Real-Time Flow Computations using an Image Based Depth Sensor and GPU Acceleration

Conference Paper · January 2013

CITATIONS
3

READS
361

3 authors:

Mark Mawson
Science and Technology Facilities Council
3 PUBLICATIONS **32** CITATIONS
SEE PROFILE

George W Leaver
The University of Manchester
8 PUBLICATIONS **15** CITATIONS
SEE PROFILE

Alistair Revell
The University of Manchester
120 PUBLICATIONS **1,109** CITATIONS
SEE PROFILE

Real-time Flow Computations using an Image Based Depth Sensor and GPU Acceleration

Mr. M. Mawson, Mr. G. Leaver, Dr. A. Revell (The University of Manchester)

Mr. Mark Mawson, PhD Candidate

THEME

HPC Session

SUMMARY

This work presents a recently developed computational fluid dynamics (CFD) solver based on the Lattice Boltzmann Method (LBM) that is algorithmically optimized for CUDA, allowing for real-time user interaction via stylus input, or a Microsoft Kinect sensor. The linear operations and the local nature of the Lattice Boltzmann equations are well suited to massively parallel architecture and thus ideal for Graphical Processing Units (GPUs). The 2D LBM solver is capable of speeds of up to 820 million lattice updates per second (MLUPS), the equivalent of 2600 time steps per second at a resolution of 640x480 points.

Results are also presented from a 3D version of the LBM solver capable of updating over 210 MLUPS, the equivalent of 50 time steps per second at a resolution of 160^3 points. Existing algorithms are used to distinguish people, and objects they may be holding, from background scenery; the extracted silhouette is then used to define boundary conditions within the fluid flow.

To facilitate interaction with the fluid flow, contour plots of macroscopic values (i.e. velocity and pressure) are rendered in real-time on the GPU, together with any boundary conditions captured from the Kinect. Several Image Based Flow Visualization (IBFV) techniques are implemented to provide a range of visual insight into the flow field; including spot noise, streamlines using the Line Integral Convolution method, and dye injection. The visualization methods are implemented using a combination of CUDA kernels and OpenGL texture functions, which removes visualization tasks from the CPU and minimizes host-device transfers, needed to maintain real-time performance. Output from the 3D LBM solver is currently rendered as 2D slices through the 3D domain using IBFV techniques.

KEYWORDS

CUDA, GPU, Kinect, Lattice Boltzmann Method

1 Introduction

As GPU technology becomes more prominent within scientific computing, it is inevitable that further uses will be identified and developed for the high performance computing at our disposal. Implementation of the Lattice Boltzmann Method (LBM) on GPU for solving fluid dynamics problems has already been the focus of several studies (see, for example, (Habich et al. 2011; Kuznik et al. 2010; Rinaldi et al. 2012). These works are indicative of an increasing trend to focus on the inherent good performance of LBM on GPU; typically in the order of 10^8 discrete fluid point updates/second. Performance of this magnitude brings about the possibility of real-time visualization of flow; 10^8 point updates per second on a domain consisting of 500x500 points in theory allows up to 400 complete iterations across the entire domain per second. This is more than enough for real-time output, which typically requires 24-60fps. In this work 2D and 3D LBM solvers are combined with OpenGL output, to allow the creation of first order boundary conditions within the flow using mouse input.

The advent of the Kinect sensor (Microsoft 2012) has provided an even more direct means of geometry extraction which can be exploited in the present work. This device was originally released as a user-interface to a video-gaming console but has since been employed for a range of novel applications. The Kinect sensor uses an infrared projector and a CMOS sensor to calculate the depth of points on an image captured by the camera (PrimeSense 2013). Software tools can then be applied to extract target components of the image; for example to identify and remove the floor plane in the image and, if necessary, to identify human shapes. In this paper a threshold is applied to the depth component, and if a pixel falls within this threshold it is implemented as a solid boundary within the flow. In this way complex real world shapes can be directly imported into the solver as boundary conditions.

A number of techniques have been implemented to visualize the simulation in real-time as it progresses. The simplest approach is to map a flow quantity such as the velocity magnitude to a standard colour texture. While this serves to highlight how the bulk flow behaves information about the flow directionality, or the evolution of flow from one pat of the domain to another, it is difficult using this method to ascertain the local direction of flow including details such as vortices and eddies. To allow the visualization of such features the Image Based Flow Visualization (IBFV) technique has been implemented (J. J. van Wijk 1991). This is a texture-based approach to visualizing flow that can emulate methods such as Line Integral Convolution (Cabral & Leedom 1993) and spot noise (J. Van Wijk 2002), but with lower computational overhead. A dye injection technique was also employed to render streak lines within flow from user selected seed points. By integrating the OpenGL rendering methods with the CUDA simulation expensive host/device transfers of simulation data

to rendering functions are avoided. All data is maintained on the GPU allowing real time visualization of the simulation.

In the following section a brief overview of GPU computing is given, before also providing background information on the flow visualisation techniques used and the Kinect sensor itself. Subsequently the specific form of the Lattice Boltzmann Method is provided along with details about how optimizations were achieved. This method is then validated and subsequently assessed for it's potential to deliver meaningful real-time flow computation.

2 GPU Computing

Modern GPUs use a Unified Shader Architecture (Luebke et al. 2007), in which blocks of processing cores (capable of performing operations within all parts of the graphics pipeline) are favored over hardware in which the architecture matches the flow of the graphics pipeline. For the purpose of this paper it is sufficient to note that the graphics pipeline takes the vertices of triangles as inputs and performs several vertex operations, such as spatial transformations and the application of a lighting model, to create a 'scene'. The scene is then rasterized to create a 2D image which can then have textures placed over the component pixels to create the final image. For a more comprehensive introduction to the graphics pipeline and how older GPUs matched their architecture to it see (Pharr & Fernando 2005).

The architecture of a generic Unified Shader based GPU is shown alongside that of a generic CPU in Figure 1. Processing cores can be seen arranged into rows with small amounts of cache and control hardware; the combination of all three is known as a Streaming Multiprocessor (SMX). Comparison with a generic CPU highlights the following key differences:

- GPUs sacrifice larger amounts of cache and control units for a greater number of processing cores.
- The cores of a SMX take up less die space than those of a CPU, and as a result are required to be more simple in design.

These attributes render the GPU suitable for performing computation on large datasets where little control is needed, i.e. the same task is performed across the entire dataset. Indeed, it is this aspect which has created interest in GPU computing for the Lattice Boltzmann Method, as identical independent operations are performed across the majority of the fluid domain, boundary conditions being the obvious exception.

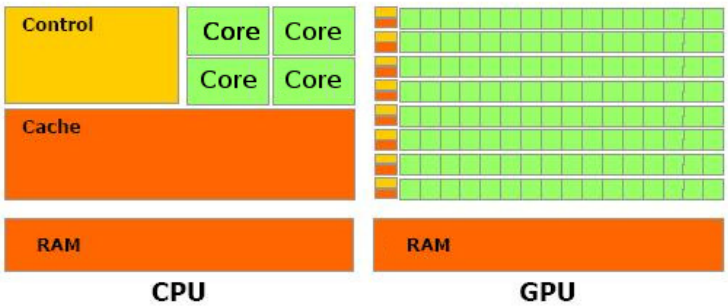


Figure 1: Typical CPU and GPU Architectures .

2.1 Threads and Blocks of Processing

Code written for NVIDIA GPUs is generally parallelized at two levels; the computation is divided into blocks which contain component parallel threads (see Figure 2). A single block of threads is allocated to a SMX at any one time, with the component threads divided into groups of 32 called 'warps'. The threads within a warp are executed in parallel, and all threads within the warp must execute the same instruction or stall. Figure 2 shows an example of threads and blocks being allocated two-dimensionally; the division of threads and blocks can be performed in *n* dimensions, depending on the problem.

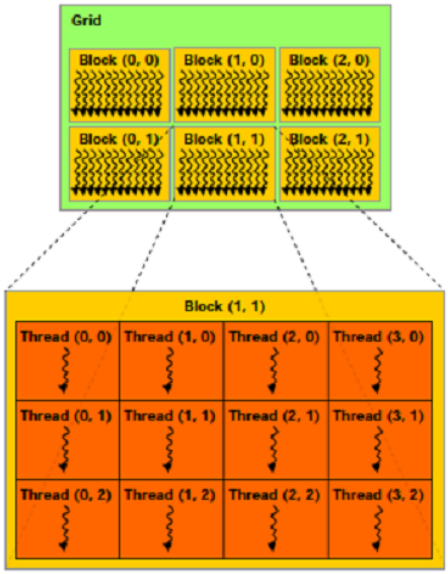


Figure 2: Blocks and Threads in CUDA Programming (NVIDIA 2010).

2.2 The Quadro K5000M GPU

The GPU used in this paper, a Quadro K5000M based on the NVIDIA GK104 architecture (NVIDIA 2012)) consists of 7 Stream Multiprocessors (SMX). Each SMX containis 192 Single Instruction Multiple Thread (SIMT)

processing cores operating at 601Mhz, a 64KB joint shared memory/cache and a number of control units and special function units. A 512KB layer of L2 cache is shared between all SMXs to provide cache coherency. In addition, all SMXs have access to 4GB of off-chip, and therefore higher latency, Device RAM (DRAM). A common characteristic of most GPUs, which stems from their original development for graphics rather than scientific computing, is that performance available when using double precision is often less than $1/2$ of that possible if only single precision performance is used. In the case of the K5000M used in this work this ratio is closer to 1/24; as a result all calculations reported herein were performed in single precision to maximise potential for 'real-time' simulations. It is worth emphasising that the K5000M , used for all computations in this work, is packaged inside a single laptop computer; compact and portable when compared to other computational resources commonly used for CFD.

3 Real-time OpenGL Visualization

Image Based Flow Visualization (IBFV) simulates the advection of particles through an unsteady vector field (in this case the macroscopic **u** field) to show local flow features. Prior to image-based methods, techniques such as streamline visualization required the user to select a set of seed points from which particles were advected. It was difficult for the user to locate seed points which would result in streamlines passing through the regions of flow of interest. IBFV uses 2D noise textures to represent a dense set of particles that cover the entire simulation domain. Hence the user is not required to find seed points that, after advection, highlight interesting regions of the simulation. Instead particles are seeded from every point in the simulation domain. The dense particles (i.e., noise textures) are advected forwards using texture mapping on a distorted polygonal mesh as indicated in Figure 3.

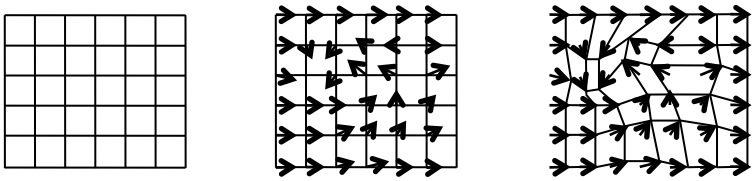


Figure 3: A CUDA kernel calculates a displaced polygonal mesh using the macroscopic u field at each frame.

The basic approach is to texture the previous frame's image **M** on to a distorted polygonal mesh and then blend the resulting image with a noise texture **N** according to some blending factor α. The noise texture provides random high contrast particles (typically a black and white random image is used). Repeated application of this process streaks the noise image in the direction of local

flow. In effect the framebuffer accumulates the smearing of random noise texture over time where the local direction of smearing is given by the unsteady flow field. This process is described by the recurrence relation:

$$M_i(\mathbf{x}) = (1-\alpha)M_{i-1}(\mathbf{x} + \mathbf{u}_i(\mathbf{x}, t_i)\Delta t) + \alpha N_i(\mathbf{x})$$

To create the distorted mesh onto which the previous frame's image is textured, a regular grid is placed over the simulation domain and the macroscopic **u** field is sampled at each vertex of the grid, thereby assigning a reference velocity to each vertex.

The velocities are used to displace the mesh vertices using a forward integration step (a first order Euler approximation). Texture coordinates (which are not displaced in this process) are assigned to each vertex to allow the previous frame's image to be sampled and effectively advected forward by the displaced mesh. It should be noted that the random element of the technique is purely for visualisation purposes does not directly modify the flow quantities; indeed it can be completely removed by setting α to zero.

The mesh resolution used to sample the simulation domain may equal that of the simulation for a 1-1 representation of the computed flow with the visualized flow. Otherwise the resolution can be downsampled in order to reduce the rendering load. Clearly, the effectivenes of the visualization tool is reduced if the mesh resolution is set too low in regions of high velocity gradients. The mesh geometry is held in an OpenGL Vertex Buffer Object and generated on the GPU using a CUDA kernel (an alternative approach would be to use an OpenGL vertex shader). It is important to highlight that the mapping of simulation **u** data to renderable textured geometry requires no host/device data transfer.

Dye injection uses a similar method. Point geometry is drawn at user specified seed points which act as a simple coloured regions in an otherwise transparent texture. The texture is applied to the distorted mesh described above. By repeatedly rendering the previous frame's dye image to the mesh (distorted according to flow velocity) the dye is streaked from the seed point through the flow. Of course, the requirement to specify seed points means that dye injection may miss interesting regions of flow. The user can, however, move the seed points and can specify multiple points until the desired visualization is achieved. The convective nature of this approach is often felt to be more intuitive, since the user can relate the evolution of the dye to real-world experience.

Flow and dye images are rendered seperately to offscreen textures. This allows one or both visualization techniques to be displayed. Both techniques require the use of the previous frame's image in the current frame. Rendering to a double buffered offscreen texture ensures that previous images remain available without the need to perform an expensive pixel readback operation

from the visible framebuffer, although this comes at the expense of extra GPU texture memory overhead. In total texture memory is therefore required to hold four offscreen textures: two for the flow visualization and two for dye injection. The offscreen textures have a fixed resolution, given by the simulation domain resolution. To display the offscreen textures a full screen quad is textured at the end of the rendering function, which relies on the GPU's texture filtering hardware to scale the images if the user has increased the size of the display window. This keeps GPU texture memory usage constant.

In order to permit simultaneous display of visualization techniques and the geometry itself, a texture representing the solid regions captured by the Kinect camera is drawn over the flow and dye images.

4 The Microsoft Kinect Sensor

The Microsoft Kinect sensor is a commodity accessory for the Xbox 360 games console developed by PrimeSense (Zalevsky et al. 2007), originally intended to allow greater user interaction with videogames. After its release in 2010 it was quickly hacked to allow use with a PC (Adafruit Industries 2010) and soon after an API was provided by PrimeSense (Mitchell 2010).

Depth is calculated by projecting a known light pattern from the infrared emitter, and measuring variances from this pattern seen by a CMOS sensor , a detailed explanation of this procedure can be found in (Liebe et al. 2006). Figure 4 shows this for the simple case of a book being placed in front of the sensor, the black object seen at the bottom of the image is simply a non-reflective stand.

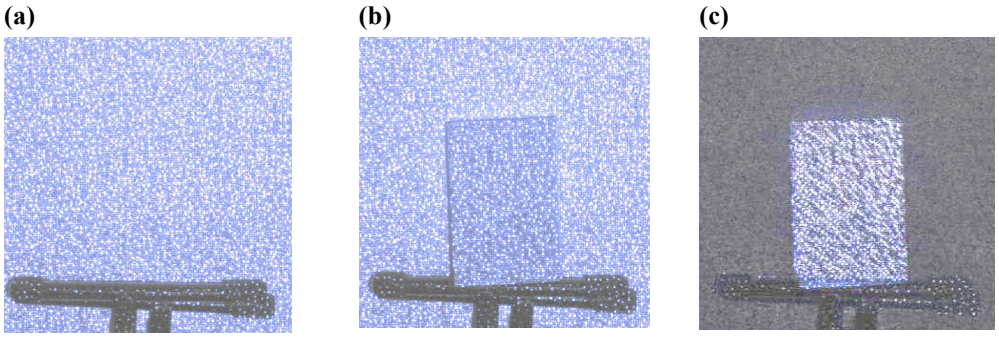


Figure 4: (a) The infrared light pattern produced by a Kinect, **(b)** with an object distorting the pattern and **(c)** the difference between the two images. From (FUTUREPICTURE 2010).

The depth information at each pixel location is stored as an 11-bit value, along with information about whether or not the pixel forms part of a shape

determined to be 'human' by the Kinect runtime software. The technical specification of the Kinect device used in this work are listed in Table 1.

Viewing Angle	43° vertical by 57° horizontal
Frame Rate	30fps
Depth Image Resolutions	80x60, 320x240, 640x480
Depth Image Range	800mm to 4000mm
Minimum/Maximum Depth Error	<2mm/40mm (Khoshelham 2011)

Table 1: Specifications of the Kinect depth sensor

5 The Lattice Boltzmann Method

5.1 Numerical method

In this study the Lattice Boltzmann Method is used to simulate fluid flow, this method is based on microscopic models and mesoscopic kinetic equations; in contrast to Navier-Stokes which is in terms of macroscale variables. The Boltzmann equation for the probability distribution function $f = f(\mathbf{x}, \mathbf{e}, t)$ is given as follows:

$$\frac{\partial f}{\partial t} + \mathbf{e} \cdot \nabla_{\mathbf{x}} f = \Omega_{12}, \tag{1}$$

where $\mathbf{x}$ are the space coordinates, and $\mathbf{e}$ is the particle velocity. The collision operator Ω_{12} is simplified using the 'BGK' single time relaxation approach found in (Bhatnagar & Gross 1954) , in which context, it is assumed that local particle distributions relax to an equilibrium state, $f^{(eq)}$ in time τ :

$$\Omega_{12} = \frac{1}{\tau}\left(f^{(eq)} - f\right). \tag{2}$$

The discretized form of Eqn. 1 is obtained via Taylor series expansion following (He & Luo 1997), as shown in Eqn. 3, where i refers to the discrete directions of f. The dimensionality of the model and spatial discretisation of f is given in the 'DmQn' format, in which the lattice has m dimensions and f has n discrete directional components. In the current work the D2Q9 and D3Q19 models are used, in which the discrete velocity is defined according to Eqns. 4 and 5 respectively. Since spatial and temporal discretization in the lattice are set to unity, the lattice speed $c = \Delta x/\Delta t = 1$.

$$f_i(\mathbf{x}+\mathbf{e_i}\Delta t, t+\Delta t) - f_i(\mathbf{x},t) = \frac{\Delta t}{\tau}\left(f^{(eq)}(\mathbf{x},t) - f(\mathbf{x},t)\right) \tag{3}$$

$$\mathbf{e_i} = c\begin{pmatrix} 0 & 1 & -1 & 0 & 0 & 1 & -1 & 1 & -1 & 0 \\ 0 & 0 & 0 & 1 & -1 & 1 & -1 & -1 & 1 & 0 \end{pmatrix} \quad (i = 0,1,...,8) \tag{4}$$

$$\mathbf{e_i} = c\begin{pmatrix} 0 & 1 & -1 & 0 & 0 & 0 & 0 & 1 & -1 & 1 & -1 & 0 & 0 & 0 & 0 & 1 & -1 & -1 & 1 \\ 0 & 0 & 0 & 1 & -1 & 0 & 0 & 1 & -1 & -1 & 1 & 1 & -1 & 1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 & 1 & -1 & -1 & 1 & 1 & -1 & 1 & -1 \end{pmatrix} (i = 0,1,...,19) \tag{5}$$

The equilibrium function $f^{(eq)}(\mathbf{x},t)$ can be obtained by Taylor series expansion of the Maxwell-Boltzmann equilibrium distribution (Qian et al. 1992) :

$$f_i^{(eq)} = \rho\omega_i\left[1 + \frac{\mathbf{e_i}\cdot\mathbf{u}}{c_s^2} + \frac{(\mathbf{e_i}\cdot\mathbf{u})^2}{2c_s^4} - \frac{\mathbf{u}^2}{2c_s^2}\right] \tag{6}$$

In Eqn. 6, c_s is the speed of sound $c_s = 1/\sqrt{3}$ and the coefficients of ω_i are $\omega_0 = 4/9$, $\omega_i = 1/9, i = 1..4$ and $\omega_5 = 1/36, i = 5..8$ for a D2Q9 lattice and $\omega_0 = 1/3,\ \omega_i = 1/18, i = 1..6$ and $\omega_5 = 1/36, i = 7..19$ for a D3Q19 lattice. The macroscopic velocity $\mathbf{u}$ (made up of *u, v* and in 3D *w*) must satisfy the requirement for low Mach number, M , i.e. that $|\mathbf{u}|/c_s \approx M << 1$.

Macroscopic quantities (moments of the distribution function) are obtained as follows:

$$\rho = \sum_i f_i \tag{7}$$

$$\rho\mathbf{u} = \sum_i \mathbf{e_i} f_i \tag{8}$$

The multi-scale expansion of Eqn. 3 neglecting terms of O(εM^2) and using Eqns. 7 and 8 returns the Navier-Stokes equations to second order accuracy (Guo et al. 2002).

5.2 Boundary Conditions

In this work a first order 'bounce-back' boundary condition is used for objects embedded within the fluid flow. This condition simply reverses the direction of

a component of f if it is advected into a boundary (Gallivan et al. 1997). In bounce-back conditions are applied to the walls of a rectangluar domain using a D2Q9 lattice. The solid lines represent normally advected components of f and the dotted lines represent components of f that are 'bounced-back', i.e. on the top wall an incoming component f_2 would be bounced back to become f_4. In this manner f_8 takes the incoming value of f_6 and f_7 becomes f_5.

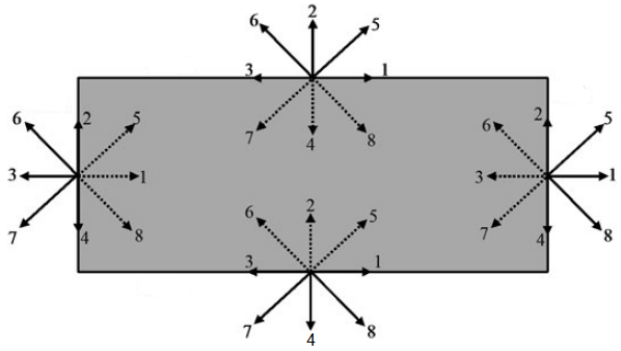


Figure 5: Bounce-back boundary conditions on a D2Q9 lattice.

While bounce-back boundaries are used for solid objects, the 2D rectangular fluid domain is bounded by symmetry conditions on the top and bottom and first order inlet and outlet conditions.

In 3D the second-order boundary conditions derived in (Hecht & Harting 2010) are employed to impose no-slip boundaries on the top and bottom walls and inlet/outlet conditions in the x direction, periodic conditions are used in the z directions. The second-order boundary conditions take the form of a bounce-back condition with additional terms provided by $f^{(eq)}$ and a correction to the momenta in the transverse directions on the boundary (Zou et al. 1997).

5.3 Implementation

When programming for GPUs there are two main considerations to ensure high performance; ensuring that as much of the code as possible can be executed independently in parallel across threads and blocks, and the arrangement of data in memory on the GPU.

5.3.1 Memory arrangement

The present code is parallelized such that one thread will perform the complete LBM algorithm at one spatial location $f(\mathbf{x})$ in the fluid domain. Each thread stores values of $f(\mathbf{x})$, ρ, $\mathbf{u}$ and information about whether or not the current location is a boundary, in a struct to minimse high latency access to DRAM once initially loaded. Within DRAM it is common practice to 'flatten' multiple dimension arrays into a single dimension, as the extra de-referencing

operations required add to the already large latency of accessing off-chip memory. In the present work, this is extended to combining and flattening the components of f into a single array, such that $f_i(\mathbf{x})$ is addressed as `f[i*Nx*Ny+y*Nx+x]`. Storing f in order of i and then by x and y coordinates neighbouring threads within a warp will enable simultaneous access to neighbouring elements of f. When this occurs the data transactions can be grouped into a single larger transaction; this is known as a coalesced access.

5.3.2 Independent LBM algorithm

The LBM is typically broken down into several steps:

1. Calculate $f^{(eq)}$ using Eqn. 6.
2. Apply the collision operator
$$f_i(\mathbf{x},t+\Delta t) = f_i(\mathbf{x},t) + \frac{\Delta t}{\tau}\left(f^{(eq)}(\mathbf{x},t) - f(\mathbf{x},t)\right).$$
3. Stream f_i in the appropriate direction $f_i(\mathbf{x}+\mathbf{e_i}\Delta t, t+\Delta t) = f_i(\mathbf{x}, t+\Delta t)$.
4. Apply boundary conditions.
5. Calculate ρ and $\mathbf{u}$ using Eqns. 7 and 8.

The algorithm in its current form poses some locality problems if it to be used in a highly parallel fashion. If a thread is launched for each location $f(\mathbf{x})$ then the operation $f_i(\mathbf{x}+\mathbf{e_i}\Delta t, t+\Delta t) = f_i(\mathbf{x}, t+\Delta t)$ is not local and synchronisation across the domain will be required before the boundary conditions, ρ and $\mathbf{u}$ can be calculated at each point in the domain. Instead the re-phased 'pull' algorithm is used, following (Wellein et al. 2006) and (Rinaldi et al. 2012):

1. Stream f_i in the appropriate direction $f_i(\mathbf{x},t) = f_i(\mathbf{x}-\mathbf{e_i}\Delta t, t-\Delta t)$.
2. Apply boundary conditions.
3. Calculate ρ and $\mathbf{u}$ using Eqns. 7 and 8.
4. Calculate $f^{(eq)}$ using Eqn. 6.
5. Apply the collision operator.
$$f_i(\mathbf{x},t+\Delta t) = f_i(\mathbf{x},t) + \frac{\Delta t}{\tau}\left(f^{(eq)}(\mathbf{x},t) - f(\mathbf{x},t)\right)$$

By providing the initial condition $f=f^{(eq)}$ these two algorithms will return identical macroscopic values, but this new algorithm enables higher levels of parallel efficiency to be obtained. The 'pull' algorithm is the operation $f_i(\mathbf{x},t) = f_i(\mathbf{x}-\mathbf{e_i}\Delta t, t-\Delta t)$, which is used instead of $f_i(\mathbf{x}+\mathbf{e_i}\Delta t, t) = f_i(\mathbf{x}, t)$. The streaming step is now a local operation, as each point gathers its own new distribution function from neighbouring points. It also eliminates the requirement to store ρ and $\mathbf{u}$, unless they are required for post-processing, as they are only used in the calculation of $f^{(eq)}$. In this work the number of OpenGL updates is generally infrequent compared to the number of lattice

updates, and so the macroscopic values are stored only in the iteration immediately before visualization.

6 Validation of the LBM code

To ensure complete validation of the implementation of the numerical solver described in the previous section, validation work was conducted on a Lid Driven Cavity and a 3D plane Poiseuille flow and the Lid Driven Cavity in both 2D and 3D.

6.1 3D Plane Poiseuille Flow

Figure 7 confirms agreement with the analytical solution for the Poiseuille flow (between two static parallel plates) with second order boundary conditions for a Reynolds number of 5. The stream wise velocity profile of is given by $u(y)=u_{max}(1-y^2/L^2)$, where y represents the distance from the centre of the y-axis, L is the distance from the centre of the y-axis to the wall and u_{max} is the maximum velocity. Second order convergence can clearly be seen in Figure 8, up until the point where the diffusive scaling of **u** causes the single precision floating point error to become dominant, and the overall error to become constant. In conducting the convergence study, the value τ was fixed while **u** was altered to ensure the same Reynolds number (Ning & Premnath 2012). This scaling causes **u** to become very small on larger domains, and at a certain point, floating point error will begin to dominate, as shown in the Figure for the smallest tested value of delta x (Hecht & Harting 2010).

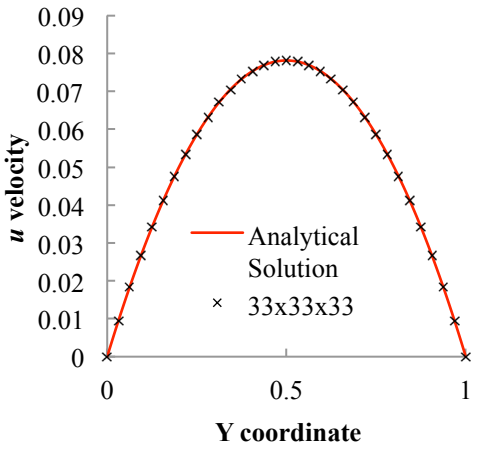


Figure 6 Poiseuille flow at Re=5.

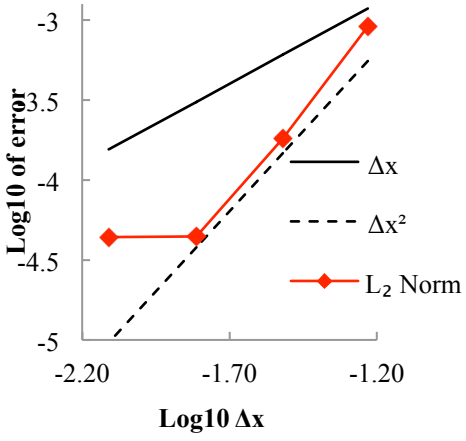


Figure 7 Convergence study.

6.2 2D and 3D Lid Driven Cavity

The Lid Driven Cavity case was performed at three Reynolds numbers {100, 400, 1000}. In each case a square domain is created with zero velocity on the left, right and bottom walls and a velocity is imposed on the top wall in the x direction, as shown in Figure 9. In 2D this is achieved with bounce-back conditions along the stationary walls and a first order velocity condition on the top wall, in 3D the second order boundary conditions from (Hecht & Harting 2010) are used to create the stationary and moving boundaries. τ is set to 0.7 and scaling is achieved according to:

$$\mathbf{u} = \frac{(\tau - 0.5)\,\mathrm{Re}}{3L}$$

In each case results are compared against those in (U. Ghia et al. 1982) in 2D and (Jiang & Lin 1994) in 3D.

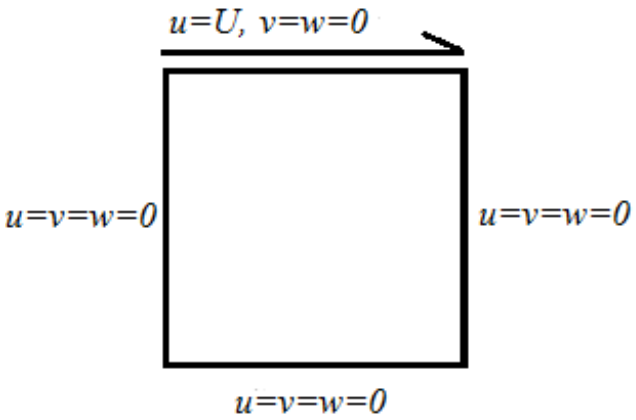
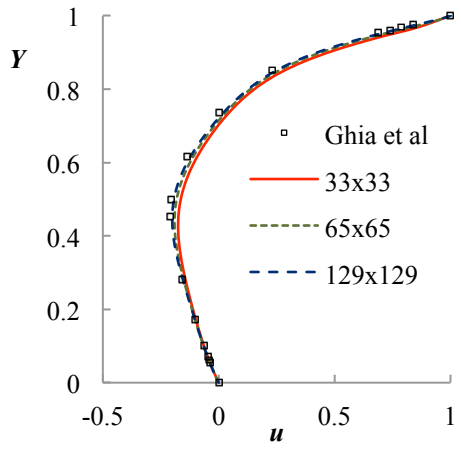
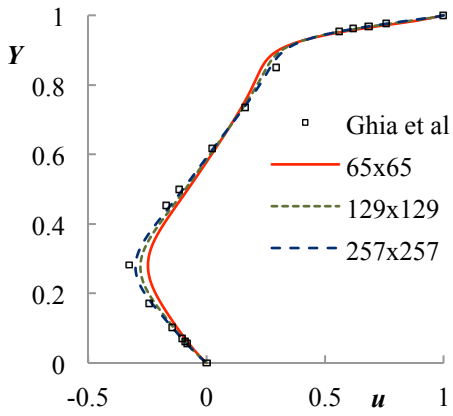


Figure 9: Lid driven cavity.

Figure 10 displays profiles of velocity extracted from the centre of the 2D domain for the three Reynolds numbers and the 3D resolutions in each case. Figure 11 displays the analogue for the 3D case. For higher Reynolds number computations the lattice numbers are increasing to reflect the higher levels of resolution required. The coarsest lattice used for the lowest Reynolds number 2D case contains just over 1000 points, whilst the finest lattice for the highest Reynolds number 3D case requires almost 17 million points. In all cases the results demonstrate a convergence for subsequent increase in lattice size and reference results are shown to be reproduced.





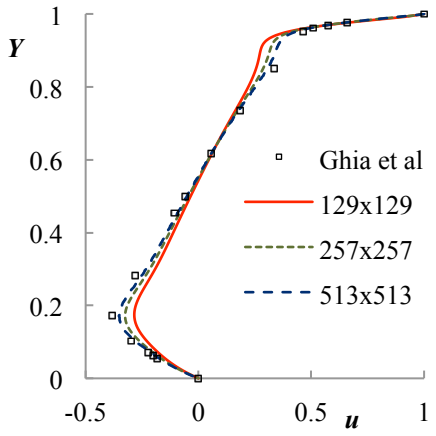
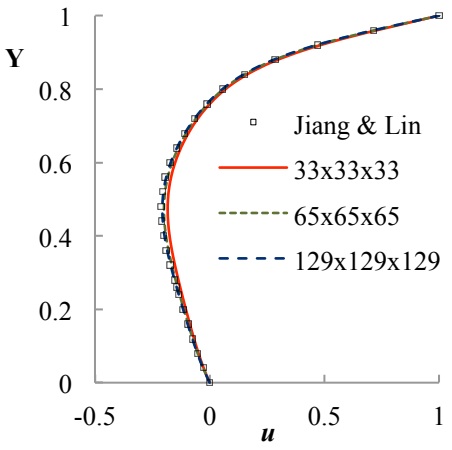
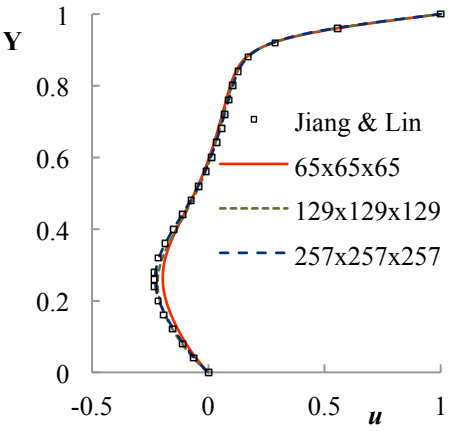


Figure 8: 2D Lid Driven cavity. From top to bottom, Re=100, 400, 1000.





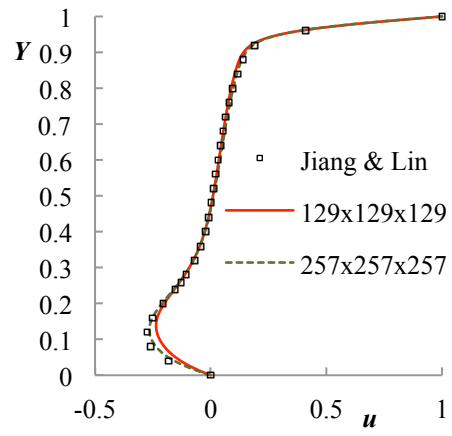


Figure 9: 3D lid driven cavity. From top to bottom, Re=100,400,1000

7 Assessment of Real-time capability

The potential of this code to deliver new applications for real-time CFD is here assessed, first with the input of 2D geometry using a mouse or stylus, and then via the capture of image based depth map with the Kinect Sensor. The code performance is also assessed and compared to recently published results.

7.1 Stylus/Mouse input

In addition to geometry extraction via the Kinect sensor, mouse/stylus input was also implemented to enable the user to manually input geometry in a highly practical and intuitive manner. **Figure 10** indicates the 2D flow field around a roughly drawn cylinder and demonstrates that it is possible to recreate the features of validated flow fields quickly and easily with the software; in this case using the die injection technique. **Figure 11** shows the 3D flow around a preset cube boundary that can be imported into the fluid via the mouse. Such rapid identification of key flow features via the intuitive input of a stylus has huge potential to bring a new dimension of analysis into industrial engineering design. In the case where detailed flow investigation is required, more accurate geometry can be achieved via import of CAD geometry as with more traditional CFD methods. There are also very useful applications of this technology in the forum of a classroom, where one can directly demonstrate the influence of a certain geometry modification (e.g. a flap on an airfoil).

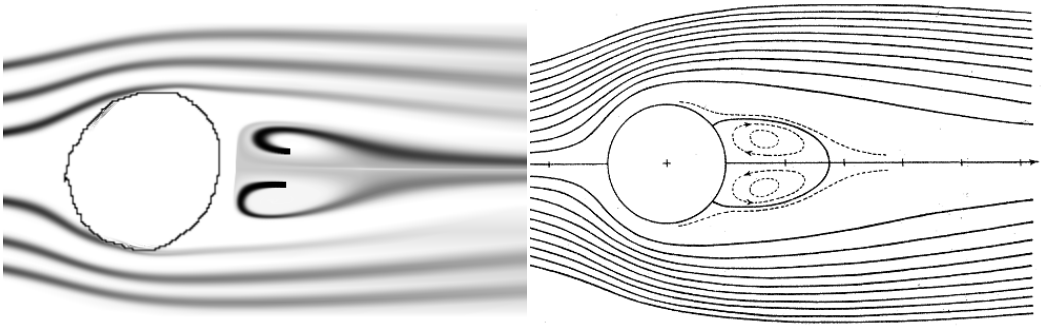


Figure 10: Streamlines (via dye injection) around a hand drawn 2D cylinder (left) and flow calculated in (Thom 1933), Re≈20.

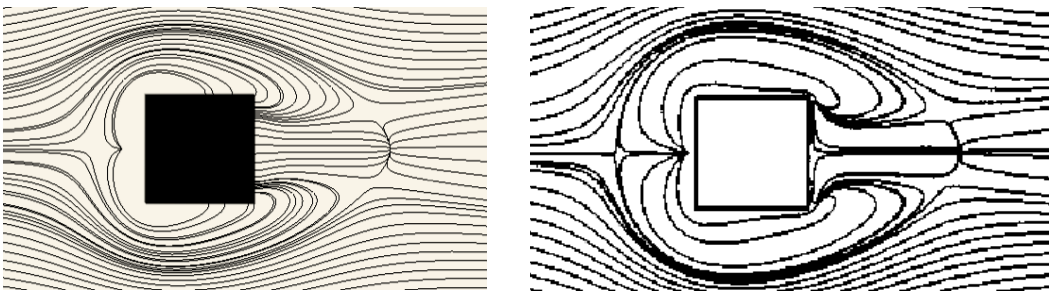


Figure 11: Streamlines (after post-processing) around a preset 3D cube (left) and from (Hwang & Yang 2004), Re≈ 50.

7.2 Image based depth sensor

The Kinect sensor allows the forward-facing geometry of objects to be imported into the fluid domain as solid boundary conditions. A depth image is read from the sensor and a distance threshold is set. Locations within this depth threshold are used to map bounce-back boundary conditions to the LBM fluid simulation. In 2D this results in a flat silhouette, while in 3D the depth information can be used to alter the location of the bounce-back boundaries on the *z*-axis.

The API provided with the Kinect also allows human shapes to be extracted from the depth image, as shown in Figure 13. Regions of low velocity fluid surround the silhouette of the person, whose shape is captured via the Kinect device. LIC method enables the visualisation of vortical structures shed in wake of the person.

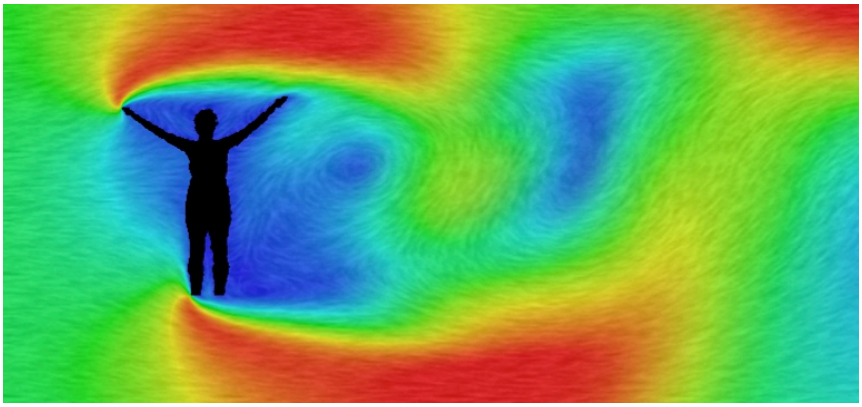


Figure 12: 2D Flow around a human shape imported using a Kinect sensor, with LIC visualization. Colour scale matches velocity magnitude from blue (low) to red (high).

With a single camera, it is of course impossible to see behind the target object, but symmetry objects can be imported by aligning the cut-off location of the depth threshold with the location (on the *z*-axis) of the object's plane of symmetry. A solid boundary is created between this mid-point and the depth given at each point on the surface of the extracted geometry, which is then mirrored behind the cut-off location to create a symmetric 3D shape. Figure 13 displays a symmetric household object captured from the Kinect sensor which has been mirrored in this way. The figure also displays colour contours of the flow velocity, taken from OpenGL output in real-time, where different slices in the z-plane are shown to indicate the 3-dimensionality of the flow.

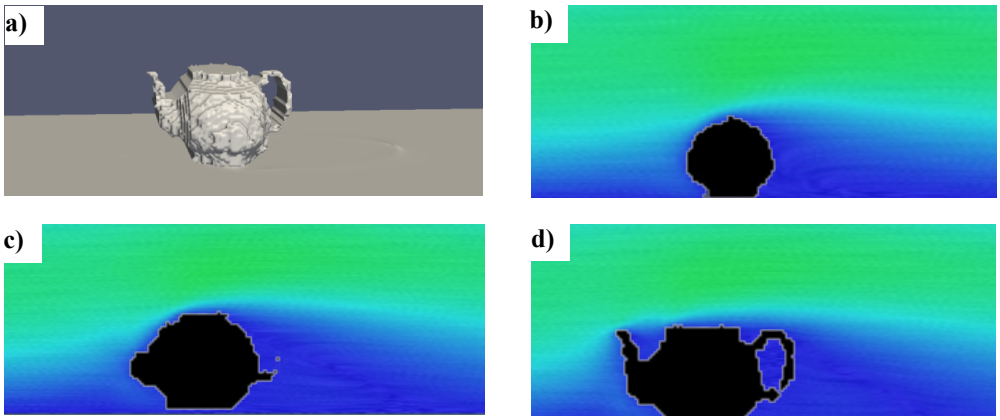


Figure 13: a) household object imported into a Poiseuille flow using the Kinect sensor, and (**b-d)** 3D flow field at different z-planes; from outside **(b)**, to symmetry plane **(d)**

Detailed flow visualization techniques such as streamlines and dye trace described in Section 3 are more complicated to implement in 3D as their path will not be limited to any given plane. The development of suitable and efficient methods to visualise fully 3D flows via OpenGL is a topic of ongoing work by the present authors.

Instead, in order to indicate the potential of the Kinect to extract complex geometry directly into a solver, it is useful to demonstrate the 3D flow in more detail by post-processing a snapshot of the simulation in Figure 14 by using more traditional means.

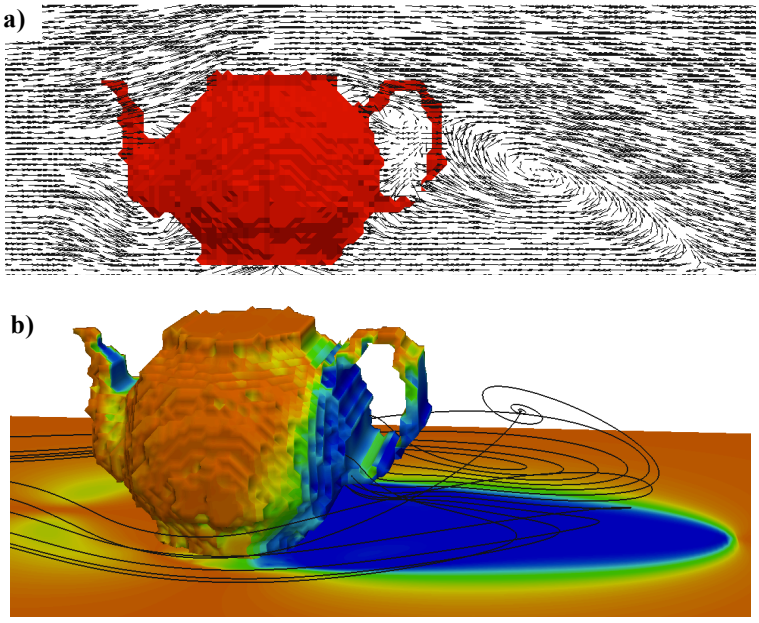


Figure 14: a) Post-processed flow field around object captured using Kinect with symmetry mapping. **b)** Velocity vectors and streamlines in the recirculation region.

Figure 15a shows the velocity vectors in the flow surrounding the target object in the symmetry plane, where several regions of recirculating flow are visible behind the object. Figure 15b displays streamlines within the main flow recirculation region, whilst the object is coloured by values of stream-wise velocity; blue indicates regions where flow has reversed. It should be noted that the somewhat pixelated surface of the object is a consequence of the current native resolution of the Kinect device, and is thus a limitation of the methodology at present. This effect can be reduced to some extent by increasing the size of the target object (where possible!).

7.3 Realtime performance

In order to achieve the effect of 'real-time' simulation, the code must be capable of iterating and outputting to OpenGL at least 24 times per second (to match the frame rate of a typical video output).

Table 2 presents performance metrics for the 2D and 3D solvers at a variety of lattice sizes, in terms of the number of Millions of Lattice Updates Per Second (MLUPS). This is compared to the speed, in MLUPS, when meeting the requirement of visualizing the flow every $1/24^{th}$ of a second. This data is also plotted in Figure 16. At larger domain sizes visualization begins to impact the solver performance, although this is less evident in 3D as (currently) only one z-plane is visualised at a time.

It is worth noting that as the K5000M is for laptop computers, it has a relatively low memory bandwidth (96GB/sec compared with 173GB/sec in K5000 GPU for desktops). Work in (Blair 2012) scales the performance of D3Q19 LBM solvers from (Astorino et al. 2011; Obrecht & Kuznik 2011; Rinaldi et al. 2012) and their own solver with respect to the bandwidth of a GTX295 (223.8GB/sec). If this scaling is repeated for the current results, the code performance is greater than those presented in the aforementioned studies, with a peak scaled performance of 926 MLUPS, as plotted in Figure 17.

Lattice Size	MLUPS	MLUPS with visualization at 24fps
256x256	751.6	751
512x512	782.5	780.6
1024x1024	812.8	805.5
2048x2048	822.4	794.8
32x32x32	324.4	324.4
64x64x64	397.2	397.2
96x96x96	392.2	392.1
128x128x128	389.8	389.6

Table 2: Performance metrics

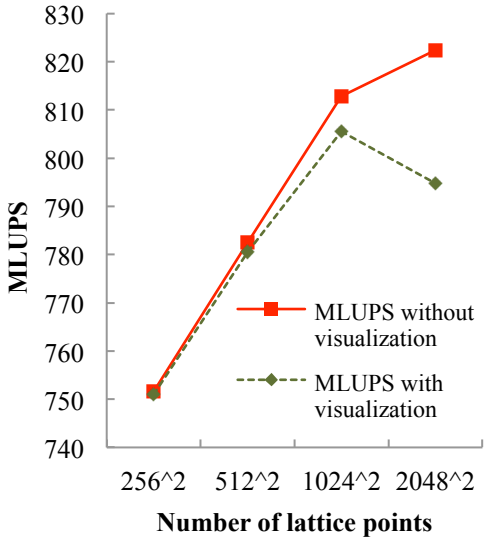


Figure 15: The effect of visualization on performance in 2D.

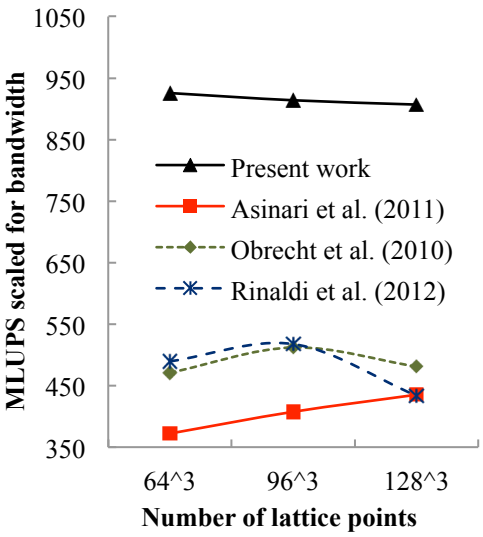


Figure 16: Scaled performance of D3Q19 solver.

8 Conclusions

This work has demonstrated the optimization and validation of 2D and 3D GPU-based Lattice Boltzmann solvers that are sufficiently fast for the cases tested to allow real-time visualisation of the fluid flow. Validation was performed on the Poiseuille flow and the lid driven cavity at three Reynolds numbers, and second order convergence was demonstrated for the boundary conditions used in the 3D solver. Qualitative agreement was demonstrated for flows around objects introduced via mouse or stylus input. A Kinect sensor was used to import both 2D silhouettes and 3D symmetrical objects into the solver, creating the potential for rapid simulation of flow around complex geometry.

Minimizing DRAM accesses on the GPU, and using an altered form of the Lattice Boltzmann algorithm to improve locality leads to a peak performance of 822 MLUPS in 2D and 397 MLUPS in 3D. Future studies will develop 3D visualisation techniques to improve insight into more complicated flows and will also include the use of multiple Kinect sensors to allow fully 3D (asymmetric) geometry to be imported into the solver.

References

Adafruit Industries, 2010. WE HAVE A WINNER - Open Kinect driver(s) released. Available at: http://www.adafruit.com/blog/2010/11/10/we-have-a-winner-open-kinect-drivers-released-winner-will-use-3k-for-more-hacking-plus-an-additional-2k-goes-to-the-eff/ [Accessed January 15, 2013].

Astorino, M., Sagredo, J. & Quarteroni, A., 2011. *A modular lattice Boltzmann solver for GPU computing processors*, Lucerne, Switzerland. Available at: http://mathicse.epfl.ch/files/content/sites/mathicse/files/Mathicse reports 2011/06.2011 MA-AQ.pdf [Accessed February 22, 2013].

Bhatnagar, P. & Gross, E., 1954. A model for collision processes in gases. I. Small amplitude processes in charged and neutral one-component systems. *Physical Review*. Available at: http://prola.aps.org/abstract/PR/v94/i3/p511_1 [Accessed October 3, 2011].

Blair, S.R., 2012. *Lattice Boltzmann Methods For Fluid Structure Interaction*. Naval Postgraduate School. Available at: https://www.google.co.uk/url?sa=t&rct=j&q=&esrc=s&source=web&cd=9&cad=rja&ved=0CFsQFjAI&url=http%3A%2F%2Fcalhoun.nps.edu%2Fpublic%2Fbitstream%2Fhandle%2F10945%2F17325%2F12Sep_Blair_Stuart.pdf%3Fsequence%3D1&ei=1LUkUfSSG4O40QXVs4CoDw&usg=AFQjCNFzBict2XfKq8xOruEQsxzJ-GV1ug&bvm=bv.42661473,d.d2k.

Cabral, B. & Leedom, L., 1993. Imaging vector fields using line integral convolution. *Proceedings of the 20th annual conference on Computer graphics and interactive techniques*. Available at: http://dl.acm.org/citation.cfm?id=166151 [Accessed January 21, 2013].

FUTUREPICTURE, 2010. Kinect Hacking 101: Hack a Powershot A540 for Infrared Sensitivity. Available at: http://www.futurepicture.org/?p=97.

Gallivan, M. a. et al., 1997. An Evaluation of the Bounce Back Boundary Condition for Lattice Boltzmann Simulations. *International Journal for Numerical Methods in Fluids*, 25(3), pp.249–263. Available at: http://doi.wiley.com/10.1002/(SICI)1097-0363(19970815)25:3<249::AID-FLD546>3.3.CO;2-Z.

Ghia, U., Ghia, K. & Shin, C., 1982. High-Re solutions for incompressible flow using the Navier-Stokes equations and a multigrid method• 1. *Journal of computational physics*, 48(3), pp.387–411. Available at: http://linkinghub.elsevier.com/retrieve/pii/0021999182900584 [Accessed January 14, 2011].

Guo, Z., Zheng, C. & Shi, B., 2002. Discrete lattice effects on the forcing term in the lattice Boltzmann method. *Physical Review E*, 65(4), pp.1–6. Available at: http://link.aps.org/doi/10.1103/PhysRevE.65.046308 [Accessed March 1, 2012].

Habich, J. et al., 2011. Performance analysis and optimization strategies for a D3Q19 lattice Boltzmann kernel on nVIDIA GPUs using CUDA. *Advances in Engineering Software*, 42(5), pp.266–272. Available at: http://linkinghub.elsevier.com/retrieve/pii/S0965997810001274 [Accessed September 19, 2011].

He, X. & Luo, L.-S., 1997. Lattice Boltzmann Model for the Incompressible Navier–Stokes Equation. *Journal of Statistical Physics*, 88(3/4), pp.927–944. Available at:

http://www.springerlink.com/openurl.asp?id=doi:10.1023/B:JOSS.0000015179.12689.e4 .

Hecht, M. & Harting, J., 2010. Implementation of on-site velocity boundary conditions for D3Q19 lattice Boltzmann simulations. *Journal of Statistical Mechanics: Theory and Experiment*, 2010(01), p.P01018. Available at: http://stacks.iop.org/1742-5468/2010/i=01/a=P01018?key=crossref.99ac72342da7ece69e08c1882bb3a4ed [Accessed March 15, 2012].

Hwang, J.-Y. & Yang, K.-S., 2004. Numerical study of vortical structures around a wall-mounted cubic obstacle in channel flow. *Physics of Fluids*, 16(7), p.2382. Available at: http://link.aip.org/link/PHFLE6/v16/i7/p2382/s1&Agg=doi [Accessed February 11, 2013].

Jiang, B. & Lin, T.., 1994. Large-scale computation of incompressible viscous flow by least-squares finite element method. *Computer Methods in Applied Mechanics*, 114(3-4), pp.213–231. Available at: http://linkinghub.elsevier.com/retrieve/pii/0045782594901724 [Accessed June 7, 2012].

Khoshelham, K., 2011. Accuracy analysis of kinect depth data. *ISPRS workshop laser scanning*, XXXVIII(August), pp.29–31. Available at: http://www.int-arch-photogramm-remote-sens-spatial-inf-sci.net/XXXVIII-5-W12/133/2011/isprsarchives-XXXVIII-5-W12-133-2011.pdf [Accessed February 22, 2013].

Kuznik, F. et al., 2010. LBM based flow simulation using GPU computing processor. *Computers & Mathematics with Applications*, 59(7), pp.2380–2392. Available at: http://linkinghub.elsevier.com/retrieve/pii/S0898122109006361 [Accessed December 13, 2011].

Liebe, C.C. et al., 2006. Spacecraft Hazard Avoidance Utilizing Structured Light. *2006 IEEE Aerospace Conference*, pp.1–10. Available at: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1655898.

Luebke, D., Humphreys, G. & Res, N., 2007. How gpus work. *Computer*, 40(2), pp.96–100. Available at: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4085637 [Accessed September 20, 2010].

Microsoft, 2012. Kinect for Windows. Available at: http://www.microsoft.com/en-us/kinectforwindows/ [Accessed January 17, 2013].

Mitchell, R., 2010. PrimeSense releases open source drivers, middleware that work with Kinect. Available at: http://www.joystiq.com/2010/12/10/primesense-releases-open-source-drivers-middleware-for-kinect/ [Accessed January 15, 2013].

Ning, Y. & Premnath, K.N., 2012. Numerical Study of the Properties of the Central Moment Lattice Boltzmann Method. *ArXiv e-prints*. Available at: http://adsabs.harvard.edu/abs/2012arXiv1202.6351N [Accessed February 8, 2013].

NVIDIA, 2010. *NVIDIA CUDA C Programming Guide Version 3.2*, Available at: http://developer.nvidia.com/cuda-toolkit-32-downloads.

NVIDIA, 2012. NVIDIA GeForce GTX 680 The fastest, most efficient GPU ever built. Available at: http://www.geforce.com/Active/en_US/en_US/pdf/GeForce-GTX-680-Whitepaper-FINAL.pdf [Accessed February 1, 2013].

Obrecht, C. & Kuznik, F., 2011. A new approach to the lattice Boltzmann method for graphics processing units. *Computers & Mathematics …*. Available at: http://www.sciencedirect.com/science/article/pii/S089812211000091X [Accessed February 22, 2013].

Pharr, M. & Fernando, R., 2005. *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*, Addison-Wesley. Available at: Programming Techniques for High-Performance Graphics and General-Purpose Computation.

PrimeSense, 2013. Our Full 3D Sensing Solution. Available at: http://www.primesense.com/solutions/technology/ [Accessed January 17, 2013].

Qian, Y.H., D'Humières, D. & Lallemand, P., 1992. Lattice BGK Models for Navier-Stokes Equation. *Europhysics Letters (EPL)*, 17(6), pp.479–484. Available at: http://stacks.iop.org/0295-5075/17/i=6/a=001 [Accessed January 16, 2013].

Rinaldi, P.R. et al., 2012. A Lattice-Boltzmann solver for 3D fluid simulation on GPU. *Simulation Modelling Practice and Theory*, 25, pp.163–171. Available at: http://linkinghub.elsevier.com/retrieve/pii/S1569190X1200038X [Accessed April 13, 2012].

Thom, A., 1933. The flow past circular cylinders at low speeds. *Proceedings of the Royal Society of London. Series A, …*, 141(845), pp.651–669. Available at: http://www.jstor.org/stable/10.2307/96175 [Accessed January 25, 2013].

Wellein, G. et al., 2006. On the single processor performance of simple lattice Boltzmann kernels. *Computers & Fluids*, 35(8-9), pp.910–919. Available at: http://linkinghub.elsevier.com/retrieve/pii/S0045793005001532 [Accessed November 8, 2012].

Wijk, J. Van, 2002. Image based flow visualization. *ACM Transactions on Graphics (TOG)*. Available at: http://dl.acm.org/citation.cfm?id=566646 [Accessed January 21, 2013].

Van Wijk, J.J., 1991. Spot noise texture synthesis for data visualization. *ACM SIGGRAPH Computer Graphics*, 25(4), pp.309–318. Available at: http://portal.acm.org/citation.cfm?doid=127719.122751.

Zalevsky, Z. et al., 2007. METHOD AND SYSTEM FOR OBJECT RECONSTRUCTION. Available at: http://patentscope.wipo.int/search/en/detail.jsf?docId=WO2007043036&recNum=1&maxRec=&office=&prevFilter=&sortOption=&queryString=&tab=PCT+Biblio.

Zou, Q., He, X. & Introduction, I., 1997. On pressure and velocity boundary conditions for the lattice Boltzmann BGK model. *Physics of Fluids*, 9(6), p.1591. Available at: http://link.aip.org/link/PHFLE6/v9/i6/p1591/s1&Agg=doi.