A new approach to the lattice Boltzmann method for graphics processing units

Christian Obrecht, Frédéric Kuznik, Bernard Tourancheau, Jean-Jacques Roux

A new approach to the lattice Boltzmann method for graphics processing units

Christian Obrecht[a,b,*], Frédéric Kuznik[a], Bernard Tourancheau[b], Jean-Jacques Roux[a]

[a] *Centre de Thermique de Lyon, UMR5008, CNRS, INSA-Lyon, Université de Lyon*
[b] *Laboratoire de l'Informatique du Parallélisme, UMR 5668, CNRS, ENS de Lyon, INRIA, UCB Lyon 1*

Abstract

Emerging many-core processors, like CUDA capable nVidia GPUs, are promising platforms for regular parallel algorithms such as the Lattice Boltzmann Method (LBM). Since global memory on graphic devices shows high latency and LBM is data intensive, memory access pattern is an important issue to achieve good performances. Whenever possible, global memory loads and stores should be coalescent and aligned, but the propagation phase in LBM can lead to frequent misaligned memory accesses. Most previous CUDA implementations of 3D LBM faced this problem using low latency on chip shared memory. Instead of this, our CUDA implementation of LBM follows carefully chosen data transfer schemes in global memory. On the 3D lid-driven cavity test case, we obtained 2× to 3× speed-up over previously published performances, achieving up to 86% of the global memory maximal throughput. As a consequence we show that highly efficient implementations of LBM on GPU are possible, even for complex models.

Key words: GPU programming, CUDA, Lattice Boltzmann method, Parallel computing

1. Introduction

During the last decade, the computational power of commodity graphics hardware has dramatically increased, as shown in figure 1, nearing 1 GFlop/s with nVidia's latest GT200. Yet, one should be aware that this performance is attainable only for single precision computations, which are not fully IEEE-754 compliant. Nonetheless, due to their low cost, GPUs become more and more popular for scientific computations (see [1, 2]).

Lattice Boltzmann method, which originates from the lattice gas automata methods, is an efficient alternative to the numerical solving of Navier-Stokes equations for simulations of complex fluid systems. Besides its numerical stability and accuracy, one of the major advantage of LBM is its data parallel nature. Nevertheless, using LBM for practical purposes requires large computational power. Thus, several attempts to implement LBM on GPUs were made recently.

In this paper, we intend to present some optimisation principles for CUDA programming. These principles led us to a GPU implementation of 3D LBM which appears to be more efficient than the previously published ones.

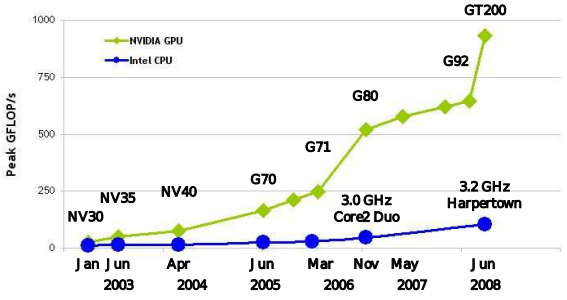

Figure 1: Peak performances GPU vs CPU (source nVidia)

2. CUDA

The Compute Unified Device Architecture (CUDA), released by nVidia in early 2007, is up to now the leading technology for general purpose GPU programming (see [3]). It consists of hardware specifications, a specific programming model, and a programming environment (API and SDK).

*Corresponding author: christian.obrecht@insa-lyon.fr

2.1. Architecture

General purpose GPU programming usually requires to take some architectural aspects into consideration. CUDA hardware specifications make the optimisation process easier by providing a general model for the nVidia GPUs architecture from the G80 generation on.

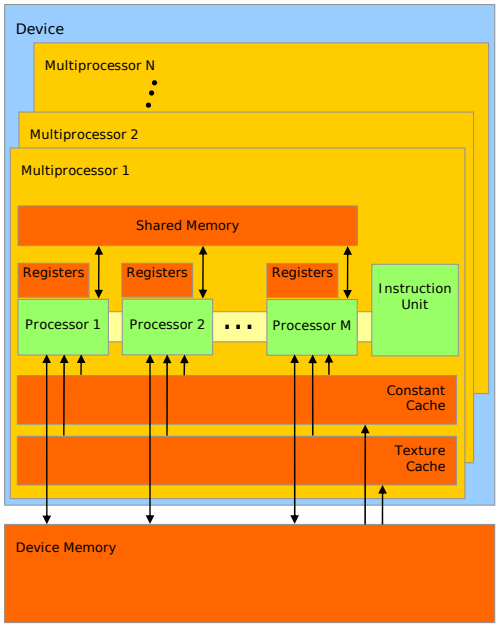

Figure 2: CUDA hardware (source nVidia)

Figure 2 shows the main aspects of the CUDA hardware specifications. A GPU consists in several *Streaming Multiprocessors* (SMs). Each SM contains *Scalar Processors* (SPs), an instruction unit, and a shared memory, concurrently accessible by the SPs through 16 memory banks. Two cached, read-only memories for constants and textures are also available. The device memory, usually named *global memory* is accessible by both the GPU and the CPU. Table 1 specifies some of the features of the GT200 processor on which our implementations were tested.

SPs are only able to perform single precision computations. From compute capability 1.3 on, CUDA supports double precision. On this kind of hardware, each SM is linked to a double precision computation unit. Both single and double precision calculations are mostly IEEE-754 compliant. Divergences from the standard are mainly:

- No denormalized numbers. Numbers with null exponent are considered as zero.

Number of SMs	30
Number of SPs per SM	8
Registers per SM	16,384
Shared Memory	16 KB
Constant Cache	8 KB
Texture Cache	8 KB
Global Memory	896 MB or 1 GB

Table 1: Features of the GT200

- Partial support of rounding modes.
- No floating point exception mechanism.
- Multiply-add operations with truncated intermediate results.
- Non compliant implementations of some operations like division or square root.

2.2. Programming

CUDA programming model (see [4]) relies on the concept of kernel. A kernel is a function that is executed in concurrent threads on the GPU. Threads are grouped into blocks which in turn form the execution grid (see figure 3).

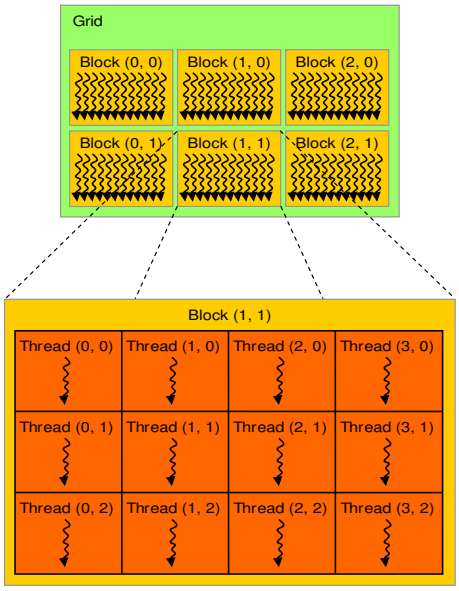

Figure 3: CUDA programming model (source nVidia)

The CUDA technology makes use of a slightly modified version of the C (or C++) language as a

programming language. The code of a CUDA application consists in functions which can be classified in four categories:

1. Sequential functions run by the CPU.
2. Launching functions allowing the CPU to start a kernel.
3. Kernels run by the GPU.
4. Auxiliary functions which are inlined into the kernels at compile time.

The execution grid's layout is specified at run time. A grid may have one or two dimensions. The blocks of threads within a grid must be identical and may have up to three dimensions. A thread is identified in respect of the grid using the two structures `threadIdx` and `blockIdx`, containing the three fields `x`, `y`, and `z`.

A block may only be executed on a single SM, which yields to an upper bound of the number of threads within a block[1]. Scheduling is carried out at hardware level and may not be adjusted. It is yet possible to place synchronisation barriers, but their scope is limited to blocks. The only way to ensure global synchronisation is to use a kernel for each step.

The local variables of a kernel are stored in the registers of the SMs. Their scope is limited to threads and they cannot be used for communication purposes. Data exchanges between threads require the use of shared memory. The management of these exchanges is left to the programmer. It is worth noting that no protection mechanism is available, hence concurrent writes at the same memory location yield unpredictable results. The shared memory's scope is limited to blocks. Communication between threads belonging to different blocks requires the use of global memory.

3. Optimisation principles

3.1. Computational aspect

Generally speaking, the occupancy rate of the SPs, i.e. the ratio between the number of threads run and the maximum number of executable threads, is an important aspect to take into consideration for the optimisation of a CUDA kernel. Even though a block may only be run on a single SM, it is possible however to execute several blocks concurrently on the same SM. Hence tuning the execution grid's layout allows to increase the occupancy rate. Nevertheless, reaching the maximal occupancy is usually not possible: the threads executed in parallel on one SM have to share the available registers. On compute capability 1.3 architectures, for instance, maximal occupancy is achieved only for kernels using at most 16 registers, that is to say only the simplest ones. It should be noted that shared memory, which is rather scarce too, may also be a limiting factor for the occupancy.

The rather elementary optimisation technique consisting in common sub-expression elimination should be used with care. As a matter of fact, this method implies to store the values of these sub-expressions in temporary variables, thence increasing the use of registers, which in turn may lead to lower occupancy. In some cases, this common sense technique has negative effects, and it may be better to recompute some values than to store them. Anyway, general principles regarding this topic are not relevant. Since the compiler performs aggressive optimisation, the number of register needed for a given kernel is scarcely predictable.

The hardware scheduler groups threads in warps of 32 threads. Though not mandatory, the number of threads in a block should be a multiple of the warp size. Whenever a warp is running, all the corresponding threads are executed concurrently by the SPs, except when conditional branching occurs. Divergent branches are executed sequentially by the SM. Even though serialisation only happens at warp level, conditional structures should be avoided as much as possible, being likely to have a major impact on actual parallelism.

Regarding optimisation, the cost of arithmetic operations (in clock cycles) must also be taken in consideration. Table 2 displays the time needed for a warp to perform the most common single precision floating point operations :

Operation	Cycles
Add, multiply, multiply-add	4
Reciprocal, logarithm	16
Sine, cosine, exponential	32
Divide	36

Table 2: Cost of floating point operations

It should be noted that, since addition and mul-

[1] As of compute capability 1.3, this maximum is 1,024.

tiplication are merged in one single *axpy* operation whenever possible, evaluating the actual algorithmic complexity of a computation is not straightforward. It's also worth noting that division is rather expensive and should be used parsimoniously.

3.2. Data transfer aspect

For many applications, memory transactions optimisation appears to be even more important than computations optimisation. Registers do not arise any specific problem apart from their limited amount. Shared memory is in terms of speed similar to register but is accessed by the SPs through 16 memory banks. For efficient accesses, each thread in a half-warp must use a different bank. When this condition is not met, the transaction is repeated as much as necessary.

Global memory, being the only one accessible by both the CPU and the GPU, is a critical path for CUDA applications. Unlike registers and shared memory, global memory suffers high latency ranging from 400 to 600 clock cycles. Nonetheless, this latency can be mostly hidden by the scheduler which stalls inactive warps until data is available. Furthermore, global memory throughput is significantly less than register throughput. For data intensive applications like LBM, this aspect is generally the limiting factor.

Global memory accesses are performed by half-warp on 32, 64, or 128 bytes segments whose start addresses are multiple of the segment's size. To optimise global memory transactions, memory accesses should be coalesced and aligned whenever possible. To achieve coalescence, threads within a half-warp must access contiguous memory locations.

4. Data transfer modelling

In CUDA applications, the execution of a kernel can generally be split into three steps:

1. Reading data from global memory.
2. Processing data using registers (and possibly shared memory).
3. Writing processed data to global memory.

Code 1 follows this scheme in the case where the amount of data read and written are equal. Function `launch_kernel` calls function `kernel` with an execution grid containing L^3 threads. One may notice some syntactic specificities of the CUDA programming language: the use of the tripled angle brackets for kernel invocation, the `__global__` keyword for kernel definition, the `__device__` keyword for auxiliary functions. The kernel performs the reading and writing of N 32-bit words. The second step is simplistic, though not suppressed in order to ensure actual data transfer to the GPU. Global memory accesses are optimal, provided L is set to an appropriate value. In the present study, $L = 128$ was chosen.

```
#define id(j, k) k + SIZE*(j)

__device__ int index(void)
{
    int x = threadIdx.x;
    int y = blockIdx.x;
    int z = blockIdx.y;
    return x + y*L + z*L2;
}

__global__ void kernel(int N, float* t)
{
    int k = index();

    for (int j = 0; j <= N; j++)
    {
        t[id(j+1, k)] = t[id(j, k)]*0.5;
    }
}

extern "C" void launch_kernel(int N, float* t)
{
    dim3 grid(L, L);
    dim3 block(L);

    kernel<<<grid, block>>>(N, t);
    cudaThreadSynchronize();
}
```

Code 1: Data transfer benchmark kernel

Measuring the execution time of the `launch_kernel` function enables to estimate the average time T for data transfer between GPU and global memory relatively to the amount N of data exchanged. Figure 4 shows the results for one warp with T in nanoseconds and N in 32-bit words, obtained using a GeForce GTX 295 graphics board.

The quasi-linear aspect of these measurements reveals that, in ideal cases, the hardware scheduler is able to hide the latency of global memory. Numerically, we obtain:

$$T \approx 2.78 \times N + 0.99 \tag{1}$$

The average throughput is almost constant relatively to N and is about 90.7 GB/s for the

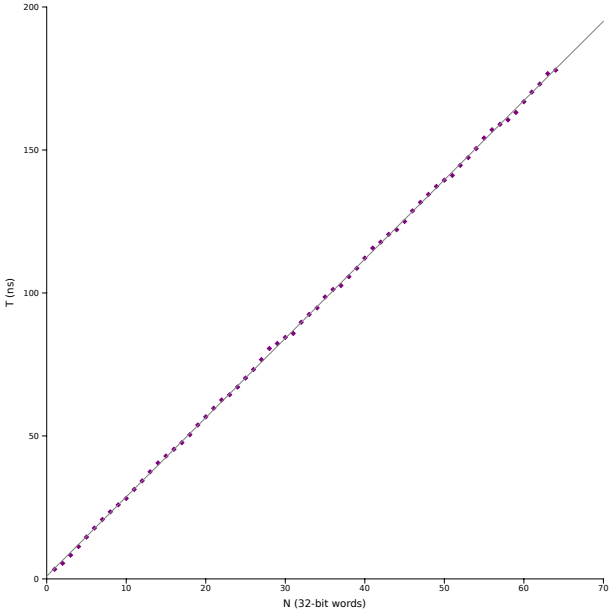

Figure 4: Average transfer time

GeForce GTX 295. Reckoning the characteristics of the benchmark program, we consider the obtained value as the effective maximal throughput for data transfer between GPU and global memory. This upper bound is useful in evaluating the performances of a CUDA application leaving aside the hardware in use. The obtained value is about 81% of the theoretical maximal throughput, which is comparable to the result found in [5].

On the same hardware, the `bandwidthTest` program from the CUDA SDK gives 91.3 GB/s, which is rather close to the value we obtained. Yet, this program uses only memory copy functions instead of a kernel, hence yielding less relevant results from a practical standpoint.

5. Lattice Boltzmann Method

The lattice Boltzmann method is based on a threefold discretisation of the Boltzmann equation: time, space and velocity (see [6]). Velocity space reduces to a finite set of well chosen velocities $\{\mathbf{e}_i \,|\, i = 0, \dots N\}$ where $\mathbf{e}_0 = \mathbf{0}$. Figure 5 illustrates the D3Q19 stencil we used.

Instead of reviewing the well-known Lattice Bhatnagar-Gross-Krook (LBGK) model (see [7]), we will outline the Multiple Relaxation Time model presented in [8]. The analogous of the one-particule

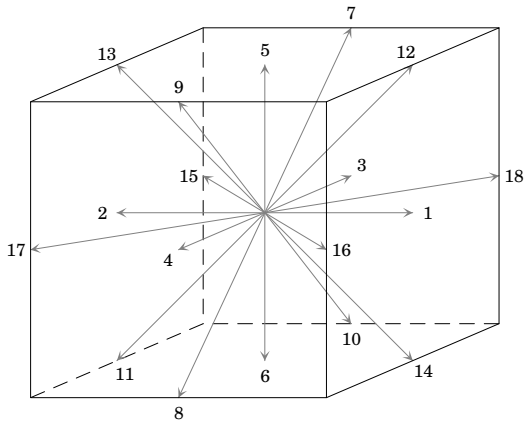

Figure 5: The D3Q19 stencil

distribution function f is a set of $N+1$ mass fractions f_i. We denote:

$$\big|f(\mathbf{x},t)\big\rangle = (f_0(\mathbf{x},t), \dots f_N(\mathbf{x},t))^{\mathsf{T}}$$

for given lattice node $\mathbf{x}$ and time t, T being the transpose operator. The mass fractions can be mapped to a set of moments $\{m_i \,|\, i = 0, \dots N\}$ by an invertible matrix M such as:

$$\big|f(\mathbf{x},t)\big\rangle = \mathsf{M}^{-1}\big|m(\mathbf{x},t)\big\rangle \tag{2}$$

where $\big|m(\mathbf{x},t)\big\rangle$ is the moment vector. With the D3Q19 stencil, the density is $\rho = m_0$, the momentum is $\mathbf{j} = (m_3, m_5, m_7)$. Higher order moments as well as matrix M are given in detail in [8, app. A]. Using these notations, the lattice Boltzmann equation can be written as:

$$\big|f(\mathbf{x}+\delta t\mathbf{e}_i, t+\delta t)\big\rangle - \big|f(\mathbf{x},t)\big\rangle = \mathsf{M}^{-1}\mathsf{S}\left(\big|m(\mathbf{x},t)\big\rangle - \big|m^{(\mathrm{eq})}(\mathbf{x},t)\big\rangle\right) \tag{3}$$

where $\big|m^{(\mathrm{eq})}(\mathbf{x},t)\big\rangle$ is the equilibrium-moment vector and:

$$\mathsf{S} = \mathrm{diag}(s_0, \dots s_N)$$

is the relaxation rates matrix. The LBGK model is a special case of MRT where all relaxation rates $s_i = 1/\tau$. In a numerical point of view, MRT should be preferred to LBGK, being more stable and accurate.

6. Previous Work

Data organisation schemes for LBM are mainly of two kinds. First, the Array of Structures (AoS) type, which for D3Q19 is equivalent to a $L^3 \times 19$ array. Second, the Structure of Arrays (SoA) type, which for D3Q19 is equivalent to a $19 \times L^3$ array. For CPU implementations of the LBM, the AoS is relevant insofar as it improves the locality of mass fractions associated to a same node (see [9]). Up to now, for all GPU implementations of LBM, a thread is allocated to each lattice node, which is probably the simplest way to take advantage of the massively parallel structure of the GPU. With this approach, ensuring coalescence of global memory accesses requires to use a SoA kind of organisation.

With values of L divisible by 16, every mass fractions associated to a half-wrap lay in a same segment of global memory. Yet, this is not sufficient to ensure optimal memory transaction. Indeed, for the minor spatial dimension, propagation corresponds to one unit shifts of memory addresses. In other words, for most mass fractions, propagation phase leads to misalignments. Getting round this problem was up to now the main issue regarding GPU implementations of the LBM.

The first attempt of implementing a D3Q19 model using CUDA is due to Ryoo *et al.* (see [10]). It consists mainly in a port of the `470.lbm` code from the SPEC CPU2006 benchmark (see [11]). In terms of optimisation, switching from AoS to SoA is the only important modification undertaken. To the best of our knowledge, misalignment problems caused by propagation are not taken into consideration. The announced speed-up factor of 12.3 is rather low compared to subsequent results.

The two-dimensional D2Q9 implantation submitted by Tölke in [5] solves the misalignment problems using one-dimensional blocks and shared memory. More precisely, propagation within one block is split in two steps: a longitudinal shift in shared memory followed by a lateral shift in global memory. This approach is outlined in figure 6.

Since blocks follow the minor dimension, no more misalignment arises. Nevertheless, because of the limited scope of shared memory, mass fractions leaving or entering a block require specific handling. The retained solution is to store out-coming mass fractions in places temporarily left vacant by in coming mass fractions (see figure 7).

A drawback of the shared memory approach is consequently the need for a second kernel exchanging data in order to place properly mass fractions located at the blocks' boundaries. Obviously, this further processing has a non negligible cost.

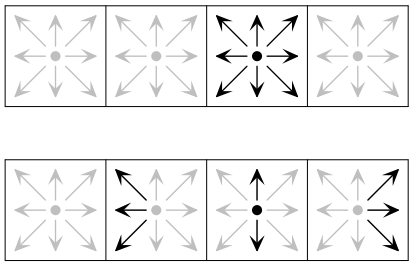

Figure 6: Propagation using shared memory

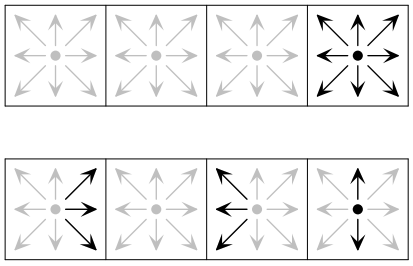

Figure 7: Storage of out-coming mass fractions

Following the same method than Tölke, Habich in [12] describes an implementation of a D3Q19 model. The transition from D2Q9 to D3Q19 leads to lower performances, achieving only 51% of the effective maximal throughput. Habich assumes this decrease is due to the low occupancy rate. As a matter of fact, given the limited amount of registers, the number of threads run in parallel on a SP cannot exceed one or two warps.

This point of view is probably erroneous as we shall see subsequently. The lower performances seem more likely due to the increase of the execution time of the second kernel, since there is dramatically more data to exchange than for two-dimensional LBM. As an example, for D2Q9 on a $2{,}048^2$ grid with 128 threads per block, there are $2{,}048 \times 16 \times 6 = 196{,}608$ mass fractions to move. For D3Q19, on a 160^3 grid with 32 threads per block, there are $160^2 \times 5 \times 14 = 1{,}792{,}000$ mass fractions to move. Relatively to the number of nodes, the ratio is about 9.3.

A way to obtain better performances for three-dimensional LBM consists in using stencils contain-

ing less mass fractions, like D3Q13. This approach was studied by Tölke and Krafczyk in [13], obtaining 61% of the effective maximal throughput. The D3Q13 stencil, which corresponds to the points of contact in a close-packing of spheres, is the simplest three-dimensional structure sufficiently isotropic for LBM. Yet, due to the lesser amount of information processed, D3Q13 is less accurate than D3Q19. Furthermore, node addressing becomes quite complex.

Bailey *et al.* in [14] announce a 20% improvement of maximal performances for their implementation of D3Q19 compared to those published in [12]. The description of the tested optimisations is not very explicit, but it seems that the main intention was to increase occupancy. One of the proposed technique consists in imposing at compile time an upper bound to the number of registers used by the computation kernel. Of course this directive causes the compiler to fall back on register spilling. Taking the cost of global memory accesses into account, we consider this approach as not relevant.

7. Proposed Implementations

All but one CUDA implementations of LBM mentioned in the former section use shared memory for propagation. As formerly outlined, this approach imposes the use of a second kernel taking care of the mass fractions crossing the blocks' boundaries. Though rather basic, the CUDA profiler allows to gather some informations during kernel run time (see [15]). Concerning LBM, this tool led us to make two assumptions:

1. The additional cost caused by misalignment has the same order of magnitude than the one caused by the exchange kernel.
2. The cost of a misaligned read is less than the cost of a misaligned write.

Hence we adopted the following approach for our implementations of D3Q19:

- SoA type of data organisation.
- One-dimensional blocks following the minor dimension.
- Propagation performed by global memory transactions.
- Deferment of misalignment on reading.

We experimented two propagation schemes: a split scheme and a reversed scheme. The split scheme was tested with a LBGK model and on-grid boundary conditions. The reversed scheme was tested with a MRT model and mid-grid boundary conditions. To ease cross platform development, we employed the CMake build system (see [16]). Moreover, we used the XML based VTK format for output (see [17]).

7.1. Split scheme

With the split scheme, propagation is parted in two components: shifts that induce misalignment are performed at reading, the others are performed at writing, as outlined in figure 8. For the sake of simplicity, the diagram shows the two-dimensional case.

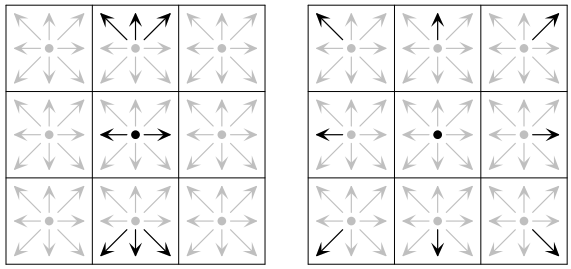

Figure 8: Split propagation scheme

Boundary conditions are implemented using on-grid bounce back: nodes at the cavity's borders, except the lid, are considered as solid and simply return the in-coming mass fractions in the opposite direction.

To summarise, the corresponding kernel breaks up into:

1. Reading along with propagation in minor dimension.
2. On-grid bounce back boundary conditions.
3. Computations using LBGK model.
4. Writing along with propagation in major dimensions.

7.2. Reversed scheme

With reversed scheme, propagation is entirely performed at reading, as outlined in figure 9. Again, the diagram shows the two-dimensional case only.

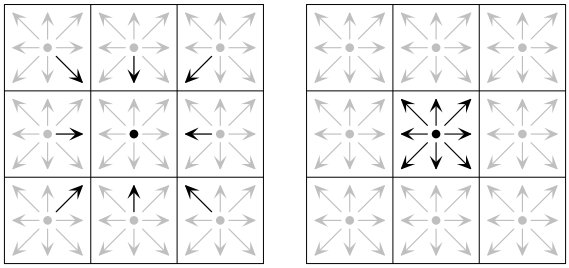

Figure 9: Reversed propagation scheme

Boundary conditions are implemented using mid-grid bounce back: nodes at the cavity's borders, except the lid, are considered as fluid with null velocity. Unknown mass fractions are determined using:

$$f_i - f_i^{\mathrm{eq}} = f_j - f_j^{\mathrm{eq}} \tag{4}$$

whith i and j such as $\mathbf{e}_i = -\mathbf{e}_j$ (see [18]). For null velocity, $f_i^{\mathrm{eq}} = f_j^{\mathrm{eq}}$. Hence the former equation yields $f_i = f_j$.

To summarise, the corresponding kernel breaks up into:

1. Reading carrying out propagation.
2. Mid-grid bounce back boundary conditions.
3. Computations using MRT model.
4. Writing without propagation.

8. Results

8.1. Validation

Numerical validation is an important issue in GPU computing, since most calculations are performed using single precision. This topic being thoroughly studied for GPU implementations of LBM in [19], we will rather focus on physical validation. We used the well-known lid-driven cavity test case, comparing velocity coordinates with results published by Albensoeder and Kuhlmann in [20]. More precisely, for velocity $\mathbf{u}(u, v, w)$, we gathered u on line $x = L/2$ and $y = L/2$, as well as w on line $z = L/2$ and $y = L/2$. For Reynolds number Re = 1,000, our LBGK code outcomes are in quite good accordance with the reference values. Not surprisingly, the MRT implementation achieves almost perfect correspondence as shown in figure 10.

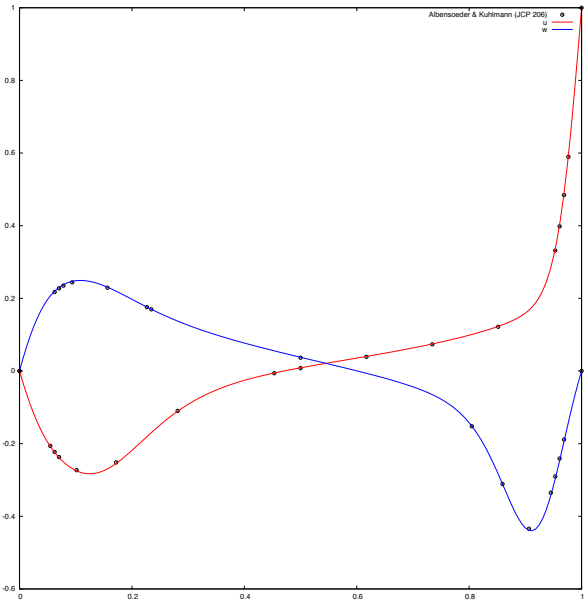

Figure 10: Validation for Re = 1,000

8.2. Performances

Binaries for nVidia GPUs are generated through a two stages process (see [21]). First, the `nvopencc` program compiles CUDA code into Parallel Thread eXecution (PTX) pseudo assembly language. Second, the `ocg` assembler translates the PTX code into actual binary. Analysing PTX outputs allows to enumerate the floating point operations in a kernel and thence to evaluate the actual algorithmic complexity of the computations. Table 3 assembles the obtained results for both the LBGK and the MRT kernels (`rcp` stands for reciprocal, `mad` for multiply-add).

	`add`	`sub`	`mul`	`div`	`rcp`	`mad`	cycles
LBGK	63	30	48	0	1	34	716
MRT	76	80	51	0	0	18	900

Table 3: Algorithmic complexity of LBGK and MRT kernels

Though being more complex than LBGK, MRT has almost the same computational cost. It is worth noting that this cost is of the same order of magnitude than one single global memory transaction, that is to say 400 to 600 cycles. Thus, taking the hardware scheduler into account, the impact of computations on global processing time is negligible. Most of the execution time of our kernels

is consumed by data transfer, the remaining being probably induced by scheduling. In terms of optimisation, increasing the occupancy rate of the SMs is not especially crucial.

The former opinion is supported by the analysis of the performances of our implementations. Million Lattice node Updates Per Second (MLUPS) is the usual unit for performance measurement in LBM. For both implementations, memory addressing is analogous to the one used in code 1. Therefore, the size of the blocks corresponds to the size of the cavity. Tables 4 and 5 show the obtained performances on a Debian GNU/Linux 5.0 workstation fitted with a GeForce GTX 295 graphics card.

	64^3	96^3	128^3	160^3
Performance (MLUPS)	471	512	481	482
Ratio to max. throughput	79%	86%	81%	81%
Occupancy rate	31%	19%	25%	16%

Table 4: Performances for LBGK

	64^3	96^3	128^3	160^3
Performance (MLUPS)	484	513	516	503
Ratio to max. throughput	81%	86%	86%	84%
Occupancy rate	25%	19%	25%	16%

Table 5: Performances for MRT

One may notice that the data transfer rate is rather close to maximum. Global memory throughput is presently the limiting factor for LBM computations on GPU. Moreover, it is worth mentioning that these satisfactory performances are achieved with quite low SM occupancy.

Confronting the obtained performances to published results corroborates our approach. Depending on the size of the cavity, we observe 2× to 3× speed-up compared to the performances mentioned in [12, 14]. Yet these studies were led on GeForce 8800 GTX graphics cards, which belong to the previous generation though being comparable to the hardware we used in terms of memory throughput. Therefore, these comparisons should be considered with care, and we additionally compared a D2Q9 version of our code to the one published in [5] on the GTX 295 obtaining a 15% betterment of the performances.

9. Summary

The present study proposes a model for data transfer on the latest generation of nVidia GPUs. Optimisation principles, leading to efficient implementations of 3D LBM on GPUs, are drawn as well. We state the impact of global memory transfer as the main limiting factor for now. Our implementations achieved up to 86% of the effective maximal throughput of global memory. On the 3D lid-driven cavity test case, we obtained 2× to 3× speed-up over previously published implementations. Moreover, we show that, compared to LBGK, the more stable and accurate MRT, despite its higher computational cost, yields equivalent performances on GPUs. Our approach, being simpler than the previous ones, exerts less pressure on hardware. Hence, our method will allow to implement more complex models in the near future.

References

[1] J. Dongarra, S. Moore, G. Peterson, S. Tomov, J. Allred, V. Natoli, D. Richie, Exploring new architectures in accelerating CFD for Air Force applications, in: Proceedings of HPCMP Users Group Conference, Citeseer, 2008, pp. 14–17.

[2] S. Tomov, J. Dongarra, M. Baboulin, Towards dense linear algebra for hybrid GPU accelerated manycore systems.

[3] T. Halfhill, Parallel processing with CUDA, Microprocessor Journal.

[4] nVidia, Compute Unified Device Architecture Programming Guide version 2.2 (April 2009).

[5] J. Tölke, Implementation of a Lattice Boltzmann kernel using the Compute Unified Device Architecture developed by nVIDIA, Computing and Visualization in Science 1–11.

[6] G. R. McNamara, G. Zanetti, Use of the Boltzmann Equation to Simulate Lattice-Gas Automata, Phys. Rev. Lett. 61 (1988) 2332–2335.

[7] Y. H. Qian, D. d'Humières, P. Lallemand, Lattice BGK models for Navier-Stokes equation, Europhys. Lett 17 (6) (1992) 479–484.

[8] D. d'Humières, I. Ginzburg, M. Krafczyk, P. Lallemand, L. Luo, Multiple-relaxation-time lattice Boltzmann models in three dimensions, Philosophical Transactions: Mathematical, Physical and Engineering Sciences (2002) 437–451.

[9] T. Pohl, M. Kowarschik, J. Wilke, K. Iglberger, U. Rüde, Optimization and Profiling of the Cache Performance of Parallel Lattice Boltzmann Codes, Parallel Processing Letters 13 (4) (2003) 549–560.

[10] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, W. H. Wen-mei, Optimization principles and application performance evaluation of a multithreaded GPU using CUDA, in: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming, ACM New York, NY, USA, 2008, pp. 73–82.

[11] J. L. Henning, SPEC CPU2006 Benchmark Descriptions, ACM SIGARCH Computer Architecture News 34 (4) (2006) 1–17.
[12] J. Habich, Performance Evaluation of Numeric Compute Kernels on nVIDIA GPUs, Master Thesis.
[13] J. Tölke, M. Krafczyk, TeraFLOP computing on a desktop PC with GPUs for 3D CFD, International Journal of Computational Fluid Dynamics 22 (7) (2008) 443–456.
[14] P. Bailey, J. Myre, S. D. C. Walsh, D. J. Lilja, M. O. Saar, Accelerating Lattice Boltzmann Fluid Flow Simulations Using Graphics Processors (2008).
[15] nVidia, CUDA Profiler version 2.2 (2009).
[16] K. Martin, B. Hoffman, Mastering CMake, A Cross-Platform Build System, Kitware Inc, Clifton Park NY, 2008.
[17] W. J. Schroeder, K. Martin, L. S. Avila, C. C. Law, The VTK User's Guide, Kitware Inc, Clifton Park NY, 2006.
[18] Q. Zou, X. He, On pressure and velocity flow boundary conditions and bounceback for the lattice Boltzmann BGK model, Arxiv preprint comp-gas/9611001.
[19] F. Kuznik, C. Obrecht, G. Rusaouën, J.-J. Roux, LBM Based Flow Simulation Using GPU Computing Processor, Computers and Mathematics with Applications (27).
[20] S. Albensoeder, H. C. Kuhlmann, Accurate three-dimensional lid-driven cavity flow, Journal of Computational Physics 206 (2) (2005) 536–558.
[21] M. Murphy, NVIDIA's Experience with Open64, nVidia.