Computer Physics Communications 185 (2014) 2566–2574

Contents lists available at ScienceDirect

Computer Physics Communications

journal homepage: www.elsevier.com/locate/cpc

Memory transfer optimization for a lattice Boltzmann solver on Kepler architecture nVidia GPUs

Mark J. Mawson *, Alistair J. Revell

University of Manchester, Oxford Road, Manchester, Greater Manchester, M13 9PL, UK

ARTICLE INFO

Article history:
Received 16 September 2013
Received in revised form
2 June 2014
Accepted 3 June 2014
Available online 12 June 2014

Keywords:
GPGPU
Lattice Boltzmann
Computational fluid dynamics
CUDA

ABSTRACT

The Lattice Boltzmann method (LBM) for solving fluid flow is naturally well suited to an efficient implementation for massively parallel computing, due to the prevalence of local operations in the algorithm. This paper presents and analyses the performance of a 3D lattice Boltzmann solver, optimized for third generation nVidia GPU hardware, also known as 'Kepler'. We provide a review of previous optimization strategies and analyse data read/write times for different memory types. In LBM, the time propagation step (known as streaming), involves shifting data to adjacent locations and is central to parallel performance; here we examine three approaches which make use of different hardware options. Two of which make use of 'performance enhancing' features of the GPU; shared memory and the new *shuffle* instruction found in Kepler based GPUs. These are compared to a standard transfer of data which relies instead on optimized storage to increase coalesced access. It is shown that the more simple approach is most efficient; since the need for large numbers of registers per thread in LBM limits the block size and thus the efficiency of these special features is reduced. Detailed results are obtained for a D3Q19 LBM solver, which is benchmarked on nVidia K5000M and K20C GPUs. In the latter case the use of a read-only data cache is explored, and peak performance of over 1036 Million Lattice Updates Per Second (MLUPS) is achieved. The appearance of a periodic bottleneck in the solver performance is also reported, believed to be hardware related; spikes in iteration-time occur with a frequency of around 11 Hz for both GPUs, independent of the size of the problem.

1. Introduction

The implementation of Lattice Boltzmann method (LBM) solvers on Graphics Processing Units (GPUs) is becoming increasingly popular due to the intrinsic parallelizable nature of the algorithm. A growing literature exists in this area, though with frequent hardware changes there is a need to constantly review the means of obtaining optimal performance. As a derivative of Lattice Gas Cellular Automata (LGCA), LBM defines macroscopic flow as the collective behaviour of underlying microscopic interactions [1]. The LBM improves upon LGCA by describing each point in space using a mesoscopic particle distribution function rather than an individual particle, reducing statistical noise within the method. LBM has been used extensively in the literature over the past decade and is now regarded as a powerful and efficient alternative to the classical Navier–Stokes solvers (see [1,2] for a complete overview). Despite early suggestions in the literature to the contrary, Shan et al. [3] formally demonstrated how LBM can return exact Navier–Stokes even for high Reynolds and high Mach number flows.

Algorithmically, the method consists of a local relaxation ('collide') and a linear advection ('stream') of a number of discrete components of the distribution function, rendering the method highly parallelizable. Implementation of the LBM on GPUs can be traced back to 2003, when Li et al. [4] mapped a 2D LBM algorithm to texture and rasterization operations within the graphics pipeline. Since then, a variety of both two and three dimensional models have been implemented and GPU based LBM algorithms have been proposed for a range of applications; e.g. free surface [5], thermal [6], and biomedical [7].

Another early attempt was made by Ryoo et al. [8], who tested a CUDA port of a simple LBM code from the SPEC CPU2006 benchmark [9]. This was followed by a more in-depth optimization reported by Tölke and Krafczyk [10], who implemented a 3D model with a reduced 13 component distribution function on G80 generation GPUs, specifically designed to increase maximum throughput. They used shared memory to avoid costly misaligned access to the RAM of the GPU when performing advection of the

* Corresponding author.
E-mail addresses: mark.mawson@postgrad.manchester.ac.uk, mark.mawson@stfc.ac.uk (M.J. Mawson).

http://dx.doi.org/10.1016/j.cpc.2014.06.003
0010-4655/

distribution function (although in general, using less components of f_i will reduce accuracy). A more complex split propagation method was proposed by [11], in which the data is first shifted along the contiguous dimension within shared memory for each block, before the perpendicular shifts are performed in global memory.[1] This approach demonstrated high levels of efficiency, but necessitated careful handling of data entering/leaving each block. Habich et al. [12] extended this work to the D3Q19 model (the same model presented in this work).

Obrecht et al. [13] identified that, contrary to previous attempts to avoid misalignment at all cost, the cost of a misaligned access in shared memory was actually similar to that caused by a global memory exchange; thus they proceeded to investigate the potential for performance improvement brought about by avoiding the use of shared memory altogether. Indeed, previous advances in nVidia hardware, first from compute capability 1.0–1.3, and then on to 2.0, substantially improved the efficiency of misaligned memory transactions; this had the important consequence that the use of shared memory was no longer so crucial. Furthermore, [14] reported the cost of a misaligned read to be less than a misaligned write; an observation leads one to prefer the 'pull' algorithm to the 'push', as discussed in Section 3. A peak performance of 516 Million Lattice Updates Per Second (MLUPS)[2] was reported with a maximum throughput of 86%. In addition, it is noted that previous works using Shared Memory were highly optimized, and thus obtaining substantial extra performance would not be trivial. More importantly, the implementation of more complex code to handle e.g. multiple distribution functions or extra body force terms would only be possible with a significant reduction of performance. Indeed, the present work is building towards an efficient GPU implementation of the Immersed Boundary Method with LBM reported in [15], and thus this reasoning is of high relevance to our work (see [16]).

A comprehensive series of further work by Obrecht et al. [13,17,18] focused on compute 2.x capable hardware in their development of a multi-GPU implementation of a Hybrid thermal LBM scheme based on D3Q19, and did not make use of shared memory. Later, Habich et al. [19], also presented implementations for 'Fermi' generation GPUs without the use of shared memory.

In the present work, particular attention is paid to a comparison of three methods of performing the advection operation; the first which performs the propagation directly in the GPUs RAM (DRAM), a second that utilizes a shared memory space as an intermediate buffer and a third, new, method that performs the propagation locally within a group of 32 threads without using any intermediate memory using the *shuffle* instruction new to GPUs based on the Kepler architecture [20]. The results of this comparison are then used to implement an efficient 3D LBM solver on first and second generation Kepler GPUs (compute capabilities 3 and 3.5 respectively). In the case of second generation Kepler GPUs a read-only cache is also enabled to provide a small, but measurable, improvement in performance. In the following sections the architecture and programming model for CUDA based GPUs is introduced, before the mathematical description and specific form of the LBM used is given. Analysis of the three propagation techniques is then performed, along with consideration of other implementation aspects and estimation of the maximum achievable performance of a LBM solver based on memory requirements.

2. GPU computing

Modern GPUs use a Unified Shader Architecture [23], in which blocks of processing cores (capable of performing operations within all parts of the graphics pipeline) are favoured 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 [24].

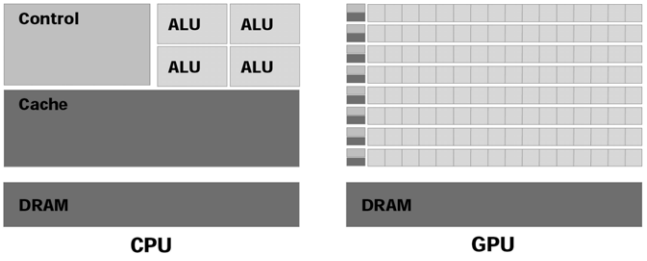


Fig. 1. Typical GPU and CPU architectures [21].

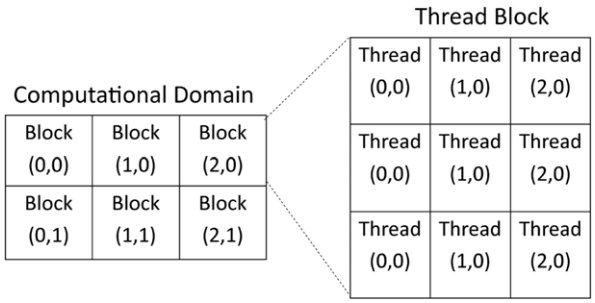


Fig. 2. Blocks & Threads in CUDA [22].

The architecture of a generic Unified Shader based GPU is shown alongside that of a generic CPU in Fig. 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:

1. GPUs sacrifice larger amounts of cache and control units for a greater number of processing cores.
2. 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 generated 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.

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 Fig. 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. Fig. 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.

[1] This paper also provides a useful introduction to LBM implementation in CUDA.

[2] This is a common performance measure based on the number of lattice points that can be updated every second.

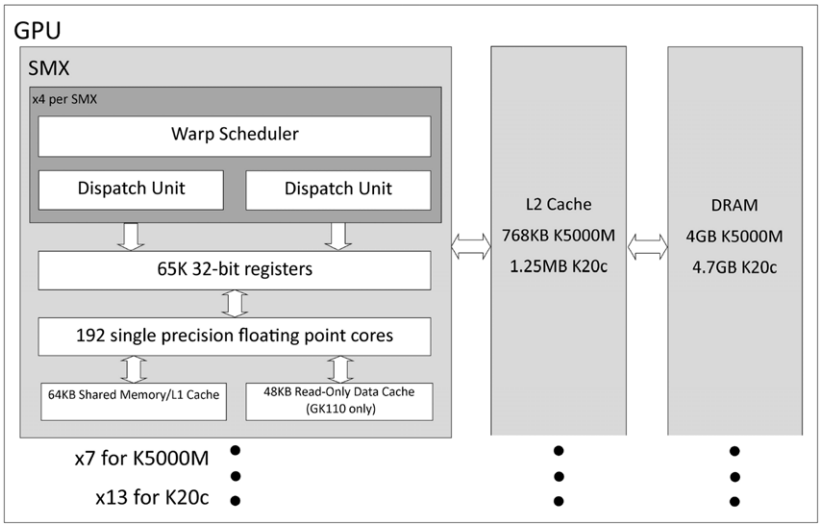


Fig. 3. Typical Kepler GPU architecture.

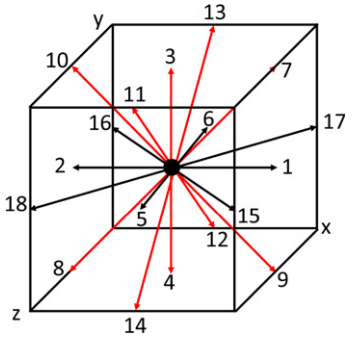


Fig. 4. The D3Q19 lattice.

2.2. nVidia Kepler architecture GPUs

In this paper two Kepler architecture GPUs are tested; the K5000M and the K20c. Details of the general hardware layout for both devices is provided in Fig. 3. The K5000M is based on the first generation GK104 Kepler architecture, and as such is limited in it's double precision performance, with a ratio of 24:1 between single and double precision peak performance. Processing cores are grouped into blocks of 192, known as Streaming Multiprocessors (SMX), in order to share access to 64 KB of configurable cache/shared memory, and four instruction schedulers capable of dispatching two instructions per clock cycle.[3] Seven SMXs make up the K5000M, and these SMXs share access to 512 KB of L2 cache and 4 GB of DRAM. In the configuration used in this paper, the K5000M is clocked at 601 MHz, giving a theoretical peak single precision performance of 1.62 TFLOPS. Peak DRAM bandwidth was measured as 64.96 GB/s using the benchmarking program included in the CUDA SDK.

The K20c is based on the newer GK110 architecture, which is largely the same as a GK104, the most significant difference being extra double precision units within each SMX to improve the single/double precision performance ratio to 3:1. A K20c contains 13 SMXs, which share access to 1.25 MB of L2 cache and 4.7 GB of DRAM. The theoretical peak single precision performance of the configuration used in this work is 3.5 TFLOPS, and measured bandwidth was 157.89 GB/s.

[3] Provided the instructions are from the same warp and independent in nature.

2.3. 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:

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

where x are the space coordinates, and e is the particle velocity. The collision operator Ω is simplified using the 'BGK' single time relaxation approach found in [25], in which context, it is assumed that local particle distributions relax to an equilibrium state, $f^{(eq)}$ in time τ:

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

The discretized form of Eq. (1) is obtained via Taylor series expansion following [26], as shown in Eq. (3), where f_i refers to the discrete directions of f. The dimensionality of the model and spatial discretization 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 D3Q19 model is used, in which the discrete velocity is defined according to Eq. (4) and is visualized in Fig. 4. Since spatial and temporal discretization in the lattice are set to unity, the lattice speed $c = \Delta x/\Delta t = 1$.

$$f_i(\mathbf{x} + \mathbf{e}_i \Delta t, t + \Delta t) = f_i(\mathbf{x}, t) + \frac{1}{\tau}\left[f_i^{eq}(\mathbf{x}, t) - f_i(\mathbf{x}, t)\right] \tag{3}$$

$$e_i = c \times \begin{pmatrix} 0 & 1 & -1 & 0 & 0 & 0 & 0 & 1 & -1 & 1 & -1 & 0 & 0 & 0 & 0 & 1 & -1 & -1 & 1 \\ 0 & 0 & 0 & 1 & -1 & 0 & 0 & 1 & -1 & -1 & 1 & 1 & -1 & 1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 & 1 & -1 & -1 & 1 & 1 & -1 & 1 & -1 \end{pmatrix}$$
$$(i = 0, 1, \ldots, 19). \tag{4}$$

The equilibrium function $f^{(eq)}(x, t)$ can be obtained by Taylor series expansion of the Maxwell–Boltzmann equilibrium distribution [27]:

$$f_i^{eq} = \rho\omega_i\left[1 + \frac{\mathbf{e}_i \cdot \mathbf{u}}{c_s^2} + \frac{(\mathbf{e}_i \cdot \mathbf{u})^2}{2c_s^4} - \frac{\mathbf{u}^2}{2c_s^2}\right]. \tag{5}$$

In Eq. (5), c_s is the speed of sound $c_s = 1/\sqrt{3}$ and the coefficients of ω_i are $\omega_0 = 1/3$, $\omega_i = 1/18$, $i = 1..6$ and $\omega_i = 1/36$, $i = 7..19$.

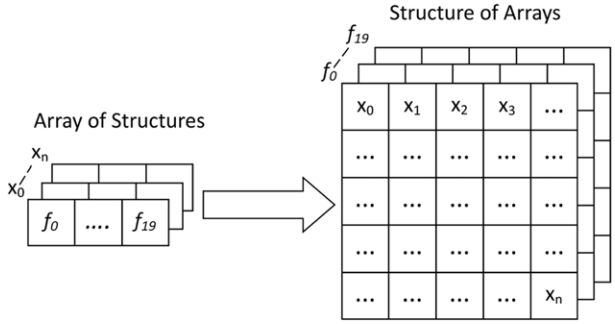


Fig. 5. Arrays of structures and structures of arrays.

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

$$\rho = \sum_i f_i \tag{6}$$

$$\rho \mathbf{u} = \sum_i \mathbf{e}_i f_i. \tag{7}$$

The multi-scale expansion of Eq. (3) neglecting terms of $\mathbf{O}(\varepsilon M^2)$ and using Eqs. (6) and (7) returns the Navier–Stokes equations to second order accuracy [26], allowing the LBM to be used for fluid simulations.

3. Implementation

3.1. Memory arrangement

The present solver is parallelized such that one thread will perform the complete LBM algorithm at one spatial location $f(x)$ in the fluid domain. Each thread stores values of $f(x)$, ρ, $\mathbf{u}$ and information about whether or not the current location is a boundary in a struct within register space to minimize 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. This is extended to combining and flattening the components of f into a single array, such that $f_i(x)$ is addressed as $f[i * Nx * Ny * Nz + z * Ny * Nx + y * Nx + x]$. Storing f in order of i and then by spatial coordinates will cause neighbouring threads within a warp to access contiguous memory in f. If these accesses are aligned within a 128-byte segment (see Section 3.3.2) the data transactions can be grouped into a single larger transaction; i.e. resulting in a coalesced access.

The ordering of the data in f is known as storing in a 'Structure of Arrays' (SoA) format; without the collapsing of the different components of f into a single array this can be seen as a structure containing 19 arrays, each one corresponding to all of the spatial locations of one component of f. The opposite 'Array of Structures' (AoS) arrangement is shown in Fig. 5 for clarity, which corresponds to a single array with one element per spatial location, each containing a data structure to store the 19 components of f. Once f has been read into a core, an analogy can be drawn between this format and the 'array' of GPU cores, each containing their own local structure. While AoS is shown to be preferable for serial CPU implementations [28], SoA is necessary to improve coalesced access to global memory with GPU versions.

Without the presence of macroscopic values, a single point in the lattice requires 19 loads from global memory, and 19 stores back to global memory during an iteration of the LBM algorithm.[4] In single precision this yields 152 bytes of data to be transferred for each lattice point. Using the measured bandwidth of 65 GB/s for the K5000M from Section 2.2, the theoretical limit is 459 MLUPS. For the K20c this limit is 1115 MLUPS. If macroscopic values are required, then an extra four storage operations are needed and the maximum theoretical performance drops to 415 MLUPS (1009 MLUPS for the K20c).

3.2. Independent LBM algorithm

The conventional LBM is typically broken down into several steps as described in Algorithm 1, which is typically the form taken for CPU implementations. This algorithm poses some locality problems if it is to be used in a highly parallel fashion. If a thread is launched for each location $f(x)$ then the non-local operation $f_i(\mathbf{x} + \mathbf{e}_i \Delta t, t + \Delta t) = f_i(\mathbf{x}, t + \Delta t)$ will require a synchronization across the domain (as shown) before the boundary conditions, ρ and $\mathbf{u}$ can be calculated.

Algorithm 1 The Conventional LBM

```
Kernel 1
for all Locations x in f_i(x, t) do
    for all i do
        Read f_i(x, t) from memory to a local store fLocal_i.
    end for
    Calculate f^eq using Eq. (5).
    Perform Collision fLocal_i = fLocal_i(x, t) + (Δt/τ)(f^(eq)(x, t) − f(x, t)).
    Stream fLocal_i to the location f_i(x + e_i Δt, t + Δt).
end for
Synchronization across f_i(x + e_i Δt, t + Δt)
Kernel 2
for all Locations x in f_i(x, t) do
    for all i do
        Read f_i(x, t) from memory to a local store fLocal_i.
    end for
    Apply boundary conditions.
    Calculate ρ and u using Eqs. (6) and (7).
end for
```

Instead, one of the two re-orderings presented in [29] can be used. These are described as 'push' or 'pull' algorithms, depending on whether the streaming operation (which causes misaligned access to DRAM, see Section 3.3.2) occurs at the end ($f_i(\mathbf{x} + \mathbf{e}_i \Delta t, t + \Delta t) = f_i(\mathbf{x}, t + \Delta t)$) or beginning ($f_i(\mathbf{x}, t) = f_i(x - \mathbf{e}_i \Delta t, t - \Delta t)$) of the algorithm. Both algorithms remove the need for an additional synchronization by placing the synchronization point at the end of an iteration, where a synchronization implicitly occurs as the loop (or kernel when implemented in CUDA) across the domain exits. This also eliminates the requirement to store ρ and $\mathbf{u}$ in DRAM, unless they are required for post-processing, as they are only used in enforcing the boundary conditions and the calculation of $f^{(eq)}$.

3.3. Read/write memory speed

As stated in the Introduction, [13] examined the cost of misaligned reads and writes for compute 1.3 devices and reported that the former were more efficient than the latter; motivating their preference for the 'pull' algorithm. In what follows, we provide results of a similar experiment for the more recent compute 3.0 and 3.5 devices. Aligned and misaligned read and writes to several large vectors were performed to mimic the behaviour of the 'push' and 'pull' algorithms, and the DRAM bandwidth achieved is measured.

Multiple vectors are used to provide improved Instruction Level Parallelism (ILP), which is a strategy to mask some memory latency by allowing a single thread to launch several independent memory

[4] The stores are actually performed on a redundant copy of f to ensure data dependency is not violated, at the end of each timestep the pointers to the original and redundant copy of f are swapped.

Algorithm 2 The Push LBM Iteration

for all Locations $\mathbf{x}$ in $f_i(\mathbf{x}, t)$ **do**
  for all i **do**
    Create a local copy of $f_i(\mathbf{x}, t)$
  end for
  Apply boundary conditions
  Calculate ρ and $\mathbf{u}$ using Eqs. (6) and (7).
  for all i **do**
    Calculate f^{eq} using Eq. (5).
    Perform Collision - $fLocal_i = fLocal_i(\mathbf{x}, t) + \frac{\Delta t}{\tau}\left(f^{(eq)}(\mathbf{x}, t) - f(\mathbf{x}, t)\right)$.
    Stream local copies of f_i to their location $f(\mathbf{x} + \mathbf{e}_i, t + 1)$
  end for
end for

Algorithm 3 The Pull LBM Iteration

for all Locations $\mathbf{x}$ in $f_i(\mathbf{x}, t)$ **do**
  for all i **do**
    Stream to $fLocal_i$ from the location $f_i(\mathbf{x} - \mathbf{e}_i \Delta t, t - \Delta t)$.
  end for
  Apply boundary conditions.
  Calculate ρ and $\mathbf{u}$ using Eqs. (6) and (7).
  Calculate f^{eq} using Eq. (5).
  for all i **do**
    Perform Collision $fLocal_i = fLocal_i(\mathbf{x}, t) + \frac{\Delta t}{\tau}\left(f^{(eq)}(\mathbf{x}, t) - f(\mathbf{x}, t)\right)$.
  end for
end for

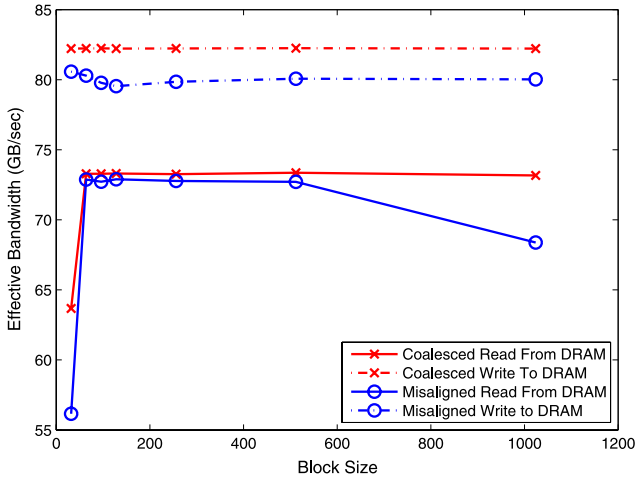


Fig. 6. Effective bandwidth on K5000m for aligned/misaligned reads and writes.

requests before previous requests have returned [30]. This helps make use of the dual-instruction dispatching feature of each warp scheduler; two instructions can only be dispatched in a single clock cycle only if they are from the same warp and the instructions are independent. In the full implementation of the LBM solver ILP is used across the various components of f_i when streaming, and also when performing the collision operation. For compute 3.5 devices the read-only data cache (previously only accessible through the use of textures) is also considered.

Fig. 6 provides a comparison of read and write bandwidth on the K5000m, for both coalesced and misaligned access to data. Fig. 7 displays the corresponding results for the K20c, in addition to times for Read-Only data access. Results show that misaligned reads incur a smaller penalty than misaligned writes when compared with their aligned access counterparts on the K5000m for reasonably large blocks that are small enough to satisfy the register requirements in Section 3.3.2, and are therefore more efficient (99% of aligned access bandwidth versus 96%). On the K20c the read-only cache is required to maintain the efficiency

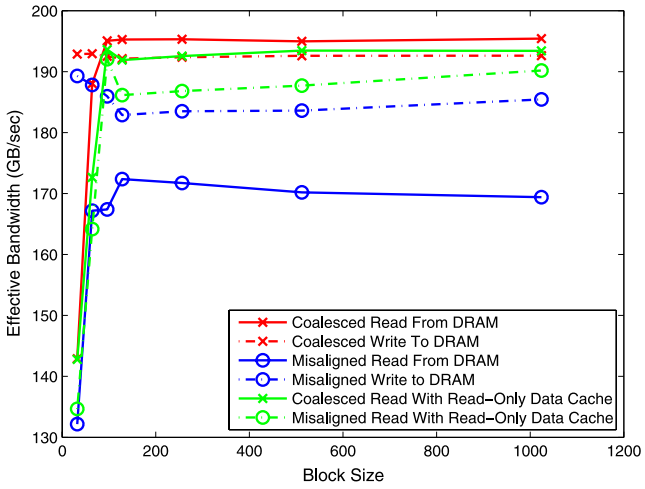


Fig. 7. Effective bandwidth on K20c for aligned/misaligned reads and writes.

of misaligned reads, efficiency drops to 88% without it. The use of read-only cache does have a small detrimental effect on the performance of aligned reading from memory due to the overhead of passing through extra hardware. Overall, it is clear that the pull algorithm is preferable.

The re-ordered *pull* algorithm is used, following [29,31] and shown in Algorithm 3. The pull algorithm takes it's name from the operation $f_i(\mathbf{x}, t) = f_i(\mathbf{x} - \mathbf{e}_i \Delta t, t - \Delta t)$, which is used instead of $f_i(\mathbf{x} + \mathbf{e}_i \Delta t, t) = f_i(\mathbf{x}, t)$; i.e. data is loaded directly into it's new location.

3.3.1. Register usage for LBM

A single SMX in compute 3.x devices contains 65 536 registers, and so the trade off between block size and grid size is best understood as follows:

$$\frac{\text{registers}}{\text{thread}} \times \frac{\text{threads}}{\text{block}} \times \frac{\text{blocks}}{\text{SMX}} \leq 65\,536.$$

The D3Q19 solver presented in this chapter is uses approximately 45 registers/thread, depending on the boundary conditions imposed.

If a block size of 1024 were used, a total of ∼45K threads would be needed for each block, and thus only one block could be launched, causing the remaining 20K registers to go unused. The authors in [17] recommend block sizes of no more than 256 threads for this reason.

3.3.2. Access patterns for the streaming operation

Access to global memory is most efficient when threads within a warp access data in the same 128-byte segment, when this occurs the 32 requests from each thread in the warp are coalesced into a single request. Within the streaming operation, alignment to a 128-byte segment is dependent on the value of the x component of $\mathbf{e}_i$. When it is zero these coalesced accesses are guaranteed, as propagation of values in the y and z directions move access to a different segment without any data misalignment. When the x component of $\mathbf{e}_i$ has a non-zero value, misaligned access to the segments will occur, and two memory requests will be required per warp; one to load values from 31 addresses in the same segment and a second to load a value from the previous (if $\mathbf{e}_{i_x} = 1$) or next (if $\mathbf{e}_{i_x} = -1$) segment (Fig. 8).

Current work on LBM solvers for GPUs either accepts this extra memory transaction (see, for example [17]), or utilizes shared memory to perform propagation in the x direction (Fig. 9), either performing global memory accesses to propagate values between

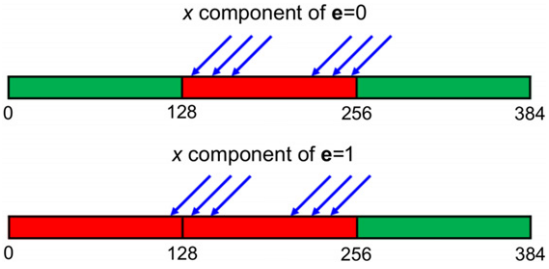


Fig. 8. Aligned and misaligned access to DRAM in the streaming operation.

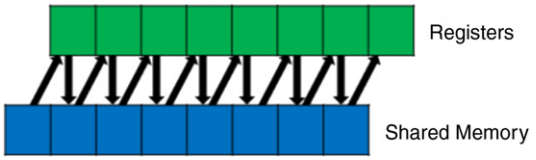


Fig. 9. Using shared memory to propagate values.

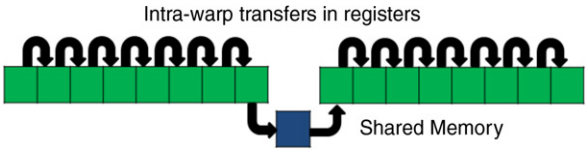


Fig. 10. Using shuffle instruction to propagate values.

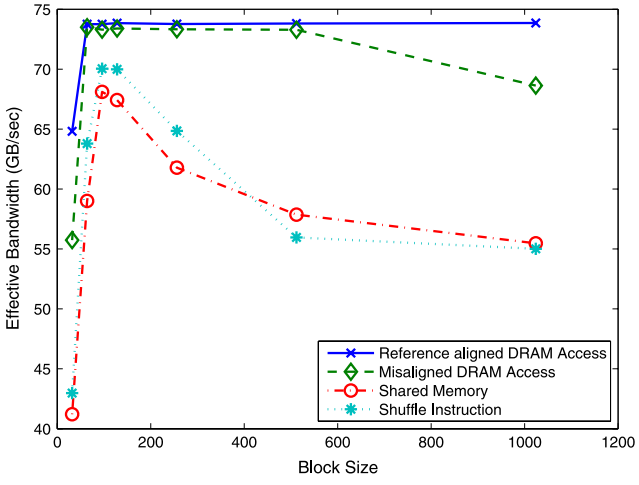


Fig. 11. Effective bandwidth on K5000m for offest-by-one DRAM reads.

blocks or matching the x dimension of the domain to the size of a block [31,32].

New instructions present within Kepler architecture GPUs allow for propagation within a warp without the use of shared memory, and only require shared memory to propagate values between warps (See Fig. 10). In order to determine the most efficient method for use within an LBM solver, the performance tests in Section 3.2 are repeated for misaligned reading using the three different methods. Figs. 11 and 12 present this comparison for both hardware derivatives, in which aligned reads from DRAM are also shown to provide a reference point, as they represent optimal memory behaviour. For both GPUs simple misalignment in DRAM is observed to be the more efficient method, achieving at least a 7.6% improvement over shared memory and 4.7% improvement over shuffle memory on a K5000m, rising to 17.6% and 17.1% on a K20c. The extra synchronization and intermediate registers required for the use of shared memory or the *shuffle* instruction lowers the achieved bandwidth. Still, for reasonable block sizes (block

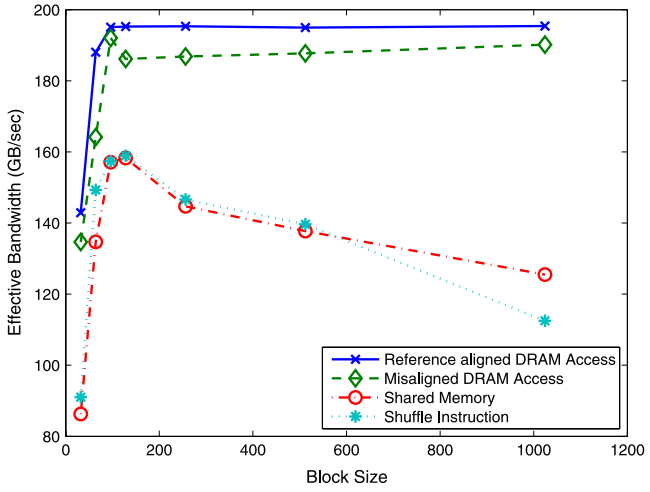


Fig. 12. Effective bandwidth on K20c for offest-by-one DRAM reads.

sizes much above 256 would be impossible to implement in a full LBM code due to the number of registers consumed) the *shuffle* instruction slightly outperforms the use of shared memory alone.

4. Validation of the LBM code

To validate the LBM solver, a lid driven cavity case was performed at three Reynolds numbers {100, 400, 1000} and compared against reference data from [33]. In each case a cubic domain is created with a non-zero x velocity on the top wall and non-slip conditions on every other wall. The boundary conditions from [34] are used to create the stationary and moving boundaries, and scaling is controlled according to

$$u = \frac{(\tau - 0.5)Re}{3L}. \tag{8}$$

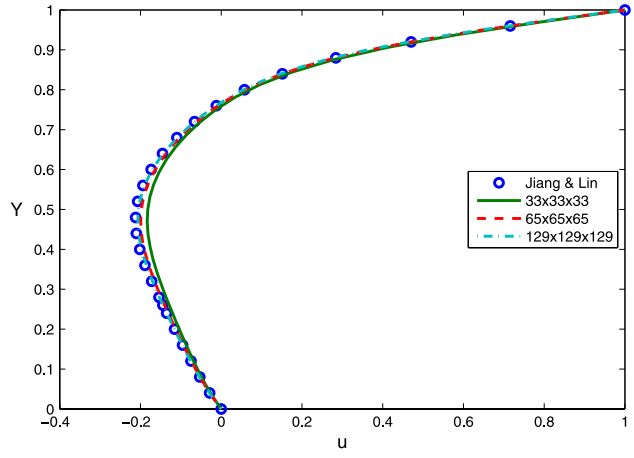
Fig. 13 displays profiles of velocity extracted from the centre of the 3D domain and the resolutions used. For higher Reynolds number computations the domain sizes are increased to reflect the higher levels of resolution required. In all cases the results demonstrate a convergence for subsequent increase in lattice size and reference results are shown to be reproduced.

5. Performance

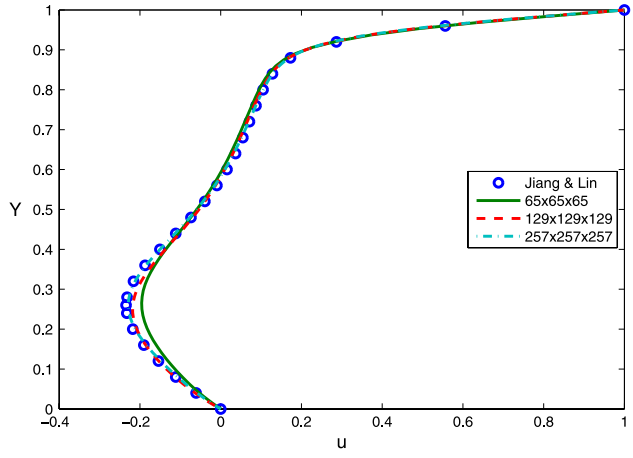
One hundred iterations of the lid driven cavity test case were performed over a variety of block sizes for domains up to a size of 256^2 in single precision on K5000m and K20c GPUs, with the domain size limited by DRAM size. The blocks are kept as x dimension dominant as possible to facilitate improved cache use, with the obvious exception that threads must not be allocated to indices outside of the computational domain. In this event multiple rows of the highest common denominator between the domain and the desired block size are used. Tables 1 and 2 show the mean performance and standard deviation of the 100 iterations in MLUPS. Peak performance is 420 MLUPS on the K5000m and 1036 MLUPS on the K20c.

Fig. 14 displays the performance of the present LBM solver compared to implementations on previous hardware generations found in work by Obrecht et al. [14] (compute 1.3 GeForce GTX 295), Rinaldi et al. [31] (compute 1.3 GeForce GTX 260) and Astorino et al. [32] (compute 2.0 GeForce GTX 480), which are also tabulated in Table 3. All results are reported for single precision calculations and include the calculation of boundary condition values.

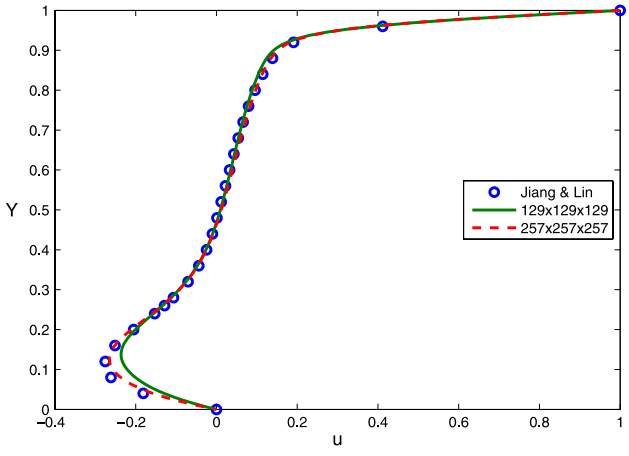
Fig. 15 scales the performance relative to the measured bandwidth of the device (Table 4 also presents results from Astorino



(a) $Re = 100$.



(b) $Re = 400$.



(c) $Re = 1000$.

Fig. 13. Lid driven cavity validation.

Table 1
Mean performance and σ in MLUPS for the 3D LBM solver on K5000m.

Domain size							
Block size	64^3	96^3	128^3	160^3	192^3	224^3	256^3
32	321.74 0.64	331.07 0.21	344.9 0.1	343.37 0.05	346.63 0.03	348.17 0.08	358.02 0.03
64	419.56 2.64	416.49 1.31	413.47 1.3	401.63 0.91	413.29 0.85	392.25 0.72	412.17 0.67
96	416.09 2.29	418.21 1.36	411.79 1.18	400.82 0.94	413.84 0.94	391.49 0.72	409.74 0.66
128	***419.93*** 2.49	416.07 1.47	415.15 1.37	400.76 0.88	413.51 0.86	391.73 0.71	413.01 0.7
160	416.18 2.38	416 1.31	412.04 1.17	***402.63*** 0.89	411.97 0.93	391.68 0.71	409.59 0.61
192	420.38 3.17	***418.89*** 1.35	413.81 1.46	400.33 0.89	***415.62*** 0.96	392.66 0.71	410.82 0.7
224	415.23 2.62	415.96 1.3	412.41 1.29	400.68 0.89	412.06 0.84	390.92 0.71	409.93 0.64
256	420.36 2.37	416.6 1.28	***416.27*** 1.57	400.75 0.93	414.52 0.96	***394.93*** 0.74	***414.47*** 0.7

et al. scaled by theoretical bandwidth) and it can be seen that the present implementation is as efficient (slightly more in the case of the K5000M) than the current best implementation found elsewhere in literature. It is worth noting that, following the examination of the use of shared memory in Section 3.3.2, the implementations found in Astorino et al. [32] and Rinaldi et al. [31] both make use of shared memory.

The L2 cache is used exclusively within the LBM solver when misaligned accesses in the stream operation occur, every other operation is fully coalesced and has no re-use of fetched data. The hit rate for L2 cache will therefore depend on the ratio of misaligned to aligned accesses in the stream operation, which will be constant, except for misaligned accesses that would fall outside the boundaries of the domain, and are therefore not performed.

Table 2
Mean performance and σ in MLUPS for the 3D LBM solver on K20c.

Domain size							
Block size	64^3	96^3	128^3	160^3	192^3	224^3	256^3
32	674.75 1.32	701.66 0.61	725.22 1.21	714.54 0.23	723.45 0.3	722.35 0.14	739.12 0.11
64	909.79 3.48	1006.73 2.02	979.22 1.07	***937.35*** 1.92	***1036.41*** 1.5	944.2 1.65	991.28 0.88
96	904.91 4.64	981.86 1.39	957.39 1.76	918.37 1.78	1008.64 0.5	990.82 3.67	962.44 1.1
128	912.9 3.62	***1007.04*** 2.22	981.79 1.43	935.4 1.96	1035.36 0.	***997.7*** 1.74	990.23 1.09
160	***913.96*** 3.17	962.88 1.29	986.46 3.6	912.16 1.6	985.47 0.77	987.15 0.56	1001.31 4.9
192	905.86 2.67	957.74 0.69	974.51 0.81	911.49 2.3	977.51 0.29	972.65 0.4	996.21 0.61
224	869.78 2.24	893.35 0.75	949.74 1.1	900.33 2.66	936.42 0.31	938.65 0.44	961.36 0.64
256	902.12 3.78	948.44 1.2	***990.39*** 1.22	912.45 1.63	974.05 0.41	977.94 0.31	***1020.59*** 0.7

Table 3
Performance expressed in MLUPS for K5000m and K20c GPUs, compared against existing work.

Domain size	K5000m	K20c	Astorino (2011)	Obrecht (2011)	Rinaldi (2012)
64^3	420	914	295	471	245
96^3	419	1007	322	512	259
128^3	416	990	343	481	217
160^3	403	937	350	482	248

Table 4
Performance normalized against DRAM bandwidth (MLUPS/GB/s).

Domain size	K5000m	K20c	Astorino (2011)	Obrecht (2011)	Rinaldi (2012)
64^3	6.027	5.3912	1.5389	5.1588	2.1934
96^3	6.0057	5.9402	1.6685	5.6079	2.3187
128^3	5.968	5.842	1.77	5.2683	1.9427
160^3	5.7725	5.5291	1.7644	5.2793	2.2202

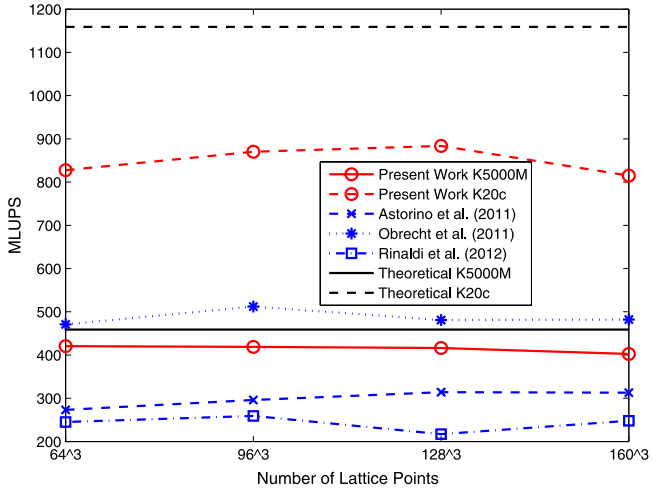


Fig. 14. Absolute performance.

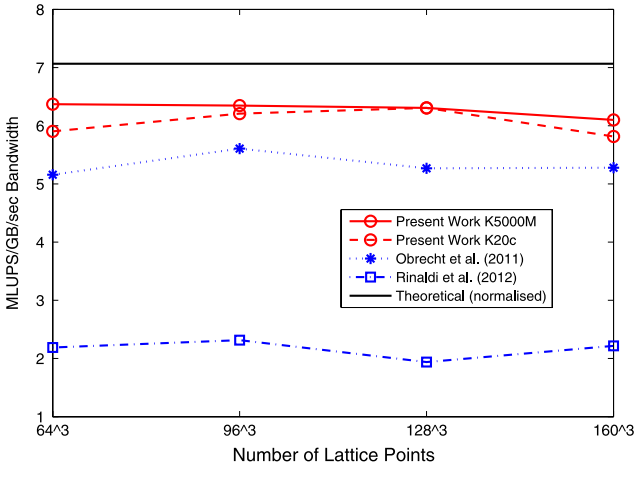


Fig. 15. Performance scaled for bandwidth.

This non-constant reduction in misaligned accesses (and therefore L2 cache use) is proportional to the ratio of boundary points to interior domain points, and will tend towards zero (and therefore constant L2 cache use) as the x dimension increases and the ratio of misaligned to aligned accesses decreases, as shown in Fig. 16.

Analysing the performance results of solver reveals interesting behaviour in the variation of performance. One would expect truly random variation to manifest itself in the form of a Gaussian-like distribution centred about the mean value. Whilst, at smaller domain sizes this is true, the vast majority of results exhibit periodic performance variation. A Fourier analysis of the results was performed to ascertain the amount of periodic contribution to performance variation, and the frequency at which it occurs. Below sample results for a domain of size 192^3 are presented.

As Fig. 17 shows, there is clearly a frequency component to the variation in performance. In this case the maximum contribution is found at a frequency of 11.71 Hz. Performing the same analysis across all of the test cases yields the following results.

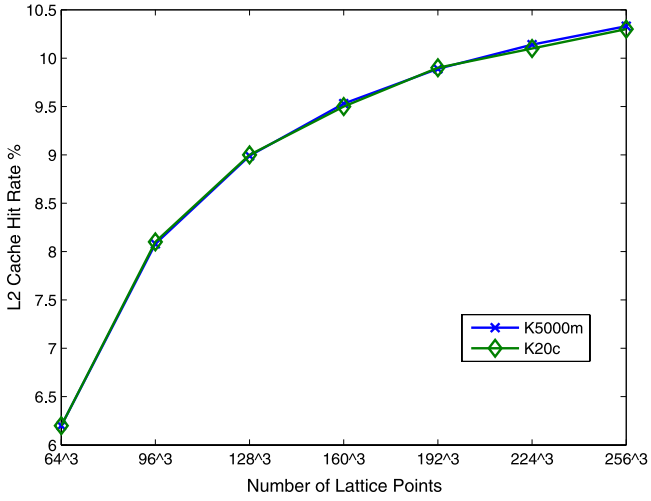


Fig. 16. L2 cache hit rates.

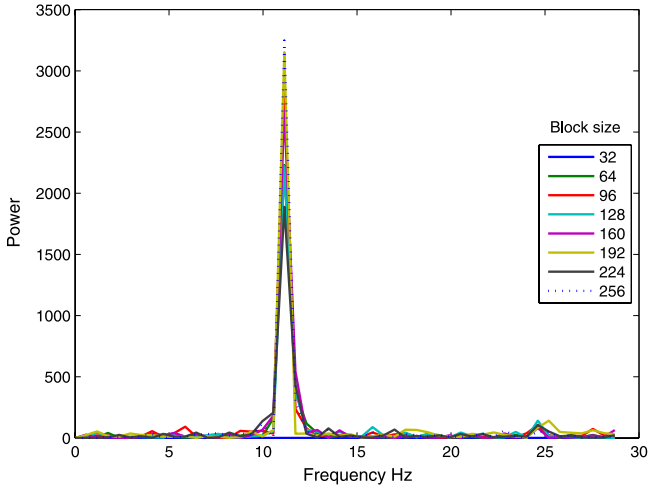


Fig. 17. Fourier analysis—domain size $= 192^3$.

No concrete explanation for this variation in performance can be provided, as it is likely a hardware level issue. Likely candidates are power and/or thermal management strategies of the GPU. The low-level analysis required to determine the cause of this variation is beyond the scope of the present work, although further study of this behaviour should be conducted as it has implications for all GPU programming.

6. Conclusion

This work has demonstrated the optimization and validation of a 3D GPU-based Lattice Boltzmann solver on Kepler architecture GPUs. The use of shared memory, and an intrinsic memory-less intra-warp *shuffle* operation, have been shown to be ineffective at improving the performance of the memory-intensive streaming operation, in spite of the fact that their use increases the number of coalesced accesses to DRAM. Instead, a 'naive' implementation that has misaligned access to DRAM is found to be more effective due to its lower register usage and no need for any additional control flow. Using this information an efficient Lattice Boltzmann solver was programmed and benchmarked on GK104 and GK110 generation GPUs, achieving a peak performance of up to 1036 MLUPS on GK110 GPUs. A review of several multicore-CPU based implementations (found in work by Groen et al. [35]) reports peak performance amongst the various solvers examined to be $\mathcal{O}(50)$ MLUPS per CPU socket on an AMD Opteron 6276 in double precision, operating at 20% efficiency. As the LBM is bandwidth-limited it is reasonable to assume that performance would double in single precision to $\mathcal{O}(100)$ MLUPS, approximately $10/times$ slower than the best GPU performance presented in this paper. Whilst this comparison is crude, the relative efficiencies of CPU and GPU implementations confirm the suitability of GPUs for LBM solvers.

Future work will involve the natural extension of the solver to multiple GPUs in order to increase the size and complexity of test cases that can be modelled. Recent work by Obrecht et al. outlines the methodologies behind such a multi-GPU implementation [18]. The findings of the present work have already been applied in the design of an interactive two dimensional LBM solver, where the high performance of the fluid solver allows boundary conditions to be captured from real-world geometry using an infrared depth sensor while maintaining real-time fluid flow evolution and visualization [16].

References

[1] S. Chen, G.D. Doolen, Annu. Rev. Fluid Mech. 30 (1998) 329–364.
[2] S. Succi, The Lattice Boltzmann Equation for Fluid Dynamics and Beyond (Numerical Mathematics and Scientific Computation), Oxford University Press, USA, 2001.
[3] X. Shan, X.-F. Yuan, H. Chen, J. Fluid Mech. 550 (2006) 413.
[4] W. Li, X. Wei, A. Kaufman, Vis. Comput. 19 (2003) 444–456.
[5] C. Janßen, M. Krafczyk, Comput. Math. Appl. 61 (2011) 3549–3563.
[6] C. Obrecht, F. Kuznik, B. Tourancheau, J.-J. Roux, Comput. & Fluids 54 (2011) 118–126.
[7] T. Miki, X. Wang, T. Aoki, Y. Imai, T. Ishikawa, K. Takase, T. Yamaguchi, Comput. Methods Biomech. Biomed. Eng. 15 (2012) 771–778.
[8] S. Ryoo, C.I. Rodrigues, S.S. Baghsorkhi, S.S. Stone, D.B. Kirk, W.-m.W. Hwu, Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP'08, 2008, p. 73.
[9] J. Henning, ACM SIGARCH Computer Architecture News, 2006.
[10] J. Tölke, M. Krafczyk, Int. J. Comput. Fluid Dyn. 22 (2008) 443–456.
[11] J. Tölke, Comput. Vis. Sci. 13 (2008) 29–39.
[12] J. Habich, T. Zeiser, G. Hager, G. Wellein, Adv. Eng. Softw. 42 (2011) 266–272.
[13] C. Obrecht, F. Kuznik, B. Tourancheau, J.-J. Roux, Proceedings of the 9th International Conference on High Performance Computing for Computational Science, Springer-Verlag, Berkeley, CA, 2011, pp. 151–161.
[14] C. Obrecht, F. Kuznik, B. Tourancheau, J.-J. Roux, Comput. Math. Appl. 61 (2011) 3628–3638.
[15] J. Favier, A. Revell, A. Pinelli, A Lattice Boltzmann-Immersed Boundary Method to Simulate the Fluid Interaction with Moving and Slender Flexible Objects, 2013.
[16] M. Mawson, G. Leaver, A. Revell, NAFEMS World Congress 2013 Summary of Proceedings, NAFEMS Ltd., Salzburg, 2013, p. 204.
[17] C. Obrecht, F. Kuznik, B. Tourancheau, J.-J. Roux, Comput. Math. Appl. 65 (2013) 252–261.
[18] C. Obrecht, F. Kuznik, B. Tourancheau, J.-J. Roux, Comput. & Fluids 80 (2013) 269–275.
[19] J. Habich, C. Feichtinger, H. Köstler, G. Hager, G. Wellein, Comput. & Fluids (2012) 1–7.
[20] NVIDIA Corporation, NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110. The Fastest, Most Efficient HPC Architecture Ever Built, 2012.
[21] NVIDIA, CUDA Programming Guide Version 2.3.1, 2009.
[22] NVIDIA, NVIDIA CUDA C Programming Guide Version 3.2, Technical Report, NVIDIA Corporation, 2010.
[23] D. Luebke, G. Humphreys, N. Res, Computer 40 (2007) 96–100.
[24] M. Pharr, R. Fernando, GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation, Addison-Wesley, 2005.
[25] P. Bhatnagar, E. Gross, M. Krook, Phys. Rev. 94 (1954) 511–525.
[26] X. He, L.-S. Luo, J. Stat. Phys. 88 (1997) 927–944.
[27] Y.H. Qian, D. D'Humières, P. Lallemand, Europhys. Lett. 17 (1992) 479–484.
[28] T. Pohl, M. Kowarschik, J. Wilke, Parallel Processing … 10 (2003).
[29] G. Wellein, T. Zeiser, G. Hager, S. Donath, Comput. & Fluids 35 (2006) 910–919.
[30] V. Volkov, Proceedings of the GPU Technology Conference.
[31] P. Rinaldi, E. Dari, M. Vénere, A. Clausse, Simul. Model. Pract. Theory 25 (2012) 163–171.
[32] M. Astorino, J.B. Sagredo, A. Quarteroni, SeMA J. 59 (2013) 53–78.
[33] B.-n. Jiang, T. Lin, Comput. Methods Appl. Mech. 114 (1994) 213–231.
[34] M. Hecht, J. Harting, J. Stat. Mech. Theory Exp. 2010 (2010) P01018.
[35] D. Groen, J. Hetherington, H.B. Carver, R.W. Nash, M.O. Bernabeu, P.V. Coveney, J. Comput. Sci. 4 (2013) 412–422.