form

```
test ? true : false
```

In this way it is easy to let each thread check if it is on the edge of the matrix and take the appropriate action if so, otherwise getting the index of its neighbour. This is done so

```
int F1 = (x==0?nx-1:x-1) + y * nx;
```

This might seem reversed compared with the Matlab- and NumPy-version, since one would expect that the last index should be wrapped around, e.g. the node reaching the maximum of the matrix. This is not the case since we now view from a single node and need to accept densities from surrounding nodes. It becomes clearer when viewing the assignment

```
F[1*size + cur] = T[1*size + F1];
```

Here the current node accepts density from the node in the calculated position, thus if the current node is at position zero, it should get density from position $nx - 1$ in the array (the end of the array), thereby propagating density in an eastward fashion. The propagation of a single node is illustrated in Figure 12

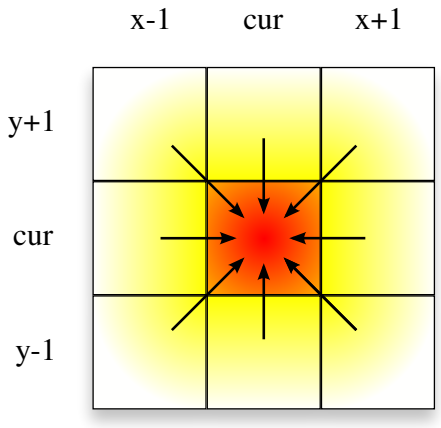


Figure 12: A single node accepting densities from surrounding neighbours

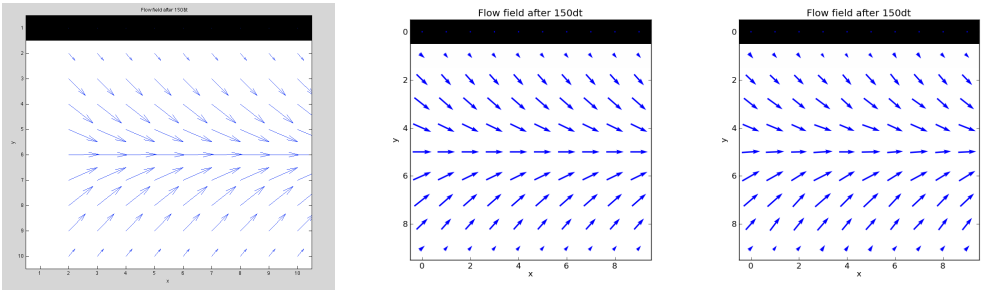
Density kernel The density kernel is found on lines 149–209 in Listing 4. In the Matlab version the bounce-back was handled by linearising the location of obstacle nodes throughout the nine matrices. This is not necessary in the PyCUDA-version since the data access is already linearised. This means that the code cleans up very nicely and becomes very efficient compared with both the Matlab- and the NumPy-version. To save the density of the obstacle nodes it is just a matter of sending the `BOUND` variable to the GPU, and testing if the thread is accessing a node for which there is an obstacle. If there exists an obstacle the densities are saved in the `BOUNCEBACK` variable.
The `DENSITY` is calculated by summing up the nine matrices for the current node. The same happens for the `UX` and `UY` variables. The inlet pressure is increased in the same way as in the Matlab- and NumPy-version. Here the scenario can be set up as in the NumPy-version, one of which is called the lid driven cavity [10]. The parameter to change scenario is supplied to the kernel via string replacement. Lastly the density for the obstacle nodes is set to zero in `D`, `UX` and `UY`. This is done in the end as to avoid divide-by-zero errors when calculating the `UX` and `UY` variables.

Equilibrium kernel The equilibrium kernel is implemented on lines 217–287 in Listing 4. It is not different from the Matlab and PyCUDA version, except that it is now working on each element individually. It is merely implemented in CUDA to save memory traffic to and from the GPU. In this way all the variables and calculations can reside on the GPU, saving a large overhead in copying variables back and forth.
As with the density kernel, constants are supplied to the kernel by string replacement.

Bounce-back kernel The bounce-back kernel works in the same way as the density kernel. It is found in Listing 4, lines 298-321. The thread uses `BOUND` to detect if it is an obstacle node. The inverted densities are then reassigned back to `F`.

4.5 Correctness

To test the correctness of the two implementations in NumPy and PyCUDA, a test was devised involving the original Matlab script. The aim of the test was of course to generate the same output in each of the different versions. A 10 by 10 field with a boundary consisting of one line along the x-axis was used to create the same conditions in each implementation. The implementation of propagation wraps around so it is as if the flow is surrounded in a horizontal tube. The Matlab script would generate a unique pattern after 150 iterations, so the test was if the two other implementations would do the same under equal conditions. Here are the results



(a) Matlab (b) NumPy (c) PyCUDA

Figure 13: Testing that the three implementations produces the same output under equal conditions

A visual inspection of Figure 13 reveals that the implementations are correct and produce the expected results according to the original Matlab implementation. There is however a small inaccuracy in the PyCUDA version on Figure 13(c), some of the velocities don't have exactly the same direction as the NumPy version in Figure 13(b). This can be attributed to floating point inaccuracy [1, p. 51].

4.6 Optimization

To further optimize the PyCUDA implementation one would look into different memory strategies. The limiting factor in most parallel algorithms is the memory access pattern. Global memory is a relatively slow memory compared with the shared memory or the texture memory. Each of these have their specific purpose, and thought needs to be given to when to use them.
Certain parts of the data could be loaded into parts of memory more suited for the use of that data. The `BOUND` variable is for example only read, and would benefit from a memory type which was spatial in layout. The texture memory would be a good candidate for this, and would increase the performance gain.
Utilizing shared memory in each block would also provide a performance gain, but this gain would come with the cost of having to synchronize each block with adjacent blocks. Many strategies could be used to resolve this, one of which is to calculate all changes on the border of the block first so that other blocks would not have to wait for a block to finish.

4.7 Issues/bugs

An earlier version of the PyCUDA implementation exhibited a bug when running the kernel with a simulated field larger than the block size (16×16) and not a multiple of 16 in each dimension. If the simulated field is chosen to say for example 25×25, the program would throw an error.
The reason for this was a mixture of miscalculations of the current thread's index, calculation of the size of the simulated field, and lacking the ability for threads to disable

when there is no data for them to process. The latter is illustrated in Figure 14. The kernels are always launched with the same block-size, and therefore will not fit with simulated fields of which the dimensions are not a multiple of the block size's dimension. If the threads are not instructed not to do so, they will try to process data, regardless of whether there is data to process or not. To solve this problem a condition was inserted to check if the thread's index (x and y) is within the simulated field, and therefore has data to process.

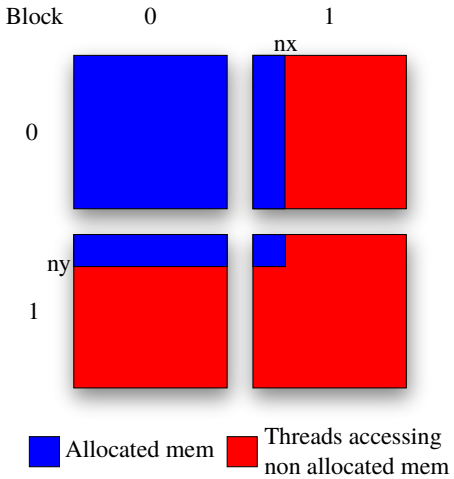


Figure 14: The allocated memory is smaller than the designated block size

Facing the problem of the miscalculations in the thread's index and size of the simulated field revealed that a mistake had been made. The size of the simulated field is not simply taking the amount of blocks times the dimension of a block, since this gives the total grid-size, which can be larger than the simulated field. The indexing calculations were done like this:

```
int x     = threadIdx.x + blockIdx.x * blockDim.x;
int y     = threadIdx.y + blockIdx.y * blockDim.y;
int nx    = blockDim.x * gridDim.x;
int ny    = blockDim.y * gridDim.y;
int cur    = x + y * nx;
int size  = nx * ny;
```

Both the `nx` and `ny` as well as the `size` was calculated incorrectly this way. The solution to this problem was not hard to find though. Instead of letting the threads calculate the simulated field, which anyway is impossible to infer from the grid-/block-structure, the simulated field's dimensions were fed to the kernels as constants by string replacement.

5 Testing (*Jacob Salomonsen*)

The object of the testing is to find if an unoptimized GPU version of the D2Q9 LBM will outperform the CPU version, in terms of running time.

5.1 System description

The tests were run on a MacBook Pro with specs listed in table 4

Spec	Value
Processor	2.66 GHz Intel Core i7
Memory	4 GB 1067 MHz DDR3
GPU	NVIDIA GeForce GT 330M on PCIe x16
VRAM	512 MB
CUDA version	3.20
Compute capability	1.2
Compute cores	48

Table 4: Specs of the system used for testing

5.2 Test design

Testing is performed with the same settings for the two versions (CPU and GPU). The number of iterations made for the simulation is the same, as well as the obstacle scenario also being the same, as seen on Figure 17. The importance of choosing the obstacle scenario is significant, since this might affect the simulation, considering that some of the operations are different for nodes which are obstacles.
The parameter which is changed is the simulated field. Here the parameter is tested with an increment of 16, where the field is always quadratic. The testing for each field size is performed in a sequence of three, and the average of these three results is taken. This is done both for the CPU- and the GPU-version.

5.3 Timing

The timing is performed by Python's built-in timing class, called *timeit*. This class can time specific parts of a program given that they are defined as functions with the `def` keyword. Only the loop which iterates over the nodes is included in the function definition so that initialization latency is not included in the calculation of running time.

5.4 Scenarios

To visually compare with other implementations of the LBM a special test case has been used, called *the lid driven cavity* [10]. This is a special setup which can help the implementer identify if the simulation behaves according to a standardized and accepted

behaviour. This will not form the basis of any conclusions about the correctness of the LBM simulation as a whole, keeping in mind that this is only a black-box implementation. It is interesting though, to see if the simulation performs as one would expect.

6 Results (*Jacob Salomonsen*)

6.1 Running time

After running the test described in section 5 on both the CPU- and the GPU-version, we got the following results. Figure 15 shows the resultant graphs by plotting the size of the simulated fields against the the average column of Table 5 and Table 6.

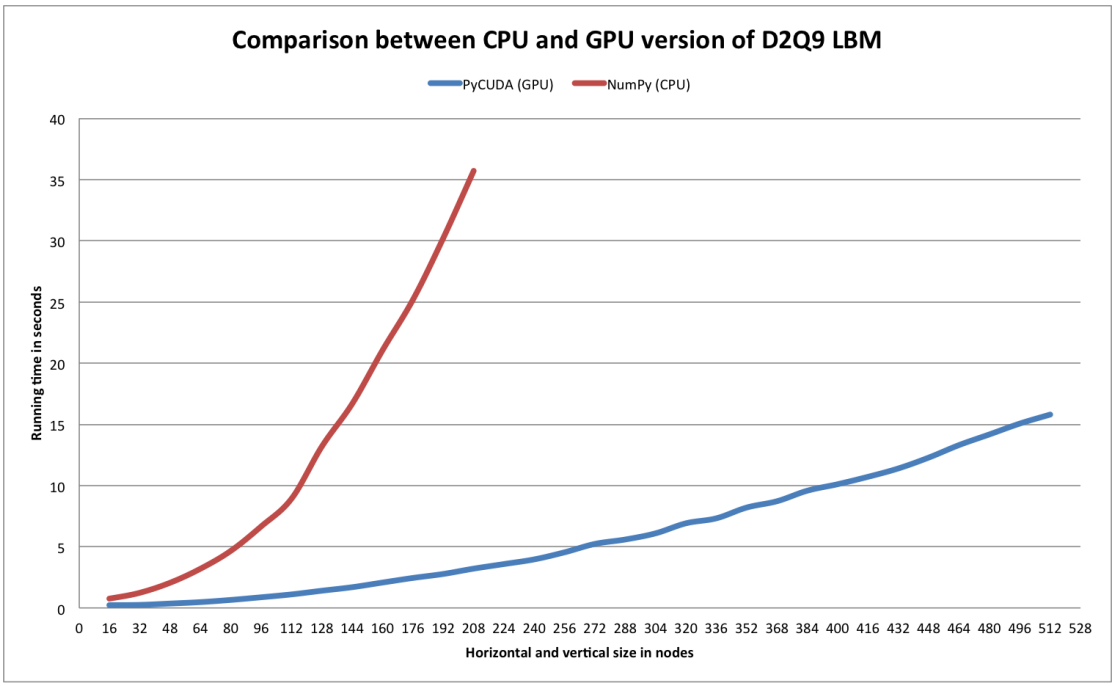
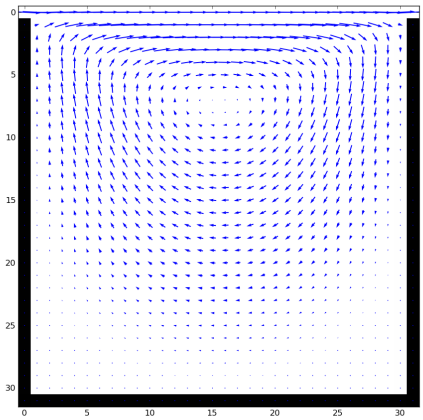


Figure 15: results

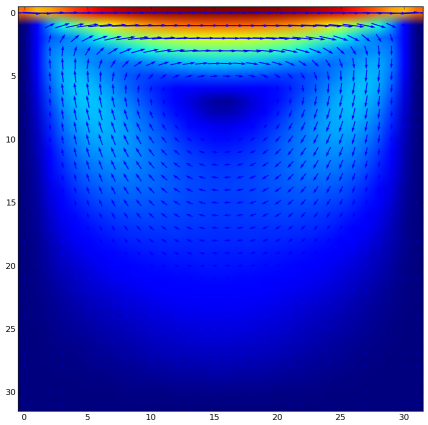
The graphs on Figure 15 shows a clear sign that the GPU-based version is faster than the CPU-based version, but that the implementations also exhibit the same time complexity. The graphs have the same shape and are reminiscent of a quadratic functions. Although this can be observed, it is beyond the scope of this project to verify it.

6.2 Other characteristics

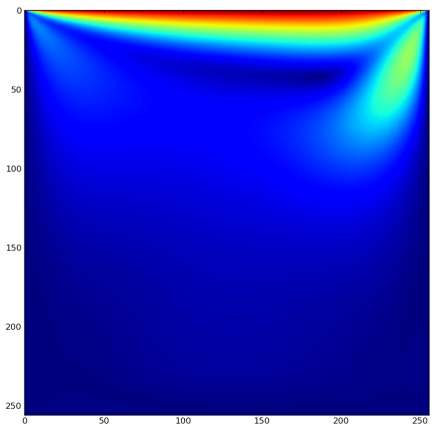
As a secondary result, the result of simulating the lid driven cavity can be compared with a reference from another implementation, to see if the implementation performs in the way one would expect. In Figure 16, the reference image is Figure 16(d), taken from [11]. Although they are not exactly the same our implementation exhibits the same pattern as in the reference solution.



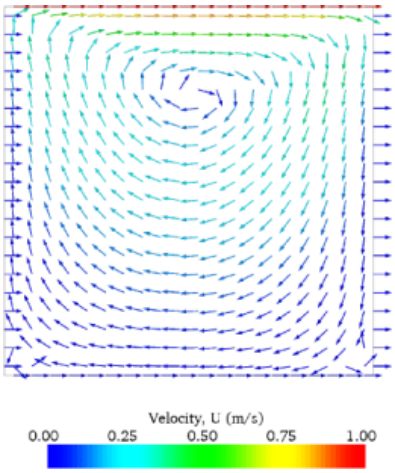
(a) Lid driven cavity flow calculated by PyCUDA



(b) The same simulation as in Figure 16(a), but with color gradient



(c) A lid driven cavity consisting of 256x256 nodes after 2700 iterations



(d) Reference lid driven cavity simulation taken from [11]

Figure 16: Lid driven cavity simulation

7 Conclusion

The revised goals of this thesis was to show the increase in execution speed, when switching a Python implementation of the Lattice Boltzmann Model from NumPy to PyCUDA. While we realize that comparing the execution speed of a NumPy implementation, which always only runs on one core, and a PyCUDA implementation, which is highly parallel in nature, running on multiple cores, scientifically speaking is like comparing apples and oranges, we wanted to show the gain in a real-life setting, and so we feel that the comparison is valid.
As outlined in subsubsection 3.1.1 the Lattice Boltzmann Model is widely used in many different circumstances, to model a large variety of things. This is why we feel it is important to show the speed increase, when switching from NumPy to PyCUDA, because that is the benefit any programmer will see, when implementing the same switch from NumPy to PyCUDA.
We started by writing a Python implementation of the supplied Matlab script, using NumPy. When we were certain of the correctness of this implementation, we wrote the PyCUDA version of the Python script, and tested the correctness of this new implementation. Then we performed time studies to see what kind, if any, of speed increase we would get. As expected the speed increase is quite significant, especially with larger problem spaces.
As mentioned in subsection 6.1 the graph shapes in Figure 15 are reminiscent of a quadratic function. But while the execution time of the NumPy version very quickly becomes unmanageable, the execution time of the PyCUDA version increases at a much slower rate. In fact the PyCUDA version is able to run problem spaces almost 3.5 times larger than the NumPy version in the same time frame.
We have clearly done what we set out to do, which was to show a speed increase when switching a Python implementation of the LBM from using NumPy to PyCUDA.

7.1 Where to go from here?

There are two obvious ways to go from here:

1. As mentioned in section 5 the PyCUDA version is unoptimized, and it could be very interesting to see what kind of speed gains there is to be found by optimizing the code, for instance following the suggestions in subsection 4.6.

2. Another interesting angle would be to examine if OpenCL is comparable to CUDA, or if it delivers even greater speed increases.

8 Appendix

The source code is available for download from https://github.com/jiekebo/LBM

8.1 Matlab version

Listing 2: Lattice Boltzmann Model, implemented in Matlab

```
% 2D Lattice Boltzmann (BGK) model of a fluid.
%  c4  c3   c2  D2Q9 model. At each timestep, particle densities propagate
%    \  |  /    outwards in the directions indicated in the figure. An
%  c5 -c9 - c1  equivalent  equilibrium  density is found, and the densities
%    /  |  \    relax towards that state, in a proportion governed by omega.
%  c6  c7   c8       Iain Haslam, March 2006.

%% init
clear all; close all;
omega=1.0;
density=1.0;
t1=4/9;
t2=1/9;
t3=1/36;
c_squ=1/3;
nx=10;
ny=10;
F=repmat(density/9,[nx ny 9]);
FEQ=F;
msize=nx*ny;
CI=0:msize:msize*7;
% for i=1:nx
%     for j=1:ny
%         if (((i-4)^2+(j-5)^2+(5-6)^2) < 6)
%             BOUND(i,j) = 1;
%         end
%     end
% end
% BOUND(nx,ny) = 1;
%a=repmat(-15:15,[31,1]);BOUND=(a.^2+a.^2)<16;BOUND(1:nx,[1 ny])=1;
BOUND=zeros(nx,ny);BOUND(1:nx,1)=1;%open channel
%BOUND=rand(nx,ny)>0.7; %extremely porous random domain
ON=find(BOUND); %matrix offset of each Occupied Node
TO_REFLECT=[ON+CI(1) ON+CI(2) ON+CI(3) ON+CI(4) ...
            ON+CI(5) ON+CI(6) ON+CI(7) ON+CI(8)];
REFLECTED= [ON+CI(5) ON+CI(6) ON+CI(7) ON+CI(8) ...
            ON+CI(1) ON+CI(2) ON+CI(3) ON+CI(4)];

avu=1; prevavu=1; ts=0; deltaU=1e-7; numactivenodes=sum(sum(1-BOUND));
it = 150;
%% while loop
%while (ts<4000 && 1e-10<abs((prevavu-avu)/avu)) || ts<100
while ts<it
    %% Propagate
    % propagate nearest neighbours
    F(:,:,1)=F([nx 1:nx-1],:,1);
    F(:,:,3)=F(:,[ny 1:ny-1],3);
    F(:,:,5)=F([2:nx 1],:,5);
    F(:,:,7)=F(:,[2:ny 1],7);

```

```
51      % propagate next-nearest neighbours
52      F(:,:,2)=F([nx 1:nx-1],[ny 1:ny-1],2);
53      F(:,:,4)=F([2:nx 1],[ny 1:ny-1],4);
54      F(:,:,6)=F([2:nx 1],[2:ny 1],6);
55      F(:,:,8)=F([nx 1:nx-1],[2:ny 1],8);
56
57      %% Density
58      BOUNCEDBACK=F(TO_REFLECT); % Densities bouncing back at next timestep
59      DENSITY=sum(F,3);
60      UX=(sum(F(:,:,[1 2 8]),3)-sum(F(:,:,[4 5 6]),3))./DENSITY;
61      UY=(sum(F(:,:,[2 3 4]),3)-sum(F(:,:,[6 7 8]),3))./DENSITY;
62
63      %% Equilibrium
64      UX(1,1:ny)=UX(1,1:ny)+deltaU; % Increase inlet pressure
65      UX(ON)=0;
66      UY(ON)=0;
67      DENSITY(ON)=0;
68      U_SQU=UX.^2+UY.^2;
69      U_C2=UX+UY;
70      U_C4=-UX+UY;
71      U_C6=-U_C2;
72      U_C8=-U_C4;
73
74      % Calculate equilibrium distribution: stationary
75      FEQ(:,:,9)=t1*DENSITY.*(1-U_SQU/(2*c_squ));
76
77      % nearest-neighbours
78      FEQ(:,:,1)=t2*DENSITY.*(1+UX/c_squ+0.5*(UX/c_squ).^2-U_SQU/(2*c_squ));
79      FEQ(:,:,3)=t2*DENSITY.*(1+UY/c_squ+0.5*(UY/c_squ).^2-U_SQU/(2*c_squ));
80      FEQ(:,:,5)=t2*DENSITY.*(1-UX/c_squ+0.5*(UX/c_squ).^2-U_SQU/(2*c_squ));
81      FEQ(:,:,7)=t2*DENSITY.*(1-UY/c_squ+0.5*(UY/c_squ).^2-U_SQU/(2*c_squ));
82
83      % next-nearest neighbours
84      FEQ(:,:,2)=t3*DENSITY.*(1+U_C2/c_squ+0.5*(U_C2/c_squ).^2-U_SQU/(2*c_squ));
85      FEQ(:,:,4)=t3*DENSITY.*(1+U_C4/c_squ+0.5*(U_C4/c_squ).^2-U_SQU/(2*c_squ));
86      FEQ(:,:,6)=t3*DENSITY.*(1+U_C6/c_squ+0.5*(U_C6/c_squ).^2-U_SQU/(2*c_squ));
87      FEQ(:,:,8)=t3*DENSITY.*(1+U_C8/c_squ+0.5*(U_C8/c_squ).^2-U_SQU/(2*c_squ));
88      F=omega*FEQ+(1-omega)*F;
89
90      %% Bounce back
91      F(REFLECTED)=BOUNCEDBACK;
92      prevavu=avu;
93      avu=sum(sum(UX))/numactivenodes; ts=ts+1;
94  end
95
96  %% Display result
97  figure;
98  colormap(gray(2));
99  image(2-BOUND);
100 hold on;
101 quiver(2:nx,1:ny,UX(2:nx,:),UY(2:nx,:));
102 title([ Flow field after ,num2str(ts), \deltat ]);xlabel( x );ylabel( y );
```

8.2 NumPy version

Listing 3: Lattice Boltzmann Model, implemented in NumPy

```
1
2  Created on May 23, 2011
3
4  @author: Jacob Salomonsen
5
6
7  import numpy as np
8
9    Simulation attributes
10 nx = 23
11 ny = 23
12 it = 900
13
14   Constants
15 omega   = 1.0
16 density = 1.0
17 t1      = 4/9.0
18 t2      = 1/9.0
19 t3      = 1/36.0
20 deltaU  = 1e-7
21 c_squ   = 1/3.0
22
23   Create the main arrays
24 F            = np.zeros((9,nx,ny), dtype=float)
25 T            = np.zeros((9,nx,ny), dtype=float)
26 F[:,:,:]    += density/9.0
27 FEQ          = np.copy(F)
28 DENSITY      = np.zeros((nx,ny), dtype=float)
29 UX           = np.copy(DENSITY)
30 UY           = np.copy(DENSITY)
31 BOUND        = np.copy(DENSITY)
32 BOUNDi       = np.ones(BOUND.shape, dtype=float)
33
34   Create the scenery
35 scenery = 0
36
37 # Tunnel
38 if scenery == 0:
39     BOUND  [0,:] = 1.0
40     BOUNDi [0,:] = 0.0
41 # Circle
42 elif scenery == 1:
43     for i in xrange(nx):
44         for j in xrange(ny):
45             if ((i-4)**2+(j-5)**2+(5-6)**2) < 6:
46                 BOUND  [i,j] = 1.0
47                 BOUNDi [i,j] = 0.0
48     BOUND  [:,0] = 1.0
49     BOUNDi [:,0] = 0.0
50 # Random porous domain
51 elif scenery == 2:
52     BOUND  = np.random.randint(2, size=(nx,ny)).astype(np.float32)
53     for i in xrange(nx):
54         for j in xrange(ny):
55             if BOUND[i,j] == 1.0:
56                 BOUNDi [i,j] = 0.0
```

```
57 # Lid driven cavity cavity
58  elif scenery == 3:
59      BOUND  [-1,:] = 1.0
60      BOUNDi [-1,:] = 0.0
61      BOUND  [1:,0]  = 1.0
62      BOUNDi [1:,0]  = 0.0
63      BOUND  [1:,-1] = 1.0
64      BOUNDi [1:,-1] = 0.0
65
66  def loop(it):
67      ts=0
68      while(ts<it):
69          global F, UX, UY
70          T[:] = F
71          # propagate nearest neigbours
72          F[1,:,0]      = T[1,:,-1]
73          F[1,:,1:]     = T[1,:,:-1]          # +x
74
75          F[3,0,:]      = T[3,-1,:]
76          F[3,1:,:]     = T[3,:-1,:]          # +y
77
78          F[5,:,-1]     = T[5,:,0]
79          F[5,:,:-1]    = T[5,:,1:]           # -x
80
81          F[7,-1,:]     = T[7,0,:]
82          F[7,:-1,:]    = T[7,1:,:]           # -y
83
84          # propagate next-nearest neightbours
85          F[2,0,0]      = T[2,-1,-1]
86          F[2,0,1:]     = T[2,-1,:-1]
87          F[2,1:,0]     = T[2,:-1,-1]
88          F[2,1:,1:]    = T[2,:-1,:-1]        # +x+y
89
90          F[4,0,-1]     = T[4,-1,0]
91          F[4,0,:-1]    = T[4,-1,1:]
92          F[4,1:,-1]    = T[4,:-1,0]
93          F[4,1:,:-1]   = T[4,:-1,1:]         # -x+y
94
95          F[6,-1,-1]    = T[6,0,0]
96          F[6,-1,:-1]   = T[6,0,1:]
97          F[6,:-1,-1]   = T[6,1:,0]
98          F[6,:-1,:-1]  = T[6,1:,1:]          # -x-y
99
100         F[8,-1,0]     = T[8,0,-1]
101         F[8,-1,1:]    = T[8,0,:-1]
102         F[8,:-1,0]    = T[8,1:,-1]
103         F[8,:-1,1:]   = T[8,1:,:-1]         # +x-y
104
105         # Densities bouncing back at next timestep
106         BOUNCEBACK = np.zeros(F.shape, dtype=float)
107         T[:] = F
108
109         T[1:,:,:] *= BOUND[np.newaxis,:,:]
110         BOUNCEBACK[1] += T[5,:,:]
111         BOUNCEBACK[2] += T[6,:,:]
112         BOUNCEBACK[3] += T[7,:,:]
113         BOUNCEBACK[4] += T[8,:,:]
114         BOUNCEBACK[5] += T[1,:,:]
115         BOUNCEBACK[6] += T[2,:,:]
116         BOUNCEBACK[7] += T[3,:,:]
117         BOUNCEBACK[8] += T[4,:,:]
118
```

```
119             DENSITY = np.add.reduce(F)
120
121             T1 = F[1,:,:]+F[2,:,:]+F[8,:,:]
122             T2 = F[4,:,:]+F[5,:,:]+F[6,:,:]
123             UX = (T1-T2)/DENSITY
124
125             T1 = F[2,:,:]+F[3,:,:]+F[4,:,:]
126             T2 = F[6,:,:]+F[7,:,:]+F[8,:,:]
127             UY = (T1-T2)/DENSITY
128
129             # Increase inlet pressure
130             if scenery != 3:
131                 UX[:,0] += deltaU
132             else:
133                 UX[0,:] += deltaU
134
135             UX[:,:] *= BOUNDi
136             UY[:,:] *= BOUNDi
137             DENSITY[:,:] *= BOUNDi
138
139             U_SQU = UX**2 + UY**2
140             U_C2=UX+UY
141             U_C4=-UX+UY
142             U_C6=-U_C2
143             U_C8=-U_C4
144
145             # Calculate equilibrium distribution: stationary
146             FEQ[0,:,:]=t1*DENSITY*(1-U_SQU/(2*c_squ))
147
148             # nearest-neighbours
149             FEQ[1,:,:]=t2*DENSITY*(1+UX/c_squ+0.5*(UX/c_squ)**2-U_SQU/(2*c_squ))
150             FEQ[3,:,:]=t2*DENSITY*(1+UY/c_squ+0.5*(UY/c_squ)**2-U_SQU/(2*c_squ))
151             FEQ[5,:,:]=t2*DENSITY*(1-UX/c_squ+0.5*(UX/c_squ)**2-U_SQU/(2*c_squ))
152             FEQ[7,:,:]=t2*DENSITY*(1-UY/c_squ+0.5*(UY/c_squ)**2-U_SQU/(2*c_squ))
153
154             # next-nearest neighbours
155             FEQ[2,:,:]=t3*DENSITY*(1+U_C2/c_squ+0.5*(U_C2/c_squ)**2-
156                                                         U_SQU/(2*c_squ))
157             FEQ[4,:,:]=t3*DENSITY*(1+U_C4/c_squ+0.5*(U_C4/c_squ)**2-
158                                                         U_SQU/(2*c_squ))
159             FEQ[6,:,:]=t3*DENSITY*(1+U_C6/c_squ+0.5*(U_C6/c_squ)**2-
160                                                         U_SQU/(2*c_squ))
161             FEQ[8,:,:]=t3*DENSITY*(1+U_C8/c_squ+0.5*(U_C8/c_squ)**2-
162                                                         U_SQU/(2*c_squ))
163
164             F=omega*FEQ+(1.0-omega)*F
165
166             #Densities bouncing back at next timestep
167             F[1:,:,:] *= BOUNDi[np.newaxis,:,:]
168             F[1:,:,:] += BOUNCEBACK[1:,:,:]
169
170             ts += 1
171
172       Run the loop
173     loop(it)
174
175     import matplotlib.pyplot as plt
176     UY *= -1
177     plt.hold(True)
178     plt.xlabel( x )
179     plt.ylabel( y )
180     plt.title( Flow field after %sdt  % it)
```

```
181  plt.quiver(UX,UY, pivot= middle , color= blue )
182  plt.imshow(BOUND, interpolation= nearest , cmap= gist_yarg )
183 #plt.imshow(np.sqrt(UX**2+UY**2))
184  plt.show()
```

8.3 PyCUDA version

Listing 4: Lattice Boltzmann Model, implemented in PyCUDA

```
1
2  Created on May 16, 2011
3
4  @author: Jacob Salomonsen
5
6
7  import numpy as np
8  import pycuda.driver as cuda
9  import pycuda.autoinit
10 from pycuda.compiler import SourceModule
11
12   Simulation attributes
13 nx = 23
14 ny = 23
15 it = 900
16
17   Constants
18 omega    = 1.0
19 density  = 1.0
20 t1       = 4/9.0
21 t2       = 1/9.0
22 t3       = 1/36.0
23 deltaU   = 1e-7
24 c_squ    = 1/3.0
25
26   Create the main arrays
27 F            = np.zeros((9,nx,ny), dtype=float).astype(np.float32)
28 F[:,:,:]    += density/9.0
29 T            = np.copy(F)
30 FEQ          = np.copy(F)
31 BOUNCEBACK   = np.zeros(F.shape, dtype=float).astype(np.float32)
32 DENSITY      = np.zeros((nx,ny), dtype=float).astype(np.float32)
33 UX           = np.copy(DENSITY)
34 UY           = np.copy(DENSITY)
35 BOUND        = np.copy(DENSITY)
36
37   Create the scenery
38 scenery = 0
39
40 # Tunnel
41 if scenery == 0:
42     BOUND  [0,:] = 1.0
43 # Circle
44 elif scenery == 1:
45     for i in xrange(nx):
46         for j in xrange(ny):
47             if ((i-4)**2+(j-5)**2+(5-6)**2) < 6:
48                 BOUND  [i,j] = 1.0
49     BOUND  [:,0] = 1.0
50 # Random porous domain
51 elif scenery == 2:
52     BOUND  = np.random.randint(2, size=(nx,ny)).astype(np.float32)
53 # Lid driven cavity
54 elif scenery == 3:
55     BOUND  [-1,:]  = 1.0
56     BOUND  [1:,0]  = 1.0
```

```
57      BOUND  [1:,-1] = 1.0
58
59      CUDA specific
60    Calculate block and grid dimensions
61  #threadsPerBlock = 256
62  #blocksPerGrid   = (nx*ny + threadsPerBlock - 1) / threadsPerBlock
63  dim           = 16
64  blockDimX     = min(nx,dim)
65  blockDimY     = min(ny,dim)
66  gridDimX      = (nx+dim-1)/dim
67  gridDimY      = (ny+dim-1)/dim
68
69    Allocate memory on the GPU
70  F_gpu         = cuda.mem_alloc(F.size * F.dtype.itemsize)
71  T_gpu         = cuda.mem_alloc(T.size * F.dtype.itemsize)
72  FEQ_gpu       = cuda.mem_alloc(FEQ.size * FEQ.dtype.itemsize)
73
74  BOUND_gpu     = cuda.mem_alloc(BOUND.size * BOUND.dtype.itemsize)
75  BOUNCEBACK_gpu = cuda.mem_alloc(BOUNCEBACK.size * BOUNCEBACK.dtype.itemsize)
76  DENSITY_gpu = cuda.mem_alloc(DENSITY.size * DENSITY.dtype.itemsize)
77  UX_gpu        = cuda.mem_alloc(UX.size * UX.dtype.itemsize)
78  UY_gpu        = cuda.mem_alloc(UY.size * UY.dtype.itemsize)
79
80  U_SQU_gpu     = cuda.mem_alloc(DENSITY.size * DENSITY.dtype.itemsize)
81  U_C2_gpu      = cuda.mem_alloc(DENSITY.size * DENSITY.dtype.itemsize)
82  U_C4_gpu      = cuda.mem_alloc(DENSITY.size * DENSITY.dtype.itemsize)
83  U_C6_gpu      = cuda.mem_alloc(DENSITY.size * DENSITY.dtype.itemsize)
84  U_C8_gpu      = cuda.mem_alloc(DENSITY.size * DENSITY.dtype.itemsize)
85
86    Copy constants and variables to the gpu
87  cuda.memcpy_htod(F_gpu, F)
88  cuda.memcpy_htod(FEQ_gpu, FEQ)
89
90  cuda.memcpy_htod(BOUND_gpu, BOUND)
91  cuda.memcpy_htod(BOUNCEBACK_gpu, BOUNCEBACK)
92  cuda.memcpy_htod(DENSITY_gpu, DENSITY)
93  cuda.memcpy_htod(UX_gpu, UX)
94  cuda.memcpy_htod(UY_gpu, UY)
95
96  cuda.memcpy_htod(U_SQU_gpu, DENSITY)
97  cuda.memcpy_htod(U_C2_gpu, DENSITY)
98  cuda.memcpy_htod(U_C4_gpu, DENSITY)
99  cuda.memcpy_htod(U_C6_gpu, DENSITY)
100 cuda.memcpy_htod(U_C8_gpu, DENSITY)
101
102   Definition of kernels
103 propagateKernel = """
104     // F4  F3  F2
105     //   \ | /
106     // F5--F0--F1
107     //   / | \
108     // F6  F7  F8
109
110     __global__ void propagateKernel(float *F, float *T) {
111         int x     = threadIdx.x + blockIdx.x * blockDim.x;
112         int y     = threadIdx.y + blockIdx.y * blockDim.y;
113         int nx    = %(WIDTH)s;
114         int ny    = %(HEIGHT)s;
115         int size  = nx * ny;
116         int cur   = x + y * nx;        // current position
117
118         if(x < nx && y < ny) {
```

```
119             // nearest neighbours
120             int F1 = (x==0?nx-1:x-1) + y * nx; // +x
121             int F3 = x + (y==0?ny-1:y-1) * nx; // +y
122             int F5 = (x==nx-1?0:x+1) + y * nx; // -x
123             int F7 = x + (y==ny-1?0:y+1) * nx; // -y
124 
125             // next-nearest neighbours
126             int F2 = (x==0?nx-1:x-1) + (y==0?ny-1:y-1) * nx; //+x+y
127             int F4 = (x==nx-1?0:x+1) + (y==0?ny-1:y-1) * nx; //-x+y
128             int F6 = (x==nx-1?0:x+1) + (y==ny-1?0:y+1) * nx; //-x-y
129             int F8 = (x==0?nx-1:x-1) + (y==ny-1?0:y+1) * nx; //+x-y
130 
131             // propagate nearest
132             F[1*size + cur] = T[1*size + F1];
133             F[3*size + cur] = T[3*size + F3];
134             F[5*size + cur] = T[5*size + F5];
135             F[7*size + cur] = T[7*size + F7];
136 
137             // propagate next-nearest
138             F[2*size + cur] = T[2*size + F2];
139             F[4*size + cur] = T[4*size + F4];
140             F[6*size + cur] = T[6*size + F6];
141             F[8*size + cur] = T[8*size + F8];
142         }
143     }"""
144 propagateKernel = propagateKernel % {
145      WIDTH : nx,
146      HEIGHT : ny
147 }
148 
149 densityKernel = """
150     __global__ void densityKernel(float *F, float *BOUND, float * BOUNCEBACK,
151                                   float *D, float *UX, float *UY) {
152         int x     = threadIdx.x + blockIdx.x * blockDim.x;
153         int y     = threadIdx.y + blockIdx.y * blockDim.y;
154         int nx    = %(WIDTH)s;
155         int ny    = %(HEIGHT)s;
156         int size  = nx * ny;
157         int cur   = x + y * nx;
158 
159         if(x < nx && y < ny) {
160             if(BOUND[cur] == 1.0f) {
161                 BOUNCEBACK[1*size + cur] = F[5*size + cur];
162                 BOUNCEBACK[2*size + cur] = F[6*size + cur];
163                 BOUNCEBACK[3*size + cur] = F[7*size + cur];
164                 BOUNCEBACK[4*size + cur] = F[8*size + cur];
165                 BOUNCEBACK[5*size + cur] = F[1*size + cur];
166                 BOUNCEBACK[6*size + cur] = F[2*size + cur];
167                 BOUNCEBACK[7*size + cur] = F[3*size + cur];
168                 BOUNCEBACK[8*size + cur] = F[4*size + cur];
169             }
170 
171             float DENSITY = F[0*size + cur] +
172                             F[1*size + cur] +
173                             F[2*size + cur] +
174                             F[3*size + cur] +
175                             F[4*size + cur] +
176                             F[5*size + cur] +
177                             F[6*size + cur] +
178                             F[7*size + cur] +
179                             F[8*size + cur];
180 
```

```
181             D[cur] = DENSITY;
182
183             UX[cur] = ((F[1*size + cur] + F[2*size + cur] + F[8*size + cur]) -
184                        (F[4*size + cur] + F[5*size + cur] + F[6*size + cur]))
185                         / DENSITY;
186
187             UY[cur] = ((F[2*size + cur] + F[3*size + cur] + F[4*size + cur]) -
188                        (F[6*size + cur] + F[7*size + cur] + F[8*size + cur]))
189                         / DENSITY;
190
191             if (%(SCENERY)s == 3){
192                 // For lid driven cavity
193                 if(y == 0) {
194                     UX[cur] += 0.04f;
195                 }
196             } else {
197                 if(x == 0) {
198                     UX[cur] += %(DELTAU)sf;
199                 }
200             }
201
202             if(BOUND[cur] == 1.0f) {
203                 D[cur] = 0.0f;
204                 UX[cur] = 0.0f;
205                 UY[cur] = 0.0f;
206             }
207         }
208     }
209     """
210 densityKernel = densityKernel % {
211      WIDTH : nx,
212      HEIGHT : ny,
213      SCENERY : scenery,
214      DELTAU : deltaU
215 }
216
217 eqKernel = """
218     __global__ void eqKernel(float *F, float* FEQ, float *DENSITY, float *UX,
219                              float *UY, float *U_SQU, float *U_C2, float *U_C4,
220                              float *U_C6, float *U_C8) {
221         int x     = threadIdx.x + blockIdx.x * blockDim.x;
222         int y     = threadIdx.y + blockIdx.y * blockDim.y;
223         int nx    = %(WIDTH)s;
224         int ny    = %(HEIGHT)s;
225         int size  = nx * ny;
226         int cur   = x + y * nx;
227
228         if(x < nx && y < ny) {
229             // constants
230             float t1 = %(T1)s;
231             float t2 = %(T2)s;
232             float t3 = %(T3)s;
233             float c_squ = %(C_SQU)s;
234             float omega = %(OMEGA)s;
235
236             if(x < %(WIDTH)s || y < %(HEIGHT)s) {
237                 U_SQU[cur] = UX[cur]*UX[cur] + UY[cur]*UY[cur];
238                 U_C2[cur]  = UX[cur]+UY[cur];
239                 U_C4[cur]  = -UX[cur]+UY[cur];
240                 U_C6[cur]  = -U_C2[cur];
241                 U_C8[cur]  = -U_C4[cur];
242
```

```
243                 // Calculate equilibrium distribution: stationary
244                 FEQ[0*size + cur] = t1*DENSITY[cur]*(1-U_SQU[cur]/(2*c_squ));
245
246                 // nearest-neighbours
247                 FEQ[1*size + cur] = t2*DENSITY[cur]*(1+UX[cur]/c_squ+0.5f*
248                         (UX[cur]/c_squ)*(UX[cur]/c_squ)-U_SQU[cur]/(2*c_squ));
249                 FEQ[3*size + cur] = t2*DENSITY[cur]*(1+UY[cur]/c_squ+0.5f*
250                         (UY[cur]/c_squ)*(UY[cur]/c_squ)-U_SQU[cur]/(2*c_squ));
251                 FEQ[5*size + cur] = t2*DENSITY[cur]*(1-UX[cur]/c_squ+0.5f*
252                         (UX[cur]/c_squ)*(UX[cur]/c_squ)-U_SQU[cur]/(2*c_squ));
253                 FEQ[7*size + cur] = t2*DENSITY[cur]*(1-UY[cur]/c_squ+0.5f*
254                         (UY[cur]/c_squ)*(UY[cur]/c_squ)-U_SQU[cur]/(2*c_squ));
255
256                 // next-nearest neighbours
257                 FEQ[2*size + cur] = t3*DENSITY[cur]*(1+U_C2[cur]/c_squ+0.5f*
258                       (U_C2[cur]/c_squ)*(U_C2[cur]/c_squ)-U_SQU[cur]/(2*c_squ));
259                 FEQ[4*size + cur] = t3*DENSITY[cur]*(1+U_C4[cur]/c_squ+0.5f*
260                       (U_C4[cur]/c_squ)*(U_C4[cur]/c_squ)-U_SQU[cur]/(2*c_squ));
261                 FEQ[6*size + cur] = t3*DENSITY[cur]*(1+U_C6[cur]/c_squ+0.5f*
262                       (U_C6[cur]/c_squ)*(U_C6[cur]/c_squ)-U_SQU[cur]/(2*c_squ));
263                 FEQ[8*size + cur] = t3*DENSITY[cur]*(1+U_C8[cur]/c_squ+0.5f*
264                       (U_C8[cur]/c_squ)*(U_C8[cur]/c_squ)-U_SQU[cur]/(2*c_squ));
265
266                 F[0*size + cur] = omega*FEQ[0*size + cur] +
267                                                   (1-omega)*F[0*size + cur];
268                 F[1*size + cur] = omega*FEQ[1*size + cur] +
269                                                   (1-omega)*F[1*size + cur];
270                 F[2*size + cur] = omega*FEQ[2*size + cur] +
271                                                   (1-omega)*F[2*size + cur];
272                 F[3*size + cur] = omega*FEQ[3*size + cur] +
273                                                   (1-omega)*F[3*size + cur];
274                 F[4*size + cur] = omega*FEQ[4*size + cur] +
275                                                   (1-omega)*F[4*size + cur];
276                 F[5*size + cur] = omega*FEQ[5*size + cur] +
277                                                   (1-omega)*F[5*size + cur];
278                 F[6*size + cur] = omega*FEQ[6*size + cur] +
279                                                   (1-omega)*F[6*size + cur];
280                 F[7*size + cur] = omega*FEQ[7*size + cur] +
281                                                   (1-omega)*F[7*size + cur];
282                 F[8*size + cur] = omega*FEQ[8*size + cur] +
283                                                   (1-omega)*F[8*size + cur];
284             }
285         }
286     }
287     """
288 eqKernel = eqKernel % {
289     WIDTH : nx,
290     HEIGHT : ny,
291     T1 : t1,
292     T2 : t2,
293     T3 : t3,
294     C_SQU : c_squ,
295     OMEGA : omega
296 }
297
298 bouncebackKernel = """
299     __global__ void bouncebackKernel(float *F, float *BOUNCEBACK,
300                                      float *BOUND) {
301         int x     = threadIdx.x + blockIdx.x * blockDim.x;
302         int y     = threadIdx.y + blockIdx.y * blockDim.y;
303         int nx    = %(WIDTH)s;
304         int ny    = %(HEIGHT)s;
```

```
305             int size  = nx * ny;
306             int cur   = x + y * nx;
307 
308             if(x < nx && y < ny) {
309                 if (BOUND[cur] == 1.0f) {
310                     F[1*size + cur] = BOUNCEBACK[1*size + cur];
311                     F[2*size + cur] = BOUNCEBACK[2*size + cur];
312                     F[3*size + cur] = BOUNCEBACK[3*size + cur];
313                     F[4*size + cur] = BOUNCEBACK[4*size + cur];
314                     F[5*size + cur] = BOUNCEBACK[5*size + cur];
315                     F[6*size + cur] = BOUNCEBACK[6*size + cur];
316                     F[7*size + cur] = BOUNCEBACK[7*size + cur];
317                     F[8*size + cur] = BOUNCEBACK[8*size + cur];
318                 }
319             }
320         }
321         """
322     bouncebackKernel = bouncebackKernel % {
323          WIDTH : nx,
324          HEIGHT : ny
325     }
326 
327       Get kernel handles
328     mod          = SourceModule(propagateKernel + densityKernel +
329                                 eqKernel + bouncebackKernel)
330     prop         = mod.get_function("propagateKernel")
331     density      = mod.get_function("densityKernel")
332     eq           = mod.get_function("eqKernel")
333     bounceback   = mod.get_function("bouncebackKernel")
334 
335     def loop(iterations):
336         ts = 0
337         while(ts<iterations):
338               To avoid overwrites a temporary copy is made of F
339             T[:] = F
340             cuda.memcpy_htod(T_gpu, T)
341 
342               Propagate
343             prop(F_gpu, T_gpu,
344                  block=(blockDimX,blockDimY,1), grid=(gridDimX,gridDimY))
345 
346               Calculate density and get bounceback from obstacle nodes
347             density(F_gpu, BOUND_gpu, BOUNCEBACK_gpu, DENSITY_gpu, UX_gpu, UY_gpu,
348                     block=(blockDimX,blockDimY,1), grid=(gridDimX,gridDimY))
349 
350               Calculate equilibrium
351             eq(F_gpu, FEQ_gpu, DENSITY_gpu, UX_gpu, UY_gpu, U_SQU_gpu, U_C2_gpu,
352                U_C4_gpu, U_C6_gpu, U_C8_gpu, block=(blockDimX,blockDimY,1),
353                grid=(gridDimX,gridDimY))
354 
355               Transfer bounceback to obstacle nodes
356             bounceback(F_gpu, BOUNCEBACK_gpu, BOUND_gpu,
357                        block=(blockDimX,blockDimY,1), grid=(gridDimX,gridDimY))
358 
359               Copy F to host for copy to T in beginning of loop
360             cuda.memcpy_dtoh(F, F_gpu)
361 
362             ts += 1
363 
364       Run the loop
365     loop(it)
366 
```

```
367     Copy UX and UY back to host
368   cuda.memcpy_dtoh(UX, UX_gpu)
369   cuda.memcpy_dtoh(UY, UY_gpu)
370
371     Plot
372   import matplotlib.pyplot as plt
373   UY *= -1
374   plt.hold(True)
375   plt.xlabel( x )
376   plt.ylabel( y )
377   plt.title( Flow field after %sdt  % it)
378   plt.quiver(UX,UY, pivot= middle , color= blue )
379   plt.imshow(BOUND, interpolation= nearest , cmap= gist_yarg )
380  #plt.imshow(np.sqrt(UX*UX+UY*UY)) # fancy rainbow plot
381   plt.show()
```

8.4 Results

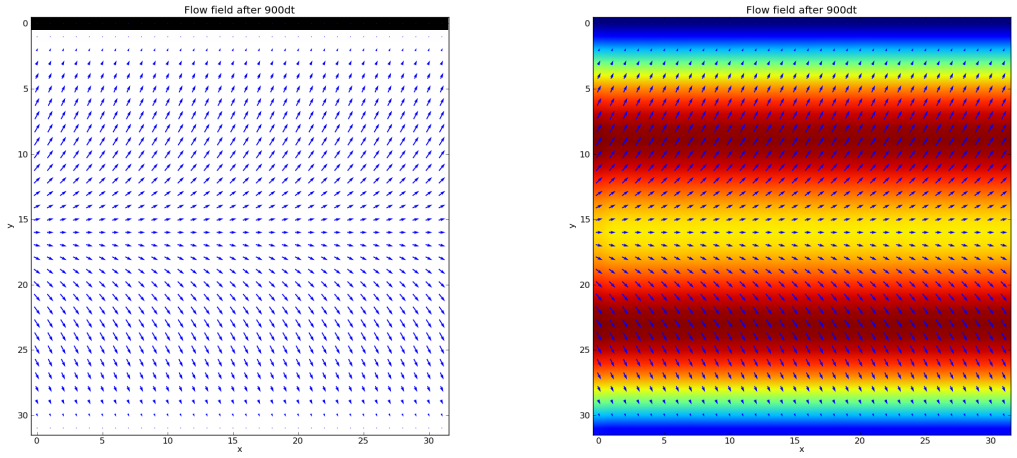
Size of field	1st run(s)	2nd run(s)	3rd run(s)	Average(s)
16	0.746133	0.734992	0.734949	0.738691333
32	1.246536	1.233322	1.240871	1.240243
48	2.071759	2.061449	2.050736	2.061314667
64	3.16993	3.206161	3.247185	3.207758667
80	4.73842	4.636399	4.638702	4.671173667
96	6.677765	6.670117	6.651408	6.66643
112	8.934494	8.932387	8.935797	8.934226
128	13.595282	13.008162	13.004377	13.202607
144	16.673446	16.704629	16.703753	16.69394267
160	21.004205	21.109045	21.096003	21.069751
176	25.085424	25.239326	25.256939	25.19389633
192	30.056787	30.267888	30.364107	30.229594
208	35.587349	35.763984	35.783916	35.71174967

Table 5: Results from running the CPU-version of D2Q9 LBM, after 1000 iterations in an open channel scenery (Figure 17). The size of the simulated field is the independent variable, and the running time is the dependent variable.

Size of field	1st run(s)	2nd run(s)	3rd run(s)	Average(s)
16	0.241659	0.23915	0.205136	0.228648333
32	0.228573	0.285473	0.234809	0.249618333
48	0.351229	0.346322	0.378153	0.358568
64	0.478636	0.479583	0.473222	0.477147
80	0.647089	0.643426	0.67618	0.655565
96	0.858993	0.879635	0.880193	0.872940333
112	1.110316	1.106407	1.105429	1.107384
128	1.410813	1.423374	1.387333	1.407173333
144	1.696993	1.699721	1.692255	1.696323
160	2.097632	2.072039	2.058629	2.0761
176	2.503845	2.421257	2.425955	2.450352333
192	2.779791	2.788359	2.759723	2.775957667
208	3.440834	3.096601	3.096115	3.211183333
224	3.668249	3.554599	3.534036	3.585628
240	3.987397	3.989254	3.929123	3.968591333
256	4.670454	4.484278	4.487079	4.547270333
272	5.450173	5.123512	5.125534	5.233073
288	5.666375	5.557551	5.555653	5.593193
304	6.105622	6.097763	6.093172	6.098852333
320	7.188123	6.79385	6.772588	6.918187
336	7.328069	7.337078	7.309121	7.324756
352	8.69325	7.965098	7.955409	8.204585667
368	8.846718	8.673317	8.643207	8.721080667
384	9.700574	9.562544	9.480546	9.581221333
400	10.36871	9.965424	9.997083	10.11040567
416	10.780117	10.691381	10.677279	10.716259
432	11.450054	11.370197	11.378999	11.39975
448	12.519018	12.172332	12.169165	12.28683833
464	13.893501	13.025537	13.011743	13.31026033
480	14.708758	13.912262	13.918446	14.179822
496	15.270144	14.979782	14.942925	15.06428367
512	16.212768	15.585513	15.59956	15.79928033

Table 6: Results from running the GPU-version of D2Q9 LBM, after 1000 iterations in an open channel scenery (Figure 17). The size of the simulated field is the independent variable, and the running time is the dependent variable.

8.5 Images



(a) Open channel simulation calculated by PyCUDA

(b) The same simulation as in Figure 17(a) but with color gradient

Figure 17: Open channel simulation, the simplest simulation used for verifying correctness of the simulation, as well as benchmarking the CPU- and GPU-version

9 Bibliography

To ensure access to the various on-line documents we have used as references, we have created a page with PDF files produced from the documents that were available when we published our thesis: http://www.snej.dk/lbm/.

References

[1] *NVIDIA CUDA Best Practices Guide 3.2*. nVidia Corporation, 2010.

[2] *NVIDIA CUDA C Programming Guide 3.2*. nVidia Corporation, 2010.

[3] Rob Farber. Dr.dobb's: Cuda, supercomputing for the masses: Part 11. http://drdobbs.com/high-performance-computing/215900921, 2009.

[4] Iain Haslam. Lattice boltzmann matlab scripts. http://www.exolete.com/code/lbm, 2006.

[5] Iain Haslam, Mads Ruben Burgdorff Kristensen, and Brian Vinter. Distnumpy d3q19 lattice boltzmann implementation. http://code.google.com/p/distnumpy/source/browse/trunk/benchmarks/lbm3d.py, 2011.

[6] David Kanter. Nvidia's gt200: Inside a parallel processor. http://www.realworldtech.com/page.cfm?ArticleID=RWT090808195242&p=3.

[7] Andreas Klöckner. Pycuda documentation. http://documen.tician.de/pycuda/.

[8] Andreas Klöckner, Nicolas Pinto, Yunsup Lee, B. Catanzaro, Paul Ivanov, and Ahmed Fasih. PyCUDA and PyOpenCL: A Scripting-Based Approach to GPU Run-Time Code Generation. Technical Report 2009-40, Scientific Computing Group, Brown University, Providence, RI, USA, November 2009.

[9] Mads Ruben Burgdorff Kristensen and Brian Vinter. Numerical python for scalable architectures. Technical report, eScience Centre, University of Copenhagen, Copenhagen, DK.

[10] CFD Online. Lid-driven cavity problem. http://www.cfd-online.com/Wiki/Lid-driven_cavity_problem.

[11] The open source CFD toolbox. Lid-driven cavity flow. http://www.openfoam.com/docs/user/cavity.php.

[12] Sauro Prof. Succi, Mauro Dr. Sbragaglia, and Dr. Ubertini Stefano. Scholarpedia: Lattice boltzmann method. http://www.scholarpedia.org/article/Lattice_Boltzmann_Method, 2010.

[13] Tom R. Halfhill. Parallel processing with cuda. *Microprocessor Report*, 2008.

[14] Jason Sanders and Edward Kandrot. *CUDA By Example*. Addison-Wesley, 2010.