GPU Parallelization of the Lattice Boltzmann CFD Method

Timothy James Guite

19/05/2017

Supervisor: Dr Ralf Detierding

Word Count: 9988

This report is submitted in partial fulfillment of the requirements for the MEng Mechanical Engineering, Faculty of Engineering and the Environment, University of Southampton

Declaration

I, Timothy James Guite, declare that this thesis and the work presented in it are my own and has been generated by me as the result of my own original research.

I confirm that:

1. This work was done wholly or mainly while in candidature for a degree at this University;

2. Where any part of this thesis has previously been submitted for any other qualification at this University or any other institution, this has been clearly stated;

3. Where I have consulted the published work of others, this is always clearly attributed;

4. Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work;

5. I have acknowledged all main sources of help;

6. Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself;

7. None of this work has been published before submission

Acknowledgements

I would like to acknowledge the following individuals for their help:

Dr Ralf Deiterding, who provided a wealth of knowledge, without which this project would not have been possible. In addition to his direct support as supervisor, he provided equipment for development and benchmarking.

Satya Jammy, who aided my understanding with some difficult concepts and helped me to create a plan for this project.

Dr Ati Sharma, who allowed me to access benchmarking equipment.

Professor Andrew Cruden, for providing support throughout the project.

Contents

Acronyms

LBM	Lattice Boltzmann Method
CPU	Central Processing Unit
GPU	Graphical Processing Unit
MFLUPS	Million Fluid Lattice Updates Per Second
Bhatnagar-Gross-Krook	BGK
Re	Reynolds Number
SGS	Sub-Grid Scale
TheLMA	Thermal LBM on Manny-core Architectures
P2P	Peer-to-Peer
FLOPS	Floating-point Operations Per Second

Abstract

In this report, the Lattice Boltzmann Method (LBM) is parallelized on a CUDA-capable Graphical Processing Unit (GPU). Validity of the implementation is assured by comparison with literature for a canonical problem. Performance benchmarking is performed and a speed up of 52.9X is achieved compared to a single-threaded Central Processing Unit (CPU) implementation. Peak performance for the GPU implementation is achieved by utilising two GPUs and is measured at 1752.8 Million Fluid Lattice Updates Per Second. The CPU and GPU implementations are described in detail and proposals for future works are given.

1. Introduction

The Lattice Boltzmann Method is a CFD method which offers an efficient alternative to traditional Navier-Stokes solvers (Valero-Lara & Jannson, 2011). It has proven successful in a wide array of applications and shows significant speed gains compared to other techniques; the ease of parallelizing LBM has allowed engineers to take advantage of the increasing power of GPUs. This work aims to implement LBM on GPU using CUDA-enabled devices. First, LBM will be implemented on CPU and applied to the canonical problem of lid-driven cavity flow. Next, a GPU implementation will be created, using the CUDA platform. Aspects of GPUs will be investigated for their performance benefits. Through the use of thousands of cores available on a GPU, significant performance increases compared to the CPU implementation are expected. Chapter 2 describes LBM and CUDA in detail, describes the lid-driven cavity problem, describes hardware, and summarises background literature. Chapter 3 describes the implementation of LBM on CPU and GPU. This work will provide the basis for integrating GPUs with Dr. Detierding's AMROC software system. The aim is to identify which aspects of GPU computation have large impacts on performance.

2. Methods

2.1 Lattice Boltzmann Method

2.1.1 Derivation of Lattice Boltzmann Equation

The LBM is based on the Boltzmann-Maxwell equation:

$$\left(\frac{\partial}{\partial t} + \boldsymbol{e} \cdot \nabla_r + \boldsymbol{a} \cdot \boldsymbol{\nabla_e}\right) f(\boldsymbol{e}, \boldsymbol{r}, t) = J \qquad (1)$$

where $f(\boldsymbol{e}, \boldsymbol{r}, t)$ is the single particle distribution function, $\boldsymbol{e}$ is the velocity of the particle, $\boldsymbol{r}$ is the position of the particle, t is the time and $\boldsymbol{a}$ is the acceleration of the particle. J represents the effects of collisions between particles (Aidun & Clausen, 2010). The Boltzmann-Maxwell equation employs a finite discrete velocity model, and describes particles with velocities that belong to a discrete set E = {$\mathbf{e}_0$ .. $\mathbf{e}_Q$} of Q velocities. If we write the probability distribution function as $f_i(\boldsymbol{r}, t) \equiv f(\boldsymbol{r}, t, \boldsymbol{e}_i)$ then the conservation equation is given by:

$$d_t f_i(r, t) + e_i \cdot \nabla f_i(r, t) = J_i(f), \qquad i = 0, \dots, Q \qquad (2)$$

Here, $f_i(r, t)$ describes the probability of finding a particle with velocity $\boldsymbol{e}$ at position $\boldsymbol{r}$ at time t. Particle interactions are described with the collision operator $J_i(f)$. By rescaling $\boldsymbol{r}$ and t such that $\partial_t = \partial_r = 1$, and projecting the conservation equation onto special lattice X, the general Lattice Boltzmann equation can be written as (McNamara & Zanetti, 1988):

$$f_i(\boldsymbol{r} + e_i, t + 1) - f_i(\boldsymbol{r}, t) \approx J_i(f)(\boldsymbol{x}, t) \qquad (3)$$

This is normally referred to as the non-linear Lattice Boltzmann Equation. It allows for collisions between multiple particles and can therefore be applied to dense fluids (Aidun & Clausen 2010).

2.1.2 Single relaxation time LBM

The non-linear LBM is often simplified with the Bhatnagar-Gross-Krook (BGK) collision operator which defines a single relaxation time, τ, and equilibrium function, f_i^{eq}:

$$f_i(\boldsymbol{r} + e_i, t + 1) - f_i(\boldsymbol{r}, t) = -\frac{1}{\tau}\left(f_i(\boldsymbol{r}, t) - f_i^{eq}(\boldsymbol{r}, t)\right) \qquad (4)$$

This model is known as the LBGK. In this model, particles have velocities along predefined directions. Figure 1 shows two of the most common models, D2Q9 and D3Q19 (In this naming scheme, D stands for the number of dimensions and Q is the number of directions per lattice cell).

Implementation	i	w_i
D2Q9	0	$4/9$
D2Q9	1, 2, 3, 6	$1/9$
D2Q9	4, 5, 7, 8	$1/36$
D3Q19	0	$1/3$
D3Q19	1..6	$1/18$
D3Q19	7..18	$1/36$

Table 1. Weighting coefficients for two implementations of LBM

The equilibrium function is calculated by:

$$f_i^{eq}(\boldsymbol{r},t) = \rho w_i \left[1 + \frac{\boldsymbol{e}_i \cdot \boldsymbol{u}}{c_s^2} + \frac{(\boldsymbol{e}_i \cdot \boldsymbol{u})^2}{2c_s^4} - \frac{\boldsymbol{u}^2}{2c_s^2}\right] \qquad (5)$$

where w_i is a weighting coefficient which depends on which direction, i, is being described. Table 1 describes w_i for the D2Q9 and D3Q19 models (Aidun & Clausen, 2010). In the BGK model, $c_s = 1/\sqrt{3}$, which is the speed of sound in the lattice. The LBGK can be expanded with the Chapman-Enskog procedure (Hou, S., Sterling, J., Chen, S., Doolen, G.D, 1996) which shows that it converges to a weakly compressible solution of the Navier-Stokes equation:

$$d_t\rho + \nabla \cdot (\rho \boldsymbol{u}) = 0, \qquad (6a)$$

$$d_t\boldsymbol{u} + \boldsymbol{u} \cdot \nabla\boldsymbol{u} = -\nabla p + v\nabla 2\boldsymbol{u} \qquad (6b)$$

It can also be demonstrated that the relaxation time, τ, collision frequency, ω, and viscosity, v, are linked (Hähnel, 2004):

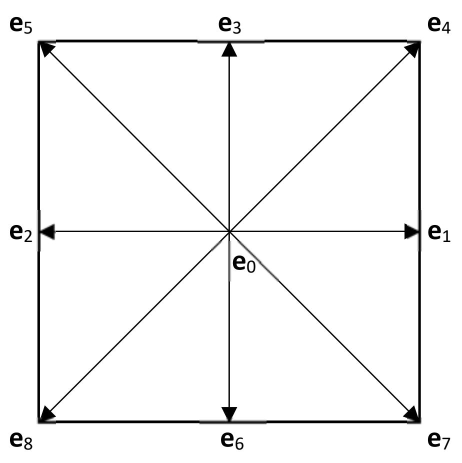
$$\tau^{-1} = \omega = \frac{c_s^2 \Delta t}{v + \Delta t c_s^2/2} \qquad (7)$$

Eq. (4) is broken down into two steps for computing LBGK: collision and stream. During the collision step, the equilibrium value of each cell is calculated and used to find the new values of the cell:

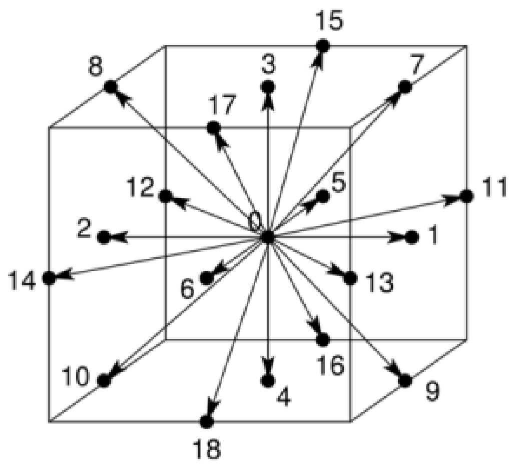
$$f_i^*(\boldsymbol{r}, t + 1) = (1 - \omega)\big(f_i(\boldsymbol{r},t)\big) + \omega\left(f_i^{eq}(\boldsymbol{r},t)\right) \qquad (8)$$

The individual velocity values are then streamed to the correct neighbouring cell based on the direction of their velocity:

$$f_i(\boldsymbol{r} + e_i, t + 1) = f_i^*(\boldsymbol{r}, t + 1) \qquad (9)$$



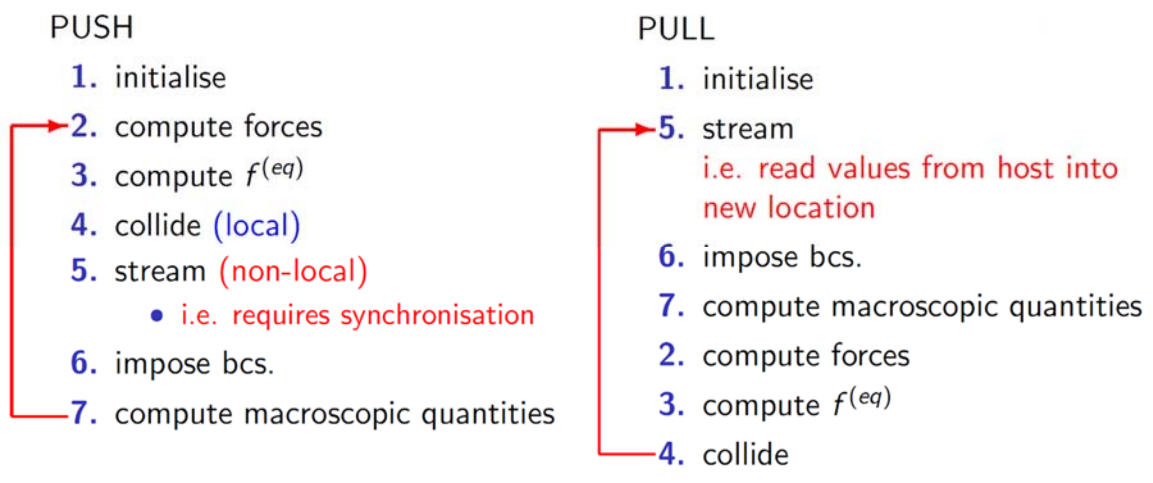
a) D2Q9



b) D3Q19 (Developers Club, 2013)

Figure 1. Discrete velocity directions for two implementations of LBM

While the collision step is a closed computation in which each cell of the lattice operates independently, the stream step requires communication between cells. This work uses the stencil approach whereby each cell has all of its velocity values streamed at the same time. This can be achieved using either the pull or push approach which are described in Figure X. These different approaches to the stream step also have an effect on when the boundary conditions are applied to the grid. By acting on each cell individually at each time step, it is clear that LBM is easily adaptable for parallelization. The only limiting factor is that the grid needs to be synchronised after each time step.


PUSH
1. initialise
2. compute forces
3. compute $f^{(eq)}$
4. collide (local)
5. stream (non-local)
    - i.e. requires synchronisation
6. impose bcs.
7. compute macroscopic quantities

PULL
1. initialise
5. stream
   i.e. read values from host into new location
6. impose bcs.
7. compute macroscopic quantities
2. compute forces
3. compute $f^{(eq)}$
4. collide

Figure 1. Push vs pull program layout (Revell, 2013)

2.1.3 Applications of LBM

As discussed above, the LBM has shown good potential as an alternative to traditional CFD methods. It is necessary to continue to test LBM in new conditions to see how it performs compared with other methods and how it can be altered to tackle difficult problems.

Perumal & Dass (2011) investigated the possibility of retrieving multiple solutions with LBM. Their test case was a square cavity with two or four driven sides. At high Re, these problems produce multiple steady state solutions. They took advantage of the fact that in LBM, by altering lid velocity, the fluid viscosity can be altered while keeping Re constant. Their results showed good agreement with results from previous studies which used the continuum approach.

Aidun & Clausen (2010) highlight other areas of success for LBM. One issue with LBM is that under high flow velocities, the distribution function becomes negative which causes numerical instabilities. Attempts to combine LBM with the H-theorem in the Entropic LBM have improved stability at high Re, although further work needs to be done.

Boundary conditions that are periodic yet impose shear have been developed. Fluids or particles which cross the top shear border are translated in the direction of shear flow and their velocity is affected by the shear flow, before they reappear at the bottom of the lattice. The effect of this is that wall conditions are removed and bulk properties of the fluid can be determined.

LBM has been used to model turbulent flows at high Re with the Entropic LBM and through combination with Sub-Grid Scale (SGS) modelling. SGS-LBM uses the Smagorinsky coefficient to alter the viscosity in areas identified as large eddies and shows good agreement with DNS-LB and DNS-NS (Dong & Sagaut, 2008).

Interesting work has been conducted on multi-phase flow with LBM. Ginzburg (2007) derived a two-relaxation-time collision operator to investigate the interface between two phases in Pouseuille flow, which has proven effective when the viscosity ratio between the phases is large (Aidun & Clausen, 2010).

A combination of approaches to modelling suspended particles in the fluid led Dupin, et al. (2008) to model the flow of Red Blood Cells (RBCs) through capillaries. RBCs are deformable particles so the LBM was combined with Lagrangian membranes for the RBCs. Assuming constant volume for each RBC, each face of the cell acts as a stiff spring. By altering the shape and rigidity of RBCs, several different blood flow phenomena were modelled in agreement with experimental data.

Deiterding & Wood (2016) applied LBM to the optimisation of wind turbine positioning, given the wake produced by each wind turbine. Their approach improved on traditional simulations by modelling a rotating turbine. They combined LBM with the Smagorinsky model, dynamic mesh adaptation and embedded structure handling. The non-Cartesian geometry of the turbine blades is modelled by adjusting the density distribution of embedded boundary cells. The geometries are calculated in a small band near the structure, to keep computations low even at increasing mesh sizes. Their model corroborates experiment results for a canonical problem described by Toomey & Eldredge (2008). It was then applied to the setup found at the U.S. Department of Energy's Scaled Wind Farm Technology (SWIFT) facility with realistic models of the Vestas V27 wind turbine. Different wind speeds and turbine RPMs were tested and results showed large differences in the wake created behind one turbine and two. The simulations were carried out on a moderately sized CPU cluster and showed good performance.

Fragner & Deiterding (2016) demonstrated that LBM can provide excellent agreement with experimental results, at significantly reduced compute times, compared to pressure-convection based solvers. They were investigating the effect of strong cross winds on high-speed trains. Again, the Smagorinsky model and dynamic mesh adaptation were employed. This allowed the mesh to be refined at points of interest, usually near the boundary, and larger elsewhere. They found this LBM application was around 16 times faster than an OpenFOAM solver on a mesh with similar cell count.

2.2 Parallel Computing

2.2.1 CUDA

CUDA is a platform and programming model for parallel computing created by NVIDIA for use on their GPUs. NVIDIA built CUDA from the ground up to provide IEEE compliance for single precision floating point calculations, and have since extended the capabilities to include double precision.

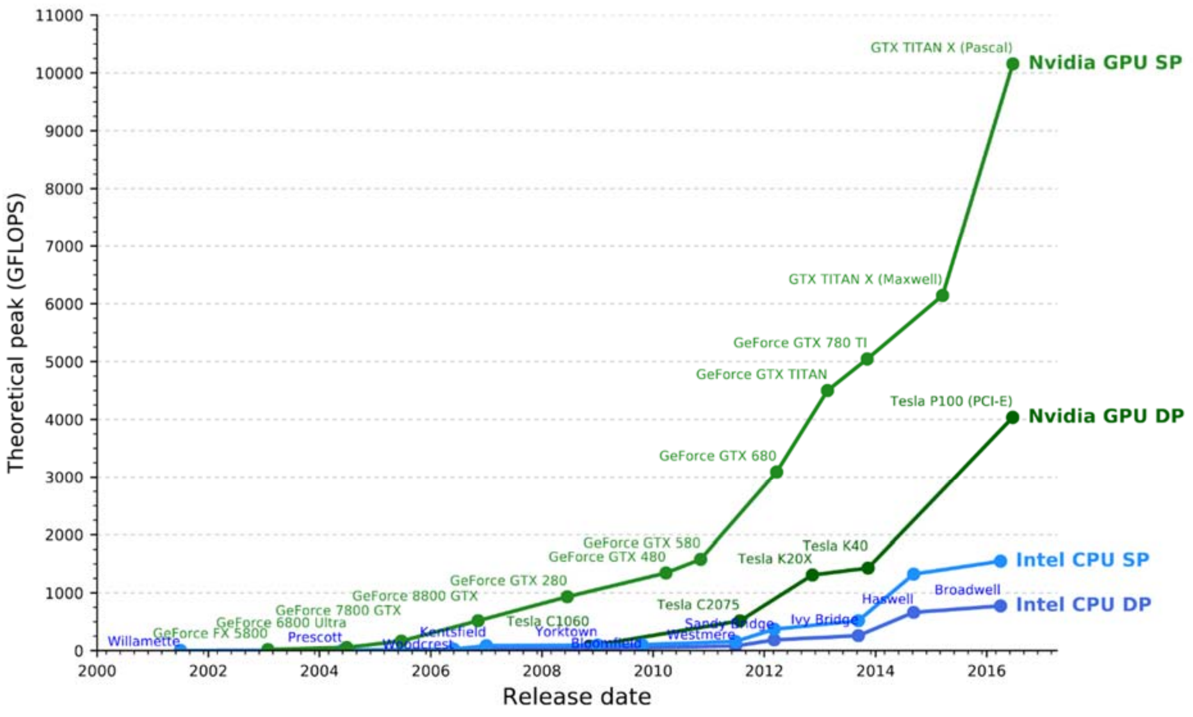


Figure 3. Comparison of GPU vs CPU FLOPS since 2001 (Galloy, 2017)

In 2006, NVIDIA released the GeForce 8800 GTX, which was the first GPU to include the CUDA architecture (Sanders et al, 2015). It became significantly easier to conduct scientific research with GPUs. This in turn led to further demand for GPUs within the scientific community and a desire for them to increase in speed along with the gains that have been made in CPU performance over the past few decades. Figure 3 shows the maximum theoretical performance of CPUs and GPUs, since 2000, in GFLOPS. It is plain to see that GPUs have made significant gains in the past decade and are now much faster than CPUs for floating point operations. Recently, efforts have been made to combine multiple GPUs within one system to further increase computational speed and problem size.

CUDA acts as an extension to the C++ language. It performs operations on the CPU, transfers data sets to the GPU to run calculations, and returns the output to the CPU. CUDA capable GPUs consist of streaming multiprocessors (SMs) each containing scalar processors (SPs) (Obrecht et al, 2013). Each SM has control units, registers, execution pipelines and caches (Woolley, 2011). One SM contains many cores, each of which can run multiple threads concurrently. Within the CUDA terminology: *host* is the CPU; *device* is the GPU; *kernels* are special functions that run on the GPU.

Blocks and threads are important concepts in CUDA. Threads are the simplest unit in the CUDA programming architecture. Threads are collected into blocks, and blocks are collected into a grid. Currently, the number of blocks is limited to 65,535 but the number of threads per block changes depending on the GPU – our maximum is 1024. Initially, it is necessary only to think of blocks and threads as having one dimension each. Each thread performs a set of instructions alongside other threads in its block and is executed by a core. Blocks execute asynchronously and often in an unpredictable order. A block executes on one SM, but one SM can execute several blocks at once. NVIDIA recommends that kernels are written such that blocks can execute completely independently. Threads within a block can be synchronised during kernel execution and can communicate with one another, but not with threads in other blocks. The architecture is known as *Single Instruction, Multiple Thread* (SIMT) and creates, manages, schedules and executes threads in groups of 32. These groups of 32 threads are called *warps* (NVIDIA, 2017). While threads within a warp are independent and can branch from one another, warps execute common instructions one at a time. Divergent threads within a warp cause the other threads to wait. The most efficient use of warps is to have all threads within a warp agree on the execution path. Warps within a block are independent from one another. The number of threads in a block should also be a multiple of the warp size. Some CUDA functions also affect *half-warps* – groups of 16 threads.

A kernel run in CUDA looks like this:

```
kernel<<<blocks, threads>>>(*data);
```

Code 1. Example of Kernel run in CUDA

Other than the triple angle operators ("<<<" and ">>>") a kernel is called like a normal function. The triple angle operators are simply a tool to define the number of blocks and threads for the kernel to run. The kernel closes once the last block has executed and through this mechanism, the blocks become synchronised. Therefore, applications that require block synchronisation are made up of many kernel calls. In order to separate device functions from host functions, CUDA employs several keywords which are employed in front of normal device declarations (Code 2). Kernels are qualified with *__global__*, they must return void; device functions, to be called by the kernel, are qualified with *__device__*; host functions can be qualified with *__host__* although, as this is the default, this is only useful for functions that will be called by both the device and host –

such functions are qualified with __*host*__ __*device*__. Kernels are the only type of function that requires triple angle operators.

```
__global__  kernel(int  *data);
__device__  dev_function(int  *f);
__host__  host_function(int  *x);
__host__  __device__  host_dev_function(int  *z);
```

Code 2. Example of CUDA function declarations

2.2.2 CUDA – Memory types

Global memory is memory which can be accessed by any thread on the GPU at any time. This is the simplest way to store data on the GPU, however, data access is very slow compared to other memory options on the GPU – hundreds of gigabytes per second – as it is stored in off-disk DRAM (Sanders, Kandrot & Dongarra, 2015). Storage size is large, 4 GB or 8 GB is common, so often this is how large datasets are stored on the GPU. Global memory must be allocated on the GPU before the kernel is called. A pointer to the memory is then passed to the kernel as a parameter. To transfer data from the host to the device, the function *cudaMemcpy* must be used. Code 3 shows two variables being created, one for the host and one for the device. The host data is then copied to the device and exists in global memory. To retrieve data from the device, the device and host variables change parameter positions and the final parameter changes from *cudaMemcpyHostToDevice* to *cudaMemcpyDeviceToHost*.

```
int  *host_var,  *device_var;

host_var  =  [1,  2,  3,  4,  5];

cudaMalloc((void**)&device_var,  5*sizeof(int));

cudaMemcpy(device_var,  host_var,  5*sizeof(int),  cudaMemcpyHostToDevice);
```

Code 3. Memory created on host and copied to global memory on device

Texture memory is a specialised read-only form of memory which was developed for use in graphical applications. It acts in a similar way to global memory but has much higher bandwidth in applications which exhibit spatial locality when accessing memory. Texture memory is cached on chip and is designed to accelerate access patterns which are not necessarily in sequence but are close together. A good example of this access pattern is shown in **Error! Reference source not found.**, where threads are accessing memory which is spatially local in a 2D array. Texture memory can be arranged in one or two dimensions, but does present some challenges. Memory is allocated by binding global memory to a texture, and it can then only be accessed by special functions. As textures are declared globally at file scope, they cannot be passed to functions as parameters.

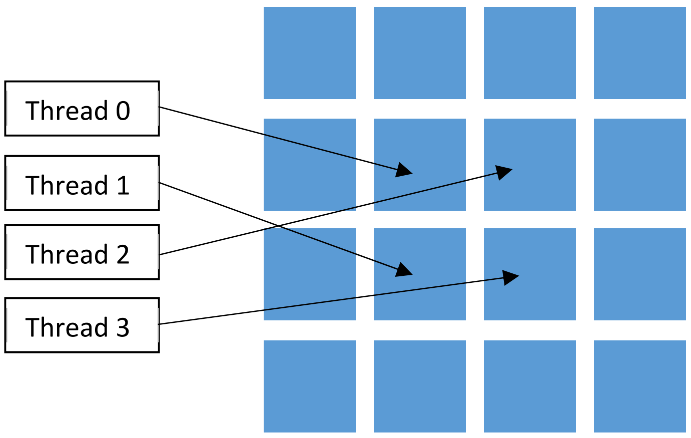


Figure 4. Mapping of threads onto 2D memory

Memory access which does not take advantage of spatial locality has similar access speeds to global memory. As a result, texture memory only provides an advantage in specialised applications.

Constant memory provides up to 64 KB of cached, read-only memory, the size of which must be defined at compile time. The advantages of constant memory are: one read can provide data to up to 16 threads (a half warp), dramatically decreasing the required bandwidth; as constant memory is cached, consecutively reading from the same address does not add to memory traffic. However, when threads in a half warp request data from different locations in constant memory, access speeds will be much lower as each half warp can only place one constant memory request at a time.

Shared memory offers many advantages. It resides physically on the GPU and has access speeds approximately 100x faster than global memory (Harris, 2014). A copy of shared memory variables is created for each block. The size limit for shared memory per block is dependent on the GPU but is generally around 48 KB. Shared memory variables can be accessed by all threads within a block, which offers a mechanism for threads to communicate with one another. Threads cannot access shared memory from other blocks. In order to control all the threads which are accessing the same shared memory, CUDA also introduces a mechanism for synchronising threads (Code 4). This ensures that all threads have completed their read or write operations before continuing. Shared

```
//Create shared memory array
__shared__ int shared_data[threadsPerBlock];
//fill shared_data with data

//Read from shared data to local variable
int data = shared_data[threadNumber];

//Synchronise threads
__syncThreads();
```

Code 4. Example of shared memory read and thread synchronisation memory allocations are destroyed when the block finishes executing which means they cannot be used to store data between kernel calls. Although this is a limit, the speed increases from shared memory are such that it is often quicker to copy a block of memory from global memory to shared memory, operate on it within the kernel and then copy it back to global memory.

Register memory is the fastest type of memory available in the CUDA architecture. It is for variables local to the thread and is created automatically when variables are declared, without a qualifier, in the kernel. There is a very limited amount of register space, so it is reserved for variables which only require a small amount of memory. For variables that require a larger amount of memory – large arrays, dynamic arrays or variables created after the register memory is filled - CUDA uses *Local memory*. Local memory has the same issues with access speed and latency as global memory.

2.2.3 Parallel computing of the LBM

As parallel computing has become more powerful, efforts have been made to improve the performance of LBM on different combinations of CPUs and GPUs. This chapter highlights the methods employed and the results.

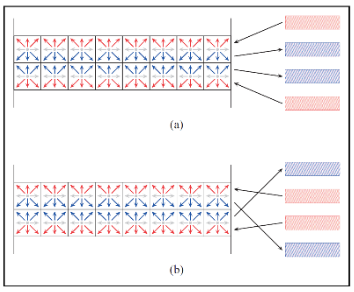


Figure 5. Inter GPU-communication a) First state b) Second state, Obrecht, Kuznik, Bernard Tourancheau & Roux (2013)

Obrecht et al. (2013) developed a system for using multiple GPU devices to compute LBM: Thermal LBM on Manny-core Architectures (TheLMA). TheLMA is a programming framework which defines C source files for computing LBM and managing POSIX threads which control the GPUs. They implemented Multi-Relaxation Time LBM, a refinement of LBGK which is more stable at high Re (Aidun & Clausen, 2010), with D3Q19 stencils on grid sizes of 192^3 and 256^3 with up to 8 GPUs. To transfer data between GPUs, they took advantage of zero-copy memory. Zero-copy memory is a type of CUDA pinned memory on the CPU which is directly accessible from within a kernel. Although copying memory between the host and device is slow, the memory access speed can be compensated for by allowing other blocks to continue to work while the copy is taking place. Having split the grid up into regular cuboids to provide maximum memory coalescence (Chapter 3.2.4) each GPU writes boundary cells to a pair of buffers. It then reads another pair of buffers into its boundary cells. After each time step, the role of the buffer arrays is swapped by switching pointers (Figure 5). Maximum measured performance was obtained from 6 GPUs on a

192^3 grid of 2150 MFLUPS. By focussing further on the limits of inter-GPU communication, they showed that TheLMA has excellent communication speeds and the performance speed rivalled parallel implementations on homogenous supercomputers at the time.

One way to increase the performance of CFD methods, and LBM in particular, is to decrease the number of cells. Unfortunately, this can lead to inaccuracies. Eitel-Amoret al. (2013) investigated the implementation of hierarchical meshes on a cell-centred lattice in two and three dimensions. They optimized their implementation for use on parallel machines. The LBGK is adapted so that cells can be locally refined. Communication from coarse to fine cells takes place through linear interpolation of the values at of the coarse cell given the relative position of the fine cell. Communication from fine to coarse cells requires averaging the values of a number of fine cells to effectively produce one coarse cell, which then communicates to other coarse cells. This is only necessary at the interface between cells of different sizes. Relaxation time is also altered depending on the refinement of the cells:

$$\tau_f = m\left(\tau_c - \frac{1}{2}\right) + \frac{1}{2} \qquad (10)$$

where τ_f and τ_c are the relaxation times for fine cells and coarse cells respectively, and m is the ratio between cell sizes, which is limited to 2 in this case. There is also a rescaling of the non-equilibrium components $f_i^{neq} = f_i - f_i^{eq}$. Two sensors are used: absolute value of the vorticity vector and difference of the total pressure at low mach numbers. When both of these sensors are above respective threshold values for a given cell, the cell is refined. When both sensors drop below separate threshold values, the cell is coarsened. Results from canonical test cases of flow past a cylinder and a sphere show good agreement with the literature. The refinement approach in 2D reduced the cell count by 35% and led to significant speedups, reducing CPU time by a factor of 50 without any accuracy decrease.

Valero-Lara & Jansson (2015) built on this work by applying a similar refinement method to a heterogeneous computing platform. They combined a GPU with a multicore CPU. By using the GPU to compute fine cells and the CPU to compute coarse cells, a speedup of 30% was achieved compared to homogeneous GPU implementations. They found that performance was strongly related to the ratio of fine cells to coarse cells.

Rinaldi et al. (2012) used CUDA shared memory to improve the performance of LBM. The collision and stream steps are both computed in a single loop and the number of global memory accesses is greatly reduced. While shared memory is much faster than global memory, it restricts the maximum number of threads per multiprocessor. Despite this, utilizing shared memory results in significant speedup. On a 3D lid-driven cavity problem with a lattice size of 96^3, they achieve 259 MFLUPS and a speedup of 125 compared to a CPU implementation.

Valero-Lara (2016) defined fluid blocks within the lattice and surrounded each of them with ghost cells, a strategy which is normally applied to distributed memory systems. This allowed each of the fluid blocks to be carefully arranged in their own global memory arrays to improve memory access speeds, especially for ghost cell exchange. The total cell number was the size of the lattice plus ghost cells, compared to two lattices. The ghost cells were exchanged with a separate kernel, which led to some costs, on lattices up to 6000^2 but meant that speeds of around 620 MFLUPS remained consistent even with increasing lattice sizes. Traditional implementations face problems at grid sizes which require more memory than can be entirely stored on the GPU. By almost halving the memory requirements, this method allows bigger simulations without a performance cost.

Kuznik et al. (2010) tested double and single precision implementations of LBM on the GPU and found that there was no significant difference in the results compared with reference data. However, it did lead to a speed up of around 3.8.

Randles et al. (2013) parallelized the LBM for parallel CPU platforms. As well as the standard D3Q19 model, they implemented a D3Q39 model, in which a velocity component can be sent as far as the fifth neighbour. This enabled more complex fluid flows, such as those found in arteries and microfluidic devices. They highlight several methods for optimizing LBM. Deep halo cells, layers of ghost cells beyond a sub grid, are used to reduce communications between nodes. Although this means that several different nodes will be computing the same cells, the speedup associated with not having to communicate ghost cell velocities after every iteration was significant. Cache usage and compiler optimizations were found to have strong impacts on performance. Important relations to GPU implementations can be taken from their work on branching reduction. By restructuring the loops to reduce any branching, performance showed a marked increase. The CPU implementation mentioned here iterates over a region, as opposed to GPU iterations which generally execute on many cells simultaneously, however, branch reduction on threads within a warp will improve performance so a similar method could be applied. A method for predicting maximum performance is also introduced.

2.3 Lid driven cavity flow

In order to validate our results, the lid driven cavity flow problem was selected as a well-known and simple test case. A square cavity is filled with fluid and the lid moves with constant velocity, inducing flow (Perumal & Dass, 2011) (Ghia et al, 1982) (Marchi et al, 2009). It has proven a popular test case amongst implementations of the LBM. Despite the simplicity of the problem, it is relevant to many industrial applications and at sufficiently high Re, can induce turbulent flow within the system. Figure 6 demonstrates the problem, with u representing horizontal velocity and v representing vertical velocity. The boundary conditions for this problem have caused issues finding an analytical solution, especially at the upper corners. Marchi et al. (2009) undertook an assessment of previous data collected on the problem and generated numerical results, which align well with previous works while avoiding sources of error. Their results were used for validation. LBM is a normalized method so the cavity is a unit square.

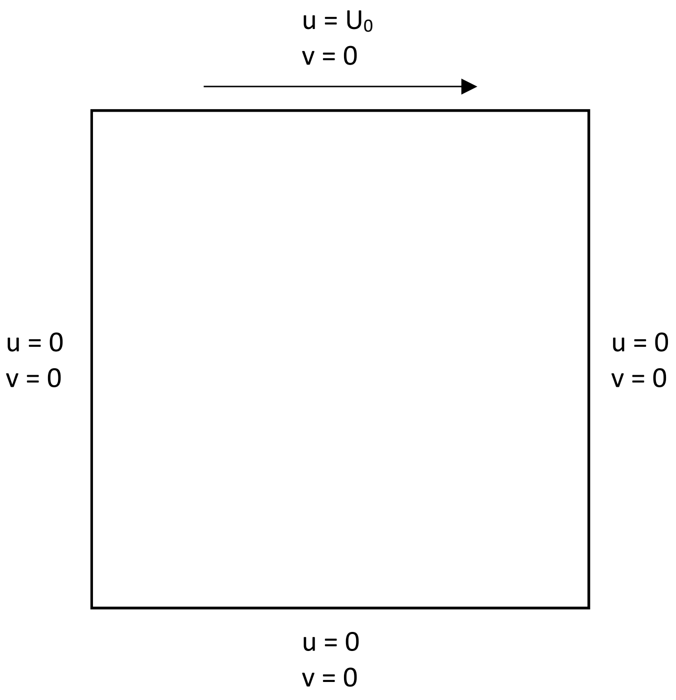


Figure 6. Lid Driven Cavity flow

2.4 Hardware

2.4.1 CPU

The CPU used for benchmarking is an Intel Core i7-6700 Processor. It has 4 cores with 2 threads on each core, a clock speed of 3.40 GHz and a maximum memory bandwidth of 34.1 GBs^{-1}. The maximum memory capacity is 64 GB and there are two memory channels (Intel, n.d). CPU implementations were compiled by GCC 6.3.1.

2.4.2 GPU

The GPU used for benchmarking is a Tesla K80, which is made up of 2 Tesla GK210s. It has a total of 4992 CUDA cores and 24 GB of memory which can be transmitted at up to 480 GBs^{-1}. The memory clock runs at 2.5 GHz but this can be raised during operation, within the thermal limit of the unit (NVIDIA, 2015). It can obtain speeds of up to 8.74 Teraflops with single precision floating points (NVIDIA, 2014). GPU implementations were compiled by NVCC with GCC 4.9.4.

```
set up grid;
for time t:
    apply boundary conditions
    // Stream - f[x, y](e_i)
    for (j=my; j>=1; y--)
       for (i=1; i<=mx; i++) {
         f[i, j](3) = f[i, j-1](3);
         f[i, j](5) = f[i+1, j-1](5);

    for (j=my; j>=1; j--)
       for (i=mx; i>=1; i--) {
         f[i,j](1) = f[i-1, j](1);
         f[i, j](4) = f[i-1, j-1](4);

    for (j=1; j<=my; j++)
       for (i=mx; i>=1; i--) {
         f[ (i, j)](6) = f[ (i, j+1)](6);
         f[ (i, j)](7) = f[ (i-1, j+1)](7);

    for (j=1; j<=my; j++)
       for (i=1; i<=mx; i++) {
         f[i, j](2) = f[index(i+1, j](2);
         f[ (i, j](8) = f[i+1, j+1](8);

    // Collide
    for (j=1; j<=my; j++)
       for (i=mx; i>=1; i++)
          compute feq
          perform collision step

end
```

Code 5. Psuedocode for CPU implementation of LBM

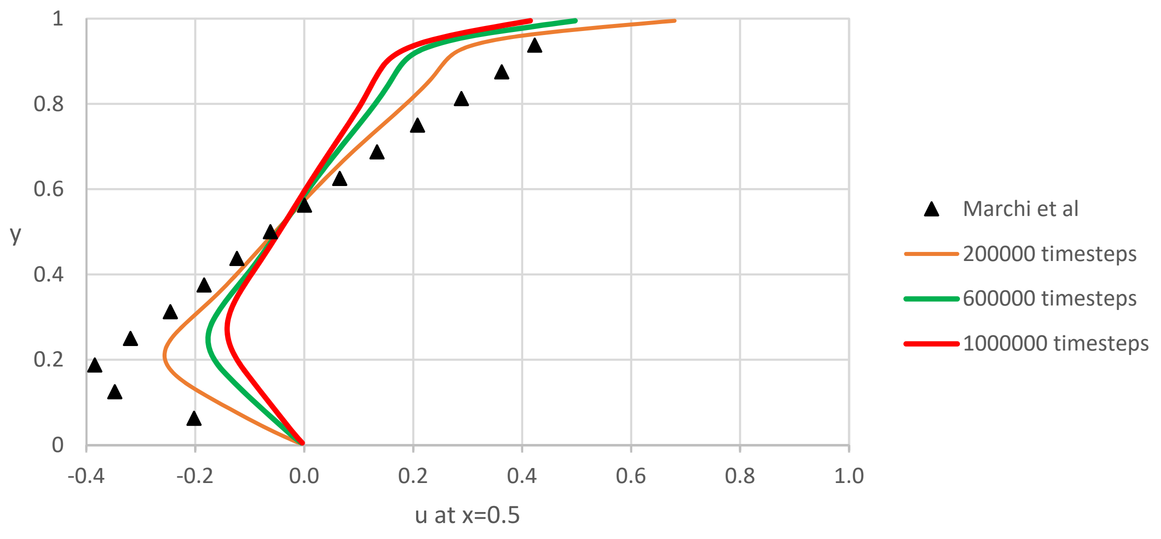


Figure 7. Horizontal velocity of 100^2 lattice

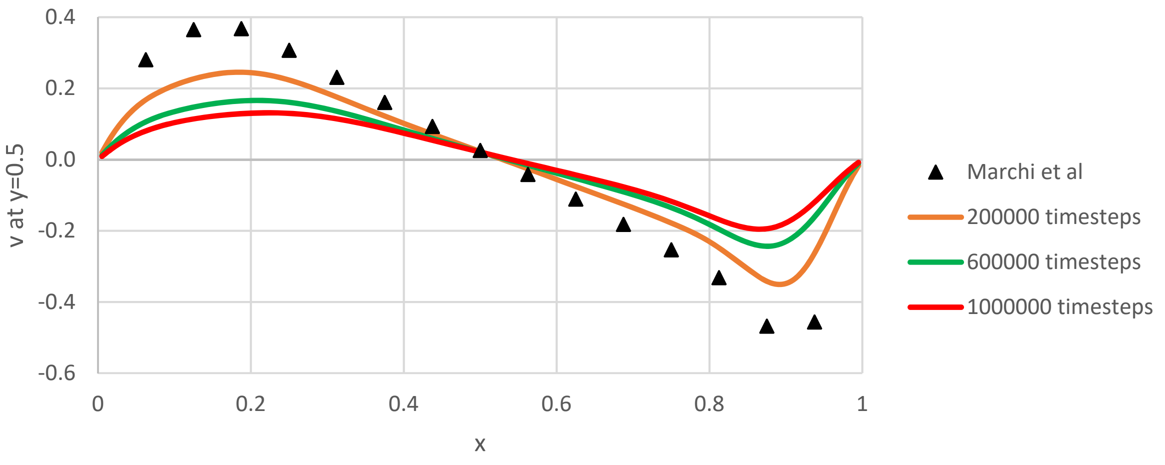


Figure 8. Vertical velocity of 100^2 lattice

3. Results

3.1 CPU Implementation

3.1.1 Description

Here, we discuss the CPU implementation of LBM. This program was based on a C++ prototype, by a student under Dr Deiterding, which applied LBM to a Poiseuille flow problem. It uses a combination of the pull and sweep methods during the stream step. Beginning in one corner of the grid, two adjacent velocities, such as $\mathbf{e}_6$ and $\mathbf{e}_7$, are streamed. The active cell moves in such a way that the values for the present time step are never overwritten. Once the whole grid has been covered, two new velocities are selected and the process repeats until all of the velocities are streamed. As $\mathbf{e}_0$ does not move, it does not have to be streamed. This allows the program to use only one array, which is significantly quicker than using two arrays. A notable departure from the

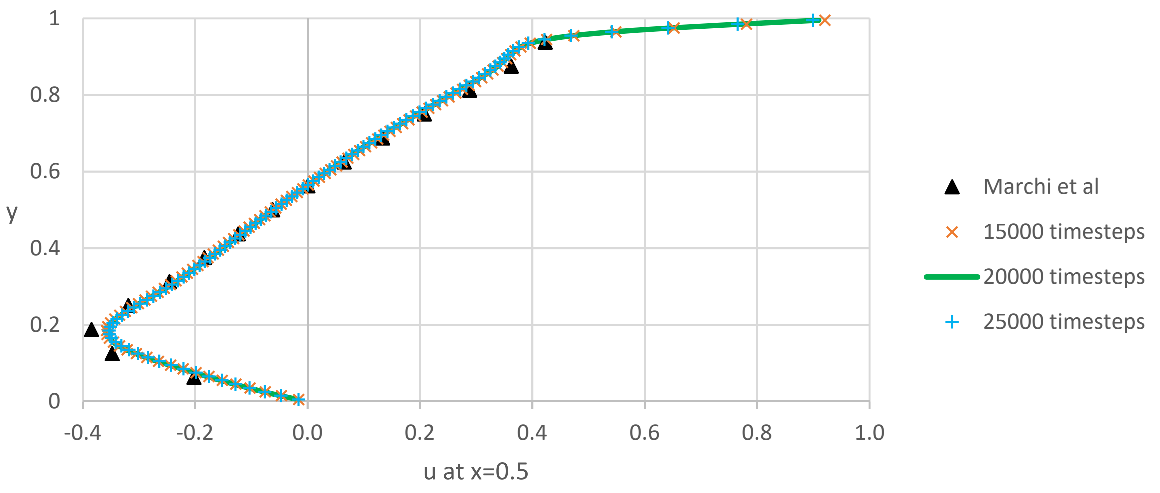


Figure 9. Horizontal velocity of 100^2 grid at stable iterations

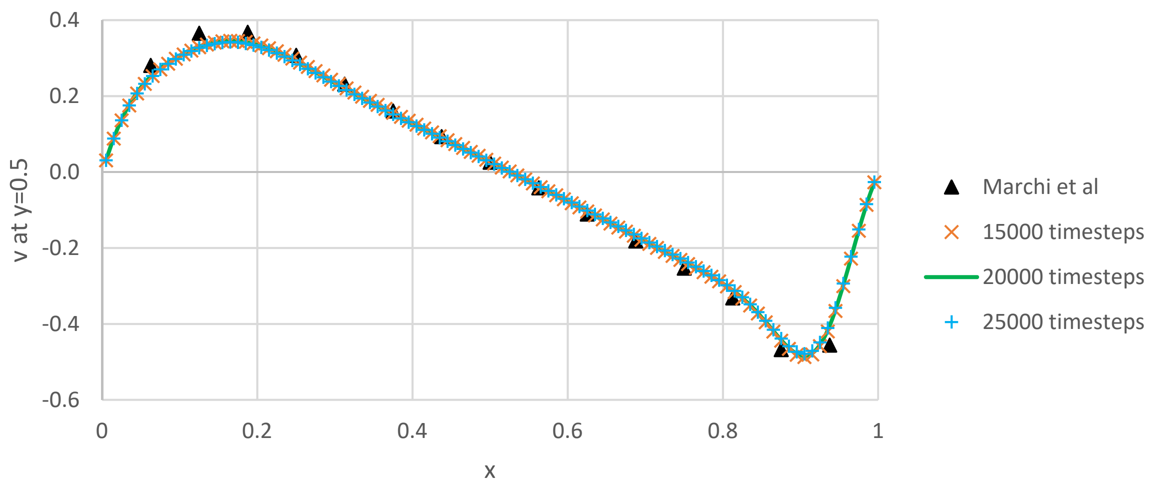


Figure 10. Vertical velocity of 100^2 grid at stable iterations

stencil approach is that the sweep method iterates over the velocities and then over the cells. This can be seen clearly in Code 5. Lid velocity is imposed by explicitly altering the horizontal and vertical velocities of the top row of ghost cells after each step. Iterating over each cell on the top row of the grid, $U(x, y+1) = 2U_0 - U(x, y)$. The other boundaries used the SBB boundary condition.

3.1.2 Convergence study and verification

Many iterative methods end once the difference in values between steps become insignificant (often around 10e-8) but this is not suitable for LBM, as a small error is introduced at each step due to calculations only going to the second derivative. Consequently, it was necessary to find the optimal number of time steps for a given grid size. Initial testing took place on a grid of size 100^2 and Re = 1000. Output was measured on the midlines and the data was compared to a highly accurate numerical solution (Marchi et al, 2009). At large iteration numbers, the simulation becomes unstable so smaller iteration numbers were investigated. Next, the grid size was steadily increased to test whether the expected convergence occurred. In normalized LBM for a square

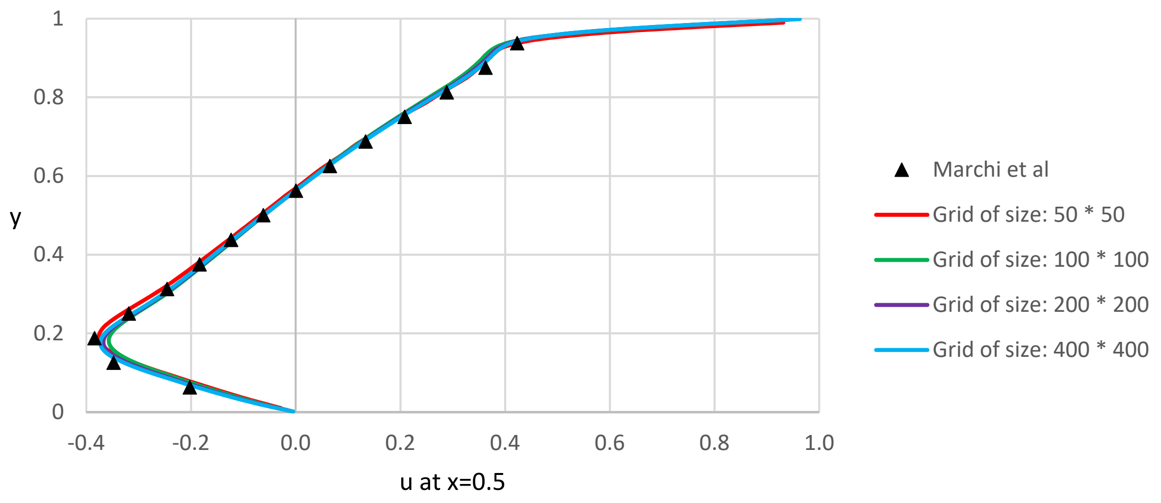


Figure 11. Horizontal velocity of converged flow on grids of varying size

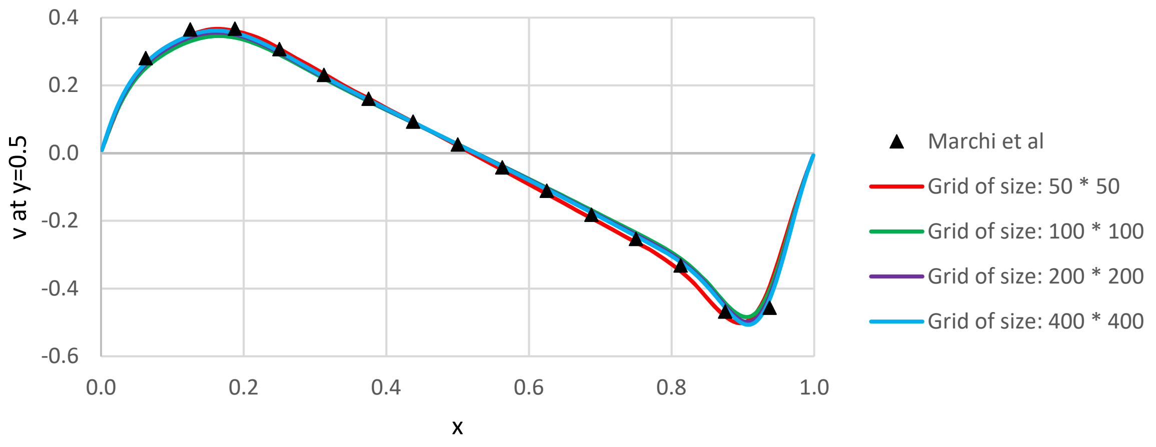


Figure 12. Vertical velocity of converged flow on grids of varying size

cavity, the required number of iterations increases linearly with grid width. It can be seen that for a square cavity of unit size, doubling the number of cells along the length will double the amount of time taken for any impulse to travel from one side to the other. In order to keep Re constant, some values must be changed. Eq. (7) shows the link between Δt, ω and v. As:

$$Re = \frac{U_0 L}{v} \qquad (11)$$

and in this application, L is equal to the number of cells along the width of the grid, and U_0, the velocity of the lid, is constant, when the grid size increases, we must alter the viscosity. This dictates that ω will have to be altered with grid size. We predict that the error at large numbers

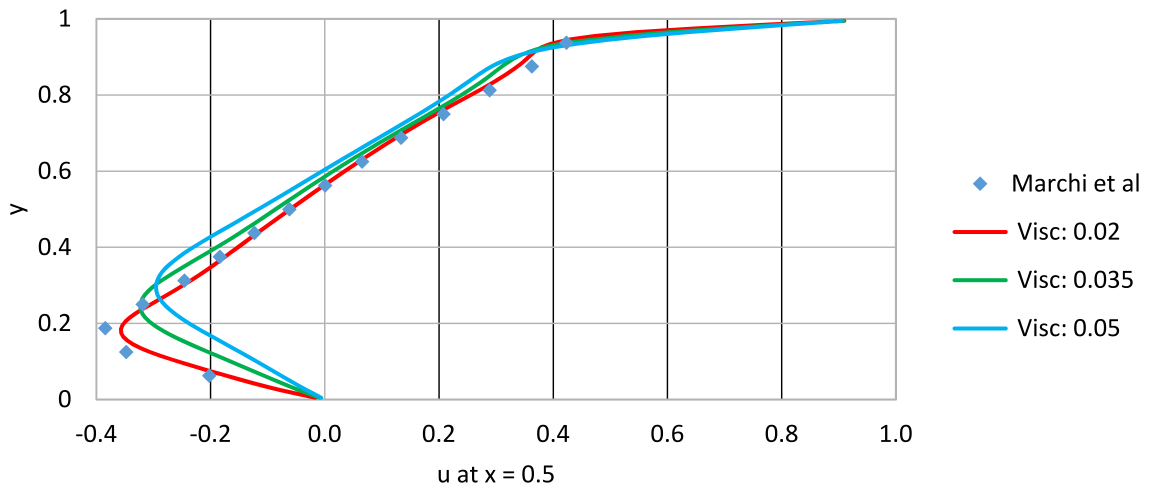


Figure 13. Horizontal velocity at different viscosities

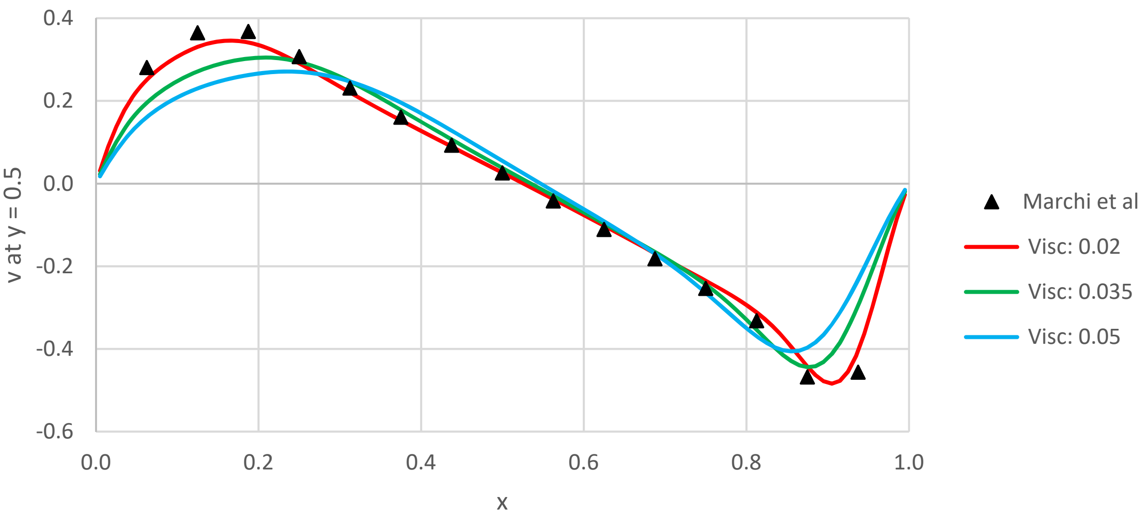


Figure 14. Horizontal velocity at different viscosities

of iterations is due to truncation errors accumulating and having a similar effect to increasing the viscosity of the fluid. Therefore we carried out tests with different viscosities assess the impact. Double precision floating point values were used for all simulations mentioned here.

Figure 7 and Figure 8 show the effect of accumulated errors at large numbers of iterations. The search for optimal iteration number was focussed empirically to between 15000 and 25000 (Figure 9 and Figure 10). Within this range, the program is stable and so the results are extremely close together. 20000 iterations was chosen as the optimal number for a 100x100 grid, through a mixture of closely investigating the numerical data and convenience. From there, larger grid sizes were tested (Figure 11 and Figure 12). As the grid size increases, the solution gets converges on the reference results. This behaviour was expected and validates the CPU implementation. The

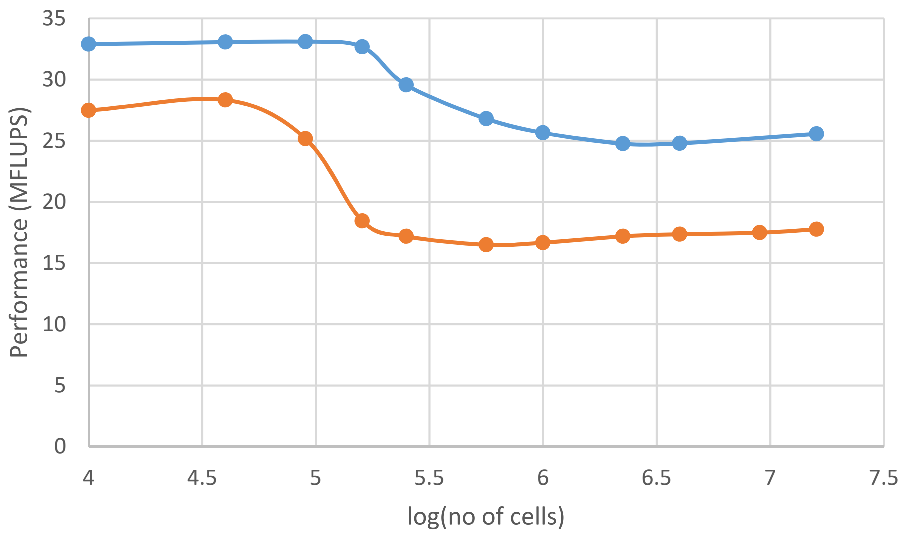


Figure 15. Performance of CPU LBM

case with increasing viscosity, show similar errors to the results at larger iteration numbers (Figure 13 and Figure 14). These simulations ran only 20000 iterations, but the effect is significant. This supports our hypothesis that error accumulation has a similar effect to increased viscosity.

3.1.3 Performance

In addition to validating the model, the CPU implementation also serves as a baseline for LBM performance against which we compare the performance of the GPU implementation. The program was run with both double and single precision floating points. It is standard practice when measuring performance of LBM implementations to use Million Fluid Lattice Updates Per Second (MFLUPS), with the formula:

$$P(MFLUPS) = \frac{s \cdot N_{fl}}{T(s) \cdot 10^6} \qquad (12)$$

where s is the number of iterations, N_{fl} is the number of fluid cells and T is the program execution time in seconds. Figure 15 shows the performance of the CPU. At small grid sizes, speeds of 33 MFLUPS are achieved, but this drops to around 25 MFLUPS for larger grids. This drop in performance is probably due to the slow speed of serialized access to a large memory array. As expected, the double precision program is consistently slower. It also experiences a distinct drop in performance, at a slightly lower grid size. This is likely due to the memory bandwidth reaching its limit sooner, given that double precision floating points contain twice as many bytes as single precision. However, the performance is significantly better than half the single precision performance, which was expected, and both implementations experience a mild increase in performance as their lattice size continues to grow.

3.2 GPU

3.2.1 Initial Program

In order to take advantage of the GPU, we implemented a stencil approach combined with pull during the stream step. This allows one thread to operate on one cell, a very convenient approach as noted in other works. Two arrays are set up, f_0 and f_1. Although it only offers limited bandwidth, f_0 and f_1 are stored in global memory. As global memory can be seen by all threads at all times, the initial implementation is simpler. It is recommended that thread numbers per block are a multiple of 32 or 16 – warp and half-warp. The number of blocks is calculated dynamically based on the number of cells:

$$no.of\ blocks = \frac{(N + (no.of\ threads - 1)}{no.of\ threads} \qquad (13)$$

where N is the total number of cells. The program is split into two kernels. *Stream* streams velocities from f_0 to f_1. *Compute* then runs the collide step, imposes boundary conditions and returns the values to f_0. CUDA defines several built in variables which are used to place each thread on a unique cell. The method is well established: *threadIdx* is a class which gives coordinates of the thread within the block, *blockIdx* performs the same function for blocks on the GPU grid and *blockDim* gives the total number of threads within a block. This is seen in Code 6 and Code 7, where *nxm* is the total number of cells that lie on the width of the grid, equal to the number of active cells plus two ghost cells. If the calculation for y gives a floating point, it is always rounded down to an integer. Therefore, each thread has a unique cell. The *index* function translates a two dimensional coordinate onto one dimensional arrays. As well as coordinates, it takes a length as a parameter which defines the width of the grid the coordinates are on. Single precision floating-point values are used due to their performance advantages over double precision. It is necessary to check that each cell lies within the range of active cells or ghost cells because on some grid sizes, more threads are than is required. In this case, the extra threads would attempt to access memory beyond the global arrays and would introduce errors into the program.

```
int tid = threadIdx.x + blockIdx.x * blockDim.x;
int x = (tid % nxm);
int y = (tid / nxm);
```

Code 6. Coordinate calculation in CUDA

The initial program shows good speed across a range of lattice sizes. The speed of the program increases slightly with lattice size and iteration number. The performance figures shown in Figure 15 are representative of the performance as it converges with iteration number. This effect is less

```
__global__  void  stream(float  f[][nz],  float  f_1[][nz],  float  om){
        //Stream from f to f_1
        int  tid  =  threadIdx.x  +  blockIdx.x  *  blockDim.x;

        //Get coordinates
        int  x  =  (tid  %  nxm);
        int  y  =  (tid  /  nxm);

        //If active cell
        if  ((x  >=  1)  &&  (x  <=  nx)  &&  (y  >=  1)  &&  (y  <=  ny)){
                //Stream
                f_1[index(x,  y, nxm)][0]  =  f[index(x,  y, nxm)][0];
                f_1[index(x,  y, nxm)][1]  =  f[index(x-1,  y, nxm)][1];
                f_1[index(x,  y, nxm)][2]  =  f[index(x+1,  y, nxm)][2];
                f_1[index(x,  y, nxm)][3]  =  f[index(x,  y-1, nxm)][3];
                f_1[index(x,  y, nxm)][4]  =  f[index(x-1,  y-1, nxm)][4];
                f_1[index(x,  y, nxm)][5]  =  f[index(x+1,  y-1, nxm)][5];
                f_1[index(x,  y, nxm)][6]  =  f[index(x,  y+1, nxm)][6];
                f_1[index(x,  y, nxm)][7]  =  f[index(x-1,  y+1, nxm)][7];
                f_1[index(x,  y, nxm)][8]  =  f[index(x+1,  y+1, nxm)][8];
        }
        else{
                //If ghost cell
                f_1[tid][0]  =  f[tid][0];
                f_1[tid][1]  =  f[tid][1];
                f_1[tid][2]  =  f[tid][2];
                f_1[tid][3]  =  f[tid][3];
                f_1[tid][4]  =  f[tid][4];
                f_1[tid][5]  =  f[tid][5];
                f_1[tid][6]  =  f[tid][6];
                f_1[tid][7]  =  f[tid][7];
                f_1[tid][8]  =  f[tid][8];
        }
}
```

Code 7. Stream kernel on initial GPU implementation

prevalent on large grids. On small grids, there is simply not enough work to keep all the CUDA cores busy. As the lattice size increases, the proportion of blocks which are operating entirely on internal cells with no boundary conditions to compute increases. These blocks can be more efficiently placed and they mask the extra time spent transferring data by blocks with boundary cells. The initial program shows good accuracy compared to the CPU program and reaches speeds of 224 MFLUPS.

3.2.2 Refinement to active cells only

The initial program operated on every single cell in the grid, including the ghost cells outside the boundary. As ghost cells are used only to store boundary velocities, this wastes threads which could be computing. Altering the program to work only on active cells (within the boundary) involved offsetting the coordinate system and decreasing the number of threads. This is expected to lead to a speed increase, but it also makes the program much simpler from a programming point of view, as there are fewer divergent paths. Boundary cells on sides other than the top only write only three velocities to the ghost cells. The three velocities of the ghost cell closest to the

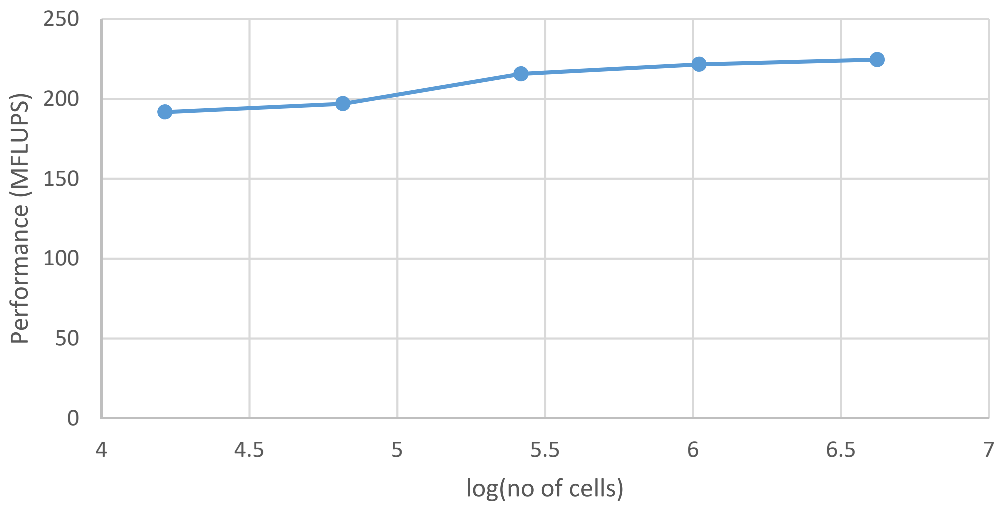


Figure 16. Performance of initial GPU implementation

grid are the only values that are read during the stream step, so this reduces the number of memory accesses without impacting on the accuracy of the program (Figure 19). It does, however, slightly increase the amount of work done by boundary cells, which now have to write ghost cell velocities as well as their own. As blocks wait for the last thread to finish executing before they close, this could create a new time penalty.

3.2.3 Optimal number of threads per block

Having previously chosen to use 256 threads per block, here we investigate the effects of different thread numbers. As threads are arranged in warps of 32, it is known that thread multiples of 32 will produce the best results. The kernels are already set up to calculate a unique coordinate based on the thread and block number, so the only change is specifying the number of blocks and threads within the triple angle brackets when calling the kernel:

```
kernel<<<(cells+(threadsPerBlock-1))/threadsPerBlock,  threadsPerBlock>>>();
```

where cells is the total number of active cells on the grid. The calculation for the number of blocks guarantees that the minimum number of blocks is created. The result of the previous two refinements is shown in Figure 17. As expected, the performance of the 64 thread program is good and increases with lattice size – it shows a peak speed of 329 MFLUPS. However, the 128 and 256 thread programs show a distinct decrease in speed that was not present in the initial program. These programs still produce accurate results so the decrease in performance is attributed to conflicts during memory access. Although their performance at large lattice sizes is somewhat diminished, increase in performance with lattice size is demonstrated after the initial decrease.

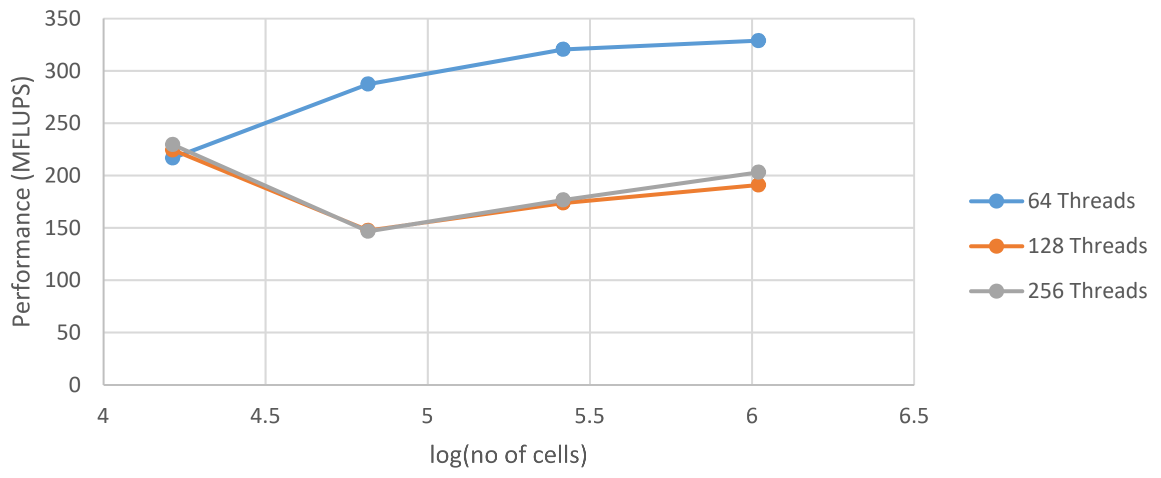


Figure 17. Performance of LBM with different numbers of threads per block

3.2.4 Reshaping of blocks with dim3

CUDA blocks and threads can be defined by up to three dimensions. So far, blocks and threads have only been created in one dimension. This aligns with the global arrays, which are one dimensional to increase access speed. The effect is that blocks are arranged on the grid as horizontal lines which only move up the grid if they overlap the end of a row. This means that blocks placed at the top and bottom of the grid are computing a much larger number of boundary cells than blocks in the interior of the grid. As the primary bottleneck for this program is memory access and the kernel has to wait for the last block to finish before it can synchronise, it is possible that creating a more equal distribution of boundary cells amongst the blocks will result in some speedup in program execution. This is achieved by arranging the blocks as rectangular sub-grids on the larger grid. Each block will have a much larger number of interior cells than boundary cells and many threads will be able to bypass the boundary condition work. CUDA makes this very easy to achieve with dim3 variables (Code 8). Within the kernel, coordinates are now calculated by x = threadIdx.x + blockIdx.x * blockDim.x and y = threadIdx.y + blockIdx.y * blockDim.y.

```
//Dimensions of block
dim3  blockDim(bx,  by);
dim3  threads(bw,  bh);

stream<<<blockDim,  threads>>>();
```

Code 8. Example of CUDA dim3 variables

Blocks of 256 were implemented in varying dimensions to investigate what effect this would have on performance. The results are shown in Figure 18. The results follow the same pattern as the 256 thread program in Figure 17 but for large grids there are clear differences. The square thread

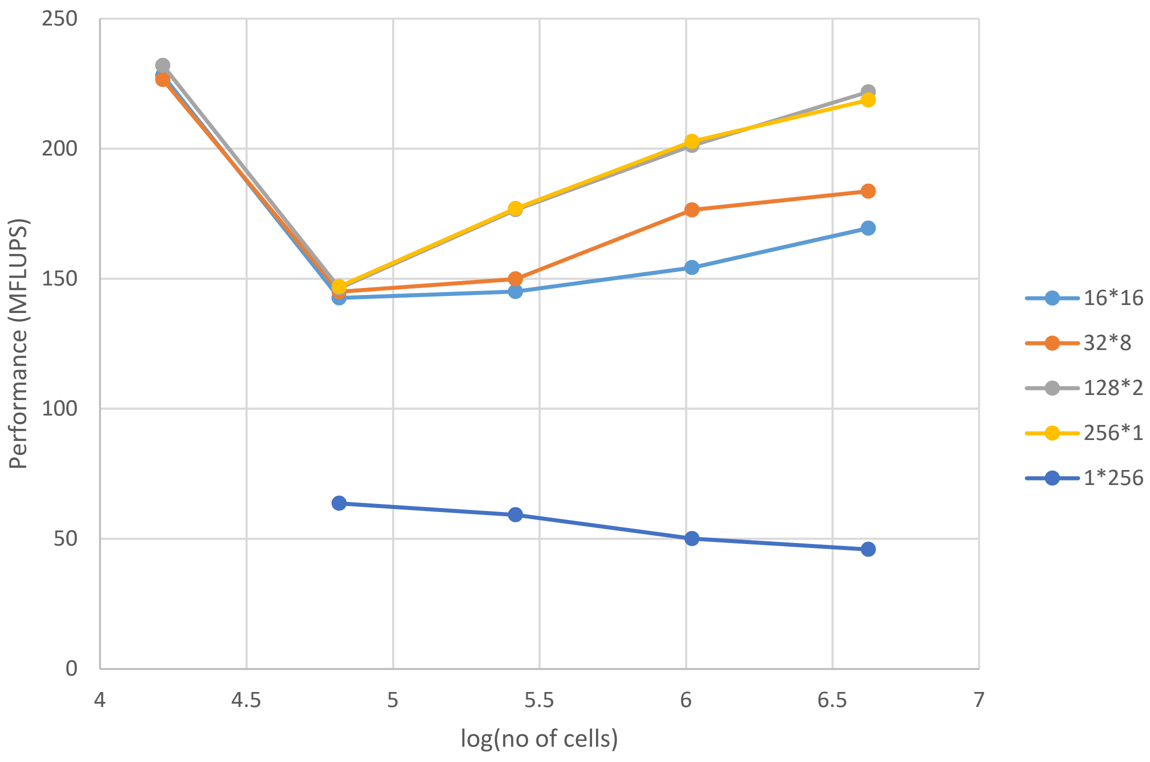


Figure 18. Performance of different block layouts

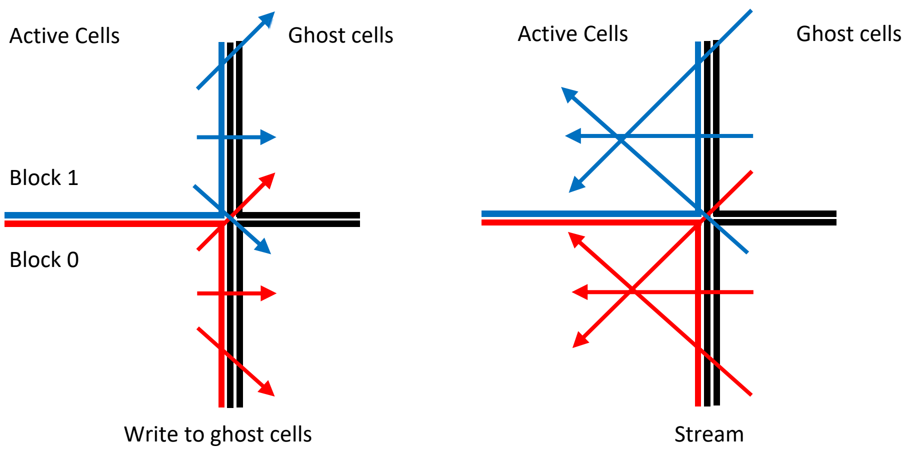


Figure 19. Write to and stream from ghost cells in neighbouring blocks

arrangement performs poorly compared to flatter orientations. The programs with 128 and 256 thread wide blocks show almost identical performance and perform the best. The block arranged 256 threads high performs very poorly and performance decreases with lattice size. This is a good demonstration of the *memory coalescence* effect, which must be utilized for high speed memory access to global memory on CUDA. As the memory is stored in a one-dimensional array, cells which are close spatially, on a grid, can be far away in memory. When CUDA blocks are accessing data which is all close together in memory, this data access is said to be *coalesced*. It is dramatically faster than uncoalesced data access because CUDA devices can load 32, 64 or 128 bytes in a single transaction. While this explains the advantage of flatter cell shapes in this context, it is likely that the performance increase seen by the 128 and 256 thread wide blocks is due to decreased thread branching, compared to more compact blocks.

3.2.5 Shared Memory

As discussed in Chapter 2, shared memory offers much quicker access speed than global memory, so to maximise the performance of LBM on GPU, this shared memory must be properly leveraged. Although blocks synchronise at the end of kernels, communication only occurs between time steps when all velocities from the current iteration have been written. As the blocks are arranged on the gird the same way for both the stream and compute kernels, it can be seen that the blocks are effectively acting independently from one another. As shared memory allocations are destroyed once a block has executed, data must be stored in global arrays between kernel runs. Here, we describe the most successful implementation of shared memory. Firstly, the two kernels are combined into one kernel. This prevents the need to write to and read from global memory between the stream and compute steps. One shared memory array is created per block and is sized to provide enough memory for all of the cells within the block, but no ghost cells or halo cells. Two coordinate sets are calculated, one for the position of the thread within the block, for accessing shared memory, and one for the position of the thread on the lattice, for accessing global memory. Next, the stream step is combined with a read from global memory. This process is shown in Code 9. To ensure that all of the threads in the block have successfully streamed, before moving onto the collide step, the command threads are synchronised. This prevents memory conflicts between threads that have moved further in the execution path and are writing data to global memory and threads that are trying to read from global memory. Next, the collide step is computed. Each thread overwrites variables from its own cell in shared memory. Boundary conditions are then calculated and written directly to global memory. By previously limiting the boundary cells to writing only the necessary variables, here we prevent conflicts between

```
//Grid coordinates to block coordinates
grid[index(s_x, s_y, bw)][0] = f[index(x, y, nxm)][0];
grid[index(s_x, s_y, bw)][1] = f[index(x-1, y, nxm)][1];
grid[index(s_x, s_y, bw)][2] = f[index(x+1, y, nxm)][2];
grid[index(s_x, s_y, bw)][3] = f[index(x, y-1, nxm)][3];
grid[index(s_x, s_y, bw)][4] = f[index(x-1, y-1, nxm)][4];
grid[index(s_x, s_y, bw)][5] = f[index(x+1, y-1, nxm)][5];
grid[index(s_x, s_y, bw)][6] = f[index(x, y+1, nxm)][6];
grid[index(s_x, s_y, bw)][7] = f[index(x-1, y+1, nxm)][7];
grid[index(s_x, s_y, bw)][8] = f[index(x+1, y+1, nxm)][8];
```

Code 9. Combined read/stream from global to shared memory

bordering blocks that are writing to ghost cells that will be read by another block during the next iteration. Every thread then writes its cell in the block to global memory. The threads are synchronised again before the kernel ends. This is not strictly necessary, as threads should

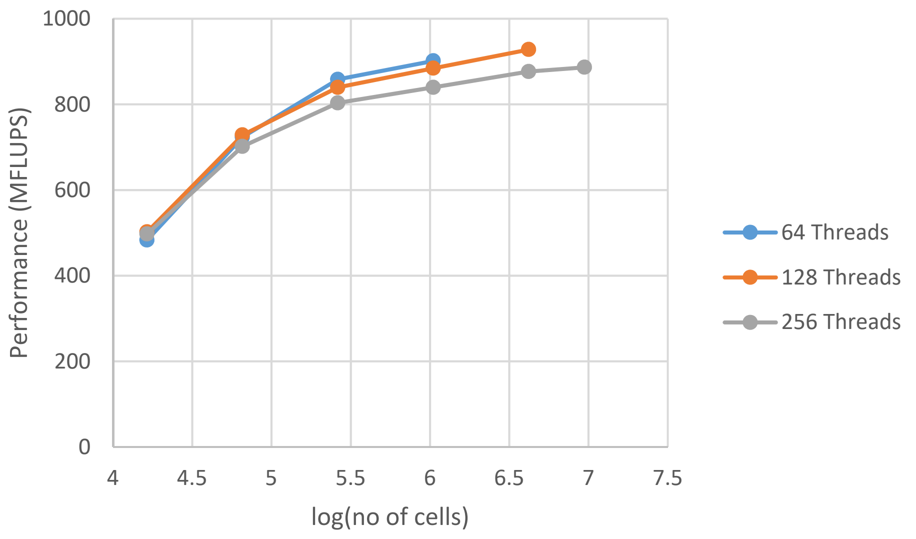


Figure 20. Performance of shared memory program

synchronise before the block finishes, but it was found to increase accuracy. The Tesla K80 offers up to 49152 bytes of shared memory per block, which with one array could store 1365 cells. However, with the optimal number of threads being between 64 and 256, and using one cell per thread, we will not approach this limit.

This implementation was tested with 64, 128 and 256 threads per block. From Figure 20, once more there is a performance curve showing a positive correlation with grid size. The smaller thread sizes are limited to smaller grids but perform marginally better than 256 threads per block. All of the programs reach around 900 MFLUPS and peak performance of 928.0 MFLUPS is achieved with 128 threads per block on a 2048^2 grid. Again, the greater proportion of internal blocks on larger grids, which execute faster than border blocks due to having fewer global writes and less thread divergence, increases performance.

3.2.6 Lock-Based program (Block synchronization)

Having achieved success with shared memory, here we attempt to increase its usage in our program. Ideally, there would be almost no communication with global memory. As shared memory allocations are not preserved between kernels, it is necessary to devise a method for computing the LBM without repeatedly calling the kernel. The grid has to be synchronised between iterations, so we need a method to synchronise blocks. CUDA does not offer a method for this but Xiao and Feng (2010) encountered this problem and developed two solutions – their mutex based solution will be discussed here.

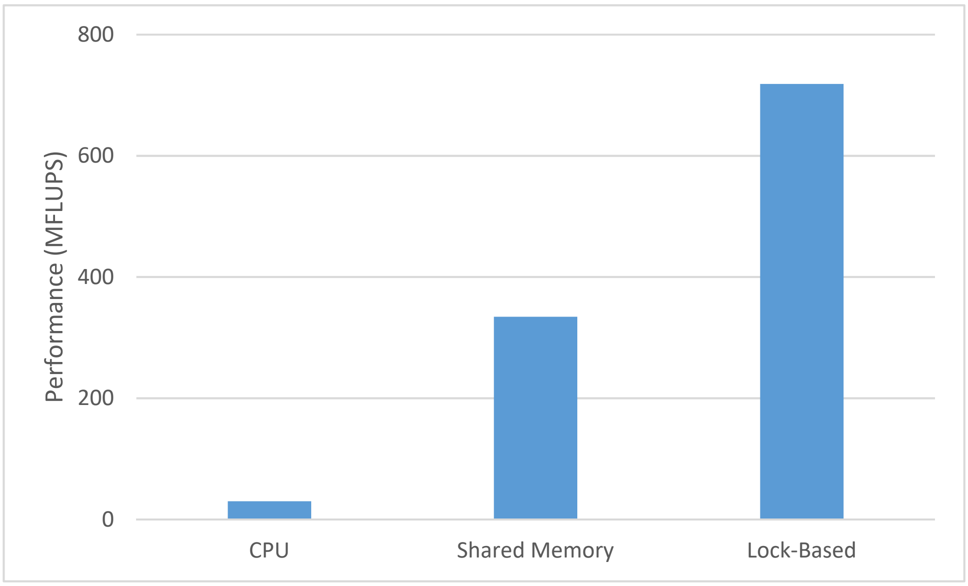


Figure 21. Performance of LBM implementations on 128^2 lattice for 1000 iterations

Atomic operations are a class of functions enable by CUDA that allow threads to have serialised access to a variable. The atomic operation used here is *atomicAdd*, which allows threads to iterate a value without conflicting with one another. In the lock-based synchronisation method, each block does its computations before one thread per block iterates a global variable. That thread then reads the global variable continuously to see if it has reached a goal value. Once it reaches the goal value, the threads synchronise again, this guarantees that the other threads wait, and the block continues. The lock-based function acts as a barrier function to the block. Xiao and Feng applied this to several common algorithms but here it is applied to LBM. In order to exchange information between blocks, velocities on the block boundary and ghost cell velocities are writen to global memory before the barrier function. Two shared memory arrays are created per block, and within the block the program works very similarly to the initial implementation, streaming from one shared memory array to the other and writing back after computing the collide step. The only difference is that at each iteration, cells on the block boundary stream velocities from global memory. Cells which are located within the block stream from shared memory. After each iteration, the goal value is increased by the number of blocks. Total number of iterations is now passed to the kernel as a parameter and after the initial copy of velocities from global to shared memory, there is a for loop within the kernel. Once the loop has finished, every velocity in the block is written to its global location in global memory.

The program was tested on a 128^2 grid for 1000 iterations and Figure 21 compares performance to a shared memory program with the same block layout and the CPU program. The lock-based program shows performance increases of over 2X compared to the shared memory program and over 24X compared to the CPU program. The number of iterations is noted here because performance of the lock-based program increases with iterations, as there is a higher proportion

```
//the mutex variable
__device__  volatile  int  g_mutex;

//GPU lock-based synchronization function
__device__  void  __gpu_sync(int  goalVal, int threadNo){

        //Only thread 0 is used for synchronization
        if  (threadNo == 0)
        {
                //Iterate mutex
                atomicAdd((int  *)&mutex,  1);
                //wait for other blocks to complete their actions
                while  (mutex  !=  goalValue){
                        //do nothing
                }
        }

        __syncthreads();
}
```

Code 10. Lock-based barrier function

of shared memory accesses compared to global memory accesses, which mostly occur at the beginning and end of the program when the whole grid has to be loaded to and from shared memory. Unfortunately, there is a limit imposed by the number of threads that can be active on the GPU at any one time. Because of the primitive busy-wait nature of the lock-based function, all of the threads in a block are occupied at all times. Eventually there is a point where the GPU cannot launch any more threads and the program will not run. For this implementation of LBM, it would not run reliably at grid sizes significantly bigger than 128^2. However, we have successfully presented an application of lock-based block synchronisation to the LBM and shown it offers significant advantages in performance.

3.2.7 Multi-GPU Program

One important aspect of parallel computing is that it is often performed on clusters of processors, which must communicate with one another. In GPU computing, this is known as Multi-GPU computing and Obrecht et al. (2013) have shown that the LBM can scale nearly linearly with the number of GPUs. By increasing the number of available cores, across GPUs, the program should run faster. Our setup is two Tesla K80s so we should expect to see the performance almost doubled from the peak performance of the shared memory program.

CUDA offers several methods for multi-GPU programming. Past efforts have been centred around zero-copy memory, but there have been advancements since then that make this process easier. Firstly, CUDA offers the function *cudaSetDevice* which sets which GPU is being addressed by the host. This simple function makes it simple to allocate or read memory and run kernels, on multiple devices at once. Unified Virtual Addressing (UVA) describes an address space, which is shared between the hosts and devices. Once memory has been allocated and a pointer has been retrieved, CUDA manages the technicality of which host or device the memory is located on. By

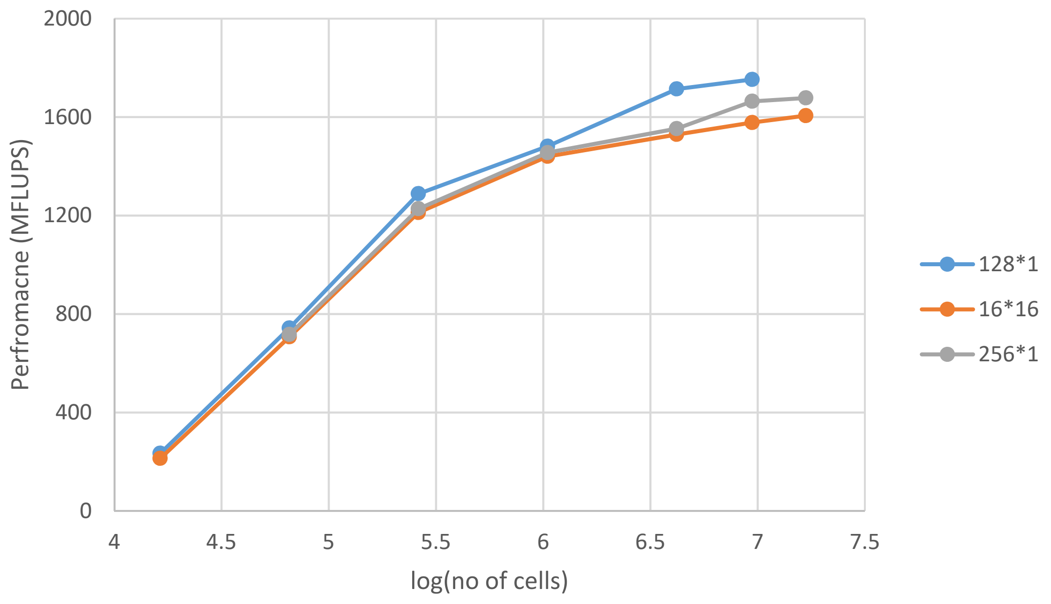


Figure 22. Performance of Multi-GPU implementation

passing the parameter *cudaMemCpyDefault* to *cudaMemCpy*, memory can easily be copied between devices or from devices to hosts, without having to specify the location of the memory. Device to device communication has been available since CUDA 4.0 through Peer-To-Peer communication (P2P). Because data is transmitted between devices through Peripheral Component Interconnect Express (PCIe), it can travel at up to 6.6 GBs^{-1}. It also avoids having to communicate with the host, which often has a slower bandwidth. Copies between devices are called from the host. Finally, CUDA offers streams on the host that are used for asynchronous operations. When using different GPUs, it is essential to use streams as each GPU has a different default stream. CUDA streams are created and managed in a similar method to time events; this is shown in Code 11.

```
cudaSetDevice(0);
cudaStream_t  stream0;
cudaStreamCreate(&stream0);
cudaEvent_t  start0;
cudaEventCreate(&start0);
```

Code 11. CUDA code for setting device and creating CUDA streams and events

When implementing the Multi-GPU LBM, it is best to split the larger lattice into shapes that offer the best memory coalescence when their border cells are copied. In this instance, the grid is split along the line $y = 1/2$. Each GPU computes a rectangular section of the grid before exchanging halo cells. Each GPU has ghost cells from the larger grid, but for accurate results they must also exchange halo cells which are on the border of the other GPU. After each iteration, the top row of

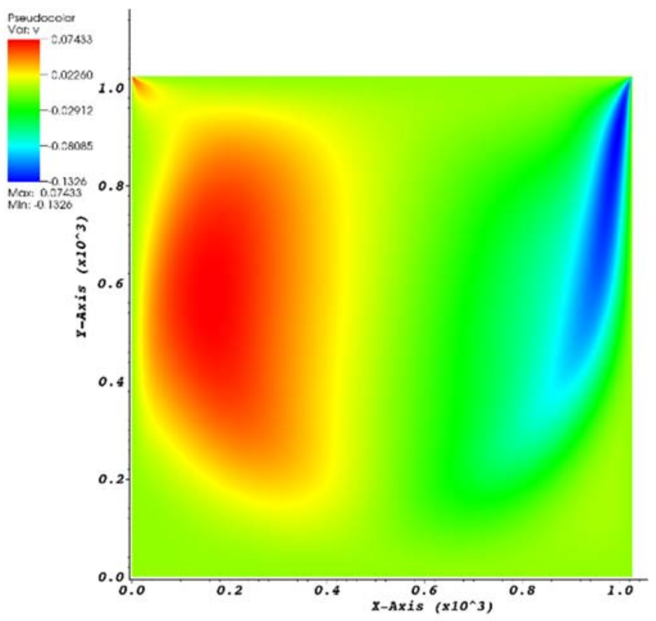


Figure 23. Visual output of vertical velocity on 1024^2 grid

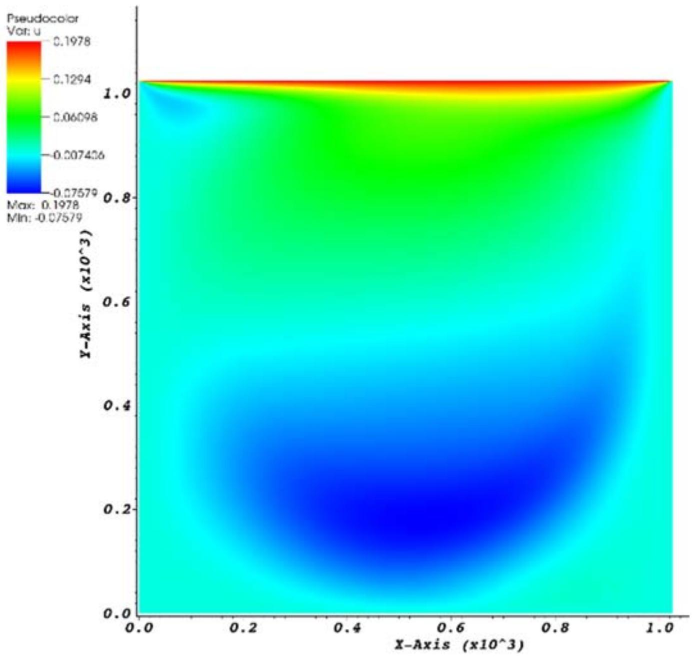


Figure 24. Visual output of horizontal velocity on 1024^2 grid

GPU_0 is copied to the halo cell row of GPU_1, and the bottom row of GPU_1 is copied to the halo cell row of GPU_0. The GPUs are then synchronized with *cudaDeviceSynchronize*, which waits until all preceding commands in all streams have completed. This is a relatively naïve implementation as the kernels cannot execute until the copy has taken place. However, it suffices for the purposes of testing a Multi-GPU LBM.

The program was tested with three different block shapes, all of which show increased performance with lattice size. This is fully expected, as the copy between devices exaggerates the

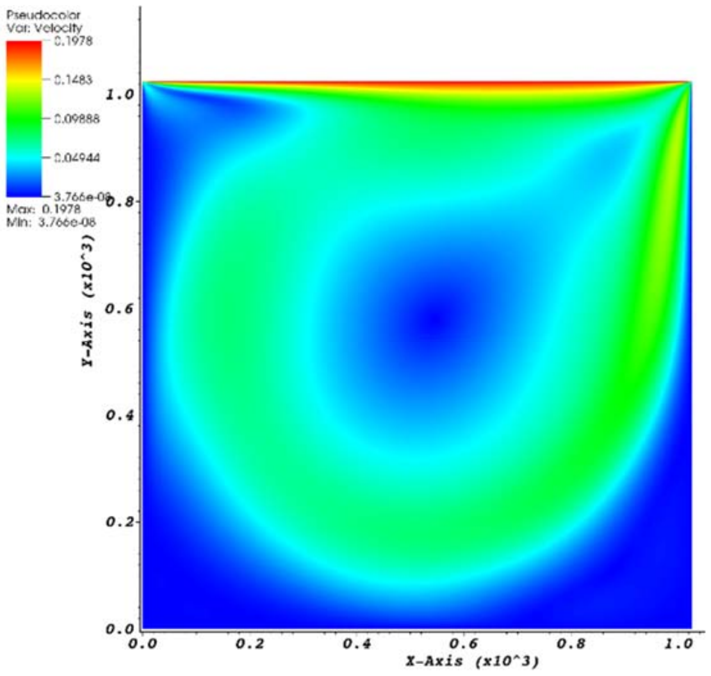


Figure 25. Visual output of total velocity on 1024^2 grid

effect of the proportion between interior and border cells. All three shapes show similar performance; peak performance of 1752.2 MFLUPS is achieved by 128 threads per block on a 3072^2 grid (Figure 22). This is a speedup of 1.89X compared to the fastest shared memory implementation, and 52.9X compared to the fastest CPU implementation. It is worth nothing that at this grid size, CPU performance is approximately 25 MFLUPS so the Multi-GPU LBM is 70X faster than the CPU on a similar lattice. To ensure that this program is properly communicating between GPUs and computing proper results, a convergence study was run on a 1024^2 grid and the output was compared to the numerical results from Marchi et al. (2009), who also used a 1024^2 lattice (Figure 26 and Figure 27). It is worth noting that the speed of this program enabled a convergence study on a much larger lattice than would have been feasible on the CPU (given other work requirements of the machine). The convergence study performed at 1481.4 MFLUPS. Also shown are visual ouputs of horizontal velocity, vertical velocity and total velocity (Figure 23, Figure 24 and Figure 25).

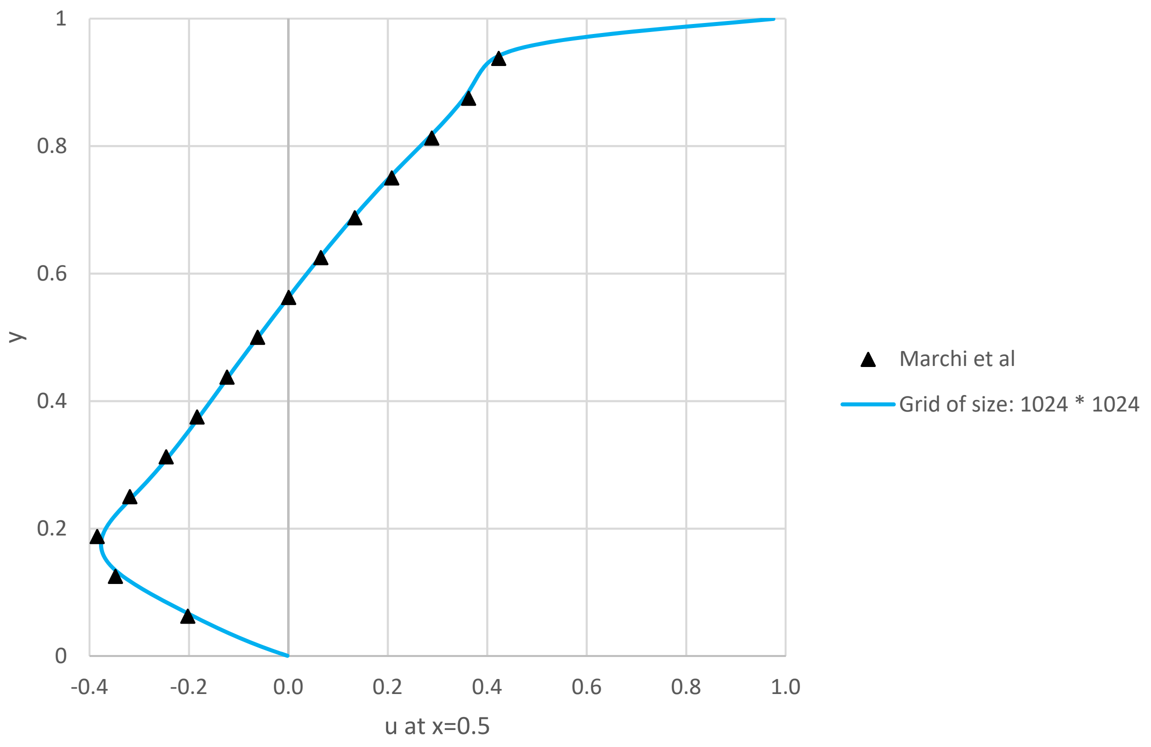


Figure 26. Horizontal velocity of 1024^2 lattice

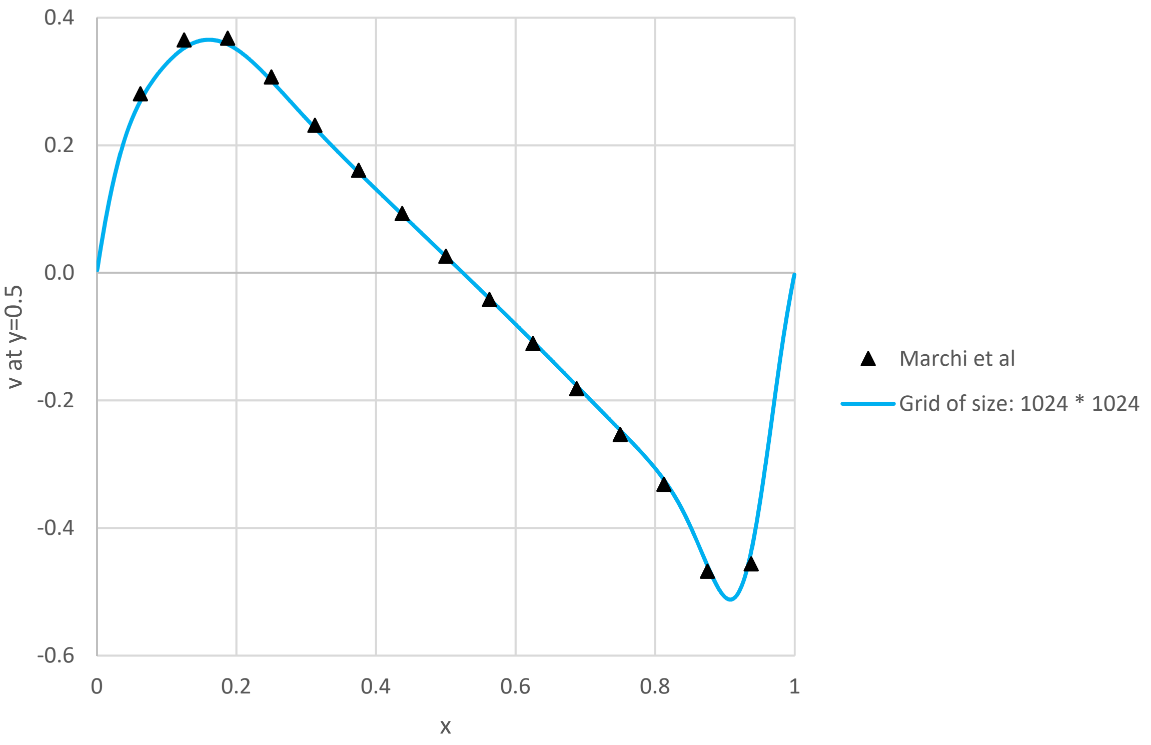


Figure 27. Vertical velocity of 1024^2 lattice

4. Discussion

The results (Figure 28) show that there are three components which had a significant effect on the performance of this GPU LBM implementation. Firstly, parallelizing LBM with a stencil-based approach and applying it to many cores gave a distinct increase in performance over the CPU implementation. Secondly, the use of shared memory to store velocities during iterations. Although there is still a high proportion of global memory access to shared memory access, the use of shared memory during the compute step, where each velocity is read and written, combined with coalesced memory access at the end of each iteration gave large increases in performance, particularly on large grids. This was emphasised by the lock-based program, which almost solely used shared memory and performed very well on a small grid. Finally, increasing the number of available cores by adding another GPU was shown to scale performance almost linearly. This implementation was not able to show true linear scaling, as others have achieved (Obrecht, et al, 2013), but it does show the potential for this kernel to expand to many GPUs: the main barrier to performance is exchanging halo cells, which could be masked by true asynchronous computing of the halo cells and other grid cells.

In their paper "Debunking the 100X GPU vs. CPU myth", Lee et al. (2010) describe optimizations for CPU programs which can make their performance more competitive with GPUs. While no effort was made here to parallelize the CPU implementation, this has been shown to give good performance boosts and CPU-GPU hybrid implementations have also been shown to give better

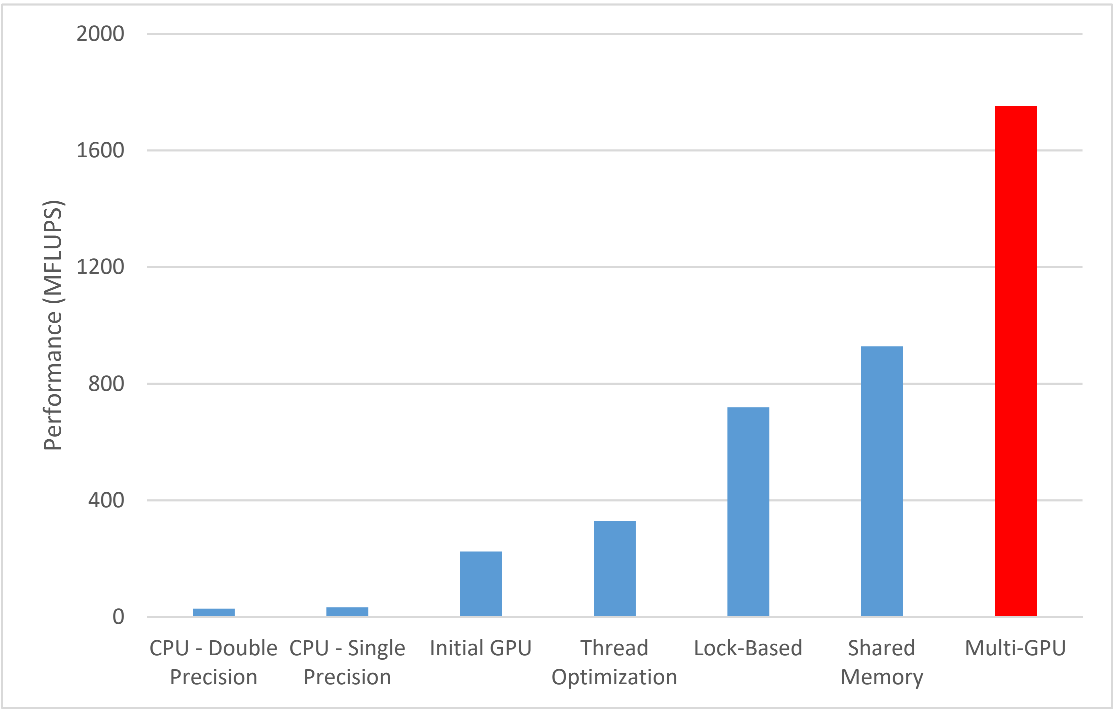


Figure 28. Peak Performance of LBM implementations

performance than homogeneous systems (Valero-Lara & Jansson, 2015). However, the single threaded CPU implementation is efficient and serves its purpose as a benchmark for performance. Kuznik et al (2010) were able to achieve peak performance of 947 MFLUPS on a single NVIDIA GTX280, which has a bandwidth of 147 GBs^{-1} and can deliver 1000 GFLOPS, for a square lid-driven cavity. With the more powerful Tesla K80, we would expect to see better performance from the shared memory implementation, which suggests there are improvements that can be made to the program.

Here, future works for the program are discussed. An effort should be made to replicate the scalability of Multi-GPU LBM by creating two global memory arrays per GPU. These will be cycled between in an A-B pattern. This enables one stream per GPU to compute only the host cells and transfer them to the ghost cells of the t+1 array on the other GPU, while another stream is computing the rest of the cells. If this is successful, an effort should be made to expand the program to more GPUs. Having replicated scalability, deep halo cells should be implemented. The program can then be expanded to hierarchical meshes and 3D LBM, which are important for simulating realistic scenarios.

5. Conclusion

CPU and GPU implementations of LBM have been presented. Validity is assured by comparing results with accurate numerical data for a lid-driven cavity. Even the basic GPU implementation performs much better than a CPU implementation and by leveraging CUDA technologies such as shared memory and P2P communication, peak performance is 50X peak CPU performance, and over 70X for large lattices. This work will be taken forward by Dr Deiterding and used to enhance the performance of the AMROC software system. In particular, it has been informative to learn which aspects of CUDA have notable impacts on performance.

6. References

Xiao and Feng (2010), Inter-block GPU communication via fast barrier synchronization, *2010 IEEE International Symposium on Parallel & Distributed Processing*

Obrecht, Kuznik, Bernard Tourancheau & Roux (2013), Multi-GPU Implementation of the Lattice Boltzmann Method, *Computers & Mathematics with Applications, 65*

Aidun & Clausen (2010), Lattice-Boltzmann Method for Complex Flows, *Annual Review of Fluid Mechanics, 42:439-72*

Valero-Lara & Jansson (2015), Multi-Domain Grid Refinement for Lattice-Boltzmann Simulations on Heterogeneous Platforms, *2015 IEEE 18th International Conference on Computational Science and Engineering*

Deiterding & Wood (2016), An Adaptive Lattice Boltzmann Method for Predicting Wake Fields Behind Wind Turbines, *New Results in Numerical and Experimental Fluid Mechanics X*

Kuzin & Derksen (2011), Introduction to the Lattice Boltzmann Method, *LBM Workshop*

E. Aslan, I. Taymaz, and A. C. Benim (2014), Investigation of the Lattice Boltzmann SRT and MRT Stability for Lid Driven Cavity Flow, *International Journal of Materials, Mechanics and Manufacturing, Vol. 2, No. 4*

Sanders, Kandrot & Dongarra (2015), *CUDA by example: an introduction to general-purpose GPU programming*, Upper Saddle River: Addison-Wesley/Pearson Education

Perumal & Dass (2011), Multiplicity of steady solutions in two-dimensional lid-driven cavity flows by Lattice-Boltzmann Method, *Computers and Mathematics with Applications, 61*

Dupin, Halliday, Care, & Munn, (2008), Lattice Boltzmann modelling of blood cell dynamics, *International Journal of Computational Fluid Dynamics, 22(7), 481-492*

Fragner & Deiterding (2016), Investigating cross-wind stability of high speed trains with large-scale parallel CFD, *Internation Journal of Comuptational Fluid Dynamics, 30*

Valero-Lara (2016), Leveraging the Performance of LBM-HPC for Large Sizes on GPUs using Ghost Cells, *International Conference on Algorithms and Architectures for Parallel Processing 2016*

Rinaldi, Dari, Vénere & Clausse (2012), A Lattice-Boltzmann solver for 3D fluid simulation on GPU, *Simulation Modelling Practice and Theory, 25*

Revell (2013), GPU Implementation of Lattice Boltzmann Method with Immersed Boundary: observations and results, *The Oxford e-Research Centre Many-Core Seminar Series*

Randles, Kale, Hammond, Gropp & Kaxiras (2013), Performance Analysis of the Lattice Boltzmann Model Beyond Navier-Stokes, *2013 IEEE 27th International Symposium on Parallel & Distributed Processing*

Eitel-Amor, Meinke & Schröder (2013), A lattice-Boltzmann method with hierarchically refined meshes, *Computers & Fluids, 75*

Peter Bailey, Joe Myre, Stuart D. C. Walsh, David J. Lilja & Martin O. Saar (2009), Accelerating Lattice Boltzmann Fluid Flow Simulations Using Graphics Processors, *2009 International Conference on Parallel Processing*

Kuznik, Obrecht, Rusaouen, & Roux (2010), LBM based flow simulation using GPU computing processor, *Computers & Mathematics with Applications, 59(7), 2380-2392*

Ghia, Ghia, Shin (1982), High-Re Solutions for Incompressible Flow Using the Navier-Stokes Equations and a Multigrid Method, *Journal of Computational Physics, 48*

Marchi, Suero & Araki (2009), The Lid-Driven Square Cavity Flow: Numerical Solution with a 1024 x 1024 Grid, *Journal of the Brazilian Society of Mechanical Sciences and Engineering, 31*

Portinari (2015) 2D and 3D verification and validation of the Lattice-Boltzmann Method, *MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES*
Bruneau & Saad (2004), The 2D lid-driven cavity problem revisited, *Computers and Fluids, 35*

Woolley (2011), CUDA Overview, Retrieved May 09, 2017, from http://www.cc.gatech.edu/~vetter/keeneland/tutorial-2011-04-14/02-cuda-overview.pdf

Developers Club (2013), Simulation of hydrodynamics: Lattice Boltzmann Method, Retrieved May 09, 2017, from http://developers-club.com/posts/190552/

Galloy (2017), CPU vs GPU since 2001, Retrieved May 09, 2017, from https://github.com/mgalloy/cpu-vs-gpu/blob/master/cpu_vs_gpu.pdf

Harris (2014), Using Shared Memory in CUDA C/C, Retrieved May 10, 2017, from https://devblogs.nvidia.com/parallelforall/using-shared-memory-cuda-cc/

NVIDIA (2017), CUDA C Programming Guide - NVIDIA Documentation, Retrieved May 11, 2017, from http://docs.nvidia.com/pdf/CUDA_C_Programming_Guide.pdf

INTEL (n.d), Intel® Core™ i7-6700 Processor, Retrieved May 16, 2017, from http://www.intel.co.uk/content/www/uk/en/products/processors/core/core-vpro/i7-6700.html

NVIDIA (2015), Tesla K80 GPU Accelerator, Retrieved May 16, 2017, from http://images.nvidia.com/content/pdf/kepler/Tesla-K80-BoardSpec-07317-001-v05.pdf

NVIDIA (2014), NVIDIA Tesla K80, Retrieved May 16, 2017, from http://images.nvidia.com/content/tesla/pdf/nvidia-tesla-k80-overview.pdf

Lee, Hammarlund, Singhal, Dubey, Kim, Chhugani, . . . Chennupaty (2010), Debunking the 100X GPU vs. CPU myth, *ACM SIGARCH Computer Architecture News, 38(3), 451*