High Performance Lattice Boltzmann Solvers on Massively Parallel Architectures with Applications to Building Aeraulics

Christian Obrecht

N° d'ordre 2012ISAL0134
Année 2012

Thèse

High Performance Lattice Boltzmann Solvers on Massively Parallel Architectures with Applications to Building Aeraulics

Présentée devant
L'institut national des sciences appliquées de Lyon

Pour obtenir
Le grade de docteur

Formation doctorale
Génie civil
École doctorale
École doctorale MEGA (mécanique, énergétique, génie civil, acoustique)

Par
Christian Obrecht

Soutenue le 11 décembre 2012 devant la commission d'examen

Jury

J.-L. Hubert	Ingénieur-Chercheur (EDF R&D)	*Examinateur*
Ch. Inard	Professeur (Université de La Rochelle)	*Président*
M. Krafczyk	Professeur (TU Braunschweig)	*Rapporteur*
F. Kuznik	Maître de conférences HDR (INSA de Lyon)	*Co-directeur*
J. Roman	Directeur de recherche (INRIA)	*Rapporteur*
J.-J. Roux	Professeur (INSA de Lyon)	*Directeur*
B. Tourancheau	Professeur (UJF – Grenoble)	*Co-directeur*

INSA Direction de la Recherche - Ecoles Doctorales – Quinquennal 2011-2015

SIGLE	ECOLE DOCTORALE	NOM ET COORDONNEES DU RESPONSABLE
CHIMIE	**CHIMIE DE LYON** **http://www.edchimie-lyon.fr** Insa : R. GOURDON	M. Jean Marc LANCELIN Université de Lyon – Collège Doctoral Bât ESCPE 43 bd du 11 novembre 1918 69622 VILLEURBANNE Cedex Tél : 04.72.43 13 95 directeur@edchimie-lyon.fr
E.E.A.	**ELECTRONIQUE, ELECTROTECHNIQUE, AUTOMATIQUE** **http://edeea.ec-lyon.fr** Secrétariat : M.C. HAVGOUDOUKIAN eea@ec-lyon.fr	M. Gérard SCORLETTI Ecole Centrale de Lyon 36 avenue Guy de Collongue 69134 ECULLY Tél : 04.72.18 60 97 Fax : 04 78 43 37 17 Gerard.scorletti@ec-lyon.fr
E2M2	**EVOLUTION, ECOSYSTEME, MICROBIOLOGIE, MODELISATION** **http://e2m2.universite-lyon.fr** Insa : H. CHARLES	Mme Gudrun BORNETTE CNRS UMR 5023 LEHNA Université Claude Bernard Lyon 1 Bât Forel 43 bd du 11 novembre 1918 69622 VILLEURBANNE Cédex Tél : 04.72.43.12.94 e2m2@biomserv.univ-lyon1.fr
EDISS	**INTERDISCIPLINAIRE SCIENCES-SANTE** **http://ww2.ibcp.fr/ediss** Sec : Safia AIT CHALAL Insa : M. LAGARDE	M. Didier REVEL Hôpital Louis Pradel Bâtiment Central 28 Avenue Doyen Lépine 69677 BRON Tél : 04.72.68 49 09 Fax :04 72 35 49 16 Didier.revel@creatis.uni-lyon1.fr
INFOMATHS	**INFORMATIQUE ET MATHEMATIQUES** http://infomaths.univ-lyon1.fr	M. Johannes KELLENDONK Université Claude Bernard Lyon 1 INFOMATHS Bâtiment Braconnier 43 bd du 11 novembre 1918 69622 VILLEURBANNE Cedex Tél : 04.72. 44.82.94 Fax 04 72 43 16 87 infomaths@univ-lyon1.fr
Matériaux	**MATERIAUX DE LYON** Secrétariat : M. LABOUNE PM : 71.70 –Fax : 87.12 Bat. Saint Exupéry Ed.materiaux@insa-lyon.fr	M. Jean-Yves BUFFIERE INSA de Lyon MATEIS Bâtiment Saint Exupéry 7 avenue Jean Capelle 69621 VILLEURBANNE Cédex Tél : 04.72.43 83 18 Fax 04 72 43 85 28 Jean-yves.buffiere@insa-lyon.fr
MEGA	**MECANIQUE, ENERGETIQUE, GENIE CIVIL, ACOUSTIQUE** Secrétariat : M. LABOUNE PM : 71.70 –Fax : 87.12 Bat. Saint Exupéry mega@insa-lyon.fr	M. Philippe BOISSE INSA de Lyon Laboratoire LAMCOS Bâtiment Jacquard 25 bis avenue Jean Capelle 69621 VILLEURBANNE Cedex Tél :04.72.43.71.70 Fax : 04 72 43 72 37 Philippe.boisse@insa-lyon.fr
ScSo	**ScSo*** M. OBADIA Lionel Sec : Viviane POLSINELLI Insa : J.Y. TOUSSAINT	M. OBADIA Lionel Université Lyon 2 86 rue Pasteur 69365 LYON Cedex 07 Tél : 04.78.69.72.76 Fax : 04.37.28.04.48 Lionel.Obadia@univ-lyon2.fr

*ScSo : Histoire, Geographie, Aménagement, Urbanisme, Archéologie, Science politique, Sociologie, Anthropologie

High Performance Lattice Boltzmann Solvers on Massively Parallel Architectures with Applications to Building Aeraulics

Christian Obrecht

2012

I dedicate this work to my sons Adrien and Nicolas, for:

The lyf so short, the craft so longe to lerne,
Th'assay so hard, so sharp the conquerynge, ...

Geoffrey Chaucer — The Parlement of Foules

Acknowledgements

First and foremost, I wish to acknowledge my deepest gratitude to my advisors Jean-Jacques Roux, Frédéric Kuznik, and Bernard Tourancheau whose trust and constant care made this whole endeavour possible. As well, I would like to express my particular appreciation to the EDF company for funding my research and to Jean-Luc Hubert for his kind consideration. I also wish to thank my fellows at the CETHIL: Miguel, Samuel, Andrea, Kévyn, and many others for their joyous company and the administrative staff, especially Christine, Agnès, and Florence, for their diligent work. Special acknowledgements go to Gilles Rusaouën for sharing his deep understanding of many topics related to my research work, and to the members of the ICMMES community for the so many fruitful discussions we had during our annual meetings. Last and most of all, I thank my beloved wife and sons for their everyday affection and support.

Résumé

Avec l'émergence des bâtiments à haute efficacité énergétique, il est devenu indispensable de pouvoir prédire de manière fiable le comportement énergétique des bâtiments. Or, à l'heure actuelle, la prise en compte des effets thermo-aérauliques dans les modèles se cantonne le plus souvent à l'utilisation d'approches simplifiées voire empiriques qui ne sauraient atteindre la précision requise. Le recours à la simulation numérique des écoulements semble donc incontournable, mais il est limité par un coût calculatoire généralement prohibitif. L'utilisation conjointe d'approches innovantes telle que la méthode de Boltzmann sur gaz réseau (LBM) et d'outils de calcul massivement parallèles comme les processeurs graphiques (GPU) pourrait permettre de s'affranchir de ces limites. Le présent travail de recherche s'attache à en explorer les potentialités.

La méthode de Boltzmann sur gaz réseau, qui repose sur une forme discrétisée de l'équation de Boltzmann, est une approche explicite qui jouit de nombreuses qualités : précision, stabilité, prise en compte de géométries complexes, etc. Elle constitue donc une alternative intéressante à la résolution directe des équations de Navier-Stokes par une méthode numérique classique. De par ses caractéristiques algorithmiques, elle se révèle bien adaptée au calcul parallèle. L'utilisation de processeurs graphiques pour mener des calculs généralistes est de plus en plus répandue dans le domaine du calcul intensif. Ces processeurs à l'architecture massivement parallèle offrent des performances inégalées à ce jour pour un coût relativement modéré. Néanmoins, nombre de contraintes matérielles en rendent la programmation complexe et les gains en termes de performances dépendent fortement de la nature de l'algorithme considéré. Dans le cas de la LBM, les implantations GPU affichent couramment des performances supérieures de deux ordres de grandeur à celle d'une implantation CPU séquentielle modérément optimisée.

Le présent mémoire de thèse est constitué d'un ensemble de neuf articles de revues internationales et d'actes de conférences internationales (le dernier étant en cours d'évaluation). Dans ces travaux sont abordés les problématiques liées tant à l'implantation mono-GPU de la LBM et à l'optimisation des accès en mémoire, qu'aux implantations multi-GPU et à la modélisation des communications inter-GPU et inter-nœuds. En complément, sont détaillées diverses extensions à la LBM indispensables pour envisager une utilisation en thermo-aéraulique des bâtiments. Les cas d'études utilisés pour la validation des codes permettent de juger du fort potentiel de cette approche en pratique.

Mots-clefs : calcul intensif, méthode de Boltzmann sur gaz réseau, processeurs graphiques, aéraulique des bâtiments.

Abstract

With the advent of low-energy buildings, the need for accurate building performance simulations has significantly increased. However, for the time being, the thermo-aeraulic effects are often taken into account through simplified or even empirical models, which fail to provide the expected accuracy. Resorting to computational fluid dynamics seems therefore unavoidable, but the required computational effort is in general prohibitive. The joint use of innovative approaches such as the lattice Boltzmann method (LBM) and massively parallel computing devices such as graphics processing units (GPUs) could help to overcome these limits. The present research work is devoted to explore the potential of such a strategy.

The lattice Boltzmann method, which is based on a discretised version of the Boltzmann equation, is an explicit approach offering numerous attractive features: accuracy, stability, ability to handle complex geometries, etc. It is therefore an interesting alternative to the direct solving of the Navier-Stokes equations using classic numerical analysis. From an algorithmic standpoint, the LBM is well-suited for parallel implementations. The use of graphics processors to perform general purpose computations is increasingly widespread in high performance computing. These massively parallel circuits provide up to now unrivalled performance at a rather moderate cost. Yet, due to numerous hardware induced constraints, GPU programming is quite complex and the possible benefits in performance depend strongly on the algorithmic nature of the targeted application. For LBM, GPU implementations currently provide performance two orders of magnitude higher than a weakly optimised sequential CPU implementation.

The present thesis consists of a collection of nine articles published in international journals and proceedings of international conferences (the last one being under review). These contributions address the issues related to single-GPU implementations of the LBM and the optimisation of memory accesses, as well as multi-GPU implementations and the modelling of inter-GPU and inter-node communication. In addition, we outline several extensions to the LBM, which appear essential to perform actual building thermo-aeraulic simulations. The test cases we used to validate our codes account for the strong potential of GPU LBM solvers in practice.

Keywords: high performance computing, lattice Boltzmann method, graphics processing units, building aeraulics.

Foreword

The present document is a thesis by publication consisting of a general introduction and the nine following articles:

[A] A New Approach to the Lattice Boltzmann Method for Graphics Processing Units. *Computers and Mathematics with Applications*, 12(61):3628–3638, June 2011.

[B] Global Memory Access Modelling for Efficient Implementation of the Lattice Boltzmann Method on Graphics Processing Units. *Lecture Notes in Computer Science 6449*, High Performance Computing for Computational Science – VECPAR 2010 Revised Selected Papers, pages 151–161, February 2011.

[C] The TheLMA project: a thermal lattice Boltzmann solver for the GPU. *Computers and Fluids*, 54:118–126, January 2012.

[D] Multi-GPU Implementation of the Lattice Boltzmann Method. *Computers and Mathematics with Applications*, published online March 17, 2011.

[E] The TheLMA project: Multi-GPU Implementation of the Lattice Boltzmann Method. *International Journal of High Performance Computing Applications*, 25(3):295–303, August 2011.

[F] Towards Urban-Scale Flow Simulations Using the Lattice Boltzmann Method. In *Proceedings of the Building Simulation 2011 Conference*, pages 933–940. IBPSA, 2011.

[G] Multi-GPU Implementation of a Hybrid Thermal Lattice Boltzmann Solver using the TheLMA Framework. *Computers and Fluids*, published online March 6, 2012.

[H] Efficient GPU Implementation of the Linearly Interpolated Bounce-Back Boundary Condition. *Computers and Mathematics with Applications*, published online June 28, 2012.

[I] Scalable Lattice Boltzmann Solvers for CUDA GPU Cluster. Submitted to *Parallel Computing*, August 22, 2012.

All the papers are authored by Christian Obrecht, Frédéric Kuznik, Bernard Tourancheau, and Jean-Jacques Roux. They are hereinafter referred to as Art. A, Art. B, and so forth. The page numbering is specific to each article, e.g. B–1, B–2, etc.

Contents of the General Introduction

General introduction

> *The real problem is that programmers have spent far too much time worrying about efficiency in the wrong places and at the wrong times; premature optimization is the root of all evil (or at least most of it) in programming.*
>
> D.E. Knuth — Computer Programming as an Art, 1974

Is accurate building performance simulation possible? Taking into account the tremendous computational power of modern computers, one might be tempted to give an immediate positive answer to this question. Yet, buildings are complex systems interacting in numerous ways with their environment, at various time and length scales, and accurate simulations often appear to be of prohibitive computational cost.

The design of low or zero energy buildings even increases the need for accurate modelling of the heat and mass transfer between the outdoor or indoor environment and the envelope of a building. The commonplace practice regarding building aeraulics is to use simplified empirical or semi-empirical formulae. However, as outlined in [4], such approaches only provide a crude indication of the relevant parameters.

Regarding indoor simulations, nodal [6] and zonal [22] models are also widespread. In the former approach, the air in a given room is represented by a single node; in the latter, the room is divided into macroscopic cells. These methods, which require low computational efforts but simulate large volumes of air using single nodes, often lead to unsatisfactory results when compared to experimental data [9].

To achieve adequate accuracy, the use of computational fluid dynamics (CFD) in building aeraulics appears therefore to be mandatory. Yet, the computational cost and memory requirements of CFD simulations are often so high that they may not be carried out on personal computers. Since the access to high performance computing (HPC) facilities is limited and expensive, alternative approaches improving the performance of CFD solvers are of major practical interest in numerous engineering fields.

The present work explores the potential of graphics processing units (GPUs) to perform CFD simulations using the lattice Boltzmann method (LBM). This method is a rather recent approach for fluid flow simulation which is well-suited to HPC implementations as we shall see later. The investigations led to the design and creation of a framework going by the name of TheLMA, which stands for "Thermal LBM on Many-core Architectures". Several LBM solvers based on this framework were developed, ranging from single-GPU to GPU

cluster implementations, and addressing issues such as thermal flow simulation, turbulent flow simulation, or complex geometries representation.

The remainder of this introduction is organised as follows. Section I is an overview of general purpose computation technologies for the GPU. In section II, a description of the principles and algorithmic aspects of the lattice Boltzmann method is given. Section III focuses on single-GPU implementations of the LBM. Section IV outlines several extensions to the LBM, regarding thermal simulations, high Reynolds number flows and representation of complex geometries. Section V focuses on multi-GPU implementations of the LBM, either single-node or multi-node. Section VI concludes, giving a summary of the potential applications of the TheLMA framework in building simulation, and some perspective on the remaining issues.

I General purpose computing on graphics processors

a. A brief history

Graphics accelerators are meant to alleviate the CPU's workload in graphics rendering. These specialised electronic circuits became commonplace hardware in consumer-level computers during the 1990s. In these early years, graphics accelerators were chiefly rasterisation devices using fixed-function logic, and had therefore limited interest from a computing standpoint, although some attempts were made [21]. The first electronic circuit marketed as "graphics processing unit", was Nvidia's GeForce 256 released in 1999, which introduced dedicated hardware to process transform and lighting operations, but was still not programmable.

In 2001, Nvidia implemented a novel architecture in the GeForce 3, based on programmable shading units. With this technology, the graphics pipeline incorporates *shaders*, which are programs responsible for processing vertices and pixels according to the properties of the scene to render. From then on, GPUs could be regarded as *single instruction multiple data* (SIMD) parallel processors. At first, shaders had to be written in hardware-specific assembly language; nonetheless the new versatility of GPUs increased their potential for numerical computations [41].

The two leading 3D graphics APIs both introduced a high level shading language in 2002: HLSL for Microsoft's Direct3D, which is also known as Nvidia Cg [30], and GLSL for OpenGL. Both languages use a C-like syntax and a stream-oriented programming paradigm: a series of operations (the *kernel*) is applied to each element of a set of data (the *stream*). High level languages contributed to significantly enlarge the repertoire of general purpose computing on GPU (GPGPU) [35]. In many situations, because of their massively parallel architecture, GPUs would outperform CPUs. Nevertheless, GPGPU development remained cumbersome because of the graphics processing orientation of both Cg and GLSL. It further evolved with the release of BrookGPU, which provides a run-time environment and compiler for Brook, a general purpose stream processing language [7].

During the last decade, because of several issues such as the "frequency wall", the computational power of CPUs per core grew modestly, to say the least. Meanwhile, the improvements of the CMOS fabrication process made possible to constantly increase the number of shading units per GPU, and therefore, to increase the raw computational power. This situation led the Nvidia company to consider the HPC market as a new target and to develop a novel technology labeled "Compute Unified Device Architecture" (CUDA) [24].

The advent of the CUDA technology in 2007 is probably the most significant breakthrough in the GPGPU field up to now. The core concepts consist of a set of abstract hardware specifications, together with a parallel programming model. CUDA often refers to the "C for CUDA" language, which is an extension

to the C/C++ language (with some restrictions) based on this programming model. Compared to previous technologies, CUDA provides unprecedented flexibility in GPGPU software development, which contributed to bring GPU computing to the forefront of HPC.

The proprietary status of the CUDA technology is a major drawback: only Nvidia GPUs are capable of running CUDA programs. The OpenCL framework [17], first released in 2008, provides a more generic approach, since it is designed for heterogeneous platforms. OpenCL is supported by most GPU vendors today. It is worth noting that the OpenCL programming model and the OpenCL language share many similarities with their CUDA counterparts. From an HPC standpoint, OpenCL is a promising standard which might override CUDA in future.

The present work focuses on CUDA platforms, mainly because, at the time it began, the CUDA technology was by far more mature than OpenCL. However, it should be mentioned that, since CUDA and OpenCL are closely related, several contributions of this research work might also be valuable on OpenCL platforms.

b. CUDA hardware and execution model

The CUDA hardware model consists of abstract specifications that apply to each Nvidia GPU architecture since the G80. The differences in features between GPU generations is represented by the "compute capability" which consist of a major and a minor version number. The GT200 GPU featured in the Tesla C1060 computing board, which was used for most of the present work, has compute capability 1.3. The GF100 GPU, also known as "Fermi", has compute capability 2.0.

A CUDA capable GPU is described in the hardware model as a set of "streaming multiprocessors" (SMs). An SM consists merely of an instruction unit, several "scalar processors" (SPs) (namely 8 for the GT200 and 32 for the GF100), a register file partitioned among SPs, and shared memory. Both register memory and shared memory are rather scarce, the later being for instance limited to 64 KB with the GF100. In the CUDA terminology, the off-chip memory associated to the GPU is called "device memory". In order to reduce latency when accessing to device memory, SMs contain several caches: textures, constants, and—as of compute capability 2.0—data.

The CUDA execution model, coined by Nvidia as "single instruction multiple threads" (SIMT), is rather complex because of the dual level hardware hierarchy. The SMs belong to the SIMD category, although they do not reduce to pure vector processors, being e.g. able to access scattered data. At global level, execution is better described as *single program multiple data* (SPMD) since SMs are not synchronised.[1] It should also be mentioned that, whereas efficient com-

[1] Starting with the Fermi generation, the execution at GPU level might even be described as *multiple program multiple data* (MPMD) since several kernels can be run concurrently.

munication within an SM is possible through on-chip shared memory, sharing information at global level requires the use of device memory and may therefore suffer from high latency.

Unlike streaming languages, which are data centred, the CUDA programming paradigm is task centred. An elementary process is designated as a *thread*, the corresponding series of operations is named *kernel*. To execute a CUDA kernel, it is mandatory to specify an "execution grid". A *grid* is a multidimensional array of identical *blocks*, which are multidimensional arrays of threads.[2] This dual level scheme is induced by the architecture: a block is to be processed by a single SM; consequently, the number of threads a block can hold is limited by the local resources.

Depending on the available resources, an SM may run several blocks concurrently. For computationally intensive application, it might be of great interest to tailor the dimensions of the blocks in order to achieve high arithmetic intensity, i.e. for the SMs, to host as many active threads as possible. It is worth mentioning that grid dimensions are less constraint than block dimensions. As described in [8], blocks are processed asynchronously in batches. A grid may therefore contain far more blocks than a GPU is actually able to run concurrently.

Because of the SIMD nature of an SM, the threads are not run individually but in groups named *warps*. Up to now, a warp contains 32 threads; yet, this value is implementation dependent and might change for future hardware generations. The warp being an atomic unit, it is good practice to ensure that the total number of threads in a block is a multiple of the warp size. Warps induce several limitations such as in conditional branching, for instance: when branch divergence occurs within a warp, the processing of the branches is serialised. Whenever possible, branch granularity should be greater than the warp size. In many situations, the design of an execution grid is not a trivial task, and might be of cardinal importance to achieve satisfactory performance.

c. Memory hierarchy and data transfer

As outlined in the preceding subsection, the CUDA memory hierarchy is fairly complex from an architectural standpoint, but it is even more intricate from a programming standpoint. In CUDA C, a kernel is merely a void-valued function. Grid and block dimensions as well as block and thread indices are accessible within the kernel using built-in read-only variables. CUDA memory spaces fall into five categories: local, shared, global, constant, and textures.

Automatic variables in a kernel, which are proper to each thread, are stored in registers whenever possible. Arrays and structures that would consume too much registers, as well as arrays which are accessed using unpredictable in-

[2] As of compute capability 2.0, grids and blocks may have up to three dimensions, prior to this, grids were limited to two dimensions.

dices, are stored in local memory.[3] Local memory, which is hosted in device memory, is also used to spill registers if need be. Prior to the Fermi generation, local memory was not cached. Greatest care had therefore to be taken in kernel design in order to avoid register shortage.

Threads may also access to shared and global memory. The scope and lifetime of shared memory is limited to the local block. Being on chip, it may be as fast as registers, provided no bank conflicts occurs.[4] Global memory is visible by all threads and is persistent during the lifetime of the application. It resides in device memory which is the only accessible by the host system. Transferring from and to global memory is the usual way for an application to provide data to a kernel before launch and to retrieve results once kernel execution has completed.

Constant and shared memory are additional memory spaces hosted in device memory, which may be accessed read-only by the threads. Both are modifiable by the host. The former is useful to store parameters that remain valid across the lifetime of the application; the later is of little use in GPGPU and shall not be discussed further.

With devices of compute capability 1.3, global memory is not cached. Transactions are carried out on aligned segments[5] of either 32, 64, or 128 bytes. At SM level, global memory requests are issued by half-warps and are serviced in as few segment transactions as possible. Devices of compute capability 2.0 feature L1 and L2 caches for global memory. L1 is local to each SM and may be disabled at compile time, whereas L2 is visible to all SMs. Cache lines are mapped to 128 bytes aligned segments in device memory. Memory accesses are serviced with 128-byte memory transactions when both caches are activated and with 32-byte memory transactions when L1 is disabled, which may reduce over-fetch in case of scattered requests.

It is worth stressing that for both architectures, although being significantly different, global memory bandwidth is best used when consecutive threads access to consecutive memory locations. Data layout is therefore an important optimisation issue when designing data-intensive application for CUDA.

[3] Registers being not addressable, an array may only be stored in the register file when its addressing is known at compile time.

[4] The shared memory is partitioned in several memory banks. A bank conflict occurs when several threads try to access concurrently to the same bank. To resolve the conflict, the transactions must be serialised.

[5] An *aligned* segment is a block of memory whose start address is a multiple of its size.

II Principles of the lattice Boltzmann method

a. Origins of the lattice Boltzmann method

The path followed by usual approaches in CFD may be sketched as "top-down": a set of non-linear partial differential equations is discretised and solved using some numerical technique such as finite differences, finite volumes, finite elements, or spectral methods. As pointed out in [51], attention is often drawn on truncation errors induced by the discretisation. However, from a physical standpoint, the fact that the desired conservation properties still hold for the discretised equations is of major importance, especially for long-term simulations in which small variations may by accumulation lead to physically unsound results. The preservation of these conservation properties is generally not guaranteed by the discretisation method.

Alternatives to the former methods may be described as "bottom-up" approaches, molecular dynamics (MD) being the most emblematic one. In MD, the macroscopic behaviour of fluids is obtained by simulating as accurately as possible the behaviour of individual molecules. Because of the tremendous computational effort required by even small scale MD simulations, the scope of applications is limited. Lattice-gas automata (LGA) and LBM fall into the same bottom-up category than MD although they act at a less microscopic scale, which is often referred to as *mesoscopic*.

LGA are a class of cellular automata attempting to describe hydrodynamics through the discrete motion and collision of fictitious particles. In LGA, space is represented by a regular lattice in which particles hop from one node to another at each time step. Most LGA models obey to an *exclusion principle*, which allows only one particle for each lattice direction to enter a node at a time. A node may therefore only adopt a finite number of states, which are usually represented by a bit field. The simulation process consist of an alternation of *collisions*, in which the state of each node determines the local set of out-going particles, and *propagation*, during which the out-going particles are advected to the appropriate neighbour nodes.

The first two-dimensional LGA model was proposed by Hardy, Pomeau, and de Pazzis in 1973 [18]. It is known as HPP after its inventors. HPP operates on a square lattice with collision rules conserving mass and momentum. Yet, HPP lacks rotational invariance and therefore cannot yield the Navier-Stokes equations in the macroscopic limit, which drastically reduces its practical interest. In the two-dimensional case, this goal was first achieved with the FHP model introduced in 1986 by Frisch, Hasslacher, and Pomeau [15]. Increased isotropy is obtained by using an hexagonal grid instead of a square lattice.

In the three-dimensional case, achieving sufficient isotropy is by far more difficult than with two dimensions, and this could only be solved by using a four-dimensional face-centred hypercube (FCHC) lattice [12]. FCHC yields 24-bit wide states which in turn causes the look-up tables of the collision rules to be

very large. This issue, beside the complexity induced by the fourth dimension, is a serious drawback for the use of 3D LGA models in practice.

Another general disease of LGA models is the statistical noise caused by the use of Boolean variables. To address this issue, McNamara and Zanetti in 1988 [31], proposed the first lattice Boltzmann model, replacing the Boolean fields by continuous distributions over the FHP and FCHC lattices. Later on, in 1992, Qian, d'Humières, and Lallemand [37], replaced the Fermi-Dirac by the Maxwell–Boltzmann equilibrium distribution and the collision operator inherited from the LGA models by a linearised operator based on the Bhatnagar–Gross–Krook (BGK) approximation [2]. This model, often referred to as LBGK (for lattice BGK), is still widely in use today.

A conclusion to this emergence period was brought by the work of He and Luo in 1997 [20]. The lattice Boltzmann equation (LBE), i.e. the governing equation of the LBM, may be derived directly from the continuous Boltzmann equation. From a theoretical standpoint, the LBM is therefore a standalone approach sharing similarities with LGA but no actual dependency.

b. Isothermal fluid flow simulation

The Boltzmann equation describes the hydrodynamic behaviour of a fluid by the means of a one-particle distribution function f over phase space, i.e. particle position $\boldsymbol{x}$ and velocity $\boldsymbol{\xi}$, and time:

$$\partial_t f + \boldsymbol{\xi} \cdot \nabla_{\boldsymbol{x}} f + \frac{\boldsymbol{F}}{m} \cdot \nabla_{\boldsymbol{\xi}} f = \Omega(f). \tag{1}$$

In the former equation, m is the particle mass, $\boldsymbol{F}$ is an external force, and Ω denotes the collision operator. In a three-dimensional space, the macroscopic quantities describing the fluid obey:

$$\rho = \int f \, \mathrm{d}\boldsymbol{\xi} \tag{2}$$

$$\rho \boldsymbol{u} = \int f \boldsymbol{\xi} \mathrm{d}\boldsymbol{\xi} \tag{3}$$

$$\rho \boldsymbol{u}^2 + 3\rho\theta = \int f \boldsymbol{\xi}^2 \mathrm{d}\boldsymbol{\xi} \tag{4}$$

where ρ is the fluid density, $\boldsymbol{u}$ the fluid velocity, and $\theta = k_\mathrm{B} T/m$ with T the absolute temperature and k_B the Boltzmann constant.

The LBM is based on a discretised form of Eq. 1 using a constant time step δt and a regular orthogonal lattice of mesh size δx. A finite set of velocities $\{\boldsymbol{\xi}_\alpha \,|\, \alpha = 0, \dots N\}$ with $\boldsymbol{\xi}_0 = \boldsymbol{0}$, is substituted to the velocity space. The velocity set or *stencil*, is chosen in accordance with the time step and mesh size: given a lattice site $\boldsymbol{x}$, $\boldsymbol{x} + \delta t \boldsymbol{\xi}_\alpha$ is on the lattice for any α. In the three-dimensional

case, three velocity sets are commonly considered (see Fig. 1). These stencils are named D3Q15, D3Q19, and D3Q27, following the notation proposed by Qian *et al.* in 1992.[6]

The discrete counterpart of the distribution function f is a set of particle populations $\{f_\alpha \,|\, \alpha = 0, \dots N\}$ corresponding to the velocities. With $\intercal$ being the transpose operator, let us denote: $|a_\alpha\rangle = (a_0, \dots a_N)^\intercal$. In the absence of external forces, Eq. 1 becomes:

$$|f_\alpha(\boldsymbol{x} + \delta t \boldsymbol{\xi}_\alpha, t + \delta t)\rangle - |f_\alpha(\boldsymbol{x}, t)\rangle = \boldsymbol{\Omega}\left(|f_\alpha(\boldsymbol{x}, t)\rangle\right). \tag{5}$$

Regarding the macroscopic variables of the fluid, Eqs. 2 and 3 become:

$$\rho = \sum_\alpha f_\alpha, \tag{6}$$

$$\rho \boldsymbol{u} = \sum_\alpha f_\alpha \boldsymbol{\xi}_\alpha. \tag{7}$$

It is possible to show, as for instance in [29], that Eqs. 6 and 7 are exact quadratures. This demonstration uses Gauss-Hermite quadratures on the low Mach number second-order expansion of the Maxwellian equilibrium distribution function, which is sufficient to derive the Navier-Stokes equation. As a consequence, mass and momentum conservation are preserved by the numerical scheme. It should be mentioned that the Gaussian quadrature method does not yield energy conserving models and is therefore only suitable to build isothermal models.

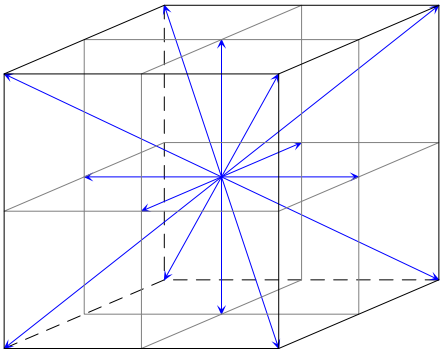
Besides LBGK, which is also known as single-relaxation-time LBM, numerous alternative models have been proposed over the past years, e.g. multiple-relaxation-time (MRT) [10], entropic lattice Boltzmann [23], or regularised lattice Boltzmann [27]. The discussion of the advantages and drawbacks of the various approaches is beyond the scope of this introduction. It should nevertheless be mentioned that MRT collision operators are implemented in all but one solver described in the present collection of articles. MRT operators are explicit, like LBGK operators, and provide increased stability and accuracy at the cost of a slightly higher arithmetic complexity. An overview of MRT is to be found e.g. in Art. D, whereas a comprehensive presentation is given in [11].

c. Algorithmic aspects

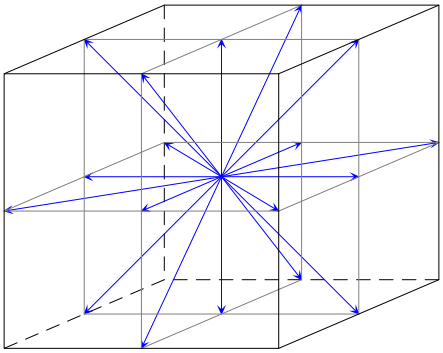
Like LGA, from an algorithmic standpoint, the LBM consists of an alternation of collision and propagation steps. The collision step is governed by:

$$|\widetilde{f}_\alpha(\boldsymbol{x}, t)\rangle = |f_\alpha(\boldsymbol{x}, t)\rangle + \boldsymbol{\Omega}\left(|f_\alpha(\boldsymbol{x}, t)\rangle\right), \tag{8}$$

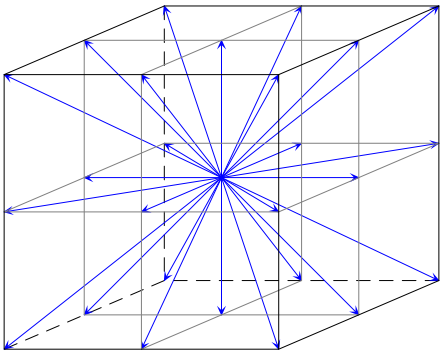
[6] In the DmQn notation, m is the spatial dimension and n is the number of velocities including the rest velocity $\boldsymbol{\xi}_0$.



(a) D3Q15



(b) D3Q19



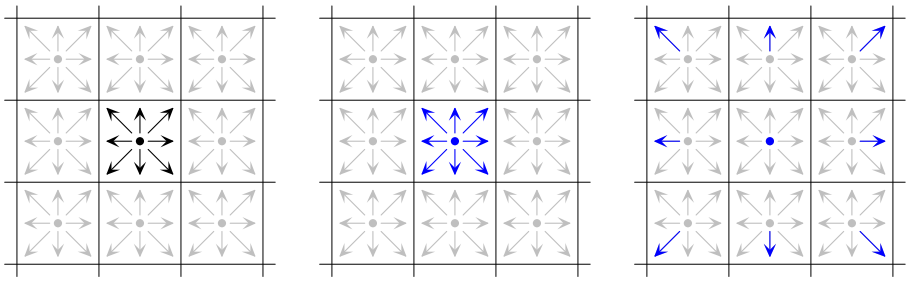
(c) D3Q27

Figure 1: Usual velocity sets in 3D LBM. — *The D3Q27 stencil links a given node to its 26 nearest neighbours in the lattice, D3Q15 and D3Q19 are degraded versions of the former.*

where $\widetilde{f}_\alpha$ denotes the post-collision particle populations. It is worth noting that the collision step is purely local to each node. The propagation step is described by:

$$\left|f_\alpha(\boldsymbol{x} + \delta t\, \boldsymbol{\xi}_\alpha, t + \delta t)\right\rangle = \left|\widetilde{f}_\alpha(\boldsymbol{x}, t)\right\rangle, \tag{9}$$

which reduces to mere data transfer. This two phase process is outlined for the two-dimensional D2Q9 stencil by Fig. 2.



(a) Pre-collision (b) Post-collision (c) Post-propagation

Figure 2: Collision and propagation. — *The nine pre-collision particle populations of the central cell are drawn in black, whereas the nine post-collision populations are drawn in blue. After collision, these populations are advected to the neighbouring nodes in accordance with the velocity set.*

The breaking of Eq. 5 makes the data-parallel nature of LBM obvious. The LBM is therefore well-suited for massively parallel implementations, provided the next-neighbours synchronisation constraint is taken care of. The simplest way to address this issue is to use two instances of the lattice, one for even time steps and the other for odd time steps. A now commonplace method, known as *grid compression* [50], makes possible to save almost half of the memory by overlaying the two lattices with a diagonal shift of one unit in each direction. Yet, this technique requires control over the schedule of node updates in order to enforce data dependency. It does therefore not apply in an asynchronous execution model.

In a shared memory environment, when using non-overlaid lattices, the data layout for 3D LBM simulations often reduces to a five-dimensional array: three dimensions for space, one for velocity indices, and one for time. Depending on the target architecture and memory hierarchy, the ordering of the array may have major impact on performance. The design of the data layout is therefore a key optimisation phase for HPC implementations of LBM.

III GPU implementations of the LBM

a. Early implementations

As mentioned in the previous section, data-parallel applications such as the LBM are usually well-suited for massively parallel hardware. Attempts to implement the LBM for the GPU were made in the very early days of GPGPU, starting with the contribution of Li *et al.* in 2003 [28]. At that time, because of the unavailability of a general purpose programming framework, the particle distribution had to be stored as a stack of two-dimensional texture arrays. With this approach, the LBE needs to be translated into rendering operations of the texturing units and the frame buffer, which is of course extremely awkward and hardware dependent. Beside issues such as the rather low accuracy of the computations, the authors of [28] had also to face the limited amount of on-board memory which only made possible very coarse simulations.[7] This work was later extended by Fan *et al.* in 2004 [14], which describes the first GPU cluster implementation of the LBM. The authors report a simulation on a $480 \times 400 \times 80$ lattice with 30 GPUs, demonstrating the practical interest of GPU LBM solvers.

In 2008, Tölke reported the first CUDA implementation of 2D LBM [43], and later the same year, with Krafczyk, the first CUDA implementation of 3D LBM [44]. The general implementation principles are the same in both works and remain, for the most part, valid until today. The authors focus on minimising the cost of data transfer between GPU and global memory, since the target architecture, i.e. the G80, like the later GF100 does not provide cache for global memory. The following list gives an outline of these principles:

1. Fuse collision and propagation in a single kernel to avoid unnecessary data transfer.

2. Map the CUDA execution grid to the lattice, i.e. assign one thread to each lattice node. This approach, by creating a large number of threads, is likely to take advantage of the massive parallelism of the GPU and to hide the latency of the global memory.

3. Use an appropriate data layout in order for the global memory transactions issued by the warps to be coalesced whenever possible.

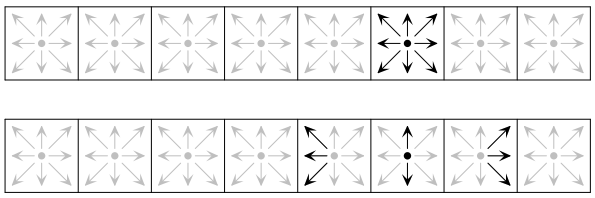
4. Launch the collision and propagation kernel at each time step and use two instances of the lattice in order to enforce local synchronisation.

The implementations described in [43] and [44] use one-dimensional blocks of threads and specific two- or three-dimensional arrays for each velocity index. It should be noted that, because of the constraints on the block size, the size

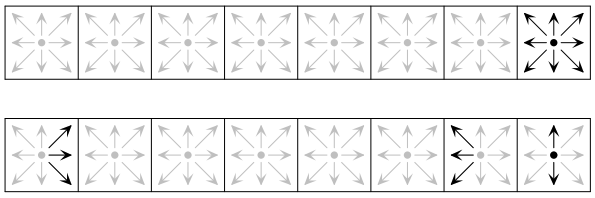
[7] The Nvidia GeForce4 used for the implementation has only 128 MB of memory.

of the lattice along the blocks' direction is better chosen to be a multiple of the warp size. Such a set-up allows the memory accesses to be coalesced when fetching the pre-collision distributions. Yet, special care is taken for the propagation step. Propagation reduces to data shifts along the spatial directions, including the minor dimension of the distribution arrays. Thus, directly writing back to global memory leads to memory transactions on non-aligned segments. With the G80, this would have a dramatic impact on performance, since in this case memory accesses are serviced individually.

To address the former issue, Tölke proposes to store the particle distribution within a block in shared memory and to break propagation in two steps. At first, a partial propagation is performed along the blocks' direction within shared memory, then propagation is completed while storing the distribution back to global memory. Fig. 3 outlines the method (using eight nodes wide blocks for the sake of clarity). By suppressing shifts along the minor dimension of the distribution arrays, this approach fully eliminates misalignments for global memory write accesses during propagation. However, when the blocks do not span the entire width of the domain, which may occur at least with large two-dimensional simulations, nodes located at the borders of the blocks but not on the domain's boundary need special handling: the particle populations leaving the block are temporarily stored in the unused places at the opposite border, as illustrated in Fig. 3b. After the execution of the main kernel, a second kernel is then needed to reorder the distribution arrays.



(a) General case



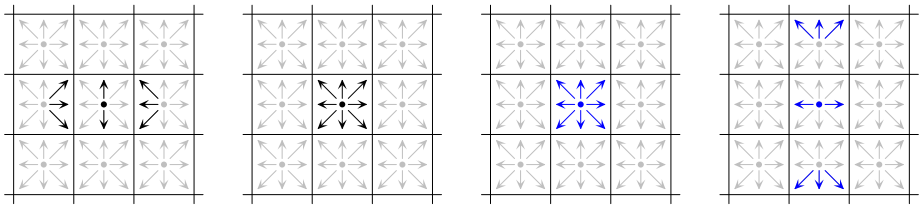
(b) Borders of the block

Figure 3: Partial propagation in shared memory. — *In the general case, the relevant particle populations are simply advected along the blocks' direction. For the border nodes, the populations leaving the block are stored in the unused array cells corresponding to the populations entering the block at the opposite border.*

b. Direct propagation schemes

Implementing the LBM for CUDA devices of compute capability 1.0, as the aforementioned G80, is quite a challenging task because of the constraints induced by hardware limitations on data transfer. The work of Tölke and Krafczyk achieves this goal in a rather efficient way since the authors report up to 61% of the maximum sustained throughput in 3D. Transition from compute capability 1.0 to 1.3 significantly decreased the cost of misaligned memory transactions. For 32-bit word accesses, e.g. single precision floating point numbers, a stride of one word yields an additional 32 B transaction by half-warp, i.e. a 50% increase of transferred data. It seems therefore reasonable to consider direct propagation schemes as alternatives to the shared memory approach. The results of our experiments are reported in Art. A.

Basic investigations on a GeForce GTX 295 led us to the conclusion that misaligned memory transactions still have noticeable influence on data transfer for CUDA devices of compute capability 1.3, but that misaligned read accesses are significantly less expensive than misaligned write accesses. We therefore experimented two alternatives to the elementary out-of-place propagation scheme. The first one, which we named *split scheme*, consist in breaking the propagation in two half-steps as illustrated in Fig. 4. After collision, the updated particle distributions are advected in global memory in all directions except along the minor dimension. The remainder of the propagation is carried out before collision at the next time step. With such a procedure, misalignement occurs only when loading data.

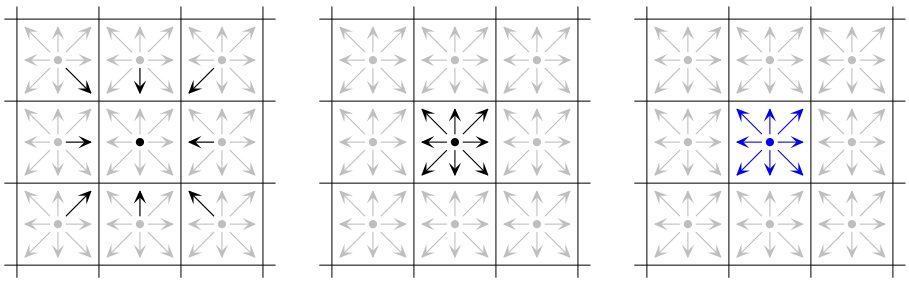


(a) Initial state (b) Pre-collision (c) Post-collision (d) Final state

Figure 4: Split propagation scheme. — *The data transfer corresponding to the propagation along the minor dimension is represented by the transition between (a) and (b) whereas the second propagation half-step is represented by the transition between (c) and (d).*

The second alternative propagation scheme is the *in-place scheme*, also referred to as *reversed scheme* in Art. A and B. As shown in Fig. 5, the in-place scheme consists in reversing collision and propagation. At each time step, the kernel performs propagation while gathering the local particle distribution with the appropriate strides. After collision, the updated particle distribution is stored back in global memory without any shift. As for the split scheme, the

in-place scheme avoids any misaligned memory transaction when writing data to global memory.



(a) Initial state (b) Pre-collision (c) Post-collision

Figure 5: In-place propagation scheme. — *Post-collision particle populations from the former time step are gathered to the current node. Collision is then carried out and the new particle distribution is written back to global memory.*

In Art. A, we describe two implementations of 3D LBM; the first one is based on the split scheme and the LBGK collision operator, and the second one is based on the in-place scheme and the MRT collision operator. Both solvers show similar performance with about 500 million lattice node updates per second (MLUPS) in single precision, the data throughput being over 80% of the maximum sustained throughput. Hence, communication appears to be the limiting factor as for the implementation of Tölke and Krafczyk. For 3D simulations, provided no auxiliary kernel is needed, the shared memory approach is likely to provide even better performance than the split propagation or the in-place propagation approach. However, these direct propagation schemes are of genuine practical interest, since they lead to significantly simpler code and exert less pressure on hardware, leaving the shared memory free for additional computations as we shall see in the case of thermal simulations.

The performance obtained by CUDA LBM solvers (in single precision) using a single GPU is more than one order of magnitude higher than the performance (in double precision) reported for HPC systems such as the NEC SX6+ or the Cray X1 [49]. Considering furthermore the moderate cost of GPU based computing devices makes obvious the great potential of GPU LBM solvers for realistic engineering applications.

c. Global memory access modelling

CUDA implementations of the LBM tend to be communication-bound. Although some information is given in the CUDA programming guide [33], most of the data transfer mechanisms between GPU and device memory remain undocumented. Art. B reports the investigations we undertook to gain a better understanding of these aspects. Our study aims at devising possible optimisation

strategies for future CUDA LBM implementations, and at providing a performance model suitable for a wide range of LBM based simulation programs.

The GT200 GPU we used for our benchmarks is divided into ten texture processing clusters (TPCs), each containing three SMs. Within a TPC, the SMs are synchronised and the corresponding hardware counter is accessible via the CUDA `clock()` function. We implemented a program able to generate benchmark kernels mimicking the data transfer performed by an actual LBM kernel. At start, the benchmark kernel loads a given number N of registers from global memory which are afterwards stored back. Several parameters such as the number of misaligned load or store transactions, or the number k of warps per SM are tunable. The generated code incorporates appropriate data dependencies in order to avoid aggressive compiler optimisations. The `clock()` function is used to measure the durations of both the loading phase and the storing phase, making possible to evaluate the actual throughput in both directions.

The results reported in Art. B show that the behaviour of the memory interface is not uniform, depending on whether $N \leq 20$ or not. In the first case, we estimate the time to process k warps per SM with $T = \ell + T_R + T_W$, where ℓ is the launch time of the last warp, T_R is the average read time, and T_W is the average write time. We show that ℓ only depends on k, and that, for a given number of misalignments, T_R and T_W depend linearly on N. The knowledge of T for the appropriate parameters, makes possible to evaluate the expected performance P (in MLUPS) of an LBM kernel: $P = (K/T) \times F$, where K is the global number of active threads and F is the GPU frequency in MHz. Comparing this performance model to actual performance values is useful to evaluate the opportunity of additional optimisations.

IV Extensions to the lattice Boltzmann method

a. Hybrid thermal lattice Boltzmann method

Although isothermal fluid flow solvers are of practical interest in building aeraulics, it is often desirable to take thermal effects into account. Art. C and G report our endeavour to implement thermal LBM solvers for the GPU. Usual lattice Boltzmann models such as the D3Q19 MRT fail to conserve energy and numerous attempts to apply the LBM to thermo-hydrodynamics are to be found in literature. Beside others, one should mention multi-speed models [36], using larger velocity sets in order to enforce energy conservation, double-population models [19], using an energy distribution in addition to the particle distribution, or hybrid models, in which the energy equation is solved by other means such as finite differences. As shown in [26], both multi-speed models and double-population models suffer from inherent numerical instabilities. Moreover, from an algorithmic standpoint, both approaches significantly increase the requirements in global memory as well as the volume of transfered data which has a direct impact on performance for communication-bound applications such as GPU LBM solvers.

We therefore chose to implement the hybrid thermal lattice Boltzmann model described in [26]. The hydrodynamic part is solved using a slightly modified MRT model in which a temperature coupling term is added to the equilibrium of the internal energy moment. The temperature T is solved using a finite difference equation. In the case where the ratio of specific heats $\gamma = C_P/C_V$ is set to $\gamma = 1$, this equation may be written as:

$$\partial_t^* T = \kappa \Delta^* T - \boldsymbol{j} \cdot \nabla^* T \tag{10}$$

where $\boldsymbol{j}$ is the momentum, κ is the thermal diffusivity, and where ∂_t^*, Δ^*, and ∇^* denote the finite difference operators, the two later using the same stencil as the associated lattice Boltzmann equation.

The single-GPU implementation of this model, which is described in Art. C, is derived from our single-GPU D3Q19 MRT solver. We use a two-dimensional execution grid of one-dimensional blocks spanning the simulation domain in the x-direction. Processing Eq. 10 for a given node requires to have access to the nineteen temperatures corresponding to the D3Q19 stencil, but leads to only one additional store operation for the local temperature. In order to reduce read redundancy, the kernel fetches at start the temperatures of all the nodes neighbouring the current block into shared memory. As illustrated by Fig. 6, each thread is responsible for reading the temperatures of the nodes sharing the same abscissa. By reducing the amount of additional reads by more than one half, this approach leads to rather satisfying performance with around 300 MLUPS in single precision using a single Tesla C1060 computing device.

The single-GPU solver described in Art. C and the multi-GPU solver described in Art. G were both tested using the differentially heated cubic cavity.

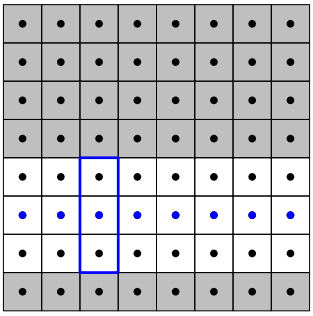


Figure 6: Read access pattern for temperature. — *The nodes associated to the current block are represented by blue dots. The white background cells stand for the nodes whose temperature is needed to solve the finite difference equation. The blue frame represents the zone assigned to a thread for temperature fetching.*

For validation purpose, we computed the Nusselt numbers at the isothermal walls. The obtained values are in good agreement with previously published results [42, 46]. Using the multi-GPU version, we could perform simulations for Rayleigh numbers up to 10^9.

b. Large eddy simulations

Multi-GPU LBM solvers, such as the one described in Art. D and E, make possible to perform fluid simulations on very large computation domains. When using a Tyan B7015 server equipped with eight Tesla C1060, each providing 4 GB of memory, a cubic computation domain may be as large as 576^3 in single precision, i.e. contain more than 190 million nodes. Such a fine resolution allows the Reynolds number of the simulation to be of the order of 10^3 to 10^4. In external building aeraulics, however, the typical Reynolds numbers are of the order of 10^6. Because of turbulence phenomena, performing direct numerical simulation for such applications appears therefore to be out of reach at the present time.

We see that, to be of practical interest in external building aeraulics, LBM solvers need to incorporate a sub-grid scale model such as large eddy simulation (LES). As a first attempt, we chose to implement the most elementary LES model, proposed by Smagorinsky in 1963 [40]. The simulation results we obtained using this extended version of our multi-GPU isothermal LBM solver are reported in Art. F. In the Smagorinsky model, a turbulent viscosity ν_t is added to the molecular viscosity ν_0 to obtain the kinematic viscosity ν of the simulation: $\nu = \nu_0 + \nu_t$. The turbulent viscosity is given by:

$$\nu_t = |\mathsf{S}|\,(C_S \delta x)^2, \quad |\mathsf{S}| = \sqrt{2\,\mathsf{S} : \mathsf{S}}, \tag{11}$$

where C_S is the Smagorinsky constant, which is usually set to $C_S = 0.1$, and S is

the strain rate tensor. As shown in [25], the MRT approach has the interesting feature that the strain rate tensor can be determined directly from the moments computed when applying the collision operator. The computations being fully local, the impact on the performance of a communication-bound program such as our fluid flow solver is negligible.

In Art. F, we report the simulation of the flow around a group of nine wall-mounted cubes at Reynolds number $\mathrm{Re} = 10^6$. In order to obtain relevant time-averaged pressure and velocity fields, the computations were carried out for a duration of $200T_0$, where T_0 is turn-over time corresponding to the obstacle size and the inflow velocity. These results could not be validated against experiments, due to the lack of data. However, the overall computation time being less than eighteen hours, this study demonstrates the feasibility of simulating the flow around a small group of buildings in reasonable time.

The original Smagorinsky LES model is a rather simplistic turbulence modelling approach with well-known defects such as the inability to fulfil the wall boundary law [32]. More elaborate methods such as the wall-adapting local eddy-viscosity (WALE) model proposed in [32] were reported to be suitable for the LBM [48]. Further investigations seem therefore to be required before considering using our LBM solver for actual building aeraulics simulations.

c. Interpolated bounce-back boundary conditions

With the LBM, fluid-solid boundaries result in unknown particle populations. Considering a boundary node, i.e. a fluid node next to at least one solid node, it is easily seen that the populations, which should be advected from the solid neighbouring nodes, are undefined. A common way to address this issue is the simple bounce-back (SBB) boundary condition, which consists in replacing the unknown populations with the post-collision values at the former time step for the opposite directions. Let $\boldsymbol{x}$ denote a boundary node, the SBB obeys:

$$f_{\bar{\alpha}}(\boldsymbol{x}, t) = \widetilde{f}_{\alpha}(\boldsymbol{x}, t - \delta t) \tag{12}$$

where $f_{\bar{\alpha}}(\boldsymbol{x}, t)$ is an unknown particle population and $\bar{\alpha}$ is the direction opposite to α. As shown in [16], SBB is second-order accurate in space. The fluid-solid interface is located about half-way between the boundary node and the solid nodes.

One sees from Eq. 12 that applying SBB to a boundary node is straightforward, provided the list of solid neighbours is known. Moreover, it should be noted that the SBB is well-suited not only for straight walls but for any stationary obstacle. Because of its versatility and simplicity, we chose SBB for most of our LBM implementations. We generally use an array of bit-fields to describe the neighbourhood of the nodes[8]. This array is set up at start by a specific

[8] In the case of an empty cavity, the neighbourhood bit-fields are simply determined from the coordinates in order to avoid global memory accesses.

kernel, according to the simulation layout. This approach allowed us to easily implement new test cases, even when complex geometries are involved as in the nine wall-mounted cubes simulation described in Art. F.

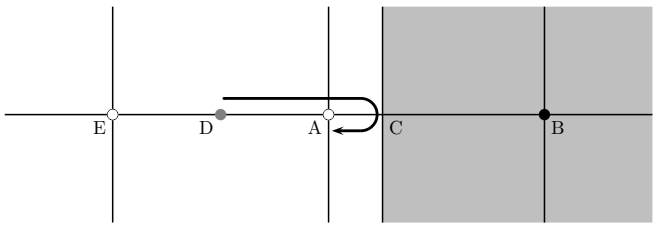
Since the fluid-solid interface has a fixed location, the use of the SBB is convincing as long as the considered bodies have flat faces parallel to the spatial directions. A curved boundary or an inclined straight wall leads to stepping effects that may have an impact on the simulated flow. In order to study these aspects, we chose to implement an extension to the SBB known as linearly interpolated bounce-back (LIBB) which takes into account the exact location of the fluid-solid interface [5].

As illustrated in Fig. 7, the LIBB is based on the determination of a fictitious particle population entering the boundary node. Two different interpolation formulae are used, depending on the location of the interface. Keeping the same notations as in Eq. 12, let q be the number such that $\boldsymbol{x} + q\delta t\boldsymbol{\xi}_\alpha$ is on the fluid-solid interface. For $q < 1/2$,

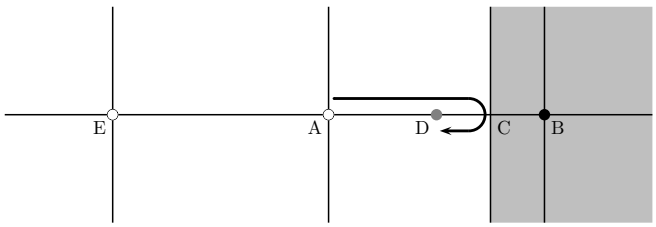
$$f_{\bar{\alpha}}(\boldsymbol{x}, t) = (1-2q)f_\alpha(\boldsymbol{x}, t) + 2q\widetilde{f}_\alpha(\boldsymbol{x}, t-\delta t) \tag{13}$$

and for $q \geq 1/2$,

$$f_{\bar{\alpha}}(\boldsymbol{x}, t) = \left(1 - \frac{1}{2q}\right)\widetilde{f}_{\bar{\alpha}}(\boldsymbol{x}, t-\delta t) + \frac{1}{2q}\widetilde{f}_\alpha(\boldsymbol{x}, t-\delta t). \tag{14}$$



(a) $q < 1/2$



(b) $q \geq 1/2$

Figure 7: Interpolation schemes of the LIBB boundary condition. — *In the first case, the fictitious particle population leaving D and entering A is interpolated from the post-collision values at E and A. In the second case, the particle population leaving A ends up at D. The population entering A is interpolated from the post-propagation values at E and D.*

The LIBB is interesting from an algorithmic standpoint since it can take advantage from the in-place propagation scheme, thus only slightly increasing the volume of communications. The details of our implementation strategy are given in Art. H. This work also reports the results obtained when simulating the flow past a sphere using either SBB or LIBB. Comparing the Strouhal numbers computed from our simulations to experimental results [39, 34] shows that LIBB improves the stability as well as the accuracy of the vortex shedding frequency. However, this conclusion does not seem to hold in general. In further investigations, yet unpublished, we simulated the flow past an inclined flat plate. For this test case, we could not see significant differences in the obtained Strouhal numbers.

The process of validating our approach in the perspective of realistic engineering applications is still at an early stage. Regarding LIBB, additional studies focusing on possibly more relevant parameters such as the drag coefficient should be carried out. Moreover, a wide variety of alternative boundary conditions for the LBM is to be found in literature, some of which being well-suited for efficient GPU implementations,could also be tested. As shown in Art. F and H, GPU LBM solvers make large scale validation studies possible, and thus may contribute to evaluate novel modelling strategies.

V Large scale lattice Boltzmann simulations

a. The TheLMA framework

CUDA, being a recent technology, imposes strong constraints to the programmer, mainly induced by hardware limitations. Although this situation has evolved with the increasing capabilities of the different hardware generations, as well as the progresses made in the compilation tool-chain, some of the most acknowledged practices of software engineering, such as library-oriented development, are still not relevant. However, our research work led us to develop multiple versions of our LBM solver, implementing a wide variety of models and simulation layouts, and targeted for several different hardware configurations. We therefore had to devise an appropriate strategy to enforce both code reusability and maintainability, which resulted in the design and creation of the TheLMA framework.

A CUDA program usually consists of a set of CUDA C source files together with some additional plain C or C++ files. Although it is possible to build a program from sole CUDA C files, it is in general good practice to split the GPU related functions from the remainder of the code. In addition to kernels, a CUDA C source file may contain *device* functions and *host* functions. The first category corresponds to functions that are run by the GPU and may only be called by CUDA threads, i.e. from within a kernel or another device function. In practice, most device functions are inlined at compile-time, which restricts their use to short auxiliary functions[9]. The second category corresponds to functions that are run by the CPU and may be called by external C or C++ function as well as launch kernels using a specific syntax to stipulate the execution grid. Host device thus provide a convenient way to launch GPU computations from an external plain C module[10].

Up to now, the most severe limitation of the CUDA compilation tool-chain is the absence of an actual linker[11]. Consequently, all compilation symbols (e.g. device functions, device constants...) related to a kernel must lie in the same compilation unit. A commonplace way to achieve some degree of modularity consist in using inclusion directives in order to merge separate CUDA source files into a single one before compilation.

The overall structure of the TheLMA framework is outlined in Fig. 8. It consists in a set of data structure definitions, C and CUDA C source files. The C files provide a collection of functions useful to process the configuration parameters, initialise the global data structures, post-process the simulation results,

[9] As of compute capability 2.0, CUDA devices are able to perform actual function calls. However, inlining is still the default behaviour.

[10] An alternative way to launch a kernel from an external module consist in using the CUDA device API, which shares many similarities with the OpenCL API. This approach gives more control over the low-level details but leads to significantly more complex codes.

[11] The CUDA 5.0 software development kit, still unreleased at the time of this writing, should provide such a feature.

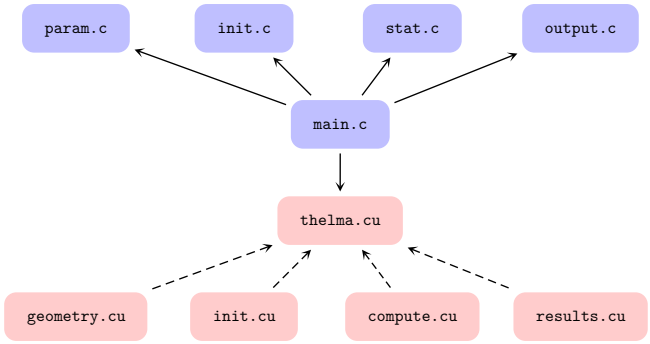


Figure 8: Structure of the TheLMA framework. — *The plain arrows symbolise the call dependencies, whereas the dashed arrows represent the inclusion operations performed by the preprocessor in order to provide a single file to the CUDA compiler.*

export data in various formats, or produce graphical outputs. The `thelma.cu` file contains some general macro definitions and device functions as well as the directives to include the other CUDA files. Each of them contains a specific kernel with the appropriate launching function. The TheLMA framework has been thought to confine code modifications to definite places in the source tree when implementing new models or new simulation layouts, thus increasing the development efficiency.

b. Multi-GPU implementations

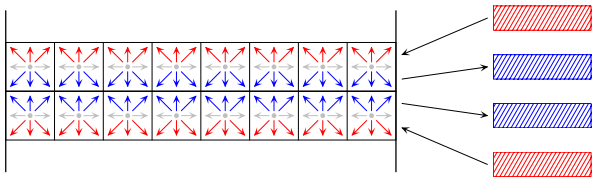
The memory shipped with CUDA computing devices reaches up to 6 GB on recent hardware such as the Tesla C2075. However, this on-board memory is not extensible. For single precision D3Q19 LBM, such an amount allows the computation domain to contain as much as 3.7×10^7 nodes, which is fairly large but might not be sufficient to carry out large scale simulations. To be of practical interest in many applications, GPU LBM solvers should therefore be able to run on multiple GPUs in parallel, which implies to address both communication and partitioning issues.

GPUs communicate with their host system through PCI Express (PCIe) links. Fermi based devices, for instance, may use up to 16 PCIe 2.0 links in parallel, which yields a maximum sustained throughput exceeding 3 GB/s in each direction. Although considerable, this performance is limited by non-negligible latencies. In the case of LBM, the simple approach, which consists in performing inter-GPU communication through host driven data transfer once kernel execution has completed, fails to give satisfactory performance [38]. A possible way to overlap communication and computations would be to process the

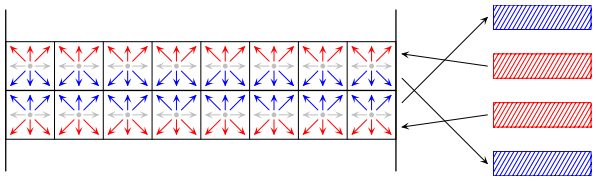
interface nodes using CUDA streams, but this would lead to a rather complex memory layout and—to our knowledge—no such attempt was ever reported in literature.

A more convenient method to overlap communication and computations arose with the introduction of the zero-copy feature, which enables the GPU to access directly to host memory, but requires the use of page-locked buffers to avoid interferences with the host operating system. As for global memory transactions, zero-copy transactions are issued per warp (or half-warp, depending on the compute capability), and should therefore be coalescent. For a multi-GPU implementation of the LBM using the same execution configuration and data layout as our generic single-GPU version, the former constraint implies that efficient data exchange is possible only for nodes located at the faces of the sub-domain parallel to the blocks' direction. Implementations based on this approach are thus limited to one- or two-dimensional domain partitions.

Recent motherboards may hold up to eight GPU computing devices. For such a small number of sub-domains, the advantages of a 2D partition over a 1D partition are dubious: the volume of transferred data is yet reduced by 43% in the best case (i.e. the cubic cavity), but the number of sub-domain interfaces raises from 7 to 10. For single node implementations, we therefore chose to restrict ourselves to one-dimensional partitioning which greatly simplifies the code. The isothermal version of our single-node multi-GPU solver is presented in Art. D and E. The thermal version, for which we had to modify our execution pattern in order to avoid shared memory shortage, is described in Art. G.



(a) Even time steps



(b) Odd time steps

Figure 9: Inter-GPU communication scheme. — *The out-going particle populations are drawn in blue whereas the in-coming populations are drawn in red. The arrows represent the data transfer occurring between the global memory of the two involved GPUs and the host memory.*

Our single-node solvers use POSIX threads to manage each GPU separately. The implemented inter-GPU communication scheme is outlined in Fig. 9. Each interface is associated to four page-locked buffers needed to load in-coming and store out-going particle populations. At each time step, the pointers to the buffers are swapped. For a cubic cavity, our single-node solvers currently reach up to 2,000 MLUPS in single precision, using eight Tesla C1060. As mentioned in Art. E, such performance is comparable to the one achieved on a Blue Gene/P computer with 4,096 cores by an optimised double precision code. Our studies in Art. D. and G show that the chosen approach yields good overlapping of communication and computations, with usually more than 80% parallelisation efficiency. In Art. E, we furthermore demonstrate that in most cases, even for fairly small computation domains, inter-GPU communication is not a limiting factor.

c. GPU cluster implementation

The single node implementation strategy outlined in the former section does not directly apply to GPU cluster systems. The inability to communicate efficiently using 3D partitions is the major issue. Beside providing increased flexibility, 3D partitioning considerably decreases the volume of communications. In the case of a cubic computation domain containing n^3 nodes and split into N^3 balanced sub-domains, the number of interface nodes is $2(N^3-1)n^2$ for the 1D partition and, with sufficiently large n, is approximately $6(N-1)n^2$ for the 3D partition. With $N = 4$ for instance, the volume of communication is thus divided by a factor of nearly 7. Moreover, with the possibility of a large number of sub-domains arises the need for a convenient way to specify the execution configuration. The MPI CUDA implementation proposed in Art. I attempts to address both issues.

With our usual data layout and execution pattern, the particle populations at the interfaces perpendicular to the blocks' direction would lie in non-contiguous memory locations. We therefore chose to store these data in auxiliary arrays. In order to enable coalesced accesses to these arrays, we use one-dimensional blocks (containing a single warp) that are not mapped any more to a single row of nodes, but to a square tile. The execution pattern at block level is summarised in Fig. 10. Before processing row by row the nodes of the tile, the in-coming populations from the auxiliary arrays are copied to shared memory. The resulting out-going populations are temporarily stored in shared memory and written back once the processing of the node has completed. Such a method allows the zero-copy transactions to be coalesced for all six possible interfaces of the sub-domain.

The execution set-up is described by a configuration file in JSON[12] format. Beside providing general parameters, such as the physical parameters of the

[12] JavaScript Object Notation. This format has the advantage of being both human-readable and easy to parse.

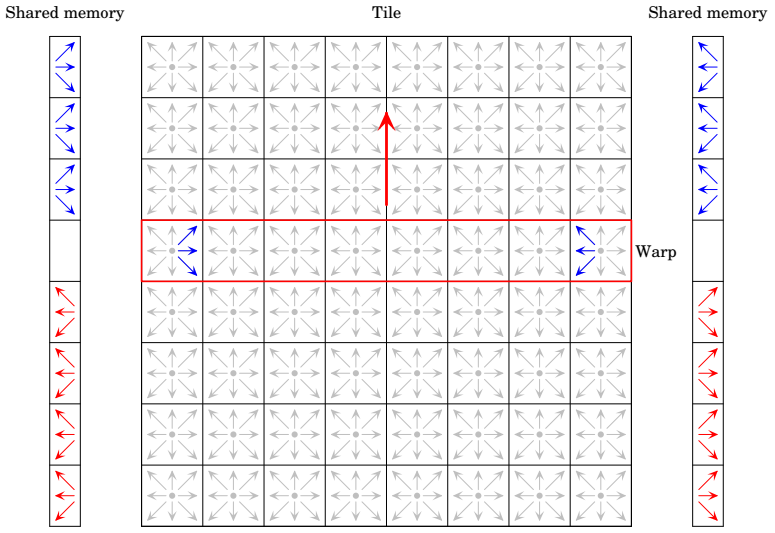


Figure 10: Kernel execution pattern. — *The current row of nodes is framed in red, the bold red arrow representing the direction of processing. The in-coming populations are drawn in red, whereas the out-going ones are drawn in blue. Once a row has been processed, the out-going post-collision populations are written over the outdated in-coming ones.*

simulation, this file is mainly used to specify for each sub-domain to which cluster node and GPU it is assigned and to which neighbouring sub-domains it is linked. A set of MPI routines is responsible for inter-GPU communication. During kernel execution, the out-going particle population located at the faces of the sub-domain are written to page-locked buffers using zero-copy transactions. The corresponding MPI process then copies the populations located at the edges to specific buffers and proceeds with message passing. Once completed, the receive buffers for faces and edges are gathered into page-locked read buffers, performing partial propagation at the same time.

We tested our solver on a nine-node cluster, each node hosting three Tesla M2070 (or M2090) computing devices. Using all 27 GPUs, we recorded up to 10,280 MLUPS in single precision with about 796 million nodes. Our performance results compare favourably with recently published works. On a 768^3 cavity for instance, we manage, using only 24 GPUs, to outperform the solver tested on the TSUBAME cluster with 96 GT200 [47]. In Art. I, our studies show that both weak and strong scalability are quite satisfactory. However, because of the limited number of sub-domains, further investigations on larger systems should be carried out. Although fully functional, our code still requires comprehensive validation studies, and may benefit from further enhancements. Nevertheless, this GPU cluster implementation of the LBM appears to be a promising tool to perform very large scale simulations.

VI Applications and perspectives

The multi-GPU thermal LBM solver described in Art. G, or possibly an extension to thermal simulations of our recent GPU cluster implementation, could be a major component for realistic indoor environment simulations as in [45]. However, the requirements in terms of spatial resolution are so high that, even with the computational power provided by GPUs, direct numerical simulations are beyond reach. The only practicable way for the time being seems to be the use of grid refinement together with a turbulence model suited for thermal LBM. Adding the grid refinement feature to our solvers is therefore of major importance, although it may have a significant impact on performance. Radiative effects should also be taken into account, especially when simulating low energy buildings, for which the direct solar contribution is usually considerable. Several approaches to simulate radiative heat transfers with GPUs are available and could be coupled to GPU LBM solvers.

Regarding external building aeraulics, the multi-GPU isothermal LES-LBM solver described in Art. F appears to be a convincing tool for simulating the flow around a building or even a group of buildings, although alternative boundary conditions and turbulence models should be tested. It may be useful to evaluate parameters such as pressure coefficients in various configurations or to carry out pedestrian wind environment studies. In this perspective, the interfacing between a geographic information system and the geometry module of the TheLMA framework would be of great interest. For external thermo-aeraulics, the situation is even more challenging than for indoor simulations, since at this level of Reynolds number, the thickness of the viscous sub-layer can go down to 100 μm [3]. The evaluation of parameters such as convective heat transfer coefficients may thus require very large computational efforts.

The TheLMA framework has been implemented in order to be as generic as possible despite of the limitations induced by the CUDA technology. Adapting it to solve similar schemes such as the recently introduced link-wise artificial compressibility method [1], or to solve PDEs such as the diffusion or the Laplace equation [52], should therefore be straightforward. Alternative many-core processors, e.g. the Intel MIC or the Kalray MPPA, share many similarities with GPUs, especially regarding data transfer mechanisms. The extension of our framework to such architectures, although it would probably necessitate to rewrite large portions of the code [13], should benefit from the general optimisation strategies we devised for the GPU implementation of LBM.

Bibliography

[1] P. Asinari, T. Ohwada, E. Chiavazzo, and A.F. Di Rienzo. Link-wise Artificial Compressibility Method. *Journal of Computational Physics*, 231(15):5109–5143, 2012.

[2] P. L. Bhatnagar, E. P. Gross, and M. Krook. A Model for Collision Processes in Gases. I. Small Amplitude Processes in Charged and Neutral One-Component Systems. *Physical Review*, 94(3):511–525, 1954.

[3] B. Blocken, T. Defraeye, D. Derome, and J. Carmeliet. High-resolution CFD simulations for forced convective heat transfer coefficients at the facade of a low-rise building. *Building and environment*, 44(12):2396–2412, 2009.

[4] B. Blocken, T. Stathopoulos, J. Carmeliet, and J.L.M. Hensen. Application of computational fluid dynamics in building performance simulation for the outdoor environment: an overview. *Journal of Building Performance Simulation*, 4(2):157–184, 2011.

[5] M. Bouzidi, M. Firdaouss, and P. Lallemand. Momentum transfer of a Boltzmann-lattice fluid with boundaries. *Physics of Fluids*, 13(11):3452–3459, 2001.

[6] H. Boyer, J.-P. Chabriat, B. Grondin-Perez, C. Tourrand, and J. Brau. Thermal building simulation and computer generation of nodal models. *Building and environment*, 31(3):207–214, 1996.

[7] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: stream computing on graphics hardware. In *ACM Transactions on Graphics*, volume 23, pages 777–786. ACM, 2004.

[8] S. Collange, D. Defour, and A. Tisserand. Power Consumption of GPUs from a Software Perspective. In *Lecture Notes in Computer Science 5544, Proceedings of the 9th International Conference on Computational Science, Part I*, pages 914–923. Springer, 2009.

[9] B. Crouse, M. Krafczyk, S. Kühner, E. Rank, and C. Van Treeck. Indoor air flow analysis based on lattice Boltzmann methods. *Energy and buildings*, 34(9):941–949, 2002.

[10] D. d'Humières. Generalized lattice-Boltzmann equations. In *Proceedings of the 18th International Symposium on Rarefied Gas Dynamics*, pages 450–458. University of British Columbia, Vancouver, Canada, 1994.

[11] D. d'Humières, I. Ginzburg, M. Krafczyk, P. Lallemand, and L.S. Luo. Multiple-relaxation-time lattice Boltzmann models in three dimensions. *Philosophical Transactions of the Royal Society A*, 360:437–451, 2002.

[12] D. d'Humières, P. Lallemand, and U. Frisch. Lattice gas models for 3D hydrodynamics. *EPL (Europhysics Letters)*, 2(4):291–297, 1986.

[13] P. Du, R. Weber, P. Luszczek, S. Tomov, G. Peterson, and J. Dongarra. From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming. *Parallel Computing*, 38(8):391–407, 2012.

[14] Z. Fan, F. Qiu, A. Kaufman, and S. Yoakum-Stover. GPU cluster for high performance computing. In *Proceedings of the 2004 ACM/IEEE Conference on Supercomputing*, pages 47–58. IEEE, 2004.

[15] U. Frisch, B. Hasslacher, and Y. Pomeau. Lattice-gas automata for the Navier-Stokes equation. *Physical review letters*, 56(14):1505–1508, 1986.

[16] I. Ginzbourg and D. d'Humières. Local second-order boundary methods for lattice Boltzmann models. *Journal of statistical physics*, 84(5):927–971, 1996.

[17] Khronos OpenCL Working Group. *The OpenCL specification*. 2008.

[18] J. Hardy, Y. Pomeau, and O. De Pazzis. Time evolution of a two-dimensional classical lattice system. *Physical Review Letters*, 31(5):276–279, 1973.

[19] X. He, S. Chen, and G. D. Doolen. A Novel Thermal Model for the Lattice Boltzmann Method in Incompressible Limit. *Journal of Computational Physics*, 146(1):282–300, 1998.

[20] X. He and L.S. Luo. Theory of the lattice Boltzmann method: From the Boltzmann equation to the lattice Boltzmann equation. *Physical Review E*, 56(6):6811–6817, 1997.

[21] K.E. Hoff III, J. Keyser, M. Lin, D. Manocha, and T. Culver. Fast computation of generalized Voronoi diagrams using graphics hardware. In *Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, pages 277–286. ACM, 1999.

[22] C. Inard, H. Bouia, and P. Dalicieux. Prediction of air temperature distribution in buildings with a zonal model. *Energy and Buildings*, 24(2):125–132, 1996.

[23] I.V. Karlin, A. Ferrante, and H.C. Öttinger. Perfect entropy functions of the Lattice Boltzmann method. *EPL (Europhysics Letters)*, 47(2):182–188, 1999.

[24] D. Kirk. Nvidia cuda software and gpu parallel computing architecture. In *Proceedings of the 6th International Symposium on Memory Management*, pages 103–104. ACM, 2007.

[25] M. Krafczyk, J. Tölke, and L.S. Luo. Large-eddy simulations with a multiple-relaxation-time LBE model. *International Journal of Modern Physics B*, 17(1):33–40, 2003.

[26] P. Lallemand and L. S. Luo. Theory of the lattice Boltzmann method: Acoustic and thermal properties in two and three dimensions. *Physical review E*, 68(3):36706(1–25), 2003.

[27] J. Latt and B. Chopard. Lattice Boltzmann method with regularized pre-collision distribution functions. *Mathematics and Computers in Simulation*, 72(2):165–168, 2006.

[28] W. Li, X. Wei, and A. Kaufman. Implementing lattice Boltzmann computation on graphics hardware. *The Visual Computer*, 19(7):444–456, 2003.

[29] S. Marié. *Étude de la méthode Boltzmann sur réseau pour les simulations en aéroacoustique*. PhD thesis, Université Pierre-et-Marie-Curie, Paris, 2008.

[30] W.R. Mark, R.S. Glanville, K. Akeley, and M.J. Kilgard. Cg: A system for programming graphics hardware in a C-like language. In *ACM Transactions on Graphics*, volume 22, pages 896–907. ACM, 2003.

[31] G. R. McNamara and G. Zanetti. Use of the Boltzmann Equation to Simulate Lattice-Gas Automata. *Physical Review Letters*, 61(20):2332–2335, 1988.

[32] F. Nicoud and F. Ducros. Subgrid-scale stress modelling based on the square of the velocity gradient tensor. *Flow, Turbulence and Combustion*, 62(3):183–200, 1999.

[33] NVIDIA. *Compute Unified Device Architecture Programming Guide version 4.0*, 2011.

[34] D. Ormières and M. Provansal. Transition to turbulence in the wake of a sphere. *Physical review letters*, 83(1):80–83, 1999.

[35] M. Pharr, editor. *GPU Gems 2 : programming techniques for high-performance graphics and general-purpose computation*. Addison-Wesley, 2005.

[36] Y. H. Qian. Simulating thermohydrodynamics with lattice BGK models. *Journal of scientific computing*, 8(3):231–242, 1993.

[37] Y. H. Qian, D. d'Humières, and P. Lallemand. Lattice BGK models for Navier-Stokes equation. *EPL (Europhysics Letters)*, 17(6):479–484, 1992.

[38] E. Riegel, T. Indinger, and N.A. Adams. Implementation of a Lattice–Boltzmann method for numerical fluid mechanics using the nVIDIA CUDA technology. *Computer Science – Research and Development*, 23(3):241–247, 2009.

[39] H. Sakamoto and H. Haniu. The formation mechanism and shedding frequency of vortices from a sphere in uniform shear flow. *Journal of Fluid Mechanics*, 287(7):151–172, 1995.

[40] J. Smagorinsky. General circulation experiments with the primitive equations. *Monthly Weather Review*, 91(3):99–164, 1963.

[41] C.J. Thompson, S. Hahn, and M. Oskin. Using modern graphics architectures for general-purpose computing: a framework and analysis. In *Proceedings of the 35th annual ACM/IEEE International Symposium on Microarchitecture*, pages 306–317. IEEE, 2002.

[42] E. Tric, G. Labrosse, and M. Betrouni. A first incursion into the 3D structure of natural convection of air in a differentially heated cubic cavity, from accurate numerical solutions. *International Journal of Heat and Mass Transfer*, 43(21):4043 – 4056, 2000.

[43] J. Tölke. Implementation of a Lattice Boltzmann kernel using the Compute Unified Device Architecture developed by nVIDIA. *Computing and Visualization in Science*, 13(1):29–39, 2010.

[44] J. Tölke and M. Krafczyk. TeraFLOP computing on a desktop PC with GPUs for 3D CFD. *International Journal of Computational Fluid Dynamics*, 22(7):443–456, 2008.

[45] C. van Treeck, E. Rank, M. Krafczyk, J. Tölke, and B. Nachtwey. Extension of a hybrid thermal LBE scheme for Large-Eddy simulations of turbulent convective flows. *Computers & Fluids*, 35(8):863–871, 2006.

[46] S. Wakashima and T.S. Saitoh. Benchmark solutions for natural convection in a cubic cavity using the high-order time-space method. *International Journal of Heat and Mass Transfer*, 47(4):853–864, 2004.

[47] X. Wang and T. Aoki. Multi-GPU performance of incompressible flow computation by lattice Boltzmann method on GPU cluster. *Parallel Computing*, 37(9):521–535, 2011.

[48] M. Weickert, G. Teike, O. Schmidt, and M. Sommerfeld. Investigation of the LES WALE turbulence model within the lattice Boltzmann framework. *Computers & Mathematics with Applications*, 59(7):2200–2214, 2010.

[49] G. Wellein, T. Zeiser, G. Hager, and S. Donath. On the single processor performance of simple lattice Boltzmann kernels. *Computers & Fluids*, 35(8):910–919, 2006.

[50] J. Wilke, T. Pohl, M. Kowarschik, and U. Rüde. Cache performance optimizations for parallel lattice Boltzmann codes. *Euro-Par 2003, LNCS 2790*, pages 441–450, 2003.

[51] D.A. Wolf-Gladrow. *Lattice-Gas Cellular Automata and Lattice Boltzmann Models: An Introduction*, volume 1725 of *Lecture Notes in Mathematics*. Springer, 2000.

[52] Y. Zhao. Lattice Boltzmann based PDE solver on the GPU. *The Visual Computer*, 24(5):323–333, 2008.

Résumé détaillé

Est-il possible de simuler avec précision le comportement énergétique d'un bâtiment ? La puissance de calcul considérable offerte par les ordinateurs actuels laisse à penser que la réponse à cette question est positive. Cela dit, les bâtiments forment des systèmes complexes interagissants de nombreuses manières avec leur environnement, à différentes échelles spatiales et temporelles. Ainsi, effectuer des simulations précises conduit souvent à des temps de calcul prohibitifs.

La conception des bâtiments à haute efficacité énergétique accroit encore les besoins en termes de modélisation des transferts de masse et de chaleur entre l'enveloppe d'un bâtiment et son environnement. La pratique usuelle en aéraulique des bâtiments est de recourir à des modèles simplifiés empiriques ou semi-empiriques. Ces approches, néanmoins, ne sauraient fournir davantage que des indications quant aux grandeurs étudiées.

En ce qui concerne l'aéraulique interne, les modèles nodaux et zonaux sont également répandus. Dans la première approche, l'air contenu dans une pièce est représenté par un nœud unique, dans la seconde, le volume correspondant à une pièce est divisé en cellules de dimensions macroscopiques. Ces méthodes, dont le coût calculatoire est faible mais qui simulent de larges volumes d'air à l'aide d'un seul nœud, conduisent souvent à des résultats éloignés des données expérimentales.

Afin d'atteindre la précision requise, le recours à la mécanique des fluides numérique (CFD[1]) en aéraulique des bâtiments semble donc incontournable. Néanmoins, la simulation numérique des écoulements a souvent un tel coût calculatoire qu'elle ne saurait être menée à bien sur une station de travail individuelle. L'accès aux centres de calculs étant limité et coûteux, la mise au point d'approches alternatives conduisant à des programmes de simulation des écoulements plus performants est donc un objectif majeur, susceptible d'avoir des retombées importantes dans de nombreux domaines de l'ingénierie.

Le travail de recherche présenté dans ces pages s'attache à explorer le potentiel lié à l'utilisation de processeurs graphiques (GPU[2]) pour mener à bien des simulations en mécanique des fluides numérique basées sur la méthode de Boltzmann sur gaz réseau (LBM[3]). Cette méthode, qui constitue une approche assez récente dans le domaine de la CFD, est bien adaptée au calcul intensif (HPC[4]) comme il sera montré plus avant. Ce travail a conduit à la conception et la création d'une plateforme logicielle nommée TheLMA[5]. Plusieurs solveurs

[1] Computational fluid dynamics.
[2] Graphics processing unit.
[3] Lattice Boltzmann method.
[4] High-performance computing.
[5] Thermal LBM for Many-core Architectures.

LBM ont été développés à partir de cette plateforme, allant de l'implantation mono-GPU à celle pour grappe de GPU, et abordant divers aspects tels que la simulation de fluides anisothermes, la simulation d'écoulements turbulents ou encore la simulation d'obstacles présentant des géométries complexes.

Le présent résumé est organisé de la façon suivante. La première section donne une vue d'ensemble des technologies de calcul généraliste sur processeurs graphiques. Dans la deuxième section, une description des principes et des aspects algorithmiques de la méthode de Boltzmann sur gaz réseau est donnée. La section III se concentre sur les implantations mono-GPU de la LBM. La section IV décrit diverses extensions à la LBM essentielles pour envisager une utilisation en aéraulique des bâtiments. La section V est consacrée aux implantations multi-GPU de la LBM, simple nœud et multi-nœuds. La section VI apporte une conclusion en donnant un aperçu des applications potentielles de la plateforme TheLMA à la simulation des bâtiments ainsi que des questions restant en suspens.

I Calcul généraliste sur processeurs graphiques

Les circuits graphiques sont destinés à soulager le travail du processeur central en ce qui concerne le rendu graphique. Ces circuits spécialisés sont devenus courants dans les ordinateurs grand public durant les années 1990. Dans un premier temps, les accélérateurs graphiques étaient pour l'essentiel des outils de rastérisation conçus pour déterminer les caractéristiques des pixels à afficher à partir d'un schéma électronique fixé. Ces circuits étaient donc d'un intérêt limité en termes de calcul bien que des essais aient été menés dès 1999. C'est au cours de cette même année que le fabricant Nvidia créa la dénomination GPU à l'occasion de la sortie de la GeForce 256.

En 2001, Nvidia a introduit une architecture innovante pour la GeForce 3, basée sur des unités de rendu programmables. Avec cette technologie, le pipeline graphique incorpore des *shaders*, c'est-à-dire des programmes affectés à la détermination des facettes et des pixels suivant les caractéristiques des scènes à afficher. À partir de là, il devint possible de considérer les GPU comme des processeurs parallèles de type SIMD[6]. Au départ, les *shaders* devaient être écrits dans un assembleur spécifique à l'architecture cible, jusqu'à l'introduction de langages de haut niveau : HLSL, également connu sous le nom de Cg, lié à l'API[7] graphique Direct3D de Microsoft et GLSL lié à l'API graphique OpenGL. Ces deux langages utilisent une syntaxe dérivée du C et un paradigme de programmation centré sur la notion de flux de données : une séquence d'opérations (le *noyau*) est effectuée pour chaque élément d'un ensemble de données (le *flux*). Ces langages de haut niveau ont largement contribué à élargir le réper-

[6] Single instruction multiple data.
[7] Application programming interface.

toire du calcul généraliste sur processeurs graphiques (GPGPU[8]). Néanmoins, le développement d'applications généralistes en Cg ou en GLSL reste délicat de par l'orientation principalement graphique de ces langages.

Durant la dernière décennie, la puissance de calcul par cœur des processeurs généralistes n'a progressé que très modestement, alors que l'amélioration de la finesse de gravure des circuits intégrés a permis aux fondeurs de multiplier le nombre d'unités de rendu graphique des GPU, augmentant d'autant la puissance de calcul théorique de ces processeurs. Cette situation a conduit le fabricant Nvidia à considérer le marché du calcul intensif comme une cible potentielle et à développer une nouvelle technologie baptisée CUDA[9]. L'arrivée de CUDA en 2007 constitue probablement le progrès le plus significatif dans le domaine du GPGPU à ce jour. Les concepts clefs consistent en un ensemble de spécifications matérielles génériques liées à un modèle de programmation parallèle. CUDA désigne souvent le langage associé à cette technologie qui dérive du C/C++ (avec quelques restrictions) mettant en œuvre ce modèle de programmation. Comparé aux technologies antérieures, CUDA apporte une flexibilité sans précédent pour le développement de logiciels destinés aux processeurs graphiques, augmentant de fait leur potentiel dans le domaine du calcul intensif.

Le statut propriétaire de la technologie CUDA est un inconvénient majeur : seuls les GPU produits par Nvidia sont à même d'exécuter des programmes CUDA. Le standard OpenCL, diffusé à partir de 2008 suit une approche plus portable dans la mesure où il se destine à tout type de plateforme. Il est à noter que le modèle et le langage de programmation OpenCL présentent de nombreuses similitudes avec leurs équivalents CUDA. Du point vue du calcul intensif, OpenCL est un standard prometteur susceptible de remplacer CUDA à l'avenir. Bien qu'énoncées dans le cadre de la technologie CUDA, les contributions du travail de recherche résumé dans ces pages devraient conserver leur pertinence avec OpenCL.

Le modèle architectural CUDA repose sur des spécifications matérielles abstraites. Un GPU CUDA est décrit comme un ensemble de multiprocesseurs (SM[10]), chacun d'entre eux contenant : un ordonnanceur, un groupe de processeur scalaires (SP[11]) disposant de leurs propres registres, ainsi que d'une mémoire partagée. La quantité de registres et de mémoire partagée par SM est assez limitée, avec par exemple au plus 64 Ko de mémoire partagée par SM sur le GF100. Les SM embarquent également de la mémoire cache destinée aux constantes, textures et, selon les générations de GPU, aux données.

Le modèle d'exécution CUDA, nommé SIMT[12] par Nvidia, est relativement complexe de par le double niveau d'organisation matérielle. Le paradigme de

[8] General purpose computing on graphics processing units.
[9] Compute Unified Device Architecture.
[10] Streaming multiprocessor.
[11] Scalar processor.
[12] Single instruction multiple threads.

programmation CUDA est centré sur la notion de tâche. Un processus élémentaire est désigné par le terme *thread*, que nous traduirons en *fil d'exécution* ou plus simplement *fil* dans la suite de ce texte, la séquence d'instruction associée étant nommée *noyau*. Pour exécuter un noyau, il est nécessaire de spécifier une *grille* d'exécution. Une grille consiste en un tableau multidimensionnel de *blocs*, qui eux-mêmes sont des tableaux multidimensionnels de fils d'exécution. Ce schéma à deux niveaux est induit par les caractéristiques architecturales : un bloc ne peut être traité qu'au sein d'un seul SM ; le nombre de fils figurant dans un bloc est par conséquent relativement limité.

Les SM pris séparément sont des processeurs de type SIMD, ce qui implique que les fils d'exécution ne sont pas traités de manière autonome, mais en groupes nommés *warp*, que nous traduirons par *trame*[13]. Pour l'instant, une trame contient 32 fils quelle que soit l'architecture considérée, mais cette valeur n'est pas fixée et est susceptible de changer pour les générations à venir. Une trame constitue un ensemble insécable, il est donc préférable de faire en sorte que le nombre de fils dans un bloc soit un multiple de la taille d'une trame. Cette organisation entraîne de sévères limitations, comme par exemple pour les branchements conditionnels : lorsqu'un chemin d'exécution diverge au sein d'une trame, le traitement des branches est séquentiel. Dans de nombreux cas, déterminer une grille d'exécution adéquate n'est pas trivial, mais peut se révéler d'une importance cardinale quant aux performances de l'application considérée.

Les variables automatiques d'un noyau, qui sont spécifiques à chaque fil, sont stockées autant que possible dans les registres. Les tableaux en général et les structures de taille trop importante, en revanche, résident en mémoire *locale*. Cet espace mémoire, qui sert également aux déchargements de registres, est hébergé par les circuits de mémoire de la carte de calcul. Les fils d'exécution ont également accès à la mémoire partagée et à la mémoire *globale*. La portée et la durée de vie de la mémoire partagée est limitée au bloc courant. La mémoire globale, quant à elle, est visible par l'ensemble des fils et persiste durant toute la durée de l'application. Comme la mémoire locale, elle est hébergée dans la mémoire externe au GPU qui est la seule accessible au système hôte. Il est à noter que pour les GPU des générations précédant le Fermi, ni la mémoire locale ni la mémoire globale ne sont pas associées à un cache.

Les accès à la mémoire globale se font par segments de taille allant de 32 à 128 octets, selon l'architecture. Pour être efficaces, les opérations de lecture et d'écriture en mémoire globale au sein d'une trame donnée doivent porter sur des adresses consécutives. L'organisation des données en mémoire est donc un aspect important pour l'optimisation des applications CUDA.

[13] Le terme *warp* appartient au vocabulaire du tissage. Sa traduction exacte en français est « chaîne ».

II Principes de la méthode de Boltzmann sur gaz réseau

Les démarches suivies en CFD peuvent dans la plupart des cas être qualifiées de « descendantes » : un système d'équations aux dérivées partielles non linéaires est discrétisé et résolu à l'aide d'une méthode d'analyse numérique telle les différences finies, volumes finis, éléments finis ou encore les approches spectrales. L'attention se porte en général sur les erreurs de troncature liées à la discrétisation. Néanmoins, d'un point de vue physique, la préservation des propriétés de conservation est essentielle, tout particulièrement pour les simulations de longue durée où de légères variations peuvent, par accumulation, conduire à des résultats totalement incorrects.

D'autres approches que l'on pourrait qualifier de « montantes » constituent des alternatives aux méthodes précédentes. Dans cette catégorie, il convient de citer la dynamique moléculaire, les automates sur gaz réseau et la méthode de Boltzmann sur gaz réseau. Alors que la première tente de simuler de la manière la plus précise possible le comportement individuel de chaque molécule, ce qui induit un coût calculatoire extrêmement élevé, les deux autres se placent à une échelle plus importante et sont souvent, de fait, qualifiées de *mésoscopiques*.

L'équation de Boltzmann décrit le comportement hydrodynamique d'un fluide par une fonction de distribution d'une particule dans l'espace des phases, regroupant la position $\boldsymbol{x}$ et la vitesse particulaire $\boldsymbol{\xi}$, et le temps :

$$\partial_t f + \boldsymbol{\xi} \cdot \nabla_{\boldsymbol{x}} f + \frac{\boldsymbol{F}}{m} \cdot \nabla_{\boldsymbol{\xi}} f = \Omega(f). \tag{1}$$

Dans l'équation précédente, m désigne la masse de la particule, $\boldsymbol{F}$ est une force extérieure et Ω correspond à l'opérateur de collision. En trois dimensions, les grandeurs macroscopiques associées au fluide sont données par :

$$\rho = \int f \, \mathrm{d}\boldsymbol{\xi} \tag{2}$$

$$\rho \boldsymbol{u} = \int f \, \boldsymbol{\xi} \mathrm{d}\boldsymbol{\xi} \tag{3}$$

$$\rho \boldsymbol{u}^2 + 3\rho\theta = \int f \, \boldsymbol{\xi}^2 \mathrm{d}\boldsymbol{\xi} \tag{4}$$

où ρ désigne la densité du fluide, $\boldsymbol{u}$ sa vitesse et θ est défini par $\theta = k_\mathrm{B} T/m$ avec T la température absolue et k_B la constante de Boltzmann.

La méthode de Boltzmann sur gaz réseau est basée sur une forme discrétisée de l'équation 1 utilisant un pas de temps constant δt et un maillage orthogonal régulier de pas δx. Un ensemble fini de vitesses $\{\boldsymbol{\xi}_\alpha \,|\, \alpha = 0, \dots N\}$ avec $\boldsymbol{\xi}_0 = \boldsymbol{0}$, tient lieu d'espace des vitesses particulaires. Cet ensemble, ou *stencil*, est choisi de telle sorte que pour un nœud $\boldsymbol{x}$ donné, $\boldsymbol{x} + \delta t \boldsymbol{\xi}_\alpha$ est également un nœud quel que soit α. En trois dimensions, le stencil le plus couramment employé

est nommé D3Q19. Il lie un nœud donné à 18 de ses plus proches voisins et compte donc 19 éléments en tout.

L'équivalent discret de la fonction de distribution f est un ensemble de densités particulaires $\{f_\alpha \,|\, \alpha = 0, \dots N\}$ associées aux vitesses. Notons $\intercal$ l'opérateur de transposition et $|a_\alpha\rangle = (a_0, \dots a_N)^\intercal$. En l'absence de force extérieure, l'équation 1 devient :

$$|f_\alpha(\boldsymbol{x} + \delta t \boldsymbol{\xi}_\alpha, t + \delta t)\rangle - |f_\alpha(\boldsymbol{x}, t)\rangle = \boldsymbol{\Omega}\left(|f_\alpha(\boldsymbol{x}, t)\rangle\right). \tag{5}$$

En ce qui concerne les grandeurs macroscopiques du fluide, les équations 2 et 3 deviennent :

$$\rho = \sum_\alpha f_\alpha, \tag{6}$$

$$\rho \boldsymbol{u} = \sum_\alpha f_\alpha \boldsymbol{\xi}_\alpha. \tag{7}$$

Il convient de mentionner le fait que les modèles LBM les plus couramment utilisés ne satisfont pas à la conservation de l'énergie. Ils ne sont donc appropriés que pour la simulation de fluides isothermes. Un composant essentiel de ces modèles est l'opérateur de collision utilisé. De nombreuses versions ont été proposées, parmi lesquelles le LBGK basé sur l'approximation de Bhatnagar–Gross–Krook, le MRT[14] recourant à des temps de relaxation multiples, les opérateurs entropiques ou encore les opérateurs régularisés. L'ensemble des solveurs LBM développés dans le cadre de ce travail de recherche, à l'exception d'un seul, intègrent l'opérateur MRT, dont une description relativement complète est donnée dans l'article D, entre autres. Par rapport au LBGK, le plus souvent employé, le MRT apporte une précision et une stabilité plus importantes au prix d'une complexité arithmétique légèrement supérieure.

D'un point de vue algorithmique, la LBM revient à une succession d'opérations de *collision* et de *propagation*. La phase de collision est décrite par :

$$|\widetilde{f_\alpha}(\boldsymbol{x}, t)\rangle = |f_\alpha(\boldsymbol{x}, t)\rangle + \boldsymbol{\Omega}\left(|f_\alpha(\boldsymbol{x}, t)\rangle\right), \tag{8}$$

où $\widetilde{f_\alpha}$ désigne les densités post-collision. Il est à noter que cette étape est purement locale. La phase de propagation obéit à :

$$|f_\alpha(\boldsymbol{x} + \delta t\, \boldsymbol{\xi}_\alpha, t + \delta t)\rangle = |\widetilde{f_\alpha}(\boldsymbol{x}, t)\rangle, \tag{9}$$

ce qui correspond à de simples transferts de données. La partition de l'équation 5 en deux relations met en lumière le parallélisme de données présent dans la LBM. Cette approche est donc bien adaptée au calcul intensif sur architectures massivement parallèles, le point essentiel étant de garantir la synchronisation des opérations entre voisins immédiats. Le moyen le plus simple

[14] Multiple relaxation time.

de gérer cette contrainte consiste à utiliser deux instances des tableaux de données, une pour les pas de temps pairs et l'autre pour les pas de temps impairs. Une méthode à présent largement utilisée et nommée *compression de grille*, permet de diminuer l'occupation mémoire presque de moitié, mais nécessite d'avoir le contrôle sur l'ordre de traitement des nœuds. Elle ne s'applique donc pas dans un contexte d'exécution asynchrone.

Dans le cadre de systèmes à mémoire partagée, l'organisation des données pour des simulations LBM en trois dimensions se réduit généralement à un tableau à cinq dimensions : trois pour l'espace, une pour les indices de vitesses particulaires et une pour le temps. Selon l'architecture cible et la hiérarchie mémoire, l'ordre des dimensions dans le tableau peut avoir un impact majeur sur les performances. La conception des structures de données est de fait une étape essentielle pour l'optimisation des implantations HPC de la LBM.

III Implantations GPU de la LBM

Les premières tentatives d'implanter la LBM sur GPU remontent aux débuts du calcul généraliste sur GPU en 2003. Aucune plateforme de programmation généraliste n'étant encore disponible, les densités particulaires devaient être stockées dans des piles de tableaux bidimensionnels de textures. Avec cette approche, les calculs associés à l'équation 5 doivent être traduits en opérations destinées aux unités de rendu graphique, ce qui, à l'évidence, est très contraignant et dépendant du matériel. La première implantation CUDA de la LBM en trois dimensions a été décrite par Tölke et Krafczyk en 2008. Les principes d'implantation proposés demeurent pour l'essentiel valables aujourd'hui. On retiendra essentiellement :

1. La fusion des étapes de collision et de propagation en un seul noyau pour éviter les transferts de données inutiles.
2. L'identification de la grille d'exécution CUDA au maillage, c'est-à-dire l'affectation d'un fil d'exécution à chaque nœud. Cette approche, créant un grand nombre de fils, permet de profiter du parallélisme massif des GPU et de masquer éventuellement la latence de la mémoire globale en l'absence de cache.
3. Le recours à une organisation des données permettant aux accès en mémoire globale effectués par les trames d'être coalescents.
4. Le lancement du noyau de collision et propagation à chaque pas de temps et l'utilisation de deux instances des tableaux de données afin de garantir la synchronisation locale.

Pour leur implantation de la LBM, Tölke et Krafczyk utilisent une grille bidimensionnelle de blocs unidimensionnels et des tableaux de distribution spécifiques à chaque vitesse particulaire. Ce dispositif permet des accès coalescents

lors de la lecture des densités avant d'opérer la collision. Néanmoins, un soin particulier doit être apporté à la réalisation de la propagation, qui consiste en des déplacements de données dans toutes les directions spatiales, y compris celle qui correspond à la dimension mineure des tableaux de distribution. Ainsi, une écriture directe en mémoire globale induit nécessairement des défauts d'alignement, ce qui, avec le G80 utilisé alors, a des conséquences délétères sur les performances, dans la mesure où tous les accès mémoire sont traités individuellement. La solution proposée par Tölke et Krafczyk consiste à utiliser la mémoire partagée pour opérer une propagation partielle le long des blocs et d'achever la propagation lors de l'écriture en mémoire globale.

La méthode de la propagation en mémoire partagée, supprimant entièrement les défauts d'alignement, semble incontournable pour les implantations destinées au G80. Pour les générations suivantes en revanche, le coût des défauts d'alignement est nettement moins conséquent, bien que toujours non négligeable. En ce qui concerne le GT200, par exemple, les accès à la mémoire globale se font par segments alignés de 32, 64 ou 128 octets, une transaction non alignée étant réalisée en un minimum d'opérations. Il apparaît donc raisonnable de considérer des approches consistant à effectuer la propagation directement en mémoire globale. Les résultats de nos recherches à ce sujet sont consignés dans l'article A.

Quelques investigations élémentaires sur une GeForce GTX 295 nous ont permis de constater que les défauts d'alignement sont significativement plus coûteux à l'écriture qu'à la lecture. Nous avons donc testé deux schémas de propagation alternatifs. Le premier, baptisé schéma *scindé*, consiste à effectuer une propagation partielle dans les directions perpendiculaires aux blocs durant la phase d'écriture, puis à compléter cette propagation durant la phase de lecture du pas de temps suivant. Le second, nommé schéma *inversé*, revient à réaliser la propagation durant la phase de lecture, aucun décalage n'étant effectué lors de l'écriture.

Les implantations de ces deux schémas obtiennent des performances comparables, de l'ordre de 500 millions de nœuds traités par seconde (MLUPS[15]) en simple précision, soit plus de 80 % du débit maximal effectif entre GPU et mémoire globale. La méthode de propagation en mémoire partagée est susceptible de mener à des performances encore supérieures à celles obtenues avec des approches de propagation directe en mémoire globale. Néanmoins, ces dernières sont d'un grand intérêt en pratique car elles conduisent à des codes plus simples et laissent la mémoire partagée vacante, ce qui permet d'envisager une utilisation pour mener à bien des calculs additionnels comme nous le verrons plus avant.

Les communications entre GPU et mémoire globale tendent à être le facteur limitant les performances des solveurs LBM pour CUDA. Bien que quelques informations soient livrées dans le guide de programmation CUDA, l'essentiel des

[15] Million lattice node updates per second.

mécanismes de transfert de données demeure non documenté. Afin de gagner en compréhension dans ce domaine et affiner nos stratégies d'optimisation, nous avons réalisé un ensemble de bancs d'essais sur l'architecture GT200 dont les résultats sont rapportés dans l'article B. Ces études nous ont permis d'énoncer un modèle pour les communications entre GPU et mémoire globale prenant en compte divers paramètres comme le nombre de défauts d'alignement éventuels. Du précédent modèle, nous dérivons une estimation de l'optimum de performance pour les implantations CUDA de la LBM, susceptible de fournir des indications utiles dans le processus d'optimisation.

IV Extensions à la LBM

En aéraulique des bâtiments, bien que les simulations d'écoulements isothermes ne soient pas dénuées d'intérêt, il se révèle souvent nécessaire de prendre en compte les aspects thermiques. Les articles C et G retracent nos tentatives d'implantation de solveurs LBM thermiques sur GPU. Comme nous l'avons déjà mentionné, les modèles de Boltzmann sur gaz réseau usuels tels le D3Q19 MRT ne satisfont pas à la conservation de l'énergie. De nombreuses voies ont été explorées pour appliquer la LBM en thermo-hydrodynamique, parmi lesquelles il convient de citer les modèles multi-vitesses, utilisant un jeu de vitesses particulaires agrandi, les modèles à double population, utilisant une distribution d'énergie en complément de la distribution particulaire, ou encore les modèles hybrides, pour lesquels l'équation de la chaleur est traitée par une méthode d'analyse numérique classique.

Notre choix s'est porté sur l'approche hybride développée par Lallemand et Luo en 2003. Elle repose sur une version légèrement modifiée du D3Q19 MRT, l'équation de la chaleur étant résolue par une méthode aux différences finies. Comparée aux modèles multi-vitesses ou double population, l'approche hybride apporte une stabilité et une précision accrues, tout en présentant un surcoût réduit en termes de communication. En effet, une seule opération d'écriture et dix-neuf opérations de lecture supplémentaires par nœud sont nécessaires. Nos implantations, utilisant la mémoire partagée afin de réduire les accès redondants, parvient à diminuer de plus de moitié le volume additionnel en lecture. Le solveur mono-GPU décrit dans l'article C et le solveur multi-GPU décrit dans l'article G ont tous deux été testés sur la cavité cubique différentiellement chauffée. Les nombres de Nusselt calculés aux parois isothermes sont en bon accord avec les données issues de la littérature. En utilisant la version multi-GPU, nous avons pu effectuer des simulations pour des nombres de Rayleigh allant jusqu'à 10^9.

Les solveurs multi-GPU tels ceux décrits dans les articles D et E permettent d'effectuer des simulations sur des domaines de calcul de tailles considérables. Avec un serveur Tyan B7015 équipé de huit Tesla C1060, comportant chacune 4 Go de mémoire vive, il est possible de travailler sur un domaine de calcul

cubique contenant jusqu'à 576^3 nœuds en simple précision, soit plus de 190 millions de nœuds. De telles résolutions permettent d'effectuer des simulations à des nombres de Reynolds de l'ordre de 10^3 à 10^4, selon les cas. Néanmoins, les valeurs typiquement atteintes en aéraulique externe des bâtiments sont plutôt de l'ordre de 10^6. À cause des phénomènes de turbulence, réaliser des simulations numériques directes pour de telles applications semble donc hors de portée pour l'instant.

Ainsi, pour être d'un intérêt pratique en aéraulique externe des bâtiments, les solveurs LBM doivent intégrer un modèle de sous-maille tel que, par exemple, la simulation aux grandes échelles (LES[16]). Afin d'évaluer la faisabilité et la pertinence d'une telle démarche, nous avons choisi d'implanter la version la plus élémentaire de la LES, à savoir le modèle originel proposé par Smagorinsky en 1963. Les résultats des simulations effectuées à l'aide d'une version étendue de notre solveur multi-GPU isotherme sont présentés dans l'article F. Dans le modèle de Smagorinsky, une viscosité turbulente obtenue à partir du tenseur des contraintes est ajoutée à la viscosité moléculaire afin d'obtenir la viscosité cinématique. L'un des intérêts de l'opérateur MRT en l'occurrence est qu'il permet aisément de retrouver les composantes du tenseur des contraintes. Les calculs étant purement locaux, l'impact sur les performances est négligeable.

Dans l'article F, nous décrivons la simulation des écoulements au voisinage d'un groupe de neuf cubes montés sur une paroi pour un nombre de Reynolds $\mathrm{Re} = 10^6$. Nous avons déterminé les champs de pression et de vitesse moyens en intégrant sur un intervalle de temps suffisamment long pour être statistiquement pertinent. En l'absence de données expérimentales, ces résultats n'ont pu être validés. En revanche, le temps de calcul étant inférieur à 18 h, notre étude démontre la possibilité de simuler les écoulements au voisinage d'un groupe de bâtiments en un temps raisonnable.

Avec la LBM, l'interface entre fluide et solide conduit à un certain nombre de densités particulaires indéterminées. En considérant un nœud à l'interface, c'est-à-dire un nœud fluide ayant au moins un nœud solide pour voisin immédiat, il est aisé de voir que les densités qui devraient être propagées depuis les nœuds solides voisins sont indéfinis. Une manière communément employée de régler ce problème consiste à utiliser la condition aux limites de *simple rebond* (SBB[17]) qui consiste à remplacer les densités inconnues par les valeurs post-collision du pas de temps précédent pour les directions opposées. Soit $\boldsymbol{x}$ un nœud à l'interface, la SBB obéit à :

$$f_{\bar{\alpha}}(\boldsymbol{x}, t) = \widetilde{f}_{\alpha}(\boldsymbol{x}, t - \delta t) \tag{10}$$

où $f_{\bar{\alpha}}(\boldsymbol{x}, t)$ est une densité particulaire inconnue et $\bar{\alpha}$ est la direction opposée à α. Il est possible de montrer que la SBB est du second ordre en espace et que la limite entre fluide et solide est située approximativement à mi-chemin entre

[16] Large eddy simulation.
[17] Simple bounce-back.

le nœud à l'interface et le nœud solide. On constate à partir de l'équation 10 que l'application de cette condition aux limites est simple, puisqu'il suffit de connaître la liste des voisins solides du nœud considéré. La SBB est utilisée pour la plupart de nos solveurs, le voisinage d'un nœud étant généralement décrit à l'aide d'un champ de bits. Cette approche nous permet de représenter aisément des obstacles possédant une géométrie complexe, comme le groupe de neuf cubes étudié dans l'article F.

Comme la position de l'interface entre fluide et solide est fixée, l'utilisation de la SBB n'est convaincante que dans le cas où les solides considérés possèdent des faces planes parallèles aux directions spatiales. Une surface incurvée voire une paroi plane inclinée conduisent à des effets de marches d'escalier qui peuvent avoir un impact non négligeable sur la simulation. Pour tenter de palier ces effets, nous avons également considéré la condition aux limites de *rebond linéairement interpolé* (LIBB[18]) qui prend en compte la position exacte de l'interface entre fluide et solide. Notre implantation met à profit des transferts de données qui demeurent inexploités avec le schéma de propagation inversé. Elle se révèle donc relativement efficace puisqu'elle n'accroit que légèrement la quantité de données à lire. Pour le cas test de l'écoulement autour d'une sphère, la comparaison avec des données expérimentales montre que la LIBB apporte une précision et une stabilité supérieures à la SBB en ce qui concerne la fréquence de détachement des vortex. Cette conclusion ne paraît cependant pas pouvoir se généraliser ; des simulations récentes et non publiées pour l'instant, portant sur les écoulements au voisinage d'une plaque plane inclinée, ne montrent pas de différences significatives entre les deux conditions aux limites. Des études plus poussées portant sur des paramètres plus pertinents tels que le coefficient de traînée, ou mettant en œuvre d'autres formes de conditions aux limites, méritent d'être menées. Comme il est montré dans les articles F et H, les solveurs LBM sur GPU rendent possibles des campagnes de validation à grande échelle et peuvent donc contribuer à l'évaluation de stratégies de modélisation innovantes.

V Simulations LBM à grande échelle

CUDA est une technologie récente qui impose des contraintes fortes au programmeur, liées principalement à des aspects matériels. Bien que la situation ait évolué favorablement avec les capacités croissantes des générations successives de GPU, ainsi que les améliorations apportées à la chaîne de compilation, de nombreuses pratiques issues l'ingénierie logicielle, tels que le développement de bibliothèques, ne sont pas pertinentes. Néanmoins, nos travaux nous ont conduits à la réalisation de nombreuses versions de nos solveurs LBM, correspondant à des modèles physiques et à des configurations très variés. Il en découle la nécessité de disposer de stratégies appropriées favorisant la réuti-

[18] Linearly interpolated bounce-back.

lisation et la maintenance du code existant. Cette situation nous a amenés à la conception et la création de la plateforme logicielle TheLMA. Elle consiste en un ensemble cohérent de définitions de structures de données et de fichiers source C et CUDA. Les fichiers C fournissent une collection de fonctions utilitaires destinées au traitement des paramètres de configuration, à l'initialisation des structures de données globales, au traitement des résultats de simulation, ou encore à la production de sorties graphiques. La partie CUDA de la plateforme se divise en plusieurs modules, chacun étant centré sur un noyau aux attributions spécifiques. La plateforme TheLMA a été conçue pour confiner les modifications à des endroits bien définis du code source lors de l'implantation de nouveaux modèles ou la mise en place d'une nouvelle simulation, permettant au développement de gagner en efficacité.

La mémoire vive disponible sur des cartes de calcul récentes comme la Tesla C2075 atteint jusqu'à 6 Go, mais n'est pas extensible. Pour une simulation LBM utilisant le stencil D3Q19 en simple précision, une telle quantité permet de stocker des domaines de calcul contenant jusqu'à 3.7×10^7 nœuds. Ce nombre, certes conséquent, peut se révéler insuffisant pour mener à bien des simulations à grande échelle. Pour avoir une portée pratique, un solveur LBM pour processeurs graphiques doit être en mesure de recourir à plusieurs GPU en parallèle, ce qui implique de traiter à la fois les problèmes liés à la communication et au partitionnement. Pour assurer un bon recouvrement des calculs et des échanges de données entre sous-domaines, nous avons choisi d'utiliser des accès directs à la mémoire centrale initiés par les processeurs graphiques eux-mêmes durant l'exécution du noyau. Comme pour les opérations en mémoire globale, ces accès, pour être efficaces, doivent être coalescents, ce qui n'est pas le cas pour les faces du sous-domaine perpendiculaires aux blocs. Cette approche ne permet donc pas d'envisager l'utilisation de partitions tridimensionnelles du domaine de calcul.

Les cartes mères actuelles sont capables de gérer jusqu'à huit cartes de calcul simultanément. Avec un nombre de sous-domaines aussi faible, les avantages d'une partition bidimensionnelle sont sujets à caution. Notre implantation multi-GPU simple nœud de la LBM isotherme, décrite dans les articles D et E se restreint volontairement aux partitions unidimensionnelles, ce qui a pour effet de simplifier le code. La version thermique, présentée dans l'article G, a nécessité une refonte partielle du schéma d'exécution habituel pour éviter les problèmes liés à la pénurie de mémoire partagée. Les performances sont de l'ordre de 2 000 MLUPS en simple précision en utilisant huit Tesla C1060, ce qui est comparable aux performances réalisées sur un ordinateur Blue Gene/P équipé de 4096 cœurs par un code optimisé en double précision. Nos études montrent que l'efficacité de parallélisation dépasse généralement les 80 % et que dans la plupart des cas, même pour des domaines de taille modeste, la communication inter-GPU n'est pas un facteur limitant.

Dès que le nombre de sous-domaines employés devient conséquent, l'utilisation d'une partition tridimensionnelle permet de réduire de façon drastique le

volume des communications. Le procédé d'échange de données employé pour les versions simple nœud n'est donc pas pertinent pour les implantations de la LBM sur grappe de GPU. Dans l'article I, nous présentons une nouvelle approche permettant une communication efficace sur toutes les faces des sous-domaines. Au lieu d'affecter un fil d'exécution à chaque nœud, nous utilisons des blocs unidimensionnels contenant une seule trame et associés à des groupes de nœuds formant une tuile carrée. Le recours à des tableaux auxiliaires permet d'échanger efficacement des données quelle que soit la direction spatiale considérée. Notre implantation pour grappe de processeurs graphiques utilise un jeu de routines MPI[19] pour assurer la communication inter-GPU. L'affectation des sous-domaines aux cartes de calcul de la grappe se fait par l'intermédiaire d'un fichier de configuration au format JSON[20].

Nous avons testé notre solveur sur une grappe de calcul regroupant neuf nœuds, chacun contenant trois Tesla M2070 (ou M2090). En utilisant 27 GPU, nous avons pu atteindre 10 280 MLUPS en simple précision sur environ 796 millions de nœuds. Nos résultats en termes de performances dépassent largement ceux, publiés récemment, qui ont été obtenus avec le super-ordinateur TSUBAME. Nos études montrent que la scalabilité, tant au sens fort qu'au sens faible, est très satisfaisante, bien que des mesures additionnelles sur des grappes plus importantes restent à effectuer. Notre solveur LBM pour grappe de GPU paraît d'ores et déjà constituer un outil prometteur pour la réalisation de simulations à très grande échelle.

VI Applications et perspectives

Le solveur LBM thermique multi-GPU décrit dans l'article G, ou éventuellement une extension à la thermique de notre récente implantation sur grappe de GPU, pourrait être un composant majeur dans la mise en œuvre de simulations réalistes en aéraulique interne. Néanmoins, les contraintes en termes de résolution spatiale sont si élevées que, même avec la puissance de calcul apportée par les processeurs graphiques, la simulation numérique directe n'est pas envisageable. Le seul chemin actuellement praticable semble être l'utilisation de maillages raffinés couplés à un modèle de turbulence adapté à la LBM thermique. L'ajout de la prise en charge du raffinement de maillage dans nos solveurs est donc d'une importance cardinale, quand bien même l'impact sur les performances pourrait être significatif. Les effets radiatifs devraient eux aussi être pris en compte, tout particulièrement dans les simulations de bâtiments à haute efficacité énergétique pour lesquels l'apport solaire est habituellement considérable. Plusieurs approches permettant de simuler les transferts radiatifs sur GPU ont été développées et devraient pouvoir être couplées à des solveurs LBM sur GPU.

[19] Message Passing Interface.
[20] JavaScript Object Notation.

Concernant l'aéraulique externe des bâtiments, le solveur LES-LBM isotherme multi-GPU décrit dans l'article F se révèle être un outil convaincant pour la simulation des écoulements au voisinage d'un bâtiment voire d'un groupe de bâtiments, quand bien même des conditions aux limites et des modèles de turbulence alternatifs devraient être testés. Il pourrait contribuer à l'évaluation de paramètres tels que des coefficients de pression pour diverses configurations ou à la réalisation d'études d'environnement aéraulique urbain. Dans cette perspective, le couplage entre des systèmes d'information géographique et le module de géométrie de la plateforme TheLMA serait d'un grand intérêt. En ce qui concerne la thermo-aéraulique externe, la situation semble d'un abord encore plus difficile que dans le cas de simulations internes, étant donné qu'à ce niveau de nombres de Reynolds, l'épaisseur de la sous-couche visqueuse peut descendre jusqu'à 100 μm. L'évaluation de paramètres tels que les coefficients d'échanges convectifs pourrait donc nécessiter des temps de calculs très conséquents.

La plateforme TheLMA a été développée de façon à être la plus générique possible en dépit des limitations induites par la technologie CUDA. Adapter TheLMA à la résolution de schémas similaires comme la méthode de compressibilité artificielle sur réseau (LW-ACM[21]) récemment proposée, ou à la résolution d'équations aux dérivées partielles telles que l'équation de diffusion ou l'équation de Laplace, ne devrait pas présenter de difficultés majeures. Les architectures massivement parallèles alternatives, comme le MIC d'Intel ou le MMPA de Kalray, possèdent de nombreux points communs avec les GPU, en particulier en ce qui concerne les mécanismes de transfert de données. L'extension de notre plateforme à ces processeurs, bien que nécessitant probablement la réécriture de larges parties du code, devrait bénéficier des stratégies générales d'optimisation mises en place pour l'implantation de la LBM sur processeurs graphiques.

[21] Link-wise artificial compressibility method.

Article A

A New Approach to the Lattice Boltzmann Method for Graphics Processing Units

Computers and Mathematics with Applications, 61(12):3628–3638, June 2011

Accepted January 29, 2010

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 the global memory on graphic devices shows high latency and LBM is data intensive, the memory access pattern is an important issue for achieving 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 addressed this problem by using low latency on chip shared memory. Instead of this, our CUDA implementation of LBM follows carefully chosen data transfer schemes in global memory. For the 3D lid-driven cavity test case, we obtained up to 86% of the global memory maximal throughput on nVidia's GT200. We show that as a consequence highly efficient implementations of LBM on GPUs are possible, even for complex models.

Keywords: 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 [4, 18]).

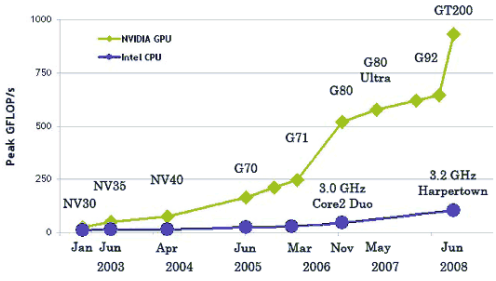


Figure 1: Peak performances GPU vs CPU (source nVidia)

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.

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 [6]). It consists of hardware specifications, a specific programming model, and a programming environment (API and SDK).

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.

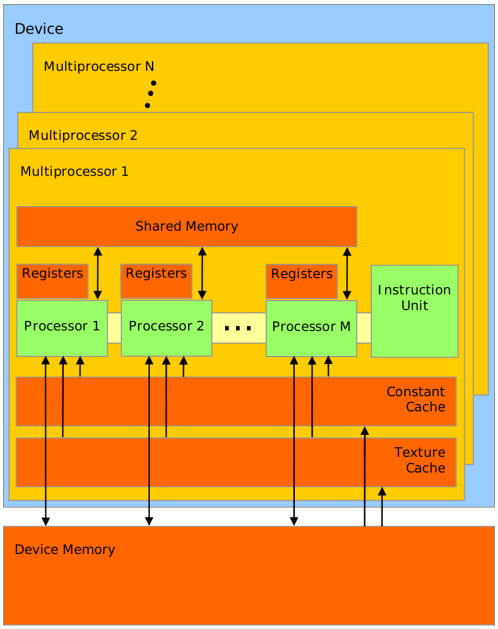


Figure 2: CUDA hardware (source nVidia)

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.

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

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.
- 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 [12]) 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).

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.

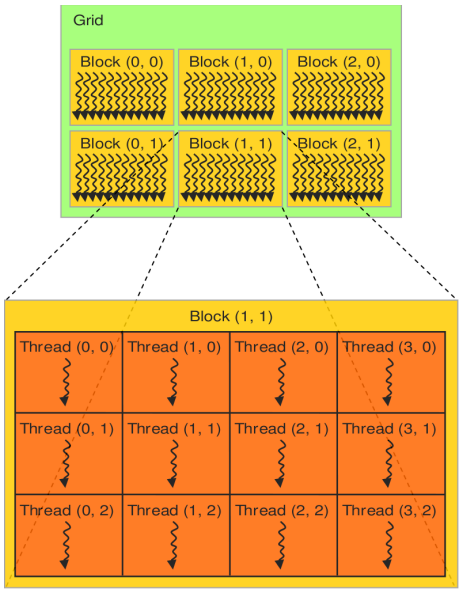


Figure 3: CUDA programming model (source nVidia)

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.

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

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 pos-

sible, 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 multiplication 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.

Measuring the execution time of the `launch_kernel` function enables us 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 GeForce GTX 295. Reckoning the characteristics of the benchmark program, we consider the obtained value as the

```
#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

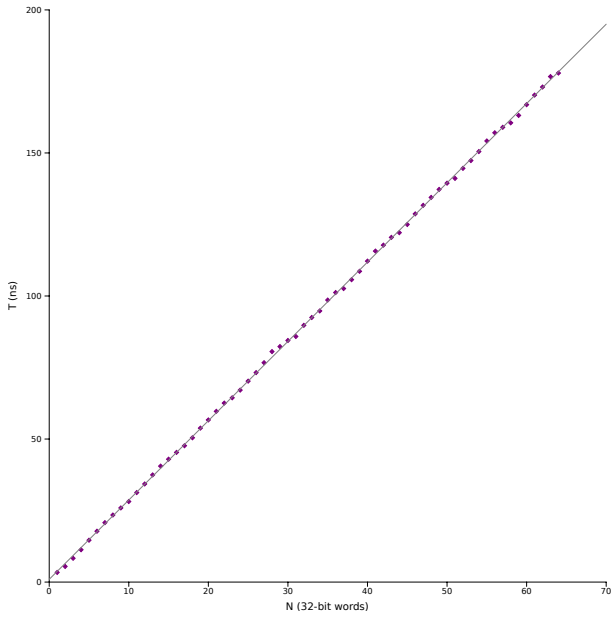


Figure 4: Average transfer time

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 [19].

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 [10]). 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.

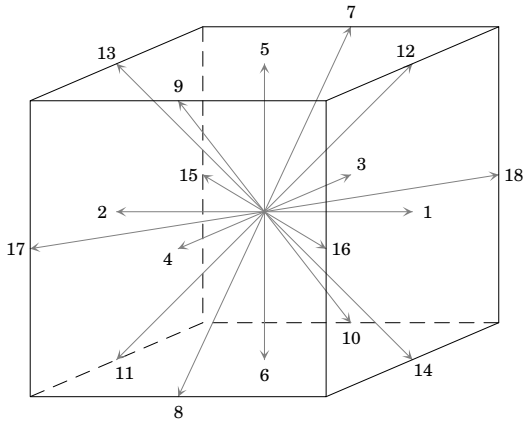


Figure 5: The D3Q19 stencil

Instead of reviewing the well-known Lattice Bhatnagar-Gross-Krook (LBGK) model (see [15]), we will outline the Multiple Relaxation Time model presented in [3]. The analogous of the one-particule distribution function f is a set of $N+1$ mass fractions f_i. We denote:

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

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 as follows:

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

where $\left|m(\mathbf{x},t)\right\rangle$ 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 [3, app. A]. Using these notations, the lattice Boltzmann equation can be written as:

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

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

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

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 [14]). 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 [16]). It consists mainly in a port of the `470.lbm` code from the SPEC CPU2006 benchmark (see [7]). 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 implementation submitted by Tölke in [19] 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.

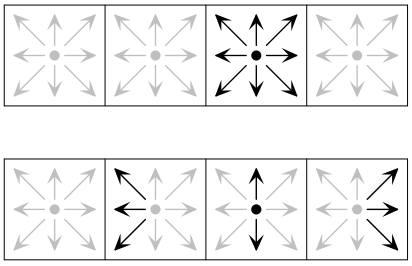


Figure 6: Propagation using shared memory

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.

Following the same method than Tölke, Habich in [5] 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

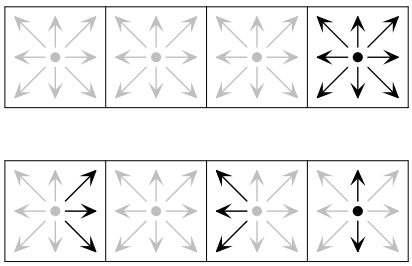


Figure 7: Storage of out-coming mass fractions

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 containing less mass fractions, like D3Q13. This approach was studied by Tölke and Krafczyk in [20], 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 [2] announce a 20% improvement of maximal performances for their implementation of D3Q19 compared to those published in [5]. 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 [13]). 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 [9]). 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.

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

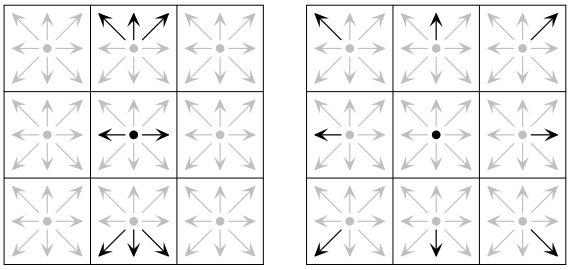


Figure 8: Split propagation scheme

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.

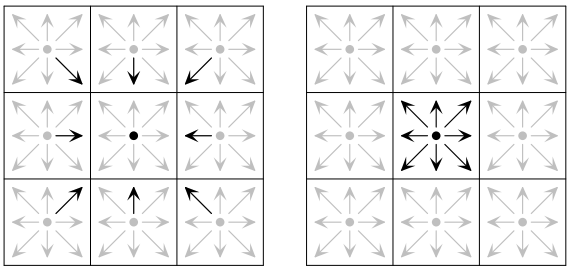


Figure 9: Reversed propagation scheme

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 that $\mathbf{e}_i = -\mathbf{e}_j$ (see [21]). 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 [8], 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 [1]. 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 $\mathrm{Re} = 1{,}000$, 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.

8.2 Performances

Binaries for nVidia GPUs are generated through a two stages process (see [11]). 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).

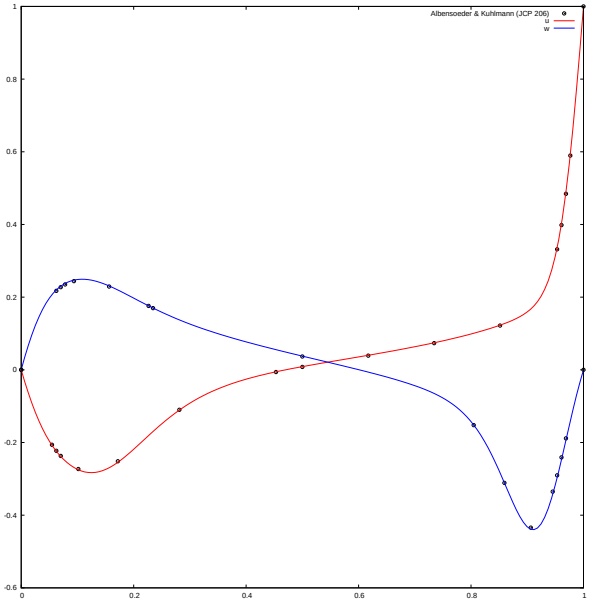


Figure 10: Validation for Re = 1,000

	`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 [5, 2]. 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 [19] 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] S. Albensoeder and H. C. Kuhlmann. Accurate three-dimensional lid-driven cavity flow. *Journal of Computational Physics*, 206(2):536–558, 2005.

[2] P. Bailey, J. Myre, S.D.C. Walsh, D.J. Lilja, and M.O. Saar. Accelerating lattice Boltzmann fluid flow simulations using graphics processors. In *Proceedings of the International Conference on Parallel Processing, 2009*, pages 550–557. IEEE, 2009.

[3] D. d'Humières, I. Ginzburg, M. Krafczyk, P. Lallemand, and L.S. Luo. Multiple-relaxation-time lattice Boltzmann models in three dimensions. *Philosophical Transactions of the Royal Society A*, 360:437–451, 2002.

[4] J. Dongarra, G. Peterson, S. Tomov, J. Allred, V. Natoli, and D. Richie. Exploring new architectures in accelerating CFD for Air Force applications. In *DoD HPCMP Users Group Conference*, pages 472–478. IEEE, 2008.

[5] J. Habich. Performance Evaluation of Numeric Compute Kernels on nVIDIA GPUs. Master's thesis, University of Erlangen-Nurnberg, 2008.

[6] T.R. Halfhill. Parallel processing with CUDA. *Microprocessor Report*, 1(28):1–8, 2008.

[7] J. L. Henning. SPEC CPU2006 Benchmark Descriptions. *ACM SIGARCH Computer Architecture News*, 34(4):1–17, 2006.

[8] F. Kuznik, C. Obrecht, G. Rusaouën, and J.-J. Roux. LBM Based Flow Simulation Using GPU Computing Processor. *Computers and Mathematics with Applications*, 59(7):2380–2392, 2010.

[9] K. Martin and B. Hoffman. *Mastering CMake, A Cross-Platform Build System*. Kitware Inc, Clifton Park NY, 2008.

[10] G. R. McNamara and G. Zanetti. Use of the Boltzmann Equation to Simulate Lattice-Gas Automata. *Physical Review Letters*, 61(20):2332–2335, 1988.

[11] M. Murphy. Nvidia's experience with open64. In *Open64 Workshop at CGO*, 2008.

[12] NVIDIA. *Compute Unified Device Architecture Programming Guide version 2.2*, 2009.

[13] NVIDIA. *CUDA Profiler version 2.2*, 2009.

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

[15] Y. H. Qian, D. d'Humières, and P. Lallemand. Lattice BGK models for Navier-Stokes equation. *EPL (Europhysics Letters)*, 17(6):479–484, 1992.

[16] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W. W. Hwu. 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*, pages 73–82. ACM, 2008.

[17] W. J. Schroeder, K. Martin, L. S. Avila, and C. C. Law. *The VTK User's Guide*. Kitware Inc, Clifton Park NY, 2006.

[18] S. Tomov, J. Dongarra, and M. Baboulin. Towards dense linear algebra for hybrid GPU accelerated manycore systems. *Parallel Computing*, 36(5):232–240, 2010.

[19] J. Tölke. Implementation of a Lattice Boltzmann kernel using the Compute Unified Device Architecture developed by nVIDIA. *Computing and Visualization in Science*, 13(1):29–39, 2010.

[20] J. Tölke and M. Krafczyk. TeraFLOP computing on a desktop PC with GPUs for 3D CFD. *International Journal of Computational Fluid Dynamics*, 22(7):443–456, 2008.

[21] Q. Zou and X. He. On pressure and velocity boundary conditions for the lattice Boltzmann BGK model. *Physics of Fluids*, 9(6):1591–1598, 1997.

Article B

Global Memory Access Modelling for Efficient Implementation of the Lattice Boltzmann Method on Graphics Processing Units

Lecture Notes in Computer Science 6449, High Performance Computing for Computational Science – VECPAR 2010 Revised Selected Papers, pages 151–161, February 2011

Accepted October 10, 2010

Abstract

In this work, we investigate the global memory access mechanism on recent GPUs. For the purpose of this study, we created specific benchmark programs, which allowed us to explore the scheduling of global memory transactions. Thus, we formulate a model capable of estimating the execution time for a large class of applications. Our main goal is to facilitate optimisation of regular data-parallel applications on GPUs. As an example, we finally describe our CUDA implementations of LBM flow solvers on which our model was able to estimate performance with less than 5% relative error.

Keywords: GPU computing, CUDA, lattice Boltzmann method, CFD

Introduction

State-of-the-art graphics processing units (GPU) have proven to be extremely efficient on regular data-parallel algorithms [3]. For many of these applications, like lattice Boltzmann method (LBM) fluid flow solvers, the computational cost is entirely hidden by global memory access. The present study intends to give some insight on the global memory access mechanism of the nVidia's GT200 GPU. The obtained results led us to optimisation elements which we used for our implementations of the LBM.

The structure of this paper is as follows. First, we briefly review nVidia's compute unified device architecture (CUDA) technology and the algorithmic aspects of the LBM. Then, we describe our measurement methodology and results. To conclude, we present our CUDA implementations of the LBM.

1 Compute Unified Device Architecture

CUDA capable GPUs, i.e. the G8x, G9x, and GT200 processors consist in a variable amount of texture processor clusters (TPC) containing two (G8x, G9x) or three (GT200) streaming multiprocessors (SM), texture units and caches [6]. Each SM contains eight scalar processors (SP), two special functions units (SFU), a register file, and shared memory. Registers and shared memory are fast but in rather limited amount, e.g. 64 KB and 16 KB per SM for the GT200. On the other hand, the off-chip global memory is large but suffers from high latency and low throughput compared to registers or shared memory.

The CUDA programming language is an extension to C/C++. Functions intended for GPU execution are named *kernels*, which are invoked on an execution grid specified at runtime. The execution grid is formed of blocks of threads. The blocks may have up to three dimensions, the grid two. During execution, blocks are dispatched to the SMs and split into warps of 32 threads.

CUDA implementations of data intensive applications are usually bound by global memory throughput. Hence, to achieve optimal efficiency, the number of global memory transactions should be minimal. Global memory transactions within a half-warp are coalesced into a single memory access whenever all the requested addresses lie in the same aligned seg-

ment of size 32, 64, or 128 bytes. Thus, improving the data access pattern of a CUDA application may dramatically increase performance.

2 Lattice Boltzmann Method

The Lattice Boltzmann Method is a rather innovative approach in computational fluid dynamics [5, 11, 2]. It is proven to be a valid alternative to the numerical integration of the Navier-Stockes equations. With the LBM, space is usually represented by a regular lattice. The physical behaviour of the simulated fluid is determined by a finite set of *mass fractions* associated to each node. From an algorithmic standpoint, the LBM may be summarised as:

for each time step **do**
  for each lattice node **do**
    if boundary node **then**
      apply boundary conditions
    end if
    compute new mass fractions
    propagate to neighbouring nodes
  end for
end for

The propagation phase follows some specific stencil. Figure 1 illustrates D3Q19, the most commonly used three-dimensional stencil, in which each node is linked to 18 of its 27 immediate neighbours.[1]

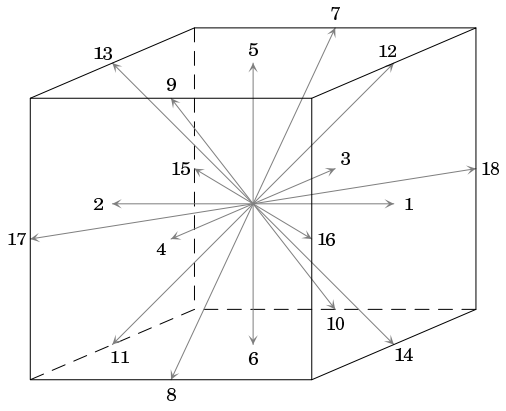


Figure 1: The D3Q19 stencil

[1] Taking the stationary mass fraction into account, the number of mass fractions per node amounts to 19, hence D3Q19.

CUDA implementations of the LBM may take advantage of its inherent data parallelism by assigning a thread to each node, the data being stored in global memory. Since there is no efficient global synchronisation barrier, a kernel has to be invoked for each time step [12]. CPU implementations of the LBM usually adopt an array of structures (AoS) data layout, which improves locality of mass fractions belonging to a same node [10]. On the other hand, CUDA implementations benefit from structure of arrays (SoA) data layouts, which allows coalesced global memory accesses [4]. However, this approach is not sufficient to ensure optimal memory transactions, since propagation corresponds to one unit shifts of global memory addresses for the minor spatial dimension. In other words, for most mass fractions, the propagation phase yields misalignments. A way to solve this issue consists in performing propagation partially in shared memory [13]. Yet, as shown in [7], this approach is less efficient than using carefully chosen propagation schemes in global memory.

3 Methodology

To study transactions between global memory and registers, we used kernels performing the following operations :

1. Store time t_0 in a register.
2. Read N words from global memory, with possibly L misalignments.
3. Store time t_1 in a register.
4. Write N words to global memory, with possibly M misalignments.
5. Store time t_2 in a register.
6. Write t_2 to global memory.

Time is accurately determined using the CUDA `clock()` function which gives access to counters that are incremented at each clock cycle. Our observations enabled us to confirm that these counters are per TPC, as described in [8], and not per SM as stated in [6]. Step 6 may influence the timings, but we shall see that it can be neglected under certain circumstances.

The parameters of our measurements are N, L, M, and k, the number of warps concurrently assigned to each SM. Number k is proportional to the occupancy rate α, which is the ratio of active warps to the maximum number of warps supported on one SM. With the GT200, this maximum number being 32, we have: $k = 32\alpha$.

We used a one-dimensional grid and one-dimensional blocks containing one single warp. Since the maximum number of blocks supported on one SM is 8, the occupancy rate is limited to 25%. Nonetheless, this rate is equivalent to the one obtained with actual CUDA applications.

We chose to create a script generating the kernels rather than using runtime parameters and loops, since the layout of the obtained code is closer to the one of actual computation kernels. We processed the CUDA binaries using decuda [14] to check whether the compiler had reliably translated our code. We carried out our measurements on a GeForce GTX 295 graphics board, featuring two GT200 processors.[2]

4 Modelling

At kernel launch, blocks are dispatched to the TPCs one by one up to k blocks per SM [1]. Since the GT200 contains ten TPCs, blocks assigned to the same TPC have identical `blockIdx.x` unit digit. This enables to extract information about the scheduling of global memory access at TPC level. In order to compare the measurements, as the clock registers are peculiar to each TPC [8], we shifted the origin of the time scale to the minimal t_0. We noticed that the obtained timings are coherent on each of the TPCs.

For a number of words read and written $N \leq 20$, we observed that:

- Reads and writes are performed in one stage, hence storing of t_2 has no noticeable influence.
- Warps 0 to 8 are launched at once (in a determined but apparently incoherent order).
- Subsequent warps are launched one after the other every ~ 63 clock cycles.

[2]In the CUDA environment, the GPUs of the GTX 295 are considered as two distinct devices. It should be noted that our benchmark programs involve only one of those devices.

For $N > 20$, reads and writes are performed in two stages. One can infer the following behaviour: if the first n warps in a SM read at least 4,096 words, where $n \in \{4, 5, 6\}$, then the processing of the subsequent warps is postponed. The number of words read by the first n warps being $n \times 32N$, this occurs whenever $n \times N \geq 128$. Hence, $n = 4$ yields $N \geq 32$, $n = 5$ yields $N \geq 26$, and $n = 6$ yields $N \geq 21$.

Time t_0 for the first $3n$ warps of a TPC follow the same pattern as in the first case. We also noticed a slight overlapping of the two stages, all the more as storing t_2 should here be taken into account. Nonetheless, the read time for the first warp in the second stage is noticeably larger than for the next ones. Therefore, we may consider, as a first approximation, that the two stages are performed sequentially.

In the targeted applications, the global amount of threads is very large. Moreover, when a set of blocks is assigned to the SMs, the scheduler waits until all blocks are completed before providing new ones. Hence, knowing the average processing time T of k warps per SM allows to estimate the global execution time.

For $N \leq 20$, we have $T = \ell + T_R + T_W$, where ℓ is time t_0 for the last launched warp, T_R is read time, and T_W is write time. Time ℓ only depends on k. For $N > 20$, we have $T = T_0 + \ell' + T'_R + T'_W$, where T_0 is the processing time of the first stage, $\ell'(i) = \ell(i - 3n + 9)$ with $i = 3k - 1$, T'_R and T'_W are read and write times for the second stage.

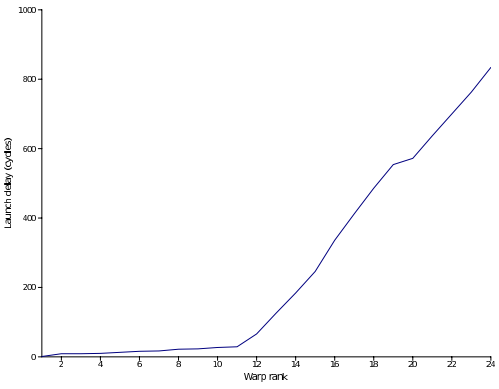


Figure 2: Launch delay in respect of warp rank

To estimate ℓ, we averaged t_0 over a large number of warps. Figure 2 shows, in increasing order, the obtained times in cycles. Numerically, we have $\ell(i) \approx$

0 for $i \leq 9$ and $\ell(i) \approx 63(i-10)+13$ otherwise.

5 Throughput

5.1 $N \leq 20$

Figures 3 and 4 show the distribution of read and write times for 96,000 warps with $N = 19$. The bimodal shape of the read time distribution is due to translation look-aside buffer (TLB) misses [15]. This aspect is reduced when adding misalignments, since the number of transactions increases while the number of misses remains constant. Using the average read time to approximate T is acceptable provided no special care is taken to avoid TLB misses.

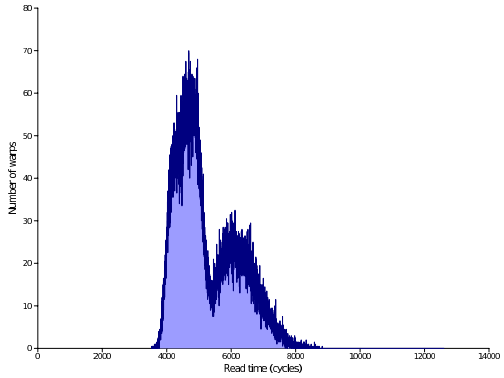


Figure 3: Read time for $N = 19$

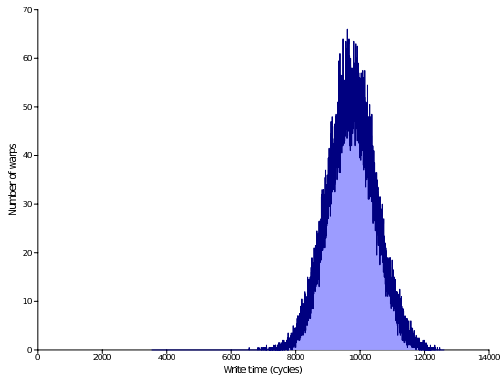


Figure 4: Write time for $N = 19$

We observed that average read and write times depend linearly of N. Numerically, with $k = 8$, we obtained:

$$T_R \approx 317(N-4)+440 \qquad T_W \approx 562(N-4)+1{,}178$$
$$T_{R'} \approx 575(N-4)+291 \qquad T_{W'} \approx 983(N-4)+2{,}030$$

where $T_{R'}$ and $T_{W'}$ are read and write times with $L = N$ and $M = N$ misalignments. Hence, we see that writes are more expensive than reads. Likewise, misalignments in writes are more expensive than misalignments in reads.

5.2 $21 \leq N \leq 39$

As shown in figures 5 and 6, T_0, T'_R, and T'_W depend linearly of N in the three intervals $\{21, \ldots 25\}$, $\{26, \ldots 32\}$, and $\{33, \ldots 39\}$. As an example, for the third interval, we obtain:

$$T_0 \approx 565(N-32)+15{,}164$$
$$T'_R \approx 112(N-32)+2{,}540 \qquad T'_W \approx 126(N-32)+3{,}988$$

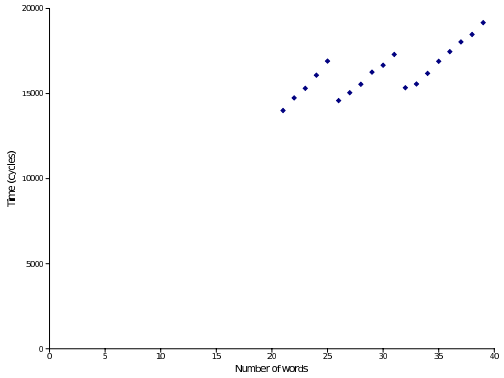


Figure 5: First stage duration

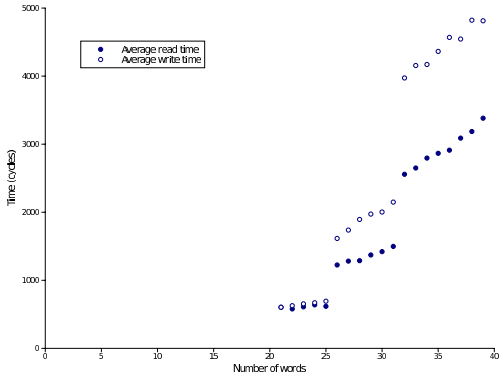


Figure 6: Timings in second stage

5.3 Complementary studies

We also investigated the impact of misalignments and occupancy rate on average read and write times. Figures 7 and 8 show obtained results for $N = 19$.

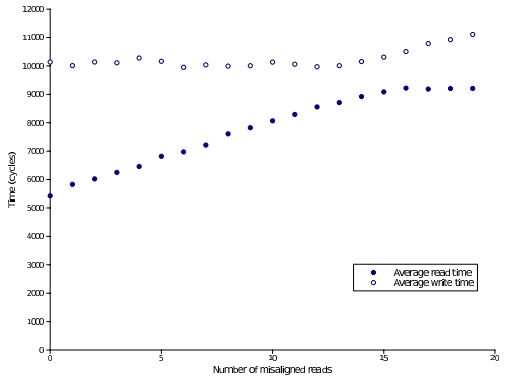


Figure 7: Misaligned reads

For misaligned reads, we observe that the average write time remains approximatively constant. Read time increases linearly with the number of misalignments until some threshold is reached. From then on, the average read time is maximal. Similar conclusion can be drawn for misaligned writes.

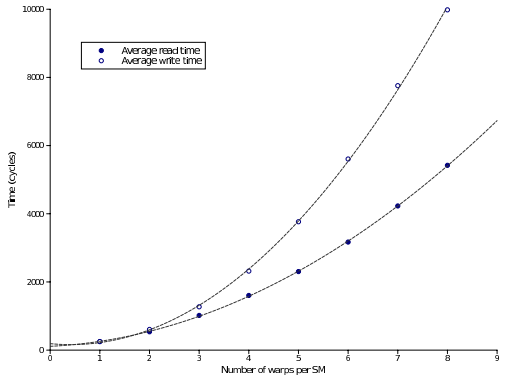


Figure 8: Occupancy impact

Average read and write times seem to depend quadratically on k. Since the amount of data transferred depends only linearly on k, this leads to think that the scheduling cost of each warp is itself proportional to k.

6 Implementations

We implemented several LBM fluid flow solvers: a D3Q19 LBGK [11], a D3Q19 MRT [2], and a double population thermal model requiring 39 words per node [9]. Our global memory access study lead us to multiple optimisations. For each implementation, we used a SoA like data layout, and a two-dimensional grid of one-dimensional blocks. Since misaligned writes are more expensive than misaligned reads, we experimented several propagation schemes in which misalignments are deferred to the read phase of the next time step. The most efficient appears to be the reversed scheme where propagation is entirely performed at reading, as outlined in figure 9. For the sake of simplicity, the diagram shows a two-dimensional version.

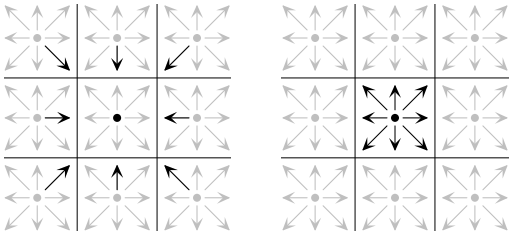


Figure 9: Reversed propagation scheme

Performance of a LBM based application is usually given in million lattice node updates per second (MLUPS). Our global memory access model enables us to give an estimate of the time T (in clock cycles) required to process k warps per SM. On the GT200, where the number of SMs is 30 and the warp size is 32, k warps per SM amounts to $K = 30 \times k \times 32 = 960k$ threads. Since one thread takes care of one single node, T is therefore the number of clock cycles needed to perform K lattice node updates. Hence, using the global memory frequency F in MHz, the expected performance in MLUPS is: $P = (K/T) \times F$.

With our D3Q19 implementations, for instance, we have $N = 19$ reads and writes, $L = 10$ misaligned reads, no misaligned writes, and 25% occupancy (thus $k = 8$). Using the estimation provided by our measurements, we obtain: $T = \ell + T_R + T_W = 15{,}594$. Since $K = 7{,}680$ and $F = 999$ MHz, we have $P = 492$ MLUPS.

To summarize, table 1 gives both the actual and estimated performances for our implementations on a 128^3 lattice. Our estimations appear to be rather

Model	Occupancy	Actual	Estimated	Relative error
D3Q19 LBGK	25%	481	492	2.3%
D3Q19 MRT	25%	516	492	4.6%
Thermal LBM	12.5%	195	196	1.0%

Table 1: Performance of LBM implementations (in MLUPS)

accurate, thus validating our model.

Summary and discussion

In this work, we present an extensive study of the global memory access mechanism between global memory and GPU for the GT200. A description of the scheduling of global memory accesses at hardware level is given. We express a model which allows to estimate the global execution time of a regular data-parallel application on GPU. The cost of individual memory transactions and the impact of misalignments is investigated as well.

We believe our model is applicable to other GPU applications provided certain conditions are met:

- The application should be data-parallel and use a regular data layout in order to ensure steady data throughput.
- The computational cost should be negligible as compared with the cost of global memory reads and writes.
- The kernel should make moderate use of branching in order to avoid branch divergence, which can dramatically impact performance. This would probably not be the case with an application dealing, for instance, with complex boundaries.

On the other hand, our model does not take possible TLB optimisation into account. Hence, some finely tuned applications may slightly outvalue our performance estimation.

The insight provided by our study, turned out to be useful in our attempts to optimize CUDA implementations of the LBM. It may contribute to efficient implementations of other applications on GPU.

References

[1] S. Collange, D. Defour, and A. Tisserand. Power Consumption of GPUs from a Software Perspective. In *Lecture Notes in Computer Science 5544, Proceedings of the 9th International Conference on Computational Science, Part I*, pages 914–923. Springer, 2009.

[2] D. d'Humières, I. Ginzburg, M. Krafczyk, P. Lallemand, and L.S. Luo. Multiple-relaxation-time lattice Boltzmann models in three dimensions. *Philosophical Transactions of the Royal Society A*, 360:437–451, 2002.

[3] J. Dongarra, G. Peterson, S. Tomov, J. Allred, V. Natoli, and D. Richie. Exploring new architectures in accelerating CFD for Air Force applications. In *DoD HPCMP Users Group Conference*, pages 472–478. IEEE, 2008.

[4] F. Kuznik, C. Obrecht, G. Rusaouën, and J.-J. Roux. LBM Based Flow Simulation Using GPU Computing Processor. *Computers and Mathematics with Applications*, 59(7):2380–2392, 2010.

[5] G. R. McNamara and G. Zanetti. Use of the Boltzmann Equation to Simulate Lattice-Gas Automata. *Physical Review Letters*, 61(20):2332–2335, 1988.

[6] NVIDIA. *Compute Unified Device Architecture Programming Guide version 2.3.1*, 2009.

[7] C. Obrecht, F. Kuznik, B. Tourancheau, and J.-J. Roux. A New Approach to the Lattice Boltzmann Method for Graphics Processing Units. *Computers and Mathematics with Applications*, 61(12):3628–3638, 2011.

[8] M.M. Papadopoulou, M. Sadooghi-Alvandi, and H. Wong. Micro-benchmarking the GT200

GPU. Technical report, University of Toronto, Canada, 2009.

[9] Y. Peng, C. Shu, and Y. T. Chew. A 3D incompressible thermal lattice Boltzmann model and its application to simulate natural convection in a cubic cavity. *Journal of Computational Physics*, 193(1):260–274, 2004.

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

[11] Y. H. Qian, D. d'Humières, and P. Lallemand. Lattice BGK models for Navier-Stokes equation. *EPL (Europhysics Letters)*, 17(6):479–484, 1992.

[12] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W. W. Hwu. 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*, pages 73–82. ACM, 2008.

[13] J. Tölke and M. Krafczyk. TeraFLOP computing on a desktop PC with GPUs for 3D CFD. *International Journal of Computational Fluid Dynamics*, 22(7):443–456, 2008.

[14] W. J. van der Laan. Decuda G80 disassembler version 0.4. Available on www.github.com/laanwj/decuda, 2007.

[15] V. Volkov and J.W. Demmel. Benchmarking GPUs to tune dense linear algebra. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*. IEEE, 2008.

Article C

The TheLMA Project: A Thermal Lattice Boltzmann Solver for the GPU

Computers and Fluids, 54:118–126, January 2012

Accepted October 8, 2011

Abstract

In this paper, we consider the implementation of a thermal flow solver based on the lattice Boltzmann method (LBM) for graphics processing units (GPUs). We first describe the hybrid thermal LBM model implemented, and give a concise review of the CUDA technology. The specific issues that arise with LBM on GPUs are outlined. We propose an approach for efficient handling of the thermal part. Performance is close to optimum and is significantly better than the one of comparable CPU solvers. We validate our code by simulating the differentially heated cubic cavity (DHC). The computed results for steady flow patterns are in good agreement with previously published ones. Finally, we use our solver to study the phenomenology of transitional flows in the DHC.

Keywords: Thermal lattice Boltzmann method, GPU programming, CUDA

1 Introduction

Originating from the lattice gas automata theory [7], the lattice Boltzmann method (LBM) was first introduced by McNamara and Zanetti in 1988 [15] and developed later on by Qian *et al.* [21]. It has since proved to be an interesting alternative to the solving of the Navier-Stokes equations. Besides other advantages over traditional methods in computational fluid dynamics, the LBM happens to be intrinsically parallel, thus easing high performance implementations.

Graphics processing units (GPUs) have nowadays outrun CPUs in terms of raw computational power. Their use in general-purpose computations [25], and more specifically in CFD [5], is promising. Successful attempts were made to implement LBM solvers on the GPU [6]. Nevertheless, the wide range of potential applications of the LBM on the GPU remains mostly unexplored, especially for problems involving heat and fluid flows.

The Compute Unified Device Architecture (CUDA), first released by nVidia in 2007, is today's leading technology for general-purpose computation on graphics processing units (GPGPU). The CUDA technology is based on general hardware specifications and a specific programming model, which allows to state generic optimization principles.

In this contribution, we shall present our CUDA implementation of a thermal flow solver. This program is an extended and improved version of the isothermal solver described in [19]. It is part of the TheLMA project [18] which aims at providing a comprehensive framework for implementing LBM solvers on GPUs and other emerging many-core architectures.

2 Hybrid thermal lattice Boltzmann model

In contrast to isothermal simulations, solving thermal fluid flows using the LBM is still a pioneering field. Up to now, adding energy-conservation constraint to the isothermal lattice Boltzmann equation seems to be a more effective method than the double-population approach. We chose to use a hybrid scheme belonging to this category. More specifically, the flow simulation is accomplished by using a D3Q19 multiple-relaxation-time (MRT) model [4],

while the heat equation is solved by using a finite-difference scheme. This model is a simplified version of the one described in [11], where the heat capacity ratio $\gamma = C_P/C_V$ is set to $\gamma = 1$, since we were not interested in acoustic effects.

The LBM can be seen as a threefold discretization of the Boltzmann equation, involving time, space and velocities. The discrete velocities $\{\boldsymbol{\xi}_i \,|\, i = 0,\ldots,N\}$ where $\boldsymbol{\xi}_0 = \mathbf{0}$, are chosen such as to link each lattice site to some of its neighbors. Figure 1 shows the D3Q19 stencil, where each node is connected to 18 of its nearest neighbors.

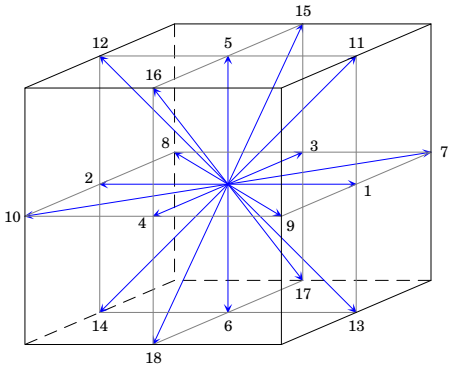


Figure 1: D3Q19 stencil

The equivalent of the single-particle distribution function in the Boltzmann equation is a discrete set of velocity distribution functions $\{f_i \,|\, i = 0,\ldots,N\}$. Let us denote:

$$\big|f_i(\boldsymbol{x},t)\big\rangle = \big(f_0(\boldsymbol{x},t),\ldots,f_N(\boldsymbol{x},t)\big)^{\mathsf{T}}$$

for given lattice node $\boldsymbol{x}$ and time t, T being the transpose operator. The lattice Boltzmann equation (LBE), i.e. the discretized version of the Boltzmann equation, thus writes:

$$\big|f_i(\boldsymbol{x}+\delta t\boldsymbol{\xi}_i,t+\delta t)\big\rangle - \big|f_i(\boldsymbol{x},t)\big\rangle = \Omega\Big(\big|f_i(\boldsymbol{x},t)\big\rangle\Big) \quad (1)$$

where Ω is the collision operator. This equation naturally breaks into the elementary steps of the LBM: the right-hand side describing *collision*, the left-hand side describing *propagation*.

In the MRT approach, the velocity distribution is mapped to a set of moments $\{m_i \,|\, i = 0,\ldots,N\}$ by an orthogonal matrix M:

$$\big|f_i(\boldsymbol{x},t)\big\rangle = \mathsf{M}^{-1}\big|m_i(\boldsymbol{x},t)\big\rangle \quad (2)$$

Matrix M for the D3Q19 stencil can be found in Appendix A of [4]. The moment vector is:

$$\big|m_i(\boldsymbol{x},t)\big\rangle = \big(\rho,e,\varepsilon,j_x,q_x,j_y,q_y,j_z,q_z,3p_{xx},3\pi_{xx},p_{ww},\pi_{ww},p_{xy},p_{yz},p_{zx},m_x,m_y,m_z\big)^{\mathsf{T}} \quad (3)$$

where ρ is the mass density, e is energy, ε is energy square, $\boldsymbol{j} = (j_x,j_y,j_z)$ is the momentum, $\boldsymbol{q} = (q_x,q_y,q_z)$ is the heat flux, p_{xx}, p_{xy}, p_{yz}, p_{zx} are components of the stress tensor and $p_{ww} = p_{yy} - p_{zz}$, π_{xx}, π_{ww} are third order moments, m_x, m_y, m_z are fourth order moments. The mass density and the momentum are the conserved moments.

The LBE becomes:

$$\big|f_i(\boldsymbol{x}+\delta t\boldsymbol{\xi}_i,t+\delta t)\big\rangle - \big|f_i(\boldsymbol{x},t)\big\rangle = -\mathsf{M}^{-1}\mathsf{S}\Big[\big|m_i(\boldsymbol{x},t)\big\rangle - \big|m_i^{(\mathrm{eq})}(\boldsymbol{x},t)\big\rangle\Big] \quad (4)$$

where S is a diagonal collision matrix and the $m_i^{(\mathrm{eq})}$ are the equilibrium values of the moments. For the sake of isotropy, S is given by:

$$\mathsf{S} = \mathrm{diag}(0,s_1,s_2,0,s_4,0,s_4,0,s_4,s_9,s_{10},s_9,s_{10},s_{13},s_{13},s_{13},s_{16},s_{16},s_{16}) \quad (5)$$

We additionally set $s_9 = s_{13}$ and the lattice speed of sound $c_s = 1/\sqrt{3}$. With T denoting the temperature, the equilibrium quantities of the non-conserved moments are given by:

$$e^{(\mathrm{eq})} = -11\rho + 19\boldsymbol{j}^2 + T \quad (6)$$

$$\varepsilon^{(\mathrm{eq})} = 3\rho \quad (7)$$

$$\boldsymbol{q}^{(\mathrm{eq})} = -\tfrac{2}{3}\boldsymbol{j} \quad (8)$$

$$3p_{xx}^{(\mathrm{eq})} = 3j_x^2 - \boldsymbol{j}^2 \quad (9)$$

$$p_{ww}^{(\mathrm{eq})} = j_y^2 - j_z^2 \quad (10)$$

$$p_{xy}^{(\mathrm{eq})} = j_x j_y \quad (11)$$

$$p_{yz}^{(\mathrm{eq})} = j_y j_z \quad (12)$$

$$p_{zx}^{(\mathrm{eq})} = j_z j_x \quad (13)$$

$$3\pi_{xx}^{(\mathrm{eq})} = \pi_{ww}^{(\mathrm{eq})} = 0 \quad (14)$$

$$m_x^{(\mathrm{eq})} = m_y^{(\mathrm{eq})} = m_z^{(\mathrm{eq})} = 0 \quad (15)$$

Since we have $\gamma = 1$, unlike [11], we chose to use T instead of $57T$ in Eq. 6, which improves numerical stability. The kinematic viscosity ν and the bulk viscosity ζ of the model are:

$$\nu = \frac{1}{3}\left(\frac{1}{s_9} - \frac{1}{2}\right) \qquad \zeta = \frac{2}{9}\left(\frac{1}{s_1} - \frac{1}{2}\right) \tag{16}$$

The temperature T evolves according to the following finite-difference equation:

$$T(\boldsymbol{x}, t+\delta t) - T(\boldsymbol{x}, t) = \kappa \Delta^* T - \boldsymbol{j} \cdot \nabla^* T \tag{17}$$

where κ denotes the thermal diffusivity, and the finite-difference operators are given by:

$$\begin{aligned} \partial_x^* f(i,j,k) &= f(i+1,j,k) - f(i-1,j,k) \\ &-\frac{1}{8}\big(f(i+1,j+1,k) - f(i-1,j+1,k) \\ &\quad + f(i+1,j-1,k) - f(i-1,j-1,k) \\ &\quad + f(i+1,j,k+1) - f(i-1,j,k+1) \\ &\quad + f(i+1,j,k-1) - f(i-1,j,k-1)\big) \end{aligned} \tag{18}$$

$$\begin{aligned} \Delta^* f(i,j,k) &= 2\big(f(i+1,j,k) + f(i-1,j,k) \\ &\quad + f(i,j+1,k) + f(i,j-1,k) \\ &\quad + f(i,j,k+1) + f(i,j,k-1)\big) \\ &-\frac{1}{4}\big(f(i+1,j+1,k) + f(i-1,j+1,k) \\ &\quad + f(i+1,j-1,k) + f(i-1,j-1,k) \\ &\quad + f(i,j+1,k+1) + f(i,j-1,k+1) \\ &\quad + f(i,j+1,k-1) + f(i,j-1,k-1) \\ &\quad + f(i+1,j,k+1) + f(i-1,j,k+1) \\ &\quad + f(i+1,j,k-1) + f(i-1,j,k-1)\big) \\ &\quad - 9f(i,j,k) \end{aligned} \tag{19}$$

It should be noted that these finite difference operators share the same symmetries as the D3Q19 stencil.

3 Review of the CUDA technology

In this section, we shall at first give a brief review of the CUDA programming model [17]. Then, we shall describe the CUDA hardware in general and the GT200 GPU we used for our computations. Last, we shall discuss the induced constraints which should be taken into account to achieve optimal efficiency.

3.1 CUDA programming model

The CUDA programming model is implemented in the CUDA C language which is an extension to C/C++. Functions in a CUDA C program belong to one of the three following categories:

1. Host code, i.e. functions run by the CPU.
2. Kernels, i.e. functions launched by host code and run by the GPU.
3. Device functions, i.e. functions run by the GPU and called by kernels or other device functions.[1]

A kernel is run in parallel on the GPU. The execution pattern is given at launch time by specifying a *grid*. Threads are grouped in identical arrays called *blocks*, which in turn are assembled to form the execution grid as shown in Fig. 2. Blocks may have up to three dimensions, a grid is one- or two-dimensional. The `x`, `y`, and `z` fields of the predefined `blockIdx` and `threadIdx` structures identify each individual thread within the execution grid.[2]

Variables local to a kernel are specific to each thread unless declared as *shared*, in which case they are accessible by all threads within a block. No mutual exclusion mechanism is available for shared variables. It is up to the programmer to manage this aspect using block-wise synchronization primitives.

Kernels may also access to *global* memory space, which is visible to each thread of the execution grid. Again, no protection mechanism is available. Nonetheless, global memory is persistent across kernel execution, hence a common way to ensure global synchronization is to perform multiple kernel launches. Global memory, being accessible to host code, is also the usual communication path between CPU and GPU.

Last, it is worth mentioning that threads may also access read-only to *constant* and *texture* memory. Constant memory is a convenient way to store parameters that will stay unchanged all along runtime.[3]

[1] Due to hardware limitations, device functions are in general inlined at compile time.

[2] When a dimension is not in use, the corresponding field is always 1.

[3] Since textures are mostly relevant in graphics processing, and is of no use in our case, we shall not discuss this feature any further.

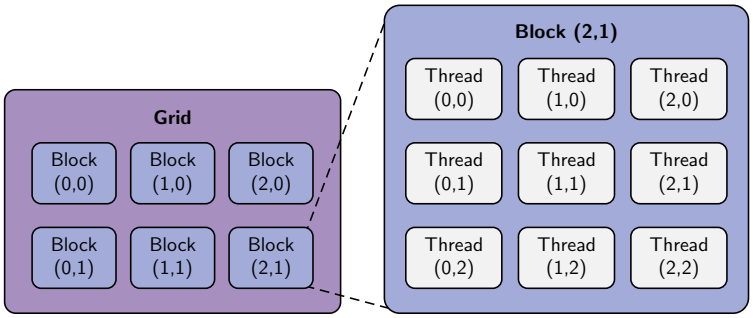


Figure 2: CUDA execution model

3.2 CUDA hardware

A CUDA capable GPU consists in a set of *streaming multiprocessors* (SMs). Each of these SMs contains several *scalar processors* (SPs), registers, shared memory, and caches for constants and textures, as shown in Fig. 3. Furthermore, the GPU is linked to off-chip device memory.

Since being specific to each SM, registers and shared memory are fast, though in rather limited amount. Device memory, in comparison, has limited throughput, suffers from high latency, but is also considerably larger. Table 1 summarizes the technical specifications of the GT200 processor we used for our computations.

Number of SMs	30
Number of SPs per SM	8
Registers per SM	16,384
Shared memory per SM	16 KB
Constant cache per SM	8 KB
Texture cache per SM	8 KB
Device memory	up to 4 GB

Table 1: Technical specifications of the GT200

The recently released Fermi architecture provides in addition L1 and L2 caches for global memory. The L1 cache is local to each SM and has configurable size: either 16 KB with 48 KB shared memory or 48 KB with 16 KB shared memory.

Variables local to a kernel are stored in registers except for arrays, since registers are not addressable. Local arrays are stored in the so-called *local* memory, which in fact is hosted in device memory, besides global, constant, and texture memory. Local memory is also used to spill registers if needed.

3.3 Optimization guidelines

A block of the execution grid can only be processed by a single SM. Yet, a SM may handle several blocks concurrently. This leads to several constraints regarding the layout of the grid which are summarized in table 2. To take advantage of the massively parallel architecture of the GPU, the number of concurrently active threads should be as large as possible. It is up to the programmer to define a grid achieving this goal, while avoiding register spilling. Nevertheless, the *occupancy rate*, i.e. the ratio of the number of active threads to the maximum, is usually not a reliable performance indicator. With data intensive applications, for instance, the limiting factor is more likely to be the global memory maximum throughput. Nonetheless, a minimal occupancy rate is required to hide global memory latency.[4]

Max. number of blocks per SM	8
Max. number of threads per SM	1,024
Max. number of threads per block	512
Maximal block dimensions	$512 \times 512 \times 64$
Maximal grid dimensions	$65\,535 \times 65\,535$

Table 2: Grid layout constraints for compute capability 1.3

When run on a SM, a block of threads is sliced into

[4] According to nVidia, at least 192 active threads per SM are required to completely hide global memory latency.

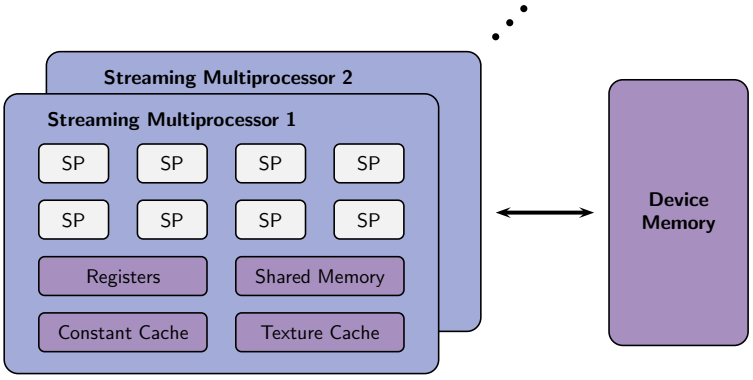


Figure 3: CUDA hardware

warps of 32 threads.[5] To achieve actual parallelism, all threads in a warp must follow the same instruction path. When branching divergence occurs, the execution of the branch paths are serialized, which may dramatically impact performance. Whenever possible, branch granularity should be a multiple of the warp size.

Shared memory is arranged in 32 bits wide memory banks. For the GT200, there are sixteen memory banks.[6] Shared memory transactions are issued by half-warps. Threads in a half-warp accessing to different memory locations lying in the same bank cause a *bank conflict*, which is resolved by serializing the transactions. Yet, shared memory may be as fast as registers, provided care is taken to avoid bank conflicts. The primary purpose of shared memory is to enable block-wise communication. Nonetheless, shared memory is also convenient to prefetch data from global memory, store small local arrays, or avoid register shortage. Such uses contribute to curtail transactions to device memory, therefore may have a major impact on performance.

For data-intensive applications, since global memory is not cached on the GT200, using a well designed memory access pattern is of crucial importance. As for shared memory, global memory transactions are issued by half-warp. These memory accesses may be coalesced into one single transaction of 32, 64, or 128 bytes, provided the address of the corresponding segment is aligned to its size. When the alignment condition is not met, several transactions are issued. As of compute capability 1.2, the hardware is able to reduce the transaction size if possible. For instance, when the threads of a half-warp read consecutive 32-bit words, it yields a single 64 bytes transaction in case of alignment, two 32 bytes transactions for a 32 bytes offset, a 32 bytes and a 64 bytes transaction otherwise.

[5] The warp size is 32 since the first generation of CUDA capable GPUs. Nevertheless, this value is implementation dependent and might change in the future.

[6] As for the warp size, the number of shared memory banks is implementation dependent.

4 Implementation

4.1 Algorithmic aspect

As described in section 2, for a given time and lattice node, the LBM breaks up in two elementary steps, namely collision and propagation. The lattice Boltzmann equation as formulated in Eq. 4 can therefore be split in:

$$\left|\widetilde{f}_i(\boldsymbol{x},t)\right\rangle = \mathsf{M}^{-1}\Bigg(\left|m_i(\boldsymbol{x},t)\right\rangle - \mathsf{S}\left[\left|m_i(\boldsymbol{x},t)\right\rangle - \left|m_i^{(\mathrm{eq})}(\boldsymbol{x},t)\right\rangle\right]\Bigg) \tag{20}$$

$$\left|f_i(\boldsymbol{x}+\delta t\boldsymbol{\xi}_i,t+\delta t)\right\rangle = \left|\widetilde{f}_i(\boldsymbol{x},t)\right\rangle \tag{21}$$

where Eq. 20 describes the collision step and Eq. 21 the propagation step. Thus, the hybrid thermal LBM outlined in section 2 corresponds to the following pseudo-code:

1. **for each** time step t **do**
2. **for each** lattice node $\boldsymbol{x}$ **do**
3. read velocity distribution $f_i(\boldsymbol{x}, t)$
4. read neighboring temperatures $T(\boldsymbol{x} + \delta t \boldsymbol{\xi}_i, t)$
5. **if** node $\boldsymbol{x}$ is on boundaries **then**
6. apply boundary conditions
7. **end if**
8. compute moments $m_i(\boldsymbol{x}, t)$
9. compute equilibrium values $m_i^{(\mathrm{eq})}(\boldsymbol{x}, t)$
10. compute updated distribution $\widetilde{f}_i(\boldsymbol{x}, t)$
11. propagate to neighboring nodes $\boldsymbol{x} + \delta t \boldsymbol{\xi}_i$
12. compute and store new temperature $T(\boldsymbol{x}, t + \delta t)$
13. **end for**
14. **end for**

4.2 GPU implementation of the LBM

To take advantage of the massive parallelism of the GPU, CUDA implementations of the LBM usually assign a thread to each lattice node [24, 10]. The layout of the execution grid needs therefore to reflect the geometry of the lattice. Global synchronization is achieved by launching a kernel at each time step.

The velocity distribution functions may be stored in either an array of structures (AoS) or a structure of arrays (SoA). The AoS approach happens to be optimal for sequential CPU implementations since it improves data locality, and thus ensures efficient use of the caches. Conversely, for GPU implementations, threads within a warp should access to consecutive global memory locations in order to enable coalesced memory transactions. Therefore, a SoA data layout (or an equivalent multi-dimensional array) is mandatory to achieve efficiency.

To meet alignment constraints, the least significant dimension of the array should be a multiple of the size of a half-warp. Nonetheless, this does not suffice to avoid misalignments, since propagation leads to one unit shifts for the minor dimension. To face this issue, an effective way is to use shared memory to perform propagation along the minor dimension. Yet, the scope of shared memory being limited to the current block, special care has to be taken of distribution values crossing the block boundaries. As described in [23], outgoing values may temporarily be stored in locations left vacant by incoming values. For large lattices, this approach requires therefore a second kernel to rearrange data.

As stated in [20], misaligned read accesses are far less expensive than misaligned write accesses. Hence, an alternative way to handle misalignment consists in replacing the usual out-of-place propagation by in-place propagation at the next time step. Figures 4 and 5 outline the two propagation schemes (in the two-dimensional case, for the sake of simplicity). It was shown in [19] that the cost of misaligned reads is of the same order of magnitude than the overhead of a rearrange kernel.

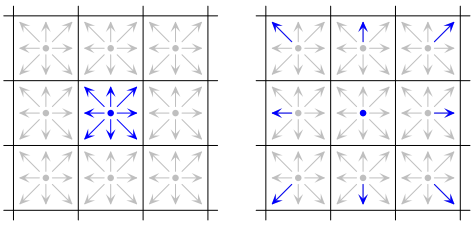


Figure 4: Out-of-place propagation scheme

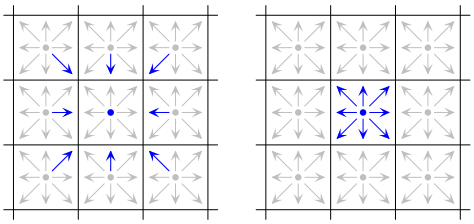


Figure 5: In-place propagation scheme

It should be noted that the in-place propagation approach is simpler and exerts less pressure on hardware than the shared memory approach. Yet, the later has only been used for isothermal LBM implementations and leaves few room for possible model enhancement. As a matter of fact, using this approach to implement on the GT200 the thermal model we chose would lead to shared memory shortage for blocks greater than 192, since ten floating point numbers per node for particle distribution and nine floating point numbers per node for temperature are required. Handling larger cavities would require the use of a rearrange kernel which has a significant

impact on performance.

4.3 Proposed implementation

Our implementation is based on the isothermal flow solver described in [19]. The lattice is a rectangular cuboid of dimensions $N_x \times N_y \times N_z$. We use one-dimensional blocks of size N_x and a two-dimensional grid of dimensions $N_y \times N_z$. For better performance, N_x should be a multiple of the warp size. Additionally, when the three lattice dimensions are different, N_x should be chosen such as to maximize the occupancy rate. Though simple, such a tilling proves to be convenient, since it ensures coalescing of global memory transactions, sufficient occupancy rate and straightforward retrieving of the node coordinates.

To store the velocity distribution functions, we used a multi-dimensional array. The velocity index may correspond to any of the dimensions but the minor one, in order to preserve coalescence. According to [25, 1], the SMs contain translation look-aside buffers (TLB) for global memory. Using the least significant dimension possible to span the velocity distribution reduces the occurrences of TLB misses. We experimented a 13% performance improvement over the major dimension version which is used in [19].

The main concern when implementing a finite-difference solver for the GPU is to curtail global memory read redundancy [16]. For a given block, the required temperatures form a $N_x \times 3 \times 3$ cuboid. Our approach is to fetch these temperatures in shared memory. To perform aligned and coalesced global memory transactions, the threads read the temperatures of nodes sharing the same abscissa. More precisely, thread of index i within block (j,k) reads the temperatures at nodes:

$$(i,j,k),\ (i,j+1,k),\ (i,j-1,k),$$
$$(i,j,k+1),\ (i,j,k-1),\ (i,j+1,k+1),$$
$$(i,j+1,k-1),\ (i,j-1,k+1),\ (i,j-1,k-1).$$

Figure 6 outlines the read access pattern we propose for temperature (in two dimensions, for the sake of simplicity). It should be noted that, using this approach, no read redundancy occurs at block level.

4.4 Boundary conditions

The solid boundaries are handled using the simple bounce-back scheme. For each direction α pointing to a solid node:

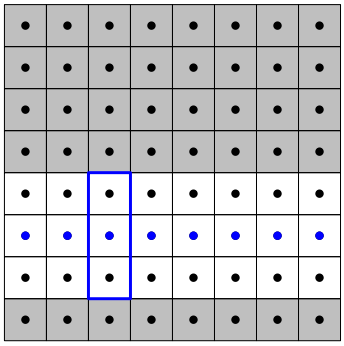


Figure 6: Read access pattern for temperature

$$f_{\bar{\alpha}}(\boldsymbol{x}, t+1) = \widetilde{f}_{\alpha}(\boldsymbol{x}, t) \tag{22}$$

where $\bar{\alpha}$ denotes the direction opposite to α. Using such scheme, the interface between fluid and solid is located at half distance between the grid nodes [8]. This aspect has to be taken into account when imposing thermal boundary conditions. In our implementation, we use halo temperatures computed by second-order extrapolations. For imposed temperature T_0, we have:

$$T(-1) = \frac{8}{3}T_0 - 2T(0) + \frac{1}{3}T(1) \tag{23}$$

For adiabatic walls, we have:

$$T(-1) = \frac{21}{23}T(0) + \frac{3}{23}T(1) - \frac{1}{23}T(2) \tag{24}$$

In blocks containing boundary nodes, the halo temperatures are computed and stored using the local temperature field in shared memory. The thermal boundary conditions therefore do not induce additional memory accesses except for adiabatic walls parallel to the block.

4.5 Performance

To evaluate performance, we carried out computations in single precision for a cubic cavity using a Tesla C1060 computing device. As usual for GPU implementations of the LBM, the limiting factor appears to be the global memory maximum throughput. For the Tesla C1060, the maximum sustained

Size of the cavity	96^3	128^3	160^3	192^3	224^3	256^3
Performance (MLUPS)	319	247	305	335	309	301
Data throughput (GB/s)	61.2	47.4	58.6	64.3	59.3	57.8
Ratio to max. throughput	83.6%	64.7%	79.9%	87.8%	81.0%	78.9%

Table 3: Performance using a Tesla C1060

throughput, provided by the CUDA `bandwidthTest` program, is about 73.3 GB/s. Table 3 gives the obtained performance in million lattice node updates per second (MLUPS) for increasing values of N, as well as the corresponding data throughput and the ratio to the maximum throughput.[7]

As shown by the rates, performance is close to optimum for most sizes. It is also worth mentioning that performance is notably higher than with state-of-the-art CPU thermal LBM implementations. For instance, we tested a Palabos [13] based single precision thermal LBM code on a dual Xeon E5560 at 2.8 GHz. We recorded 16.7 MLUPS with 16 OpenMPI processes on a 257^3 cavity.[8]

5 Differentially heated cubic cavity

5.1 Phenomenology of the differentially heated cavity

The hybrid LBM solver implemented on the GPU is used to study the differentially heated cubic cavity outlined in Fig. 7. Two opposite vertical walls have imposed temperatures $-T_0$ and $+T_0$, whereas the remaining walls are adiabatic. This configuration has been extensively studied in the two-dimensional configuration (for example [14, 3, 9]) for laminar, transitional and fully turbulent flows.

The three-dimensional configuration has been less studied in the literature than the two-dimensional case because of its computational cost. The first bifurcation is observed for $\mathrm{Ra}_1 \approx 3.3 \times 10^7$ [22] and the consequence is the unsteadiness of the flow pattern. The flow returns to a steady state for higher values of the Rayleigh number. This second transition takes place at a Rayleigh number Ra_2 belonging to the interval $[6.5 \times 10^7;\ 7 \times 10^7]$ [2]. Finally, the flow reverts to unsteadiness for $\mathrm{Ra}_3 \approx 3 \times 10^8$.

5.2 Computational procedure

The fluid is supposed to be incompressible. Applying the Boussinesq approximation, the buoyancy force $\boldsymbol{F}$ is given by:

$$\boldsymbol{F} = -\rho\beta T\boldsymbol{g} \tag{25}$$

where β is the thermal expansion coefficient, and $\boldsymbol{g}$ the gravity vector of magnitude g.

In order to conserve mass up to the second order, we add $\delta t\boldsymbol{F}$ to the momentum $\boldsymbol{j}$ in two steps: one half before collision, and one half after.

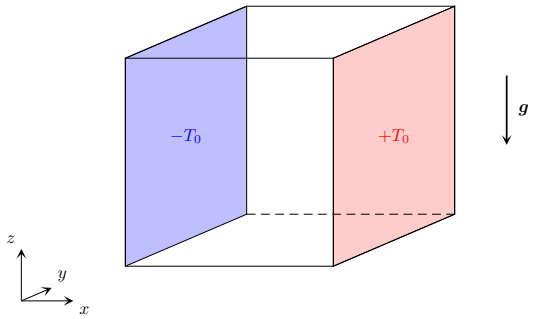


Figure 7: Differentially heated cavity

Setting the Prandtl number $\mathrm{Pr} = 0.71$, we use the Rayleigh number Ra and the kinematic viscosity ν as parameters. The thermal diffusivity κ and the value of βg are determined using the dimensionless numbers:

$$\mathrm{Ra} = \frac{2T_0\beta g N^3}{\nu\kappa} \qquad \mathrm{Pr} = \frac{\nu}{\kappa} \tag{26}$$

where $N = N_x = N_y = N_z$ is the size of the cavity. Furthermore, following [12], we set $s_1 = 1.19$, $s_2 = 1.4$, $s_4 = 1.2$, $s_{10} = 1.4$, $s_{16} = 1.98$.

[7] For each node, 48 floating point numbers are transmitted per time step: nineteen numbers are read and written for particle distribution, nine numbers are read and one number written for temperature.

[8] We used $N = 257$ instead of $N = 256$ to avoid cache thrashing.

Rayleigh number	10^4	10^5	10^6	10^7
Present	2.0560	4.3382	8.6457	16.4202
Wakashima *et al.* [26]	2.0624	4.3665	8.6973	—
Relative deviation	0.3%	0.6%	0.6%	—
Tric *et al.* [22]	2.054	4.337	8.640	16,342
Relative deviation	0.09%	0.03%	0.06%	0.5%

Table 4: Comparison of Nusselt numbers at the isothermal wall ($N = 256$)

To check for convergence, the following estimator is computed:

$$\varepsilon_n = \max_{\boldsymbol{x}} |T(\boldsymbol{x}, n\delta t) - T(\boldsymbol{x}, n\delta t - k\delta t)| \qquad (27)$$

every $k = 500$ iterations. Convergence to steadiness is declared when the criterion $\varepsilon_n < 10^{-5}$ is satisfied.

The Nusselt number at the isothermal wall is computed using:

$$\mathrm{Nu}_w = \frac{1}{2T_0 N^2} \sum_{y,z} \partial_x T\big|_{x=wall}. \qquad (28)$$

5.3 Numerical results for steady flow patterns

In order to validate our approach, the flow in the differentially heated cavity is computed for Rayleigh numbers equal to 10^4, 10^5, 10^6, and 10^7. The results are compared with data from the literature. Table 4 gives the obtained Nusselt numbers as well as the values published in [26] and [22]. As shown by the relative deviation, our results are in good accordance with the reference values.

In addition, Fig. 8 shows the isosurfaces of temperature in the cavity. As the Rayleigh number increases, the temperature in the core of the cavity becomes more stratified. The flow is less influenced by the cavity sidewalls ($y = \pm 1$) and the boundary layers at the active walls become thinner and thinner. It is also possible to confirm the centrosymmetry of the flow and temperature fields.

5.4 Numerical results concerning the first and second bifurcation

To locate the bifurcations, we proceeded a to systematic exploration using several GPUs in parallel. According to our computations, the first bifurcation occurs between 3.224×10^7 and 3.225×10^7. The critical Rayleigh number for the first bifurcation is therefore $\mathrm{Ra}_1 = 3.2245 \pm 0.0005 \times 10^7$.

For Rayleigh numbers greater than the second critical Rayleigh number Ra_2, the flow returns to a steady state. The limit is located between 6.401×10^7 and 6.402×10^7. Hence, the second critical Rayleigh number is $\mathrm{Ra}_2 = 6.4015 \pm 0.0005 \times 10^7$.

In order to exemplify the flow and temperature fields for the reversion to steadiness, the flow at $\mathrm{Ra} = 10^8$ is exhibited here. Figure 9a shows that the Nusselt number reaches a constant value of $\mathrm{Nu} = 30.2027$, which is in good agreement with the value Nu=30.311 extrapolated in [22].

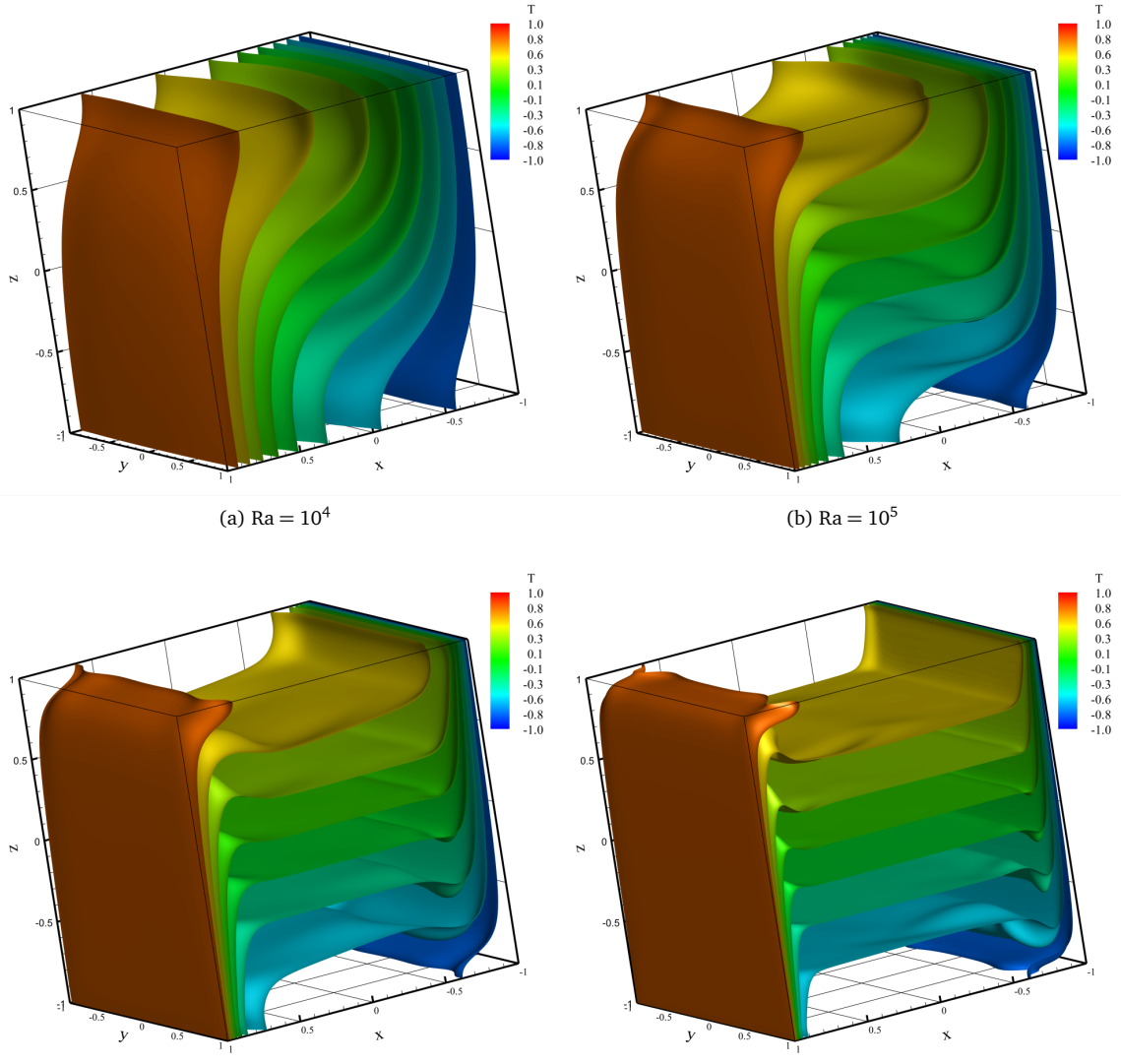
The isosurfaces of temperature, Fig. 9b, show that the thermal stratification in the core of the cavity is conserved.

The modifications of the flow field are illustrated by the isosurfaces of the u velocity component in the half-cavity (Fig. 10a) and in the entire cavity (Fig. 10b). The flow field is no more centrosymmetric. However, there is a symmetry with respect to the plane $y = 0$. The same conclusion holds for the isosurfaces of v and of w (See Fig. 11).

6 Summary

In this work, we devise general optimization strategies for programming data-parallel applications on CUDA enabled GPUs. We describe an effective implementation of a thermal LBM solver for the GPU. The proposed approach for dealing with the thermal part is likely to apply to other multiphysics coupling. Simulation results are in good agreement with available data. Performance is nearly optimal and appears to be significantly higher than for equivalent CPU implementations.

We used a dichotomous procedure to accurately



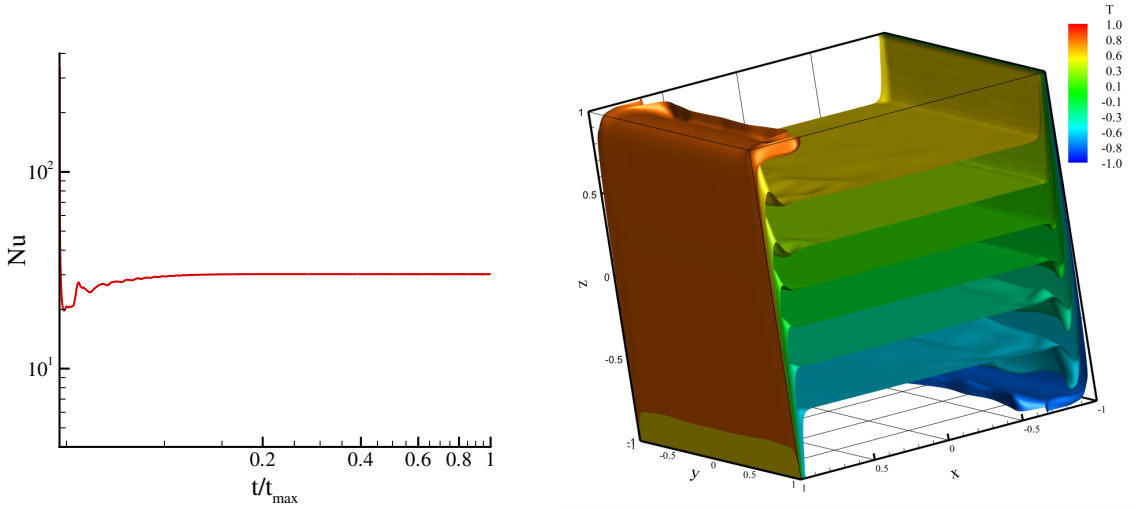
(a) Ra $= 10^4$

(b) Ra $= 10^5$

(c) Ra $= 10^6$

(d) Ra $= 10^7$

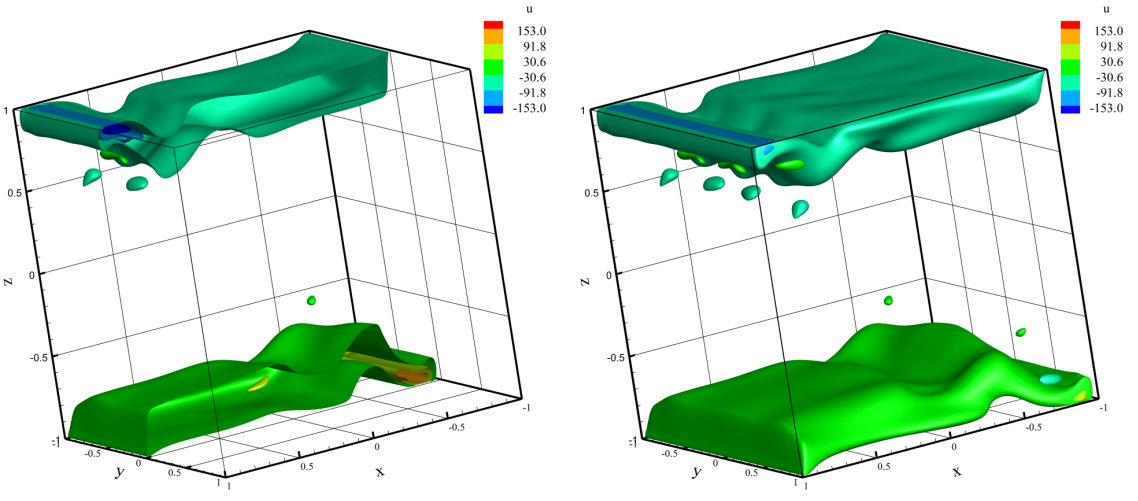
Figure 8: Isosurfaces of temperature



(a) Nusselt number at the wall

(b) Isosurfaces of temperature

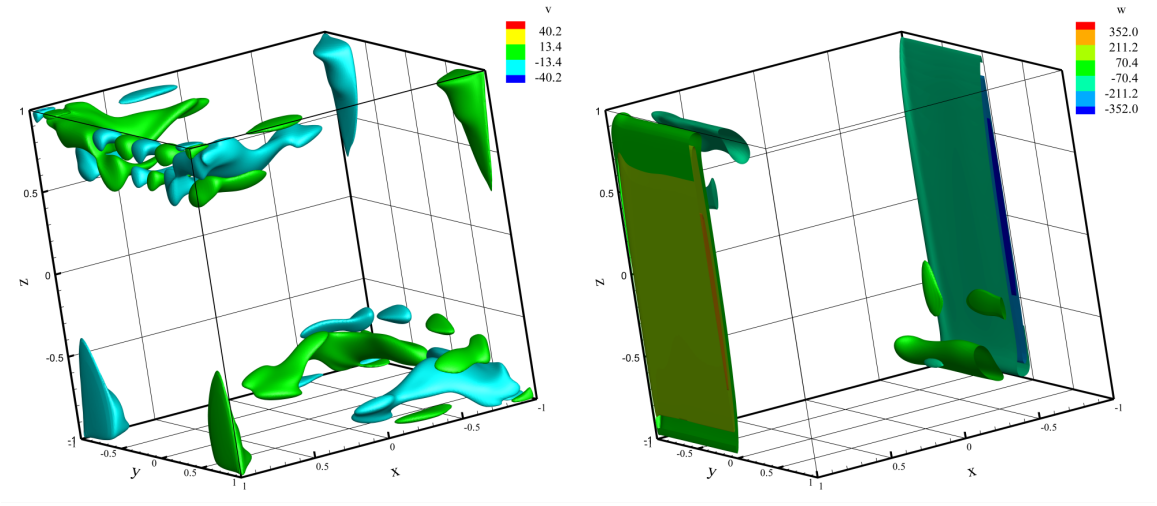
Figure 9: Results concerning the differentially heated cavity at $\mathrm{Ra} = 10^8$



(a) Half-cavity

(b) Entire cavity

Figure 10: Isosurfaces of u for $\mathrm{Ra} = 10^8$



(a) (b)

Figure 11: Isosurfaces of v and w for $\mathrm{Ra} = 10^8$

study the different flow patterns in the differentially heated cubic cavity. The flow pattern is laminar up to the first bifurcation at $\mathrm{Ra}_1 = 3.2245 \pm 0.0005 \times 10^7$. The flow becomes unsteady until $\mathrm{Ra}_2 = 6.4015 \pm 0.0005 \times 10^7$ for which it returns to a steady state. The present contribution is the first accurate determination of these critical Rayleigh numbers in the differentially heated cubic cavity. The next step of our work will be the study of the transition to turbulence around $\mathrm{Ra} = 3 \times 10^8$.

References

[1] S. Collange, D. Defour, and A. Tisserand. Power Consumption of GPUs from a Software Perspective. In *Lecture Notes in Computer Science 5544, Proceedings of the 9th International Conference on Computational Science, Part I*, pages 914–923. Springer, 2009.

[2] G. de Gassowski, S. Xin, and O. Daube. Bifurcations et solutions multiples en cavité 3D différentiellement chauffée. *Comptes Rendus Mécanique*, 331(10):705–711, 2003.

[3] G. De Vahl Davis. Natural convection of air in a square cavity: A bench mark numerical solution. *International Journal for Numerical Methods in Fluids*, 3(3):249–264, 1983.

[4] D. d'Humières, I. Ginzburg, M. Krafczyk, P. Lallemand, and L.S. Luo. Multiple-relaxation-time lattice Boltzmann models in three dimensions. *Philosophical Transactions of the Royal Society A*, 360:437–451, 2002.

[5] J. Dongarra, G. Peterson, S. Tomov, J. Allred, V. Natoli, and D. Richie. Exploring new architectures in accelerating CFD for Air Force applications. In *DoD HPCMP Users Group Conference*, pages 472–478. IEEE, 2008.

[6] Z. Fan, F. Qiu, A. Kaufman, and S. Yoakum-Stover. GPU cluster for high performance computing. In *Proceedings of the 2004 ACM/IEEE Conference on Supercomputing*, pages 47–58. IEEE, 2004.

[7] U. Frisch, B. Hasslacher, and Y. Pomeau. Lattice-Gas Automata for the Navier-Stokes Equation. *Physical Review Letters*, 56(14):1505–1508, 1986.

[8] I. Ginzbourg and D. d'Humières. Local second-order boundary method for lattice Boltzmann models. *Journal of Statistical Physics*, 84(5):927–971, 1996.

[9] F. Kuznik, Vareilles J., G. Rusaouën, and G. Krauss. A double-population lattice Boltzmann method with non-uniform mesh for the simulation of natural convection in a square cavity. *International Journal of Heat and Fluid Flow*, 28(5):862–870, 2007.

[10] F. Kuznik, C. Obrecht, G. Rusaouën, and J.-J. Roux. LBM Based Flow Simulation Using GPU Computing Processor. *Computers and Mathematics with Applications*, 59(7):2380–2392, 2010.

[11] P. Lallemand and L. S. Luo. Theory of the lattice Boltzmann method: Acoustic and thermal properties in two and three dimensions. *Physical review E*, 68(3):36706(1–25), 2003.

[12] P. Lallemand and L.S. Luo. Theory of the lattice Boltzmann method: Dispersion, dissipation, isotropy, Galilean invariance, and stability. *Physical Review E*, 61(6):6546–6562, 2000.

[13] J. Latt, O. Malaspinas, and D. Lagrava. Parallel Lattice Boltzmann Solver. Available on www.lbmethod.org/palabos, 2010.

[14] P. Le Quéré. Accurate solutions to the square thermally driven cavity at high Rayleigh number. *Computers & Fluids*, 20(1):29–41, 1991.

[15] G.R. McNamara and G. Zanetti. Use of the Boltzmann Equation to Simulate Lattice-Gas Automata. *Physical Review Letters*, 61(20):2332–2335, 1988.

[16] P. Micikevicius. 3D finite difference computation on GPUs using CUDA. In *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, pages 79–84. ACM, 2009.

[17] NVIDIA. *Compute Unified Device Architecture Programming Guide version 2.3.1*, 2009.

[18] C. Obrecht, F. Kuznik, B. Tourancheau, and J.-J. Roux. Thermal LBM on Many-core Architectures. Available on www.thelma-project.info, 2010.

[19] C. Obrecht, F. Kuznik, B. Tourancheau, and J.-J. Roux. A new approach to the lattice Boltzmann method for graphics processing units. *Computers and Mathematics with Applications*, 61(12):3628–3638, 2011.

[20] C. Obrecht, F. Kuznik, B. Tourancheau, and J.-J. Roux. Global Memory Access Modelling for Efficient Implementation of the Lattice Boltzmann Method on Graphics Processing Units. In *Lecture Notes in Computer Science 6449, High Performance Computing for Computational Science, VECPAR 2010 Revised Selected Papers*, pages 151–161. Springer, 2011.

[21] Y.H. Qian, D. d'Humières, and P. Lallemand. Lattice BGK models for Navier-Stokes equation. *EPL (Europhysics Letters)*, 17(6):479–484, 1992.

[22] E. Tric, G. Labrosse, and M. Betrouni. A first incursion into the 3D structure of natural convection of air in a differentially heated cubic cavity, from accurate numerical solutions. *International Journal of Heat and Mass Transfer*, 43(21):4043–4056, 2000.

[23] J. Tölke. Implementation of a Lattice Boltzmann kernel using the Compute Unified Device Architecture developed by nVIDIA. *Computing and Visualization in Science*, 13(1):29–39, 2010.

[24] J. Tölke and M. Krafczyk. TeraFLOP computing on a desktop PC with GPUs for 3D CFD. *International Journal of Computational Fluid Dynamics*, 22(7):443–456, 2008.

[25] V. Volkov and J.W. Demmel. Benchmarking GPUs to tune dense linear algebra. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*. IEEE, 2008.

[26] S. Wakashima and T.S. Saitoh. Benchmark solutions for natural convection in a cubic cavity using the high-order time-space method. *International Journal of Heat and Mass Transfer*, 47(4):853–864, 2004.

Article D

Multi-GPU Implementation of the Lattice Boltzmann Method

Computers and Mathematics with Applications, published online March 17, 2011

Accepted February 14, 2011

Abstract

The lattice Boltzmann method (LBM) is an increasingly popular approach for solving fluid flows in a wide range of applications. The LBM yields regular, data-parallel computations; hence, it is especially well fitted to massively parallel hardware such as graphics processing units (GPU). Up to now, though, single-GPU implementations of the LBM are of moderate practical interest since the on-board memory of GPU-based computing devices is too scarce for large scale simulations.

In this paper, we present a multi-GPU LBM solver based on the well-known D3Q19 MRT model. Using appropriate hardware, we managed to run our program on six Tesla C1060 computing devices in parallel. We observed up to 2.15×10^9 node updates per second for the lid-driven cubic cavity test case. It is worth mentioning that such a performance is comparable to the one obtained with large high performance clusters or massively parallel supercomputers.

Our solver enabled us to perform high resolution simulations for large Reynolds numbers without facing numerical instabilities. Though, we could observe symmetry breaking effects for long-extended simulations of unsteady flows. We describe the different levels of precision we implemented, showing that these effects are due to round off errors, and we discuss their relative impact on performance.

Keywords: GPU programming, CUDA, Lattice Boltzmann method, TheLMA project

1 Introduction

Although the original Moore's law [14], i.e. the exponential growth of transistor count on processors is still valid nowadays, the advances in computing performance are less straightforward. During the last decade, graphics processing units (GPU) have gradually outrun CPUs in terms of raw computational power. Using nVidia's CUDA technology [15], GPUs have proven to be effective platforms to implement various high performance computing applications, ranging from linear algebra [1] to CFD [5] and PDE solvers [13].

The lattice Boltzmann method (LBM) is a novel approach in computational fluid dynamics. It appears to be an interesting alternative to the solving of the Navier-Stockes equations for various applications such as multiphase flows or porous media. As other CFD methods, the LBM is very demanding from a computational standpoint. High performance parallel implementations are therefore necessary for the LBM to be of practical interest.

Several successful implementations for the GPU are described in literature [21, 22, 7]. Nonetheless, single-GPU implementations are bound by the device memory. The maximum available amount, when using GT200 GPUs, is 4 GB, which enables us to handle at most about 2.83×10^7 nodes in single precision using the three-dimensional D3Q19 stencil. Multi-GPU implementations are therefore mandatory to run large scale LBM simulations, but are still a pioneering field and performance is often below what is expected from such hardware [20].

Recently released motherboards are able to manage up to eight GPU based computing devices. Although an MPI based multi-GPU LBM solver would be

of great interest to run on hybrid clusters, we chose as a first step to implement a simpler POSIX thread based solver. The remainder of the paper is organised as follows. We first briefly review the LBM and the CUDA technology. Then we give some general guidelines for implementing the LBM on GPUs and describe our multi-GPU implementation. Last, we discuss numerical issues we could observe running large scale simulations at high Reynolds numbers.

2 Multiple-Relaxation-Time LBM

Although originating from the lattice-gas automata theory [6], the lattice Boltzmann method is now generally interpreted as a way to solve the linearised Boltzmann equation [12]. In our work, we used the multiple-relaxation-time (MRT) approach [3] instead of the more popular Bhatnagar-Gross-Krook (BGK) version of the LBM [19]. In this section we shall briefly describe the MRT LBM.

With the Boltzmann equation, a fluid is described using a single-particle distribution function f depending on space and particular velocity, i.e. phase space, and on time. In the LBM, space is usually represented by a regular orthogonal mesh of resolution δx and time is split in constant steps δt. The discrete counterpart of the continuous velocity space is a finite set of velocities ξ_i, carefully chosen in order to ensure sufficient isotropy. Usually, vectors $\delta t \xi_i$ link nodes to only some of their nearest neighbours. As an example, fig. 1 shows the D3Q19 stencil we used in our computations.

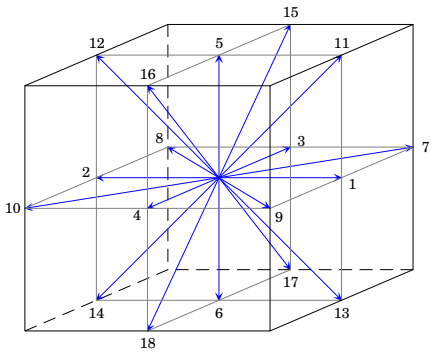


Figure 1: D3Q19 stencil

Let us denote: $\left|a_i\right\rangle = (a_0, \dots a_N)^{\mathsf{T}}$, T being the transpose operator. The lattice Boltzmann equation (LBE) writes:

$$\left|f_i(\boldsymbol{x}+\delta t\boldsymbol{\xi}_i, t+\delta t)\right\rangle - \left|f_i(\boldsymbol{x},t)\right\rangle = \Omega\Big[\left|f_i(\boldsymbol{x},t)\right\rangle\Big] \quad (1)$$

where $\{f_i \,|\, i = 0, \dots N\}$ is the discrete equivalent of f, and Ω is the collision operator.

In the MRT approach, collision is performed in moment space. The particle distribution is mapped to a set of moments $\{m_i \,|\, i = 0, \dots N\}$ by an orthogonal matrix M:

$$\left|f_i(\boldsymbol{x},t)\right\rangle = \mathsf{M}^{-1}\left|m_i(\boldsymbol{x},t)\right\rangle \quad (2)$$

where $\left|m(\boldsymbol{x},t)\right\rangle$ is the moment vector. Matrix M for the D3Q19 stencil can be found in appendix A of [4]. The corresponding moment vector is:

$$\left|m_i(\boldsymbol{x},t)\right\rangle = (\rho, e, \varepsilon, j_x, q_x, j_y, q_y, j_z, q_z, 3p_{xx}, 3\pi_{xx}, p_{ww}, \pi_{ww}, p_{xy}, p_{yz}, p_{zx}, m_x, m_y, m_z)^{\mathsf{T}} \quad (3)$$

where ρ is the mass density, e is energy, ε is energy square, $\boldsymbol{j} = (j_x, j_y, j_z)$ is the momentum, $\boldsymbol{q} = (q_x, q_y, q_z)$ is the heat flux, p_{xx}, p_{xy}, p_{yz}, p_{zx}, p_{ww} are related to the components of the stress tensor, π_{xx}, π_{ww} are fourth-order moments and m_x, m_y, m_z are third-order moments with respect to the particle velocities. The mass density and the momentum are the conserved moments.

The LBE may thus be written as:

$$\left|f_i(\boldsymbol{x}+\delta t\boldsymbol{\xi}_i, t+\delta t)\right\rangle - \left|f_i(\boldsymbol{x},t)\right\rangle = -\mathsf{M}^{-1}\mathsf{S}\left[\left|m_i(\boldsymbol{x},t)\right\rangle - \left|m_i^{(\mathrm{eq})}(\boldsymbol{x},t)\right\rangle\right] \quad (4)$$

where S is a diagonal collision matrix and the $m_i^{(\mathrm{eq})}$ are the equilibrium values of the moments. For the sake of isotropy, S obeys:

$$\mathsf{S} = \mathrm{diag}(0, s_1, s_2, 0, s_4, 0, s_4, 0, s_4, s_9, s_{10}, s_9, s_{10}, s_{13}, s_{13}, s_{13}, s_{16}, s_{16}, s_{16}) \quad (5)$$

We additionally set $s_9 = s_{13}$, the initial density $\rho_0 = 1$, and the speed of sound $c_s = 1/\sqrt{3}$, the unit of speed being $\delta x/\delta t$. The equilibrium values of the non-conserved moments are thus given by:

$$e^{(\mathrm{eq})} = -11\rho + 19\boldsymbol{j}^2 \quad (6)$$

$$\varepsilon^{(\mathrm{eq})} = -\frac{475}{63}\boldsymbol{j}^2 \tag{7}$$

$$\boldsymbol{q}^{(\mathrm{eq})} = -\frac{2}{3}\boldsymbol{j} \tag{8}$$

$$3p_{xx}^{(\mathrm{eq})} = 3j_x^2 - \boldsymbol{j}^2 \tag{9}$$

$$p_{ww}^{(\mathrm{eq})} = j_y^2 - j_z^2 \tag{10}$$

$$p_{xy}^{(\mathrm{eq})} = j_x j_y, \quad p_{yz}^{(\mathrm{eq})} = j_y j_z, \quad p_{zx}^{(\mathrm{eq})} = j_z j_x \tag{11}$$

$$3\pi_{xx}^{(\mathrm{eq})} = \pi_{ww}^{(\mathrm{eq})} = 0 \tag{12}$$

$$m_x^{(\mathrm{eq})} = m_y^{(\mathrm{eq})} = m_z^{(\mathrm{eq})} = 0 \tag{13}$$

The kinematic viscosity ν of the model is related to relaxation rate s_9 by:

$$\nu = \frac{1}{3}\left(\frac{1}{s_9} - \frac{1}{2}\right) \tag{14}$$

The other rates are set according to [8]. Namely: $s_1 = 1.19$, $s_2 = s_{10} = 1.4$, $s_4 = 1.2$, and $s_{16} = 1.98$.

3 Overview of the CUDA Technology

The *Compute Unified Device Architecture* (CUDA) is nowadays the leading technology for general purpose computations on GPUs. Initiated in late 2007 by the nVidia company, CUDA defines both a programming model and general hardware specifications. CUDA capable GPUs consist of a set of streaming multiprocessors (SM), each containing several scalar processors (SP) as outlined in fig. 2. The SPs within a SM follow a single-instruction multiple-data (SIMD) execution scheme. Yet, SMs are not globally synchronised, thus the overall execution scheme may be described as single-instruction multiple-thread (SIMT).

CUDA computing devices show a complex memory hierarchy. The main storage consists of a rather large off-chip device memory. This memory is not cached except for specific read-only data (i.e. constants and textures); hence it suffers from high latency which has to be properly hidden. Each SM provides its SPs with non-addressable registers and some addressable shared memory which allows inter-SP communication.

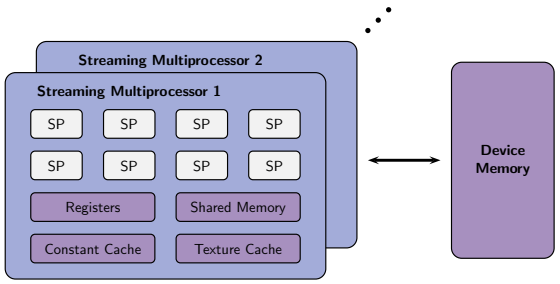


Figure 2: CUDA hardware

The CUDA programming language is an extension to C/C++ (with some restrictions). A CUDA program basically consists of CPU code and (at least) one kernel, i.e. a void returning function to be executed by the GPU. Kernels are executed in several threads with private local variables. Threads are grouped in identical blocks which may have up to three dimensions. During execution, a block cannot be partitioned and therefore must fit into a single SM. Nonetheless, an SM may execute several blocks concurrently. Threads within a block may be synchronised and have access to a shared memory space. Yet, no protection mechanism, e.g. mutexes, is available: it is up to the programmer to manage this aspect.

Blocks are grouped into a one or two-dimensional execution grid, specified at launch time. Blocks are executed asynchronously and there is no efficient dedicated mechanism to ensure global synchronisation. All threads within a grid have access to a global memory space which is hosted in the device memory and is persistent along the application life cycle (see Fig. 3). Global synchronisation is therefore achieved by performing multiple kernel launches.

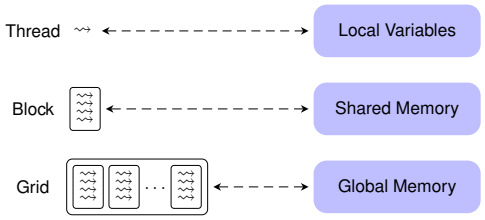


Figure 3: CUDA programming model

4 GPU Implementation Guidelines

From an algorithmic point of view, the LBM breaks into two elementary steps: *collision* in which the collision operator is applied to the particle distribution, and *propagation* in which updated particle populations are propagated to the neighbouring nodes. Equation 4 may therefore be split in :

$$\left|\tilde{f}_i(\boldsymbol{x},t)\right\rangle = \mathsf{M}^{-1}\Bigg(\left|m_i(\boldsymbol{x},t)\right\rangle - \mathsf{S}\left[\left|m_i(\boldsymbol{x},t)\right\rangle - \left|m_i^{(\mathrm{eq})}(\boldsymbol{x},t)\right\rangle\right]\Bigg) \quad (15)$$

$$\left|f_i(\boldsymbol{x}+\delta t\boldsymbol{\xi}_i, t+\delta t)\right\rangle = \left|\tilde{f}_i(\boldsymbol{x},t)\right\rangle \quad (16)$$

where eq. 15 describes the collision step and eq. 16 the propagation step. Thus, the LBM described in sec. 2, may be summarised by the following pseudo-code:

1. **for each** time step t **do**
2. **for each** lattice node $\boldsymbol{x}$ **do**
3. read velocity distribution $f_i(\boldsymbol{x}, t)$
4. **if** node $\boldsymbol{x}$ is on boundaries **then**
5. apply boundary conditions
6. **end if**
7. compute moments $m_i(\boldsymbol{x}, t)$
8. compute equilibrium values $m_i^{(\mathrm{eq})}(\boldsymbol{x}, t)$
9. compute updated distribution $\tilde{f}_i(\boldsymbol{x}, t)$
10. propagate to neighboring nodes $\boldsymbol{x} + \delta t\boldsymbol{\xi}_i$
11. **end for**
12. **end for**

The most convenient approach to take advantage of the massive parallelism of GPUs is to assign one thread to each node. Threads within a block are executed in groups of 32 threads named *warps*.[1] Global memory transactions are issued by half-warp. Best performance is achieved when these operations may be coalesced into single transactions of 32 B, 64 B, or 128 B. Yet, segment transactions face the important restriction that the segment's offset has to be a multiple of its size.

Optimised CPU implementations of the LBM generally store the particle distribution in an array of structures, which improves data locality. In order to allow coalescing, GPU implementations must adopt a reverse approach. A simple and efficient solution is to use one dimensional blocks corresponding to a given spatial direction and to store the particle distribution in a multi-dimensional array. The minor dimension of the array is chosen such that contiguous threads access contiguous memory locations.

Nonetheless, this approach is not sufficient to ensure optimal memory transactions. For most of the particle populations, the propagation step leads to one unit shifts in addresses as illustrated by fig. 4.

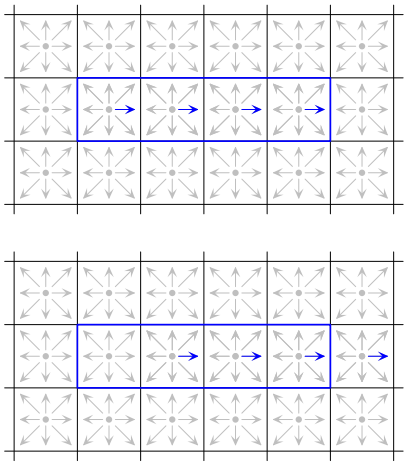


Figure 4: Misalignment issue

With the first generation of CUDA enabled GPUs, i.e. for compute capability up to 1.1, alignment is mandatory for coalescence to occur, hence misalignment has a dramatic impact on performance. To address this problem, propagation within the blocks may be performed using shared memory as described in [21]. As of compute capability 1.2, misaligned memory accesses are issued in as few segment transactions as possible. As thoroughly shown in [18], misaligned reads are far less expensive than misaligned writes, hence a rather efficient way to perform propagation is to use the out-of-place propagation scheme [17], outlined in fig. 5.

[1] The size of a warp is implementation dependent and may vary in the future.

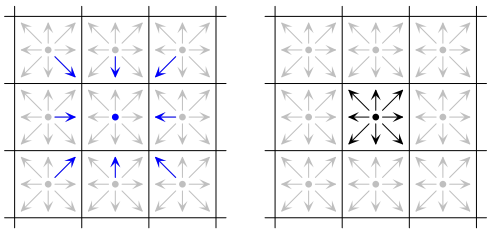


Figure 5: Out-of-place propagation

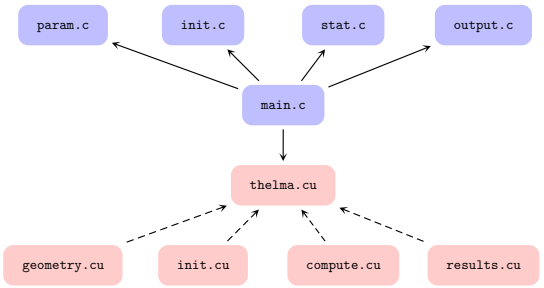


Figure 6: The TheLMA framework

5 Multi-GPU Implementation of the LBM

Developing libraries is a common and acknowledged practise in software engineering. The Palabos project [10] for instance, in the case of LBM, provides a wide set of generic functions which allows to efficiently implement a parallel CPU LBM solver with given geometry, boundary conditions, and the lattice Boltzmann model.

Nevertheless, the CUDA technology has some inherent limitations which make it difficult to follow the same path when developing GPU LBM solvers. The compilation tool chain, for instance, being unable to link GPU binaries forbids actual modular programming. Likewise, devices of compute capability up to 1.3 have limited support for functions. The so-called *device* functions, i.e. functions to be executed by the GPU, are mostly inlined at compile time, which restricts their use in practise.

In order to improve code reusability, we designed the TheLMA framework [16]. TheLMA stands for *Thermal LBM on Many-core Architectures*, thermal flow simulation being our main topic of interest. It provides a global template for multi-GPU LBM solvers on which we developed the present implementation. Figure 6 outlines the structure of the framework.

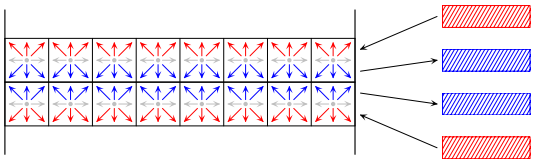
The `main.c` file contains the main loop of the simulation and may access to a set of commodity functions in order to retrieve parameters, initialise variables, perform statistical calculations, and output simulation results in various formats. The `thelma.cu` file is a hub containing some general macros and including the CUDA components responsible for setting up the geometry, initialising, running the simulation and extracting results. Each of these component contains a launch function which is accessible to the C part of the program and handles the actual kernel invocation.

At initialisation, the program creates one POSIX thread for each requested computing device in order to hold the corresponding CUDA context. A sub-domain of the global lattice is assigned to each device. As for single-GPU implementation, synchronisation within the sub-domains is achieved by launching a kernel for each time step. Global synchronisation uses standard POSIX barriers. Inter-GPU communication is performed using page-locked CPU memory and zero-copy memory transactions.

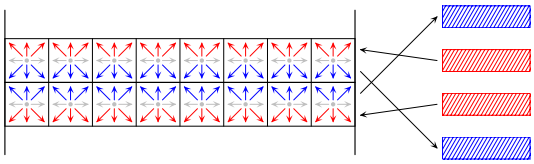
As for global memory accesses, zero-copy transactions require coalescing to achieve optimal performance. This implies that the interfaces between sub-domains should be parallel to the direction associated with the minor dimension of the particle distribution array. For the sake of simplicity, we chose to split the lattice in rectangular cuboids along the direction corresponding to the major dimension. Figure 7 outlines the inter-GPU communication scheme; incoming populations are drawn in red, outgoing populations are drawn in blue.

Each interface between sub-domains is associated to four buffers: two for incoming populations and two for outgoing. Pointers are switched after each time step. Maximal parallelisation efficiency requires perfect overlapping of computations and communication. The zero-copy feature enables such overlapping, but the overlapping ratio depends on the scheduling of memory transactions at warp level. The execution grid set-up is therefore an important optimisation target.

Another problem arise when considering the configuration of the execution grid, since it may only have up to two dimensions. The simple solution of



(a) First state



(b) Second state

Figure 7: Inter-GPU communication scheme

using one-dimensional blocks and a two-dimensional grid to span the three spatial dimensions does not apply to large lattices. On GT200 hardware, for instance, the resource requirements of an LBM kernel are likely to forbid the use of blocks greater than 256.

We therefore chose to use a two-dimensional grid of size $(\ell_x \times \ell_y \times \ell_z/2^m) \times (2^{m-n})$ with one-dimensional blocks of size 2^n; ℓ_x, ℓ_y, ℓ_z being the dimensions of the lattice, m and n being free parameters. Retrieval of coordinates is done using the following code:

```
w = blockIdx.x<<m | blockIdx.y<<n
   | threadIdx.x;
x = w % lX;
y = (w/lX) % lY;
z = w/(lX*lY);
```

The optimal values for m and n are $m = 15$ and $n = 7$, which were determined empirically. To validate our code, we implemented the well-known lid-driven cubic cavity test case in which five walls have null velocity boundary conditions and the top lid has imposed constant velocity. In order to study the scalability of the program, we chose to run performance tests on a 192^3 lattice which may be handled by one single GPU or split in two, three, four, or six identical sub-domains as well. In addition, we also ran performance tests on a 384^3 lattice using three, four, and six computing devices. Figures 8 and 9 show the obtained performance in million lattice node updates per second (MLUPS) for single precision with Tesla C1060 computing devices on a Tyan B7015 server. It is worth mentioning that maximum performance is of the same order of magnitude than the one obtained with optimised double precision code on supercomputers (see [9], for instance).

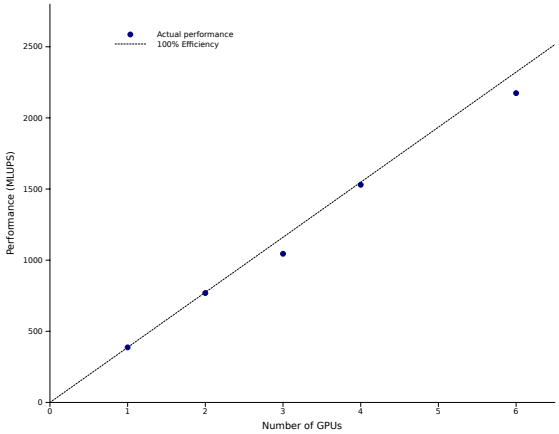


Figure 8: Performance on a 192^3 lattice

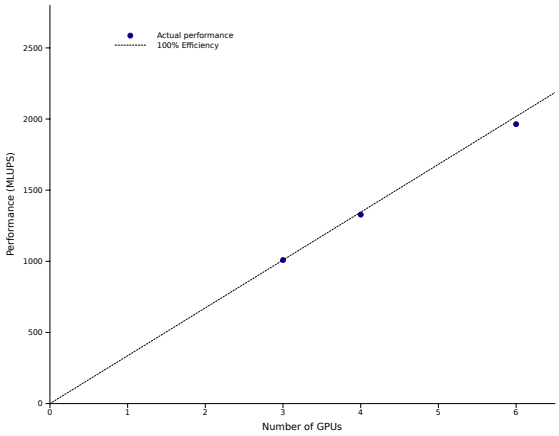


Figure 9: Performance on a 384^3 lattice

Scalability is excellent with no less than 90% parallelisation efficiency. Table 1 displays the required throughput for incoming and outgoing data at 100% efficiency on the 192^3 lattice. With the Tyan S7015 motherboard of our server, the `bandwidthTest` program that comes with the CUDA development kit gives 2.78 GB/s host to device and 1.80 GB/s device to host maximum sustained throughput. The data exchange being symmetric and the PCI-E interface being full-duplex, we may use the lower of these values as a rough estimate of the available through-

put for one PCI-E 16× slot. A comprehensive study of communications between computing devices and main memory is beyond the scope of this work and shall be given in future reference.

Table 1 shows that even with six GPUs, i.e. five sub-domain interfaces, the required throughput is comparable to the one achievable with a single PCI-E 16× slot, therefore data exchange is not likely to overflow the capacity of the PCI-E links. Furthermore, we see that the execution grid configuration we propose enables very satisfactory communication/computation overlapping.

According to the `bandwidthTest` program, the GPU to device memory maximum sustained throughput is 73.3 GB/s for the Tesla C1060. Performance in single precision using one GPU is 387 MLUPS on the 192^3 lattice which correspond to a data throughput of 80.4% of the maximum. We may therefore conclude that our single precision solver is memory bound and that performance is nearly optimal.

Performance for the double precision version of our solver on the 192^3 lattice ranges from 117 MLUPS using one GPU to 683 MLUPS using six, with similar scalability than for the single precision version. Considering one GPU, the corresponding data throughput is only 48.5% of the maximum, which implies that the double precision version is not memory bound but computation bound.

6 Numerical Issues

Although the lid-driven cubic cavity test case is well documented at low Reynolds numbers, there are–to the best of our knowledge–very few references for $\mathrm{Re} \geq 12{,}000$ [11, 2]. Using the six available Tesla C1060 cards, our solver is able to handle cubic lattices containing as much as 480^3 nodes for single precision D3Q19 and 384^3 nodes for double precision D3Q19. We could therefore run simulations for Reynolds numbers up to 30,000 without facing numerical instabilities.

According to nVidia, peak performance for the Tesla C1060 is 933 GFlops in single precision and 78 GFlops in double precision. As a matter of fact, GT200 GPUs are usually less efficient with double precision computations than with single precision. In our case, the performance ratio is about 3.2 to one. Nevertheless, the GT200 implementation of single precision is not fully IEEE 754 compliant. When first running our solver at $\mathrm{Re} = 30{,}000$ in single precision, we could see some numerical issue arise: the flow loses symmetry at a very early stage of simulation. Further investigation showed us that the average deviation from initial density decreases at a constant pace instead of fluctuating around zero.

To evaluate the impact of machine accuracy on our simulations, we experimented with three levels of precision: single precision (SP), mixed precision (MP), i.e. double precision computations with single precision storage, and double precision (DP). It has been reported that using $\delta\rho = \rho - \rho_0$ instead of ρ in moment space improves accuracy [4]. Thus we also experimented this approach for the three levels of precision: SP*, MP*, DP*.

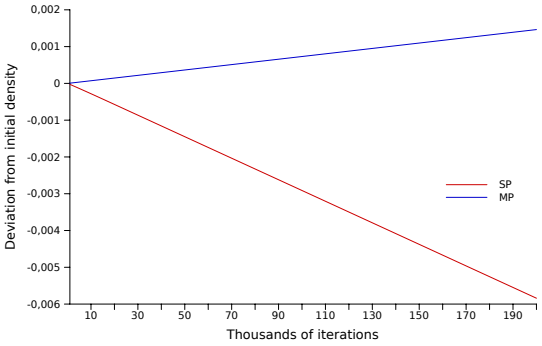


Figure 10: Deviation for SP and MP

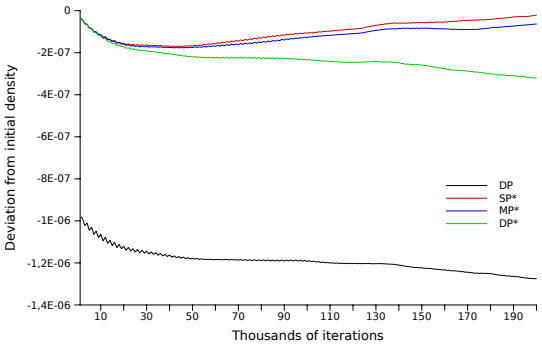


Figure 11: Deviation for DP, SP*, MP*, DP*

Figures 10 and 11 show the average deviation from initial density when running a simulation at $\mathrm{Re} = 30{,}000$ on a 384^3 lattice for the six levels of precision. We can see that, regarding conservation of mass, mixed precision does not provide significant

Number of GPUs	1	2	3	4	6
Kernel duration (ms)	18.29	9.14	6.10	4.57	3.05
Data amount (MB)	0.00	1.47	2.95	4.42	7.37
Required throughput (GB/s)	0.00	0.16	0.48	0.97	2.42

Table 1: Required throughput for data exchange at 100% efficiency

improvement over single precision, and that SP*, MP*, and DP* perform better than DP by an order of magnitude. Furthermore, we may conclude that SP and MP should not be used when simulating unsteady flows.

In order to study the loss of symmetry from a quantitative standpoint, we used the following estimator:

$$\mathscr{L} = \max_{\boldsymbol{x}} \|\boldsymbol{u}(\boldsymbol{x},t) - \bar{\boldsymbol{u}}(\bar{\boldsymbol{x}},t)\| \tag{17}$$

where $\boldsymbol{x}$ and $\bar{\boldsymbol{x}}$, and $\boldsymbol{u}$ and $\bar{\boldsymbol{u}}$ are symmetric with respect of the symmetry plane of the cavity. Figure 12 shows the evolution of $\mathscr{L}$ for the different precision levels running the same simulation than for mass conservation, i.e. $\mathrm{Re} = 30{,}000$ on a 384^3 lattice. One can deduce from this diagram that the accumulation of round-off errors is the cause for the loss of symmetry. Past a certain threshold, due to the turbulent nature of the flow, the numerical perturbations are steeply amplified.

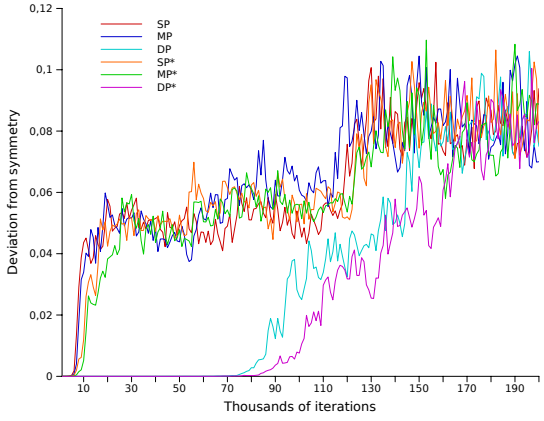


Figure 12: Evolution of $\mathscr{L}$ for the six precision levels

Figure 13 displays the evolution of $\mathscr{L}$ at different Reynolds numbers for the DP* precision level on a 384^3 lattice. This diagram shows that the more turbulent the flow pattern is, the sooner the symmetry breaking occurs, which corroborates the former point of view.

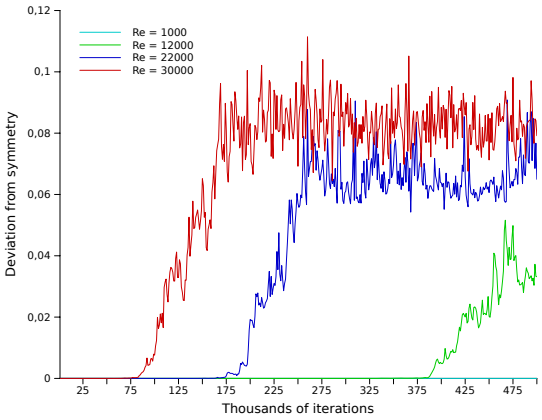


Figure 13: Evolution of $\mathscr{L}$ at different Reynolds numbers

From a performance standpoint, SP, MP, and DP behave similarly than their stared counterparts, since the difference in implementation only affects the initialisation section. Using $\delta\rho$ instead of ρ is therefore an advisable improvement. Mixed precision has almost identical performance than double precision, e.g. 602 MLUPS using six GPUs on a 192^3 lattice. In this case, the gain in accuracy is not worth the performance trade-off.

7 Conclusion

In this contribution, we describe a multi-GPU implementation of the LBM, based on rather simple technical choices, i.e. POSIX threads and basic domain tilling. Nevertheless, performance is nearly optimal, rivalling the one of supercomputer or large cluster implementations. Further investigations are needed to improve understanding of the inter-GPU communication potential. Moreover, work has to be done to design some execution grid layout and domain decomposition compatible with MPI parallelisation.

Our multi-GPU LBM solvers enables the use of

large lattices, thus allowing direct numerical simulation of unsteady flows. We describe some numerical issues that arise at high Reynolds numbers and investigate the impact of different precision levels both on accuracy and performance.

The TheLMA framework we designed to implement our flow solver is meant to improve code reusability. We are currently developing several applications based on TheLMA, including a hybrid thermal solver and a LES solver. In the near future, we plan to extend this framework to generic multi-GPU parallelisation.

References

[1] E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, and S. Tomov. Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects. In *Journal of Physics: Conference Series*, volume 180, pages 2037–2041. IOP Publishing, 2009.

[2] R. Bouffanais, M.O. Deville, and E. Leriche. Large-eddy simulation of the flow in a lid-driven cubical cavity. *Physics of Fluids*, 19(5):5108–5108, 2007.

[3] D. d'Humières. Generalized lattice-Boltzmann equations. In *Proceedings of the 18th International Symposium on Rarefied Gas Dynamics*, pages 450–458. University of British Columbia, Vancouver, Canada, 1994.

[4] D. d'Humières, I. Ginzburg, M. Krafczyk, P. Lallemand, and L.S. Luo. Multiple-relaxation-time lattice Boltzmann models in three dimensions. *Philosophical Transactions of the Royal Society A*, 360:437–451, 2002.

[5] J. Dongarra, G. Peterson, S. Tomov, J. Allred, V. Natoli, and D. Richie. Exploring new architectures in accelerating CFD for Air Force applications. In *DoD HPCMP Users Group Conference*, pages 472–478. IEEE, 2008.

[6] U. Frisch, B. Hasslacher, and Y. Pomeau. Lattice-Gas Automata for the Navier-Stokes Equation. *Physical Review Letters*, 56(14):1505–1508, 1986.

[7] F. Kuznik, C. Obrecht, G. Rusaouën, and J.-J. Roux. LBM Based Flow Simulation Using GPU Computing Processor. *Computers and Mathematics with Applications*, 59(7):2380–2392, 2010.

[8] P. Lallemand and L.S. Luo. Theory of the lattice Boltzmann method: Dispersion, dissipation, isotropy, Galilean invariance, and stability. *Physical Review E*, 61(6):6546–6562, 2000.

[9] J. Latt. Palabos Benchmarks (3D Lid-driven Cavity). Available on www.lbmethod.org/plb_wiki:benchmark:cavity_n1000, 2010.

[10] J. Latt, O. Malaspinas, and D. Lagrava. Parallel Lattice Boltzmann Solver. Available on www.lbmethod.org/palabos, 2010.

[11] E. Leriche and S. Gavrilakis. Direct numerical simulation of the flow in a lid-driven cubical cavity. *Physics of Fluids*, 1***2(6):1363–1376, 2000.

[12] G. R. McNamara and G. Zanetti. Use of the Boltzmann Equation to Simulate Lattice-Gas Automata. *Physical Review Letters*, 61(20):2332–2335, 1988.

[13] P. Micikevicius. 3D finite difference computation on GPUs using CUDA. In *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, pages 79–84. ACM, 2009.

[14] G.E. Moore. Cramming more components onto integrated circuits. *Electronics Magazine*, 38(8):114–117, 1965.

[15] NVIDIA. *Compute Unified Device Architecture Programming Guide version 3.0*, 2010.

[16] C. Obrecht, F. Kuznik, B. Tourancheau, and J.-J. Roux. Thermal LBM on Many-core Architectures. Available on www.thelma-project.info, 2010.

[17] C. Obrecht, F. Kuznik, B. Tourancheau, and J.-J. Roux. A new approach to the lattice Boltzmann method for graphics processing units. *Computers and Mathematics with Applications*, 61(12):3628–3638, 2011.

[18] C. Obrecht, F. Kuznik, B. Tourancheau, and J.-J. Roux. Global Memory Access Modelling for Efficient Implementation of the Lattice Boltzmann Method on Graphics Processing Units. In *Lecture Notes in Computer Science 6449, High Performance Computing for Computational Science, VECPAR 2010 Revised Selected Papers*, pages 151–161. Springer, 2011.

[19] Y. H. Qian, D. d'Humières, and P. Lallemand. Lattice BGK models for Navier-Stokes equation. *EPL (Europhysics Letters)*, 17(6):479–484, 1992.

[20] E. Riegel, T. Indinger, and N.A. Adams. Implementation of a Lattice–Boltzmann method for numerical fluid mechanics using the nVIDIA CUDA technology. *Computer Science – Research and Development*, 23(3):241–247, 2009.

[21] J. Tölke. Implementation of a Lattice Boltzmann kernel using the Compute Unified Device Architecture developed by nVIDIA. *Computing and Visualization in Science*, 13(1):29–39, 2010.

[22] J. Tölke and M. Krafczyk. TeraFLOP computing on a desktop PC with GPUs for 3D CFD. *International Journal of Computational Fluid Dynamics*, 22(7):443–456, 2008.

Article E

The TheLMA project: Multi-GPU Implementation of the Lattice Boltzmann Method

International Journal of High Performance Computing Applications,
25(3):295–303, August 2011

Accepted April 26, 2011

Abstract

In this paper, we describe the implementation of a multi-GPU fluid flow solver based on the lattice Boltzmann method (LBM). The LBM is a novel approach in computational fluid dynamics, with numerous interesting features from a computational, numerical, and physical standpoint. Our program is based on CUDA and uses POSIX threads to manage multiple computation devices. Using recently released hardware, our solver may therefore run eight GPUs in parallel, which allows to perform simulations at a rather large scale. Performance and scalability are excellent, the speedup over sequential implementations being at least of two orders of magnitude. In addition, we discuss tiling and communication issues for present and forthcoming implementations.

Keywords: GPU Computing, CUDA, Computational Fluid Dynamics, Lattice Boltzmann Method.

1 Introduction

First introduced by McNamara and Zanetti in 1988 [10], the lattice Boltzmann method (LBM) is an increasingly popular alternative to classic CFD methods. From a numerical and physical standpoint, the LBM possesses numerous convenient features, such as explicitness or ability to deal with complex geometries. Moreover, this novel approach is especially well-fitted for high performance CFD applications because of its inherent parallelism.

Since the advent of the CUDA technology, the use of GPUs in scientific computing has become more and more widespread [5, 1]. Several successful single-GPU implementations of the LBM using CUDA were reported during the past years [19, 8]. Nevertheless, these works are of moderate practical impact since memory in existing computing devices is too scarce for large scale simulations. Multi-GPU implementations of the LBM are still at an early stage of development and reported performance is below what is expected from such hardware [16].

Recently released motherboards are able to handle up to eight computing devices. We therefore chose to develop a POSIX thread based multi-GPU LBM solver, as a first step towards a distributed version. The structure of the paper is as follows. We first briefly present the LBM from an algorithmic perspective, and summarize the principles of LBM implementation using CUDA. Then, we describe the implementation of our solver and discuss performance, scalability, and tiling issues. To conclude, we study inter-GPU communication and propose a performance model.

2 LBM Flow Solvers

Although considered at first as an extension to the lattice gas automata method [7], the LBM is nowadays more commonly interpreted as a discrete numerical procedure to solve the Boltzmann transport equation:

$$\partial_t f + \boldsymbol{\xi} \cdot \nabla_{\boldsymbol{x}} f + \frac{\boldsymbol{F}}{m} \cdot \nabla_{\boldsymbol{\xi}} f = \Omega(f) \qquad (1)$$

where $f(\boldsymbol{x}, \boldsymbol{\xi}, t)$ describes the evolution in time t of the distribution of one particle in phase space (i.e.

position $\boldsymbol{x}$ and particle velocity $\boldsymbol{\xi}$), $\boldsymbol{F}$ is the external force field, m the mass of the particle, and Ω is the collision operator. The distribution function f is linked to the density ρ and the velocity $\boldsymbol{u}$ of the fluid by:

For LBM, Eq. 1 is usually discretized on a uniform orthogonal grid of mesh size δx with constant time steps δt. The particle velocity space is likewise replaced by a set of $N+1$ velocities $\boldsymbol{\xi}_i$ with $\boldsymbol{\xi}_0 = \mathbf{0}$, chosen such as vectors $\delta t\, \boldsymbol{\xi}_i$ link any given node to some of its neighbors.[1] Figure 1 shows the D3Q19 stencil we chose for our implementation. This stencil is a good trade-off between geometric isotropy and complexity of the numerical scheme for the three-dimensional case.

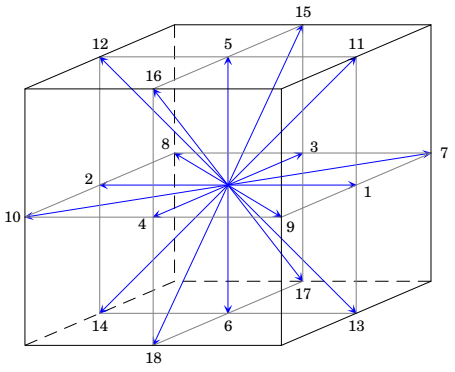


Figure 1: D3Q19 stencil

The discrete counterpart of the distribution function f is a set of $N+1$ particle population functions f_i corresponding to the finite set of particle velocities. In a null external force field, Eq. 1 writes:

$$\left|f_i(\boldsymbol{x}+\delta t\,\boldsymbol{\xi}_i, t+\delta t)\right\rangle - \left|f_i(\boldsymbol{x},t)\right\rangle = \Omega\Big(\left|f_i(\boldsymbol{x},t)\right\rangle\Big) \quad (2)$$

where $\left|a_i\right\rangle$ denotes the transpose of $(a_0,\ldots,a_N)$. Density and velocity of the fluid are given by:

$$\rho = \sum_{i=0}^{N} f_i \quad (3)$$

$$\rho\boldsymbol{u} = \sum_{i=0}^{N} f_i \boldsymbol{\xi}_i \quad (4)$$

[1] Commonly, only nearest neighbors are linked.

The description of possible collision operators is beyond the scope of this contribution. Nonetheless, it should be noted from a mathematical perspective that the collision operators are usually linear in f_i and quadratic in ρ and $\boldsymbol{u}$, as for the multiple-relaxation-time model we chose to implement [3, 4]. Equation 2 naturally breaks in two elementary steps:

$$\left|\tilde{f}_i(\boldsymbol{x},t)\right\rangle = \left|f_i(\boldsymbol{x},t)\right\rangle + \Omega\Big(\left|f_i(\boldsymbol{x},t)\right\rangle\Big) \quad (5)$$

$$\left|f_i(\boldsymbol{x}+\delta t\,\boldsymbol{\xi}_i, t+\delta t)\right\rangle = \left|\tilde{f}_i(\boldsymbol{x},t)\right\rangle \quad (6)$$

Equation 5 describes the *collision* step in which an updated particle distribution is computed, and Eq. 6 describes the *propagation* step in which the updated particle populations are transferred to the neighboring nodes. From an algorithmic standpoint, a LBM flow solvers is therefore outlined by the following pseudo-code:

1. **for each** time step t **do**
2. **for each** lattice node $\boldsymbol{x}$ **do**
3. read velocity distribution $f_i(\boldsymbol{x}, t)$
4. **if** node $\boldsymbol{x}$ is on boundaries **then**
5. apply boundary conditions
6. **end if**
7. compute updated distribution $\tilde{f}_i(\boldsymbol{x}, t)$
8. propagate to neighboring nodes $\boldsymbol{x} + \delta t\, \boldsymbol{\xi}_i$
9. **end for**
10. **end for**

To conclude, it should be noted that parallelization of the LBM is rather straightforward, ensuring synchronization between neighbors at each time step being the only constraint.

3 CUDA Implementation Guidelines

When using CUDA computing devices for LBM simulations, the particle distribution has to be stored in device memory, for obvious performance reason. The simplest strategy to ensure global synchronization is to launch one kernel at each time step. Moreover, due to the lack of control over memory transaction scheduling, two consecutive particle distributions have to be retained at once. On a Tesla C1060

computing card, with a D3Q19 stencil in single precision, the maximum number of handleable nodes is therefore about 2.83×10^7.

The LBM being a data-parallel procedure, CUDA implementations usually assign one thread per node. This option allows to take advantage of the massive parallelism of the targeted architecture. According to [11], the execution grid has to obey the following constraints and limitations:

- The grid may only have one or two dimensions.
- Blocks may have up to three dimensions but the size of the blocks is limited by the amount of available registers per streaming multiprocessor (SM).
- The size of the blocks should be a multiple of the warp size, i.e. 32 on existing hardware, the warp being the minimum execution unit.

A simple and efficient execution grid layout consists in a two-dimensional grid of one-dimensional blocks. For cubic cavities, taking the previously mentioned device memory limitation into account, this solution leads to blocks of size up to 256. Thus, there are at least 64 available registers per node, which proves to be sufficient for most three-dimensional collision models.

As for CPU implementation of the LBM, the particle distribution is stored in a multi-dimensional array. Global memory transactions are issued by half-warps [11]. Hence, in order to enable coalesced data transfer when using the former execution grid layout, the minor dimension of the array should be the spatial dimension associated to the blocks. The disposition of the remaining dimensions, including the velocity index may be tuned so as to minimize TLB misses [15]. However, this data layout does not ensure optimal global memory transactions. Propagation of particle populations along the blocks' direction yields one-unit shifts in addresses and therefore causes misaligned memory accesses. Fig. 2 sketches this issue. The red frame represent a given block, the blue arrows to updated particle populations to be propagated. For the sake of clarity, the size of the block is limited to four nodes, and the particle populations are represented as a two-dimensional stencil.

For CUDA devices of compute capability[2] 1.0 or 1.1, misaligned memory transactions may have dra-

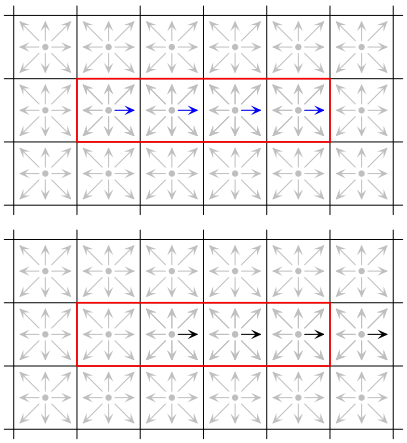


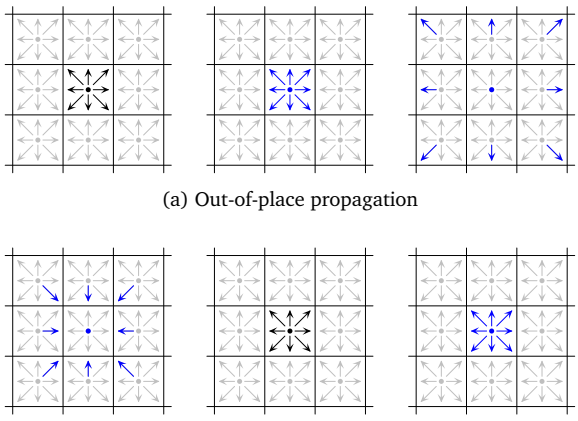
Figure 2: Misalignment issue

matic impact on performance, since sixteen accesses are issued in lieu of a single one. Performing propagation within the blocks using the shared memory as described in [18] is an efficient way to avoid this issue. For compute capability 1.2 or 1.3, however, a misaligned memory access is carried out in as few 128 B, 64 B, or 32 B aligned transactions as possible. An alternative approach for this kind of hardware consists in using in-place propagation instead of out-of-place propagation.

The two propagation schemes are outlined in Fig. 3. The diagrams are drawn from the standpoint of the central node. Again, the particle populations are represented as a two-dimensional stencil. With out-of-place propagation (Fig. 3a), the collision step is carried out *before* the propagation step, which is performed by pushing the updated populations to the neighboring nodes. With in-place propagation (Fig. 3b), the collision step is carried out *after* the propagation step, which is performed by pulling the updated populations from the neighboring nodes.

With the GT200, misaligned reads are far less expensive than misaligned writes [14]. Thus, the in-place propagation approach is only slightly less efficient than the shared memory method for simulations in three dimensions [13]. In-place propagation is simpler to implement since only global memory transactions may be used, whereas the shared mem-

[2]The compute capability number identifies the level of features supported by CUDA hardware. The G80, first CUDA enabled GPU, has capability 1.0 . The GT200, that powers the Tesla C1060, has capability 1.3 . The Fermi, most recent architecture, has capability 2.0 .



(a) Out-of-place propagation

(b) In-place propagation

Figure 3: Propagation schemes

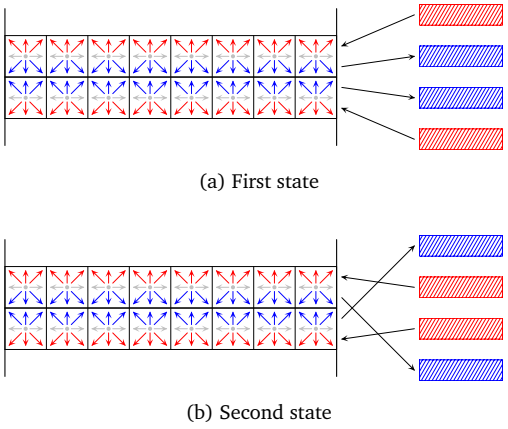
ory method needs specific handling for the propagation along the block's direction. Furthermore, in-place propagation exerts less pressure on hardware since the use of shared memory is not required.

4 Multi-GPU Implementation of the LBM

As usual in parallel computing, optimal performance for multi-GPU applications requires efficient overlapping of computation and communication. In our implementation of the LBM, we used the zero-copy feature provided by the CUDA technology which allows GPUs to directly access locked CPU memory pages. Using CUDA streams is another possible way to perform inter-GPU communication. Yet, the zero-copy feature proves to be the most convenient solution since communication is taken care of in GPU code instead of CPU code.

As for global memory accesses, zero-copy transactions should be coalescent in order to preserve performance. To enable coalescence, the blocks have to be parallel to the interfaces of the sub-domains. For the sake of simplicity, we decided to use one-dimensional blocks for our multi-GPU implementation as in our single-GPU implementation. Because of the targeted hardware, our solver has to handle at most eight sub-domains. We therefore chose to split the cavity in balanced rectangular cuboids along one direction orthogonal to the blocks.

Each sub-domain is handled by a POSIX thread which is responsible for launching the computation kernel on the associated GPU. Synchronization within the whole domain is obtained through a double barrier: one CUDA barrier for synchronization at sub-domain level and one POSIX thread barrier. Fig. 4 outlines the data flows between sub-domains. In-coming populations are drawn in red and out-going populations in blue. For a given interface, at each time step, in-coming populations are read from a couple of buffers (one for each sub-domain) and out-going are written to an alternate couple of buffers. The buffers exchange role at the next time step, which is carried out by simply switching pointers.



(a) First state

(b) Second state

Figure 4: Inter-GPU communication scheme

The splitting direction is selected such as to minimize the amount of data to exchange. Nevertheless, for large cavities, the dimension in the direction of the blocks may exceed the maximum block size. Thus, the simple grid layout described in section 3 may not be used in general. We propose instead, to use blocks of size 2^n with a two-dimensional grid of size:

$$(\ell_x \times \ell_y \times \ell_z \times 2^{-m}) \times (2^{m-n}),$$

where ℓ_x, ℓ_y, and ℓ_z are the dimensions of the sub-domain, n and m are adjustable parameters. The retrieval of the coordinates is done using a code equivalent to the following:

```
w = blockIdx.x<<m | blockIdx.y<<n
    | threadIdx.x;
x = w % lX;
y = (w/lX) % lY;
z = w/(lX*lY);
```

The proposed grid layout leads to shuffle the blocks' schedule which tends to reduce partition camping [17] and allows efficient communication-computation overlapping as shown in section 5. The optimal values for n and m, which were determined empirically, are $n = 7$ and $m = 15$.

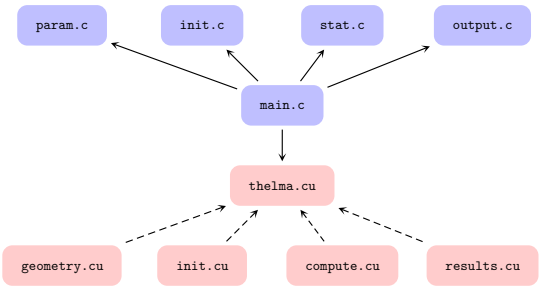


Figure 5: The TheLMA framework

From a software engineering perspective, the CUDA technology is a great improvement over the early days of GPGPU [6]. Yet, there are still some important limitations compared to usual software development. The inability of the CUDA compilation tool-chain to link several GPU binaries together for instance, makes difficult to follow an incremental, library oriented approach. To improve code reusability, we therefore developed the TheLMA framework [12]. TheLMA stands for *Thermal LBM on Many-core Architectures*, thermal flow simulations being our main topic of interest. Figure 5 outlines the structure of the framework.

The TheLMA framework mainly consists of two parts. The first part is a set of C source files which provides various utility functions such as command line parsing, POSIX threads setup, simulation statistics, and data output in various formats. The second part is a set of CUDA source files which are included in the `thelma.cu` file at compile time. The later is basically a container providing some general macro definitions.

The initialization module mainly creates CUDA contexts in order to assign a GPU to each POSIX thread. The geometry module is responsible for setting up the boundary conditions and any obstacle that may lay inside the cavity. Permitted velocity directions for each node, as well as specific boundary conditions, are encoded using bit fields. The computation module contains the core kernel of the simulation which performs both collision and propagation. Last, the result module retrieves the macroscopic variables of the fluid.

The source organization we describe is meant to ease subsequent code changing. For instance, the implementation of a different collision model or propagation scheme would mainly require the modification of the computation module. Likewise, the setting up of a given simulation configuration principally involves changes in the geometry module.

5 Performance and Scalability

To validate our code, we implemented the lid-driven cubic cavity test case in which a constant velocity is imposed at the top lid, whereas the other walls have a null velocity boundary condition. We chose to assign the x direction to the blocks and to split the cavity along the z direction. Performance for LBM solvers is usually given in MLUPS (Million Lattice node Updates Per Second). We evaluated performance in single precision for a 192^3 and a 256^3 lattice on a Tyan B7015 server with 8 Tesla C1060 computing devices running 10^5 time steps. The former lattice size, being a multiple of 24, allows to run the solver using 2, 3, 4, 6, or 8 GPUs with balanced sub-domains. Figure 6 shows our measurement for both lattices.

The maximum achieved performance is about 2,150 MLUPS obtained on the 192^3 lattice using 6 GPUs. To set a comparison, this maximum is comparable to the performance obtained with the widely used Palabos code in double precision for a $1{,}001^3$ cavity on a Blue Gene/P computer using 4,096 cores [9]. Scalability is excellent in all cases but one, with no less than 91% parallelization efficiency. Tables 1 and 2 give the inter-GPU data exchange throughput required to achieve full overlapping.[3]

With the Tyan S7015 motherboard of our server, the `bandwidthTest` program that comes with the CUDA development kit gives 5.72 GB/s host to device and 3.44 GB/s device to host maximum cumulative throughput. These values are obtained using the copy functions provided by the CUDA API with pinned memory and not zero-copy transactions as in our program. The communications being symmetric

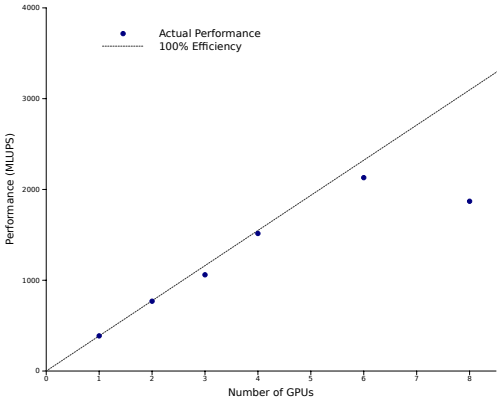
[3] Following IEEE 1541-2002, we use MB for 10^6 bytes and GB for 10^9 bytes.

Number of GPUs	1	2	3	4	6	8
Kernel duration (ms)	18.29	9.14	6.10	4.57	3.05	2.29
Data amount (MB)	0.00	1.47	2.95	4.42	7.37	10.32
Required throughput (GB/s)	0.00	0.16	0.48	0.97	2.42	4.51

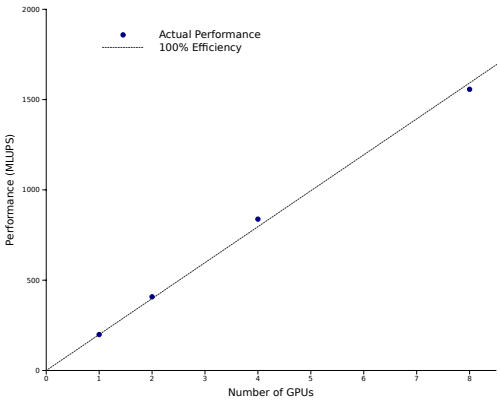
Table 1: Required throughput at 100% efficiency on a 192^3 lattice

Number of GPUs	1	2	4	8
Kernel duration (ms)	84.31	42.15	21.08	10.54
Data amount (MB)	0	2.62	7.86	18.35
Required throughput (GB/s)	0	0.06	0.37	1.74

Table 2: Required throughput at 100% efficiency on a 256^3 lattice



(a) Performance on a 192^3 lattice



(b) Performance on a 256^3 lattice

Figure 6: Performance and scalability on a cubic lattice

in our case and the PCI-E interface being full-duplex, we may use the lower of these values, i.e. 3.44 GB/s, as an estimate for sustained data exchange between GPUs and main memory. Yet, we see that, except in the case of 8 GPUs on a 192^3 lattice, the required throughput is definitely less than this estimate and is not likely to overflow the capacity of the PCI-E links. We can furthermore conclude that our data access pattern generally enables excellent communication-computation overlapping.

Considering the unfavorable case, we deduce that the number of interfaces with respect to the size of the cavity is too large to take advantage of surface to volume effects. This naturally leads to the question of a less simplistic tiling of the cavity than the one we adopted. Yet, dividing a cubic cavity in eight identical cubic tiles would lead to only three interfaces, but would yield non-coalesced, i.e. serialized, zero-copy transactions for one of the interfaces. It is possible with simple code modifications, like splitting the cavity along the x direction instead of the z direction, to evaluate the impact of serialized inter-GPU communication. Figure 7 displays the performance of our code on a 192^3 lattice after such a transformation.

We can see the overwhelming impact of serialized communication on performance, which is obviously communication bound. We may therefore conclude that gaining flexibility in tiling would most likely require to adopt a more elaborate execution grid layout.

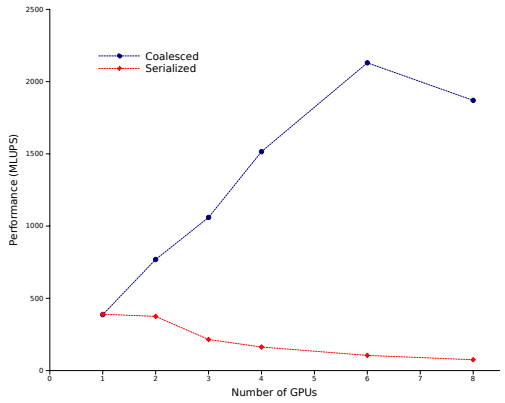


Figure 7: Performance for coalesced and serialized communication

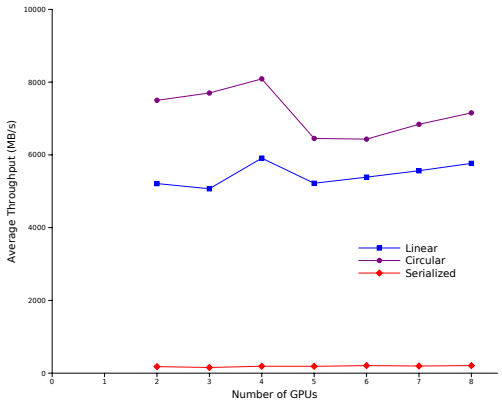


Figure 8: Inter-GPU communication throughput

6 Inter-GPU Communication

In order to get better insight into inter-GPU communication, we implemented a benchmark program based on a stripped-down version of our LBM solver. The purpose of this software is to evaluate the achievable sustained throughput when using zero-copy transactions over several GPUs. Data to exchange is formed of k two-dimensional $L \times L$ arrays of 32-bit words. The communication graph may be either circular or linear. Whereas linear graphs correspond to our solver, circular graph are useful to simulate balanced communication which is likely to occur with multi-node implementations.

The communication graph is specified by an ordered list of involved devices. In our tests, the S7015 being a two-socket motherboard, we used this parameter to obtain optimal balancing between the two northbridges. In addition, the data arrays may optionally be transposed at each access which causes serialization of the zero-copy transactions. Figure 8 shows the obtained throughput averaged over 50,000 iterations for circular graphs, linear graphs, and linear graphs with transposition. In order for our benchmark program to exchange an amount of data comparable to our solver, the values of k and L were set to $k = 5$, which is the number of exchanged particle populations when using the D3Q19 stencil, and $L = 192$, which is equivalent to the dimension of the cavity in our first performance test. Yet, extensive tests have shown that these parameters are of negligible impact over the measurements.

The variations of throughput for circular and linear graphs are similar, which is natural insofar the configurations with respect of the number of devices are identical. The throughput in the linear case is less than in the circular case since the links at the edges are only used at half capacity. The values obtained for linear communication graphs are coherent with the conclusion drawn in section 5. The throughput for the serialized version is more than one order of magnitude below the coalesced versions, i.e. about 200 MB/s. Using the measurements, we may estimate the performance of the solver under the assumption of communication-boundedness. Performance P in MLUPS obeys:

$$P = \ell^3 \times \frac{T_i}{2(i-1)D \times \ell^2} = \frac{T_i \times \ell}{2(i-1)D} \qquad (7)$$

where i is the number of devices, T_i is the corresponding throughput (in MB/s), D is the amount of out-going and in-coming data by node (in bytes), and ℓ is the dimension of the cavity (in nodes). In the case of 8 GPUs on a 192^3 lattice, the estimated performance is 1,977 MLUPS instead of 1,870 MLUPS, i.e. a 6% relative error. For the serialized communication version, Fig. 9 displays the comparison between the corresponding estimation and the actual performance.

The estimation is in good accordance with the measured values, the relative error being at most 15%, which corroborates our assumption.

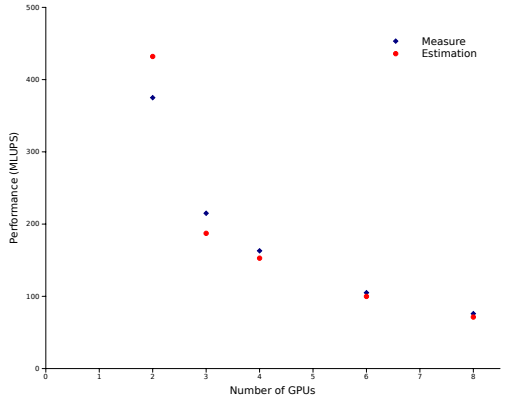


Figure 9: Performance model for serialized communication

7 Conclusion

In this contribution, we describe a multi-GPU implementation of the LBM capable of handling up to eight CUDA enabled computing devices. Performance is comparable with efficient parallel implementations on up-to-date homogeneous supercomputers and is at least two orders of magnitude higher than with optimized sequential code. We propose an execution grid layout providing excellent computation-communication overlapping and an efficient inter-GPU communication scheme. Though simple, this scheme yields excellent scalability with nearly optimal parallelization efficiency in most cases.

Our solver is implemented over the TheLMA framework, which aims at improving code reusability. A thermal LBM solver also based on TheLMA is currently under development. In the present version, management of multiple CUDA contexts is based on POSIX threads. Yet, we plan to extend our framework to more generic parallelization interfaces (e.g. MPI) in order to make possible distributed implementations. This extension requires in conjunction more elaborate data exchange procedures so as to gain flexibility in domain decomposition.

In addition, we study inter-GPU communication using a specific benchmark program. The obtained results allowed us to express a rather accurate performance model for the cases where our application is communication bound. This tool reveals moreover that data throughput depends to a certain extent on hardware locality. An extended version, using for instance the Portable Hardware Locality (hwloc) API [2], would help performing further investigation and could be a first step for adding dynamic autotuning in our framework.

References

[1] E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, and S. Tomov. Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects. In *Journal of Physics: Conference Series*, volume 180, pages 2037–2041. IOP Publishing, 2009.

[2] F. Broquedis, J. Clet-Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault, and R. Namyst. hwloc: a generic framework for managing hardware affinities in HPC applications. In *18th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP), 2010*, pages 180–186. IEEE, 2010.

[3] D. d'Humières. Generalized lattice-Boltzmann equations. In *Proceedings of the 18th International Symposium on Rarefied Gas Dynamics*, pages 450–458. University of British Columbia, Vancouver, Canada, 1994.

[4] D. d'Humières, I. Ginzburg, M. Krafczyk, P. Lallemand, and L.S. Luo. Multiple-relaxation-time lattice Boltzmann models in three dimensions. *Philosophical Transactions of the Royal Society A*, 360:437–451, 2002.

[5] J. Dongarra, G. Peterson, S. Tomov, J. Allred, V. Natoli, and D. Richie. Exploring new architectures in accelerating CFD for Air Force applications. In *DoD HPCMP Users Group Conference*, pages 472–478. IEEE, 2008.

[6] Z. Fan, F. Qiu, A. Kaufman, and S. Yoakum-Stover. GPU cluster for high performance computing. In *Proceedings of the 2004 ACM/IEEE Conference on Supercomputing*, pages 47–58. IEEE, 2004.

[7] U. Frisch, B. Hasslacher, and Y. Pomeau. Lattice-Gas Automata for the Navier-Stokes Equation. *Physical Review Letters*, 56(14):1505–1508, 1986.

[8] F. Kuznik, C. Obrecht, G. Rusaouën, and J.-J. Roux. LBM Based Flow Simulation Using GPU Computing Processor. *Computers and Mathematics with Applications*, 59(7):2380–2392, 2010.

[9] J. Latt. Palabos Benchmarks (3D Lid-driven Cavity on Blue Gene/P). Available on www.lbmethod.org/plb_wiki:benchmark:cavity_n1000, 2010.

[10] G. R. McNamara and G. Zanetti. Use of the Boltzmann Equation to Simulate Lattice-Gas Automata. *Physical Review Letters*, 61(20):2332–2335, 1988.

[11] NVIDIA. *Compute Unified Device Architecture Programming Guide version 3.1.1*, 2010.

[12] C. Obrecht, F. Kuznik, B. Tourancheau, and J.-J. Roux. Thermal LBM on Many-core Architectures. Available on www.thelma-project.info, 2010.

[13] C. Obrecht, F. Kuznik, B. Tourancheau, and J.-J. Roux. A new approach to the lattice Boltzmann method for graphics processing units. *Computers and Mathematics with Applications*, 61(12):3628–3638, 2011.

[14] C. Obrecht, F. Kuznik, B. Tourancheau, and J.-J. Roux. Global Memory Access Modelling for Efficient Implementation of the Lattice Boltzmann Method on Graphics Processing Units. In *Lecture Notes in Computer Science 6449, High Performance Computing for Computational Science, VECPAR 2010 Revised Selected Papers*, pages 151–161. Springer, 2011.

[15] M.M. Papadopoulou, M. Sadooghi-Alvandi, and H. Wong. Micro-benchmarking the GT200 GPU. Technical report, University of Toronto, Canada, 2009.

[16] E. Riegel, T. Indinger, and N.A. Adams. Implementation of a Lattice–Boltzmann method for numerical fluid mechanics using the nVIDIA CUDA technology. *Computer Science – Research and Development*, 23(3):241–247, 2009.

[17] G. Ruetsch and P. Micikevicius. *Optimizing matrix transpose in CUDA*, NVIDIA, 2009.

[18] J. Tölke. Implementation of a Lattice Boltzmann kernel using the Compute Unified Device Architecture developed by nVIDIA. *Computing and Visualization in Science*, 13(1):29–39, 2010.

[19] J. Tölke and M. Krafczyk. TeraFLOP computing on a desktop PC with GPUs for 3D CFD. *International Journal of Computational Fluid Dynamics*, 22(7):443–456, 2008.

Article F

Towards Urban-Scale Flow Simulations using the Lattice Boltzmann Method

Proceedings of the Building Simulation 2011 Conference, pages 933–940. IBPSA, 2011

Accepted September 6, 2011

Abstract

The lattice Boltzmann method (LBM) is an innovative approach in computational fluid dynamics (CFD). Due to the underlying lattice structure, the LBM is inherently parallel and therefore well suited for high performance computing. Emerging many-core devices, such as graphic processing units (GPUs), nowadays allow to run very large scale simulations on rather inexpensive hardware. In this contribution, we present some simulation results obtained using our multi-GPU LBM solver. For validation purpose, we study the flow around a wall-mounted cube and show good agreement with previously published results. Furthermore, we discuss larger scale flow simulations involving nine cubes which demonstrate the practicability of CFD simulations in building aeraulics.

1 Introduction

Because of the computational cost of flow simulations, building aeraulics is generally taken into account using simplified models. However, this approach is not satisfactory in terms of accuracy when modelling energy efficient buildings. Recent advances, in both computational fluid dynamics (CFD) and high performance computing allow to consider the practical use of explicit flow simulations in building models.

In this contribution, we shall present simulation results obtained using the lattice Boltzmann method (LBM). Being based on a mesoscopic point of view, this novel CFD approach has numerous advantages over classic macroscopic methods such as the solving of the Navier-Stokes equations. Among other benefits, it is worth mentioning the high numerical stability, the ability to deal with complex geometries and the straightforwardness of various physical couplings.

Although parallel implementations of the LBM may be rather efficient, performing large scale simulations on mainstream architectures still requires the use of expensive clusters. The present simulations were carried out using several graphics processing units (GPUs) in parallel within a single server. Performance afforded by such hardware configuration is comparable to the one obtained using large clusters at a fairly lower cost.

The remaining of the paper is organised as follows. The first section is a summary of the LBM, presenting the specific model we retained together with the subgrid-scale model we added in order to enable simulations at high Reynolds number. Then, we give a short description of state-of-the-art GPU implementations of LBM solvers and of our multi-GPU LBM framework. In the third section, for validation purpose, we present the simulation of a fully developed flow over a wall-mounted cube in a flat channel. The simulation results are compared to experimental data. The last section reports the simulation of the flow over nine identical wall-mounted cubes at high Reynolds numbers.

2 Lattice Boltzmann method

2.1 Lattice Boltzmann equation

Although originating from the lattice gas automata theory [5], the lattice Boltzmann method is nowadays usually interpreted as a discrete version of the

Boltzmann transport equation [9]:

$$\partial_t f + \boldsymbol{\xi} \cdot \nabla_{\boldsymbol{x}} f + \frac{\boldsymbol{F}}{m} \cdot \nabla_{\boldsymbol{\xi}} f = \Omega(f), \tag{1}$$

where $f(\boldsymbol{x}, \boldsymbol{\xi}, t)$ describes the evolution in time of the distribution of one particle in phase space, $\boldsymbol{F}$ is the external force field, m the mass of the particle, and Ω the collision operator.

Discretisation occurs both in time, with constant time steps δt, and phase space, generally using a regular orthogonal grid of mesh size δx and a finite set of $N+1$ particle velocities $\boldsymbol{\xi}_\alpha$ with $\boldsymbol{\xi}_0 = \boldsymbol{0}$. The later is commonly a subset of the velocities linking a node to its nearest neighbours as the D3Q19 stencil we used for our simulations (see Fig. 1).

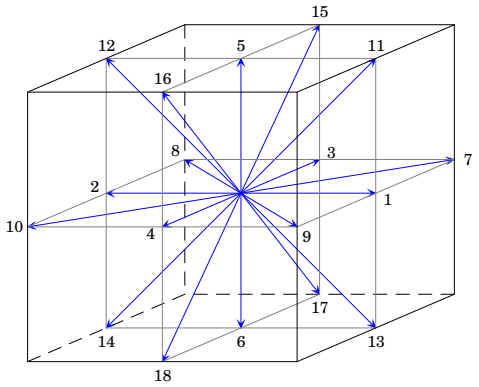


Figure 1: The D3Q19 stencil

The discrete counterpart of the distribution function f is a finite set of functions $f_\alpha(\boldsymbol{x}, t)$ associated to the particle velocities $\boldsymbol{\xi}_\alpha$. Let us denote:

$$\big|a_\alpha\big\rangle = (a_0, \dots a_N)^{\mathsf{T}},$$

T being the transpose operator. The lattice Boltzmann equation (LBE) writes:

$$\big|f_\alpha(\boldsymbol{x} + \delta t \boldsymbol{\xi}_\alpha, t + \delta t)\big\rangle - \big|f_\alpha(\boldsymbol{x}, t)\big\rangle = \Omega\Big[\big|f_\alpha(\boldsymbol{x}, t)\big\rangle\Big]. \tag{2}$$

The mass density ρ and the velocity $\boldsymbol{u}$ of the fluid are given by:

$$\rho = \sum_\alpha f_\alpha, \qquad \boldsymbol{u} = \frac{1}{\rho} \sum_\alpha f_\alpha \boldsymbol{\xi}_\alpha. \tag{3}$$

2.2 Multiple-relaxation-time LBM

The simplest (and most commonly used) way to express the collision operator is the LBGK approach [17], which uses the Bhatnagar-Gross-Krook approximation [1]. We instead chose to use the multiple-relaxation-time (MRT) approach [2]. Although of higher computational cost, MRT was shown of better accuracy and numerical stability than LBGK [7].

In the MRT approach, collision is performed in moment space. The particle distribution is mapped to a set of moments $\{m_\alpha \,|\, i = 0, \dots N\}$ by an orthogonal matrix M:

$$\big|f_\alpha(\boldsymbol{x}, t)\big\rangle = \mathsf{M}^{-1}\big|m_\alpha(\boldsymbol{x}, t)\big\rangle \tag{4}$$

where $\big|m(\boldsymbol{x}, t)\big\rangle$ is the moment vector. For the D3Q19 stencil, the orthogonal matrix M can be found in appendix A of [3]. The corresponding moment vector is:

$$\big|m_\alpha(\boldsymbol{x}, t)\big\rangle = \big(\rho, e, \varepsilon, j_x, q_x, j_y, q_y, j_z, q_z, 3p_{xx}, 3\pi_{xx}, p_{ww}, \pi_{ww}, p_{xy}, p_{yz}, p_{zx}, m_x, m_y, m_z\big)^{\mathsf{T}} \tag{5}$$

where e is energy, ε is energy square, $\boldsymbol{j} = (j_x, j_y, j_z)$ is the momentum, $\boldsymbol{q} = (q_x, q_y, q_z)$ is the heat flux, p_{xx}, p_{xy}, p_{yz}, p_{zx}, p_{ww} are related to the strain rate tensor S, π_{xx}, π_{ww} are fourth-order moments and m_x, m_y, m_z are third-order moments with respect to the particle velocities. The mass density and the momentum are the conserved moments.

The LBE thus writes:

$$\big|f_\alpha(\boldsymbol{x} + \delta t \boldsymbol{\xi}_\alpha, t + \delta t)\big\rangle - \big|f_\alpha(\boldsymbol{x}, t)\big\rangle = -\mathsf{M}^{-1}\Lambda\Big[\big|m_\alpha(\boldsymbol{x}, t)\big\rangle - \big|m_\alpha^{(\mathrm{eq})}(\boldsymbol{x}, t)\big\rangle\Big] \tag{6}$$

where Λ is a diagonal collision matrix and the $m_\alpha^{(\mathrm{eq})}$ are the equilibrium values of the moments. For the sake of isotropy, Λ obeys:

$$\Lambda = \mathrm{diag}(0, s_1, s_2, 0, s_4, 0, s_4, 0, s_4, s_9, s_{10}, s_9, s_{10}, s_{13}, s_{13}, s_{13}, s_{16}, s_{16}, s_{16}). \tag{7}$$

We additionally set $s_9 = s_{13}$. The relaxation rate s_9 is linked to the kinematic viscosity ν of the model by:

$$\frac{1}{s_9} = \frac{1}{c_s^2}\nu + \frac{1}{2}, \tag{8}$$

where the speed of sound c_s is set to:

$$c_s = \frac{1}{\sqrt{3}} \times \frac{\delta x}{\delta t}. \tag{9}$$

The other rates are set according to [7], i.e. $s_1 = 1.19$, $s_2 = s_{10} = 1.4$, $s_4 = 1.2$, and $s_{16} = 1.98$.

2.3 Large-eddy simulation

For large-eddy simulation (LES), the kinematic viscosity is $\nu = \nu_0 + \nu_t$ where ν_0 is the molecular viscosity and ν_t is the turbulent viscosity. In the Smagorinsky model [20], the turbulent viscosity is given by:

$$\nu_t = |\mathsf{S}|(C_S \delta x)^2, \quad |\mathsf{S}| = \sqrt{2\,\mathsf{S} : \mathsf{S}}, \tag{10}$$

where C_S is the Smagorinsky constant. Adding eddy viscosity to the MRT model is achieved by replacing the relaxation rate s_9 with:

$$s_9^* = \frac{1}{\tau_0 + \tau_t}, \tag{11}$$

where τ_0 and τ_t are the molecular and turbulent relaxation times:

$$\tau_0 = \frac{1}{c_s^2}\nu_0 + \frac{1}{2}, \quad \tau_t = \frac{1}{c_s^2}\nu_t. \tag{12}$$

Following [6], the second order moments obey:

$$\mathsf{P}_{ij} = \sum_\alpha \xi_{\alpha i}\xi_{\alpha j} f_\alpha = c_s^2 \rho \delta_{ij} + \rho u_i u_j - \mathsf{Q}_{ij}, \tag{13}$$

with:

$$\mathsf{Q} = \frac{2c_s^2 \rho}{s_9^*}\mathsf{S}. \tag{14}$$

Thus, the strain rate tensor may be computed from the moment vector. For the D3Q19 stencil, we obtain:

$$\mathsf{P}_{xx} = \frac{1}{57}(30\rho + e) + p_{xx}, \tag{15}$$

$$\mathsf{P}_{yy} = \frac{1}{57}(30\rho + e) + \frac{1}{2}(p_{ww} - p_{xx}), \tag{16}$$

$$\mathsf{P}_{zz} = \frac{1}{57}(30\rho + e) - \frac{1}{2}(p_{xx} + p_{ww}), \tag{17}$$

$$\mathsf{P}_{xy} = p_{xy}, \quad \mathsf{P}_{yz} = p_{yz}, \quad \mathsf{P}_{zx} = p_{zx}. \tag{18}$$

Finally, assuming that ν_t depends on S at current time, we have:

$$\tau_t = \frac{1}{2}\left(\sqrt{\tau_0^2 + 18\,|\mathsf{Q}|(C_S\delta x)^2} - \tau_0\right). \tag{19}$$

3 Multi-GPU solver

3.1 Algorithmic aspect

From an algorithmic standpoint, the LBE (Eq. 6) naturally breaks in two elementary step:

$$\left|\tilde{f}_\alpha(\boldsymbol{x}, t)\right\rangle = \left|f_\alpha(\boldsymbol{x}, t)\right\rangle + \Omega\left[\left|f_\alpha(\boldsymbol{x}, t)\right\rangle\right] \tag{20}$$

$$\left|f_\alpha(\boldsymbol{x} + \delta t\,\boldsymbol{\xi}_\alpha, t + \delta t)\right\rangle = \left|\tilde{f}_\alpha(\boldsymbol{x}, t)\right\rangle \tag{21}$$

Equation 20 describes the *collision* step in which an updated particle distribution is computed. Equation 21 describes the *propagation* step in which the updated particle populations are transferred to the neighbouring nodes as outlined by Fig. 2 (in two dimensions for the sake of simplicity).

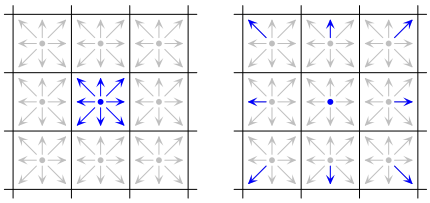


Figure 2: Propagation step

It is worth mentioning that in the first step, computations only involve informations local to each node. Moreover, in the second step, data transfers only require proper synchronisation with the nearest neighbours. As a matter of consequence, the LBM is fairly well suited for parallel implementations.

3.2 CUDA implementations

During the last decade, the computational power of GPUs has grown exponentially, reaching 1.35 Tflop/s single precision peak performance with the latest generation of NVIDIA processors [11]. Early attempts to implement the LBM on such hardware [4] were quite promising. With the introduction of the CUDA technology by NVIDIA in 2007, general purpose programming on GPUs became more practicable. Several successful CUDA implementations of the three-dimensional LBM [21, 13] were reported since.

On recent hardware, single GPU implementations are able to handle up to about 7.7×10^8 nodes per second, whereas multithreaded CPU implementations handle at most about 8.5×10^7 nodes per

second using a single quad core processor [8]. It is also worth mentioning that performance of GPU implementations is communication bound [14], while performance of CPU implementations is computation bound. Thus, making use of a model of higher algorithmic complexity (e.g. MRT instead of LBGK) has in general little impact on performance.

3.3 TheLMA framework

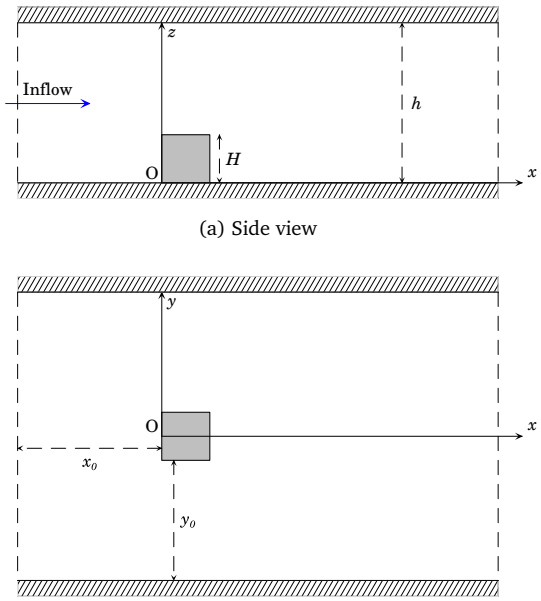
GPUs provide large computational power at fairly low cost. Yet, although growing more versatile at each generation, CUDA enabled GPUs still have numerous drawbacks. The CUDA tool chain for instance, due to hardware limitations, is unable to link several GPU binaries. In cases like LBM, this forbids the use of library oriented development techniques. The limited amount of on board memory may also be problematic. Using the latest computation devices, a single GPU implementation of the D3Q19 scheme may handle at most about 4.2×10^7 nodes in single precision.

To address both issues, we created the TheLMA framework [12]. TheLMA stands for *Thermal LBM on Many-core Architectures*, thermal simulations being our main topic of interest. The TheLMA framework is designed to improve code reusability. Setting up a new simulation usually only requires minor code modifications. Moreover, TheLMA provides native multi-GPU support [15]. For now, this support is limited to single servers, but extension to hybrid clusters is under active development.

4 Flow around a single cube

In order to validate our MRT-LBM solver, we chose to simulate a fully developed flow around a wall-mounted cube in a channel. The simulation results are compared to experimental data from [10]. Figure 3 outlines the simulation setup.

The channel is represented by a cavity containing $1{,}024 \times 768 \times 192 \approx 1.51 \times 10^8$ nodes. Solid walls are simulated using half-way bounce-back boundary condition [see for instance 16]. The inlet velocity is imposed by adding the corresponding equilibrium values to the distribution functions and the outlet condition is obtained by imposing null velocity gradient. The size of the cube is set to $H = 58\,\delta x$ in order to have $h/H \approx 3.3$ as in our reference, and the position



(a) Side view

(b) Top view

Figure 3: Simulation setup for a single cube

of the cube is such as $x_0 = 4H$. It should be mentioned that, in order to save memory, y_0 is less in our setup than in the experimental one. This allows to improve the resolution of the obstacle, with little impact on the flow since we have $y_0 > 6H$.

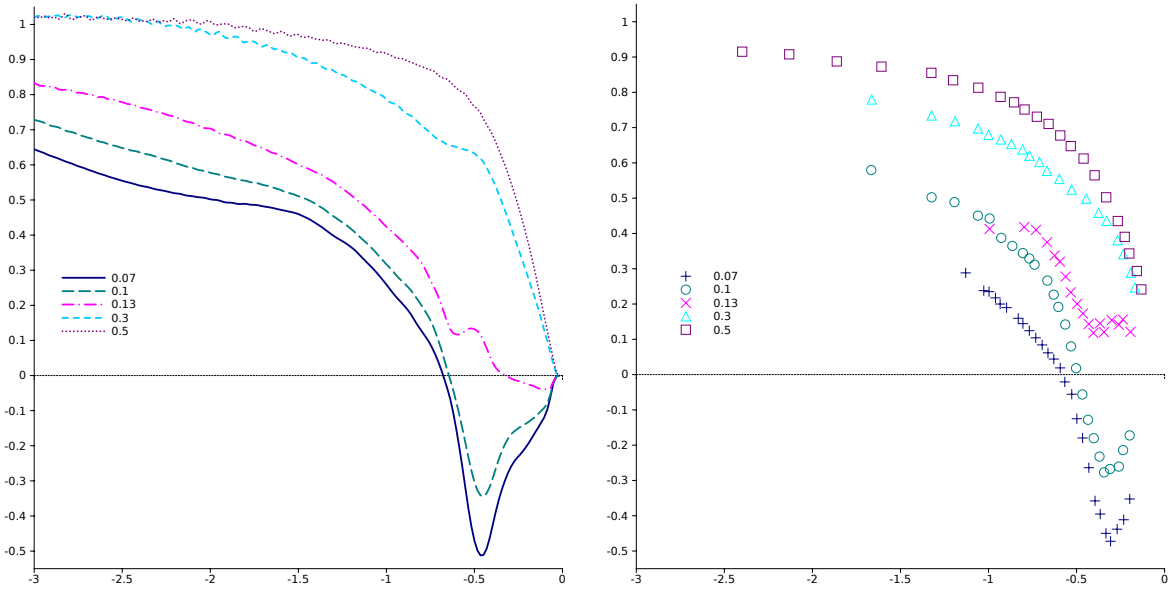
In their work, Meinders *et al.* [10] give the time-averaged streamwise velocity of the flow in the vertical symmetry plane, obtained through laser Doppler anemometry (LDA). The measurements were conducted at Reynolds number $\mathrm{Re} = 4{,}440$ where:

$$\mathrm{Re} = \frac{u_B H}{\nu} \tag{22}$$

and u_B is the bulk velocity of the inflow. In our simulation, we averaged the streamwise velocity over time from $50T_0$ to $200T_0$, where $T_0 = H/u_0$ is the turnover time and u_0 is the maximum inlet velocity. The overall computation time was less than six hours using a Tyan B7015 server with eight Tesla C1060 computing devices.

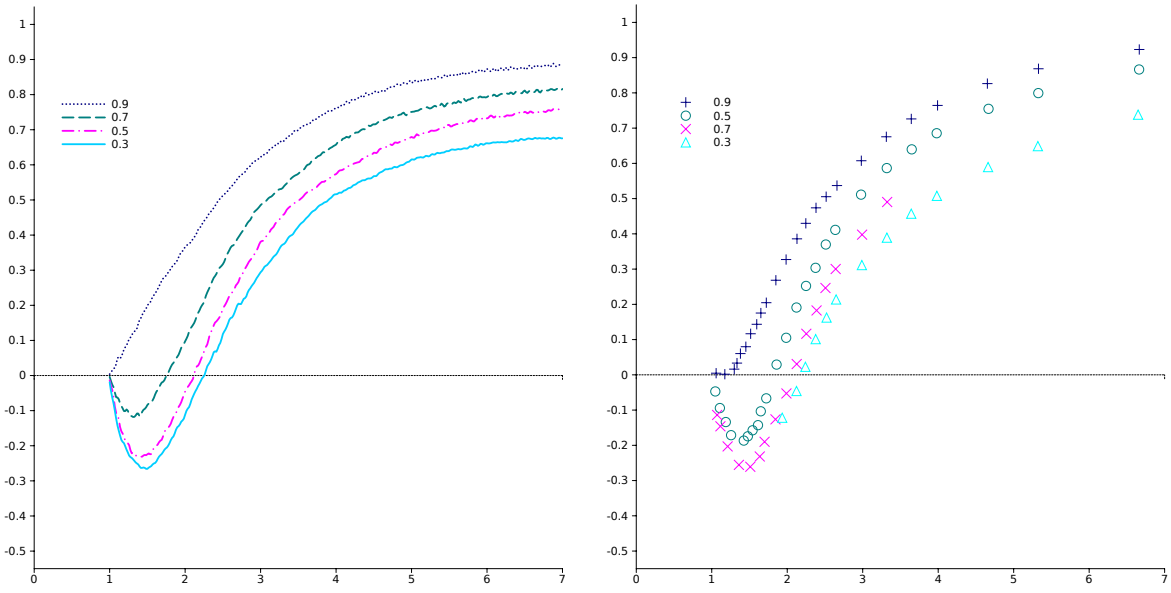
Figures 4 and 5 show upstream and downstream normalised velocity profiles with respect of x/H for both simulation and measurements.

Agreement of simulation data with experimental data is rather satisfying since uncertainties on both



(a) Simulation (Present) (b) Measurements [10]

Figure 4: Upstream normalised velocity profiles with respect of x/H for several values of z/H



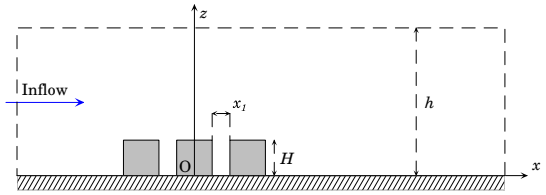
(a) Simulation (Present) (b) Measurements [10]

Figure 5: Downstream normalised velocity profiles with respect of x/H for several values of z/H

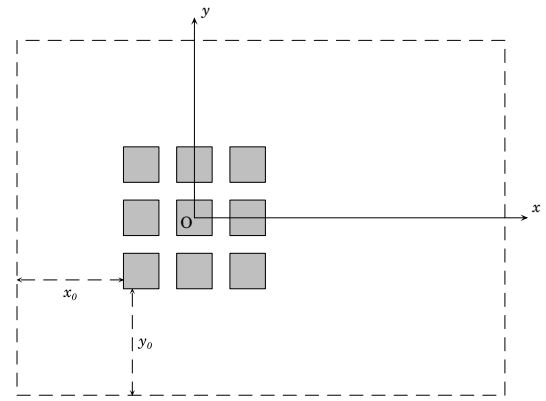
position and value of measurements should be taken into account. Unfortunately, our reference does not provide such informations, nor does it give detailed data regarding the inlet velocity profile. Although not perfect, agreement with measurements is by far better than in previously published work [see Fig. 9 in 22]. The most important flow features are also well predicted by our model.

5 Flow around nine cubes

To illustrate the possible use of multi-GPU LBM solvers in building aeraulics, we chose to simulate the flow around nine identical wall-mounted cubes. Figure 6 outlines the simulation setup.



(a) Side view



(b) Top view

Figure 6: Simulation setup for nine cubes

The simulation domain is again represented using a $1{,}024 \times 768 \times 192$ mesh. The size of the cubes is set to $H = 48\delta x$, and the position is such as $x_0 = 3H$ and $x_1 = H/2$. Thus, we have $y_0 = 6H$ and $h = 4H$. We impose logarithmic velocity profile for the inflow and constant streamwise velocity on the top lid. In this configuration, we define the Reynolds number as:

$$\mathrm{Re} = \frac{u_1 H}{\nu} \tag{23}$$

where u_1 is the inflow velocity at obstacle height. Furthermore, to reduce the impact of lateral faces on the flow, we apply the same boundary condition as for the outlet, i.e. null velocity gradient in the direction normal to the face.

We chose to run simulations at Reynolds numbers $\mathrm{Re}_1 = 40{,}000$ and $\mathrm{Re}_2 = 1{,}000{,}000$. Smagorinsky subgrid-scale models were reported satisfactory in similar situations, for Reynolds numbers up to at least Re_1 with LBM flow solvers [6], and at least Re_2 for Navier-Stockes solvers [19]. Although the LBM part in our implementation differs from the former, we may be rather confident in the results at $\mathrm{Re} = \mathrm{Re}_1$. The simulation at $\mathrm{Re} = \mathrm{Re}_2$ is more relevant at building scale, however the results should be considered with greater care.

As in the single cube simulation, we averaged density and velocity over time from $50T_0$ to $200T_0$, the turn-over time being set to $T_0 = 4H/u_0$. The overall computation time was about 17 h 19 min for 10^6 time steps. The corresponding performance is 2.4×10^9 node updates per second, which is at least a $28\times$ speedup over optimised multithreaded CPU implementations.

Figures 7 and 8 display the averaged pressure relative variation r and velocity streamlines in the vertical symmetry plane near the obstacle, with:

$$r = \frac{p - p_\infty}{p_\infty} \tag{24}$$

and p_∞ is the freestream pressure near the inflow.

Both simulations lead to rather similar mean flow patterns. The flow in the gaps between the cubes is structured in two independent cells. This specific flow feature is due to the three-dimensional effects of the flow.

The simulations also show quite similar pressure fields. At the walls, r equals to the pressure coefficient. The obtained values are within the range of coefficients used in practice and seem therefore reasonable.

6 Conclusion

In the present work, we provide building scale flow simulation results obtained using our multi-GPU im-

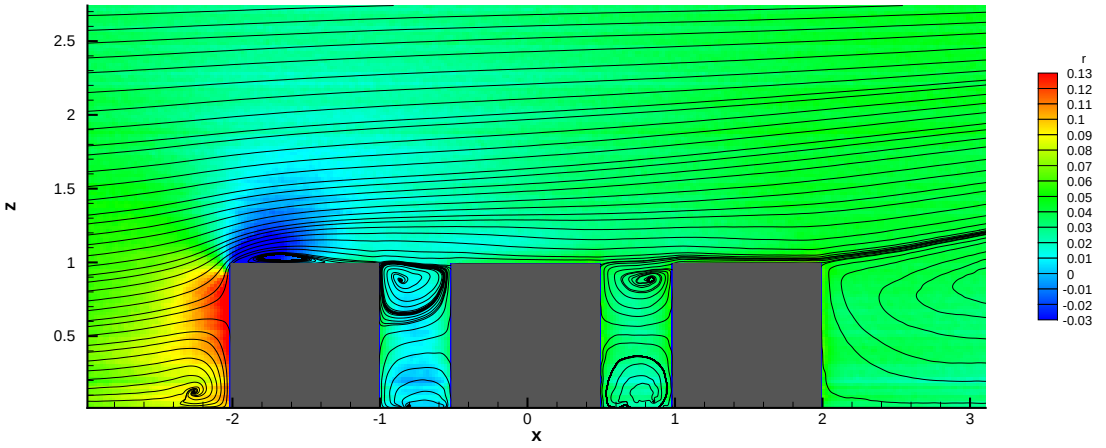


Figure 7: Time averaged streamlines and pressure relative variation at Re = 40,000

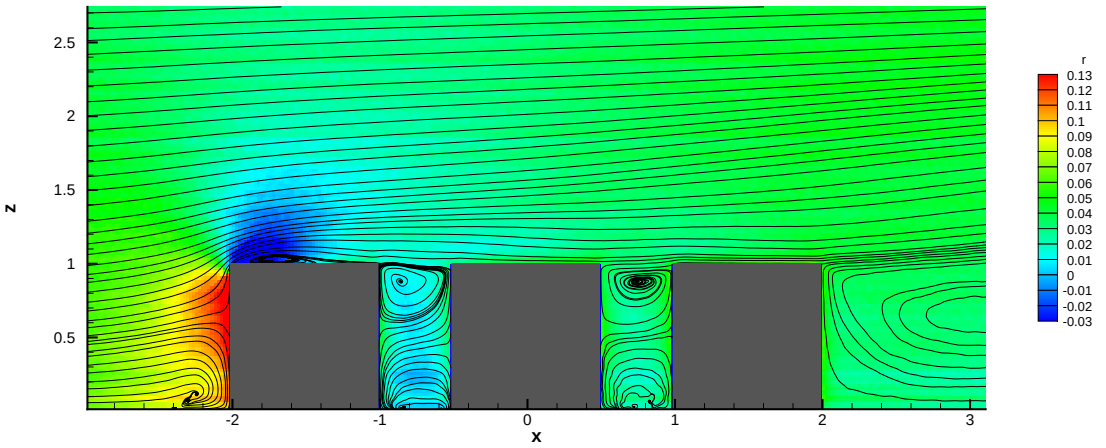


Figure 8: Time averaged streamlines and pressure relative variation at Re = 1,000,000

plementation of the LBM. We show that the required computation times remain below reasonable limits. Thus, we believe this contribution is a significant step towards the use of effective CFD simulations in building models. Moreover, it is worth mentioning that the LBM applies to a wide range of situations and therefore may be useful in other fields of building simulation than external aeraulics.

Several improvements to our approach, regarding both performance and accuracy are within reach. From a physical standpoint, the use of more elaborate subgrid-scale models than the static Smagorinsky model we implemented would be desirable. Ongoing research founded on the same mesoscopic point of view as the LBM might provide advances on this issue [18]. From a computational standpoint, porting grid refinement techniques to the GPU would be of highest practical interest and we plan to add such a feature to our framework in near future.

References

[1] P. L. Bhatnagar, E. P. Gross, and M. Krook. A Model for Collision Processes in Gases. I. Small Amplitude Processes in Charged and Neutral One-Component Systems. *Physical Review*, 94(3):511–525, 1954.

[2] D. d'Humières. Generalized lattice-Boltzmann equations. In *Proceedings of the 18th International Symposium on Rarefied Gas Dynamics*, pages 450–458. University of British Columbia, Vancouver, Canada, 1994.

[3] D. d'Humières, I. Ginzburg, M. Krafczyk, P. Lallemand, and L.S. Luo. Multiple-relaxation-time lattice Boltzmann models in three dimensions. *Philosophical Transactions of the Royal Society A*, 360:437–451, 2002.

[4] Z. Fan, F. Qiu, A. Kaufman, and S. Yoakum-Stover. GPU cluster for high performance computing. In *Proceedings of the 2004 ACM/IEEE Conference on Supercomputing*, pages 47–58. IEEE, 2004.

[5] U. Frisch, B. Hasslacher, and Y. Pomeau. Lattice-Gas Automata for the Navier-Stokes Equation. *Physical Review Letters*, 56(14):1505–1508, 1986.

[6] M. Krafczyk, J. Tölke, and L.S. Luo. Large-eddy simulations with a multiple-relaxation-time LBE model. *International Journal of Modern Physics B*, 17(1):33–40, 2003.

[7] P. Lallemand and L.S. Luo. Theory of the lattice Boltzmann method: Dispersion, dissipation, isotropy, Galilean invariance, and stability. *Physical Review E*, 61(6):6546–6562, 2000.

[8] V.W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A.D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, et al. Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. In *ACM SIGARCH Computer Architecture News*, volume 38, pages 451–460. ACM, 2010.

[9] G. R. McNamara and G. Zanetti. Use of the Boltzmann Equation to Simulate Lattice-Gas Automata. *Physical Review Letters*, 61(20):2332–2335, 1988.

[10] E.R. Meinders, K. Hanjalic, and R.J. Martinuzzi. Experimental study of the local convection heat transfer from a wall-mounted cube in turbulent channel flow. *Journal of heat transfer*, 121:564–573, 1999.

[11] NVIDIA. *Compute Unified Device Architecture Programming Guide version 3.2*, 2010.

[12] C. Obrecht, F. Kuznik, B. Tourancheau, and J.-J. Roux. Thermal LBM on Many-core Architectures. Available on www.thelma-project.info, 2010.

[13] C. Obrecht, F. Kuznik, B. Tourancheau, and J.-J. Roux. A new approach to the lattice Boltzmann method for graphics processing units. *Computers and Mathematics with Applications*, 61(12):3628–3638, 2011.

[14] C. Obrecht, F. Kuznik, B. Tourancheau, and J.-J. Roux. Global Memory Access Modelling for Efficient Implementation of the Lattice Boltzmann Method on Graphics Processing Units. In *Lecture Notes in Computer Science 6449, High Performance Computing for Computational Science, VECPAR 2010 Revised Selected Papers*, pages 151–161. Springer, 2011.

[15] C. Obrecht, F. Kuznik, B. Tourancheau, and J.-J. Roux. Multi-GPU implementation of the lattice Boltzmann method. *Computers and Mathematics with Applications*, (doi:10.1016/j.camwa.2011.02.020), 2011.

[16] C. Pan, L.S. Luo, and C.T. Miller. An evaluation of lattice Boltzmann schemes for porous medium flow simulation. *Computers & Fluids*, 35(8-9):898–909, 2006.

[17] Y. H. Qian, D. d'Humières, and P. Lallemand. Lattice BGK models for Navier-Stokes equation. *EPL (Europhysics Letters)*, 17(6):479–484, 1992.

[18] P. Sagaut. Toward advanced subgrid models for Lattice-Boltzmann-based Large-eddy simulation: Theoretical formulations. *Computers & Mathematics with Applications*, 59(7):2194–2199, 2010.

[19] S. Šarić, S. Jakirlić, A. Djugum, and C. Tropea. Computational analysis of locally forced flow over a wall-mounted hump at high-Re number. *International Journal of Heat and Fluid Flow*, 27(4):707–720, 2006.

[20] J. Smagorinsky. General circulation experiments with the primitive equations. *Monthly Weather Review*, 91(3):99–164, 1963.

[21] J. Tölke and M. Krafczyk. TeraFLOP computing on a desktop PC with GPUs for 3D CFD. *International Journal of Computational Fluid Dynamics*, 22(7):443–456, 2008.

[22] A. Yakhot, H. Liu, and N. Nikitin. Turbulent flow around a wall-mounted cube: A direct numerical simulation. *International Journal of Heat and Fluid Flow*, 27(6):994–1009, 2006.

Article G

Multi-GPU Implementation of a Hybrid Thermal Lattice Boltzmann Solver using the TheLMA Framework

Computers and Fluids, published online March 6, 2012

Accepted February 20, 2012

Abstract

In this contribution, a single-node multi-GPU thermal lattice Boltzmann solver is presented. The program is based on the TheLMA framework which was developed for that purpose. The chosen implementation and optimisation strategies are described, both for inter-GPU communication and for coupling with the thermal component of the model. Validation and performance results are provided as well.

Keywords: Thermal lattice Boltzmann method, GPU computing, CUDA

1 Introduction

Since its introduction in the late eighties, the lattice Boltzmann method (LBM) has proven to be an effective approach in computational fluid dynamics (CFD). It has been successfully applied to a wide range of engineering issues such as multiphase flows, porous media, or free surface flows. Despite of these achievements, the use of the LBM for thermal flow simulation is not very widespread yet. A possible reason for this situation is the relatively high computational cost of most thermal LBM models.

The use of emerging many-core architectures such as graphics processing units (GPUs) in CFD is fairly promising [2]. Being a regular data-parallel algorithm, the LBM is especially well adapted to such hardware. Nevertheless, implementing the lattice Boltzmann method for the GPU is still a pioneering task. Several important issues, such as multiphysics applications and efficient multi-GPU implementations, remain to be addressed. The present work, presenting a multi-GPU thermal LBM solver, faces both challenges.

The remaining of the paper is organised as follows. In the first section, we briefly present the thermal lattice Boltzmann model we chose to implement. Next, we give an overview of the TheLMA framework on which our program is based. In the third section, we describe our implementation and our optimisation strategies. Last, we provide some simulation results for validation purpose and discuss performance issues.

2 Hybrid thermal lattice Boltzmann model

The lattice Boltzmann equation (LBE), i.e. the governing equation of the LBM is interpreted as a discrete version of the Boltzmann equation [7]. In the LBM, as for the Boltzmann equation, a fluid is represented through the distribution of a single particle in phase space (i.e. position $\boldsymbol{x}$ and particle velocity ξ). Space is commonly discretised using a uniform orthogonal lattice of mesh size δx and time using constant time steps δt. Moreover, the particle velocity space is discretised into a finite set of particle velocities ξ_α. The LBM counterpart of the distribution function $f(\boldsymbol{x}, \xi, t)$ is a finite set $f_\alpha(\boldsymbol{x}, t)$ of particle distribution functions associated to the particle velocities ξ_α. The LBE writes:

$$\Big|f_\alpha(\boldsymbol{x} + \delta t \xi_\alpha, t + \delta t)\Big\rangle - \Big|f_\alpha(\boldsymbol{x}, t)\Big\rangle = \Omega\Big[\Big|f_\alpha(\boldsymbol{x}, t)\Big\rangle\Big]. \quad (1)$$

where Ω is the collision operator. The mass density ρ and the momentum $\boldsymbol{j}$ of the fluid are given by:

$$\rho = \sum_\alpha f_\alpha\,, \qquad \boldsymbol{j} = \sum_\alpha f_\alpha \boldsymbol{\xi}_\alpha\,. \tag{2}$$

The particle velocity set is usually chosen such as to link the nodes to some of their nearest neighbours, as the three-dimensional D3Q19 stencil illustrated by Fig. 1. It is well-known that such basic models are not energy conserving. To address this issue, several approaches such as multi-speed models [16] or double-population models [4] have been developed. In the former category, a larger set of particle velocities is defined allowing multiple particle speeds along some directions. In the later category, an additional set of energy distribution functions is used. Both approaches suffer from inherent numerical instabilities [5]. Moreover, from a computational standpoint, both methods lead to a markedly higher memory consumption.

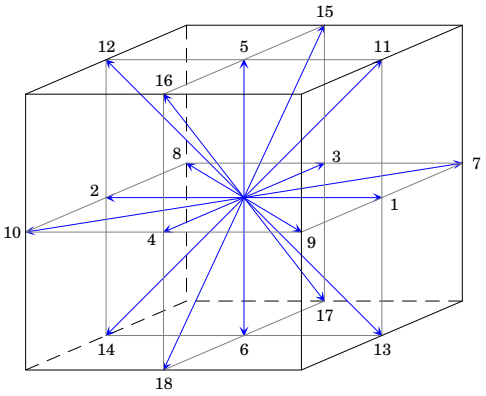


Figure 1: The D3Q19 stencil

Hybrid thermal lattice Boltzmann models constitute an alternative approach in which the flow simulation is decoupled from the solution of the heat equation. These models are free from the aforementioned drawbacks. In the present work, we implemented a simplified version of the hybrid thermal lattice Boltzmann model developed in [5]. Flow simulation is performed by multiple-relaxation-time LBM [1], using the D3Q19 stencil. In the multiple-relaxation-time approach, collision is performed in moment space and the LBE writes:

$$\begin{aligned}&\left|f_\alpha(\boldsymbol{x}+\delta t\boldsymbol{\xi}_\alpha, t+\delta t)\right\rangle - \left|f_\alpha(\boldsymbol{x},t)\right\rangle \\ &\quad = -\mathsf{M}^{-1}\mathsf{S}\left[\left|m_\alpha(\boldsymbol{x},t)\right\rangle - \left|m_\alpha^{(\mathrm{eq})}(\boldsymbol{x},t)\right\rangle\right]\end{aligned} \tag{3}$$

where M is an orthogonal matrix mapping the set of particle distributions to a set of moments m_α, and S is a diagonal matrix containing the relaxation rates. Matrices M and S as well as the equilibria of the moments for the D3Q19 stencil can be found in Appendix A of [1].

In our simulations, unlike Lallemand and Luo in [5], we set the ratio of specific heats $\gamma = C_P/C_V$ to $\gamma = 1$, which simplifies the coupling of the temperature T to the fluid momentum. Temperature is therefore obtained by solving the following finite-difference equation:

$$\partial_t^* T = \kappa \Delta^* T - \boldsymbol{j}\cdot\nabla^* T \tag{4}$$

where κ denotes the thermal diffusivity, which we assume being constant. For the sake of simplicity, we set $\delta x = 1$ and $\delta t = 1$, and define the finite-difference operators as:

$$\partial_t^* T(t) = T(t+1) - T(t) \tag{5}$$

$$\begin{aligned}\partial_x^* T(i,j,k) &= T(i+1,j,k) - T(i-1,j,k) \\ &-\frac{1}{8}\big(T(i+1,j+1,k) - T(i-1,j+1,k) \\ &\quad + T(i+1,j-1,k) - T(i-1,j-1,k) \\ &\quad + T(i+1,j,k+1) - T(i-1,j,k+1) \\ &\quad + T(i+1,j,k-1) - T(i-1,j,k-1)\big)\end{aligned} \tag{6}$$

$$\begin{aligned}\Delta^* T(i,j,k) &= 2\big(T(i+1,j,k) + T(i-1,j,k) \\ &\quad + T(i,j+1,k) + T(i,j-1,k) \\ &\quad + T(i,j,k+1) + T(i,j,k-1)\big) \\ &-\frac{1}{4}\big(T(i+1,j+1,k) + T(i-1,j+1,k) \\ &\quad + T(i+1,j-1,k) + T(i-1,j-1,k) \\ &\quad + T(i,j+1,k+1) + T(i,j-1,k+1) \\ &\quad + T(i,j+1,k-1) + T(i,j-1,k-1) \\ &\quad + T(i+1,j,k+1) + T(i-1,j,k+1) \\ &\quad + T(i+1,j,k-1) + T(i-1,j,k-1)\big) \\ &\quad - 9T(i,j,k)\end{aligned} \tag{7}$$

It should be mentioned that these operators share the same symmetries as the D3Q19 stencil. The coupling of the temperature to the momentum is explicit in Eq. 4. The coupling of the momentum to the temperature is carried out in the equilibrium of the second order moment m_2 related to internal energy:

$$m_2^{(\mathrm{eq})} = -11\rho + 19\boldsymbol{j}^2 + T \tag{8}$$

3 The TheLMA framework

Since the introduction of the CUDA technology [10] by the Nvidia company in 2007, several successful attempts to implement the LBM on the GPU were reported [18, 12]. Yet, constraints induced by low-level hardware specificities make GPU programming fairly different from usual software engineering. Beside other limitations, it is worth mentioning the fact that all symbols (e.g. device functions, device constants...) appearing in a kernel must be in the same compilation unit, which is due to the inability of the compilation tool chain to link several GPU binaries. Moreover, with hardware of compute capability 1.3 (which is the target architecture in the present work), device functions are of limited interest since inlining is compulsory in most cases.[1] Because of these constraints, library oriented development seems not relevant up to now for CUDA LBM solvers.

To improve code reusability, we designed the TheLMA framework [11], which is outlined in Fig. 2. TheLMA stands for *Thermal LBM on Many-core Architectures*, thermal flow simulations being our main topic of interest. The framework consists in a set of C and CUDA source files. The C files provide a set of utility functions to retrieve simulation parameters, initialise computation devices, extract statistical informations, and export data in various output formats. The CUDA files are included at compile time in the `thelma.cu` file which is mainly a container, additionally providing some general-purpose macros. Implementing a new lattice Boltzmann model within the framework mostly requires to alter the `compute.cu` file.

[1] Only device functions with a short list of parameters containing no pointers are actually callable and are likely to be not inlined by the compiler. Starting with hardware of compute capability 2.0, i.e. the Fermi generation, it is possible to perform actual function calls in any case, yet inlining is still the default behaviour.

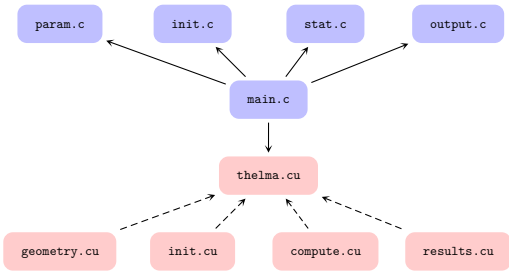


Figure 2: Overall structure of the TheLMA framework

Our framework provides native single-node multi-GPU management based on POSIX threads [15, 14]. Each computing device is managed by a specific thread which is responsible for creating the appropriate CUDA context. Communication between sub-domains is carried out using zero-copy transactions on pinned exchange buffers in CPU memory. During initialisation, one-dimensional domain decomposition is performed. The interfaces between the sub-domains are set such as to be normal to the major dimension of the lattice in memory. The read and store accesses to CPU memory can therefore be coalesced. Moreover, the chosen configuration leads to an excellent communication and computations overlapping. Figure 3 describes the inter-GPU communication scheme. For the sake of simplicity, only one GPU associated to a single sub-domain interface is presented. In order to address data dependency issues, since computations at grid level are asynchronous, our solver uses two instances of the lattice to store even time step data (in red) and odd time step data (in blue). The top line represents the GPU computations, Lattice 0 and Lattice 1 stand for the instances of the lattice stored in global memory, Read buffer 0 and Read buffer 1 for the buffers containing in-coming data, Store buffer 0 and Store buffer 1 for the buffers containing out-going data.

4 Implementation

For the implementation of a hybrid thermal lattice Boltzmann model on the GPU, there is the alternative of using a single kernel or two distinct kernels for solving the fluid flow and the thermal part. Since Eq. 3 and Eq. 4 are tightly coupled, the two kernels

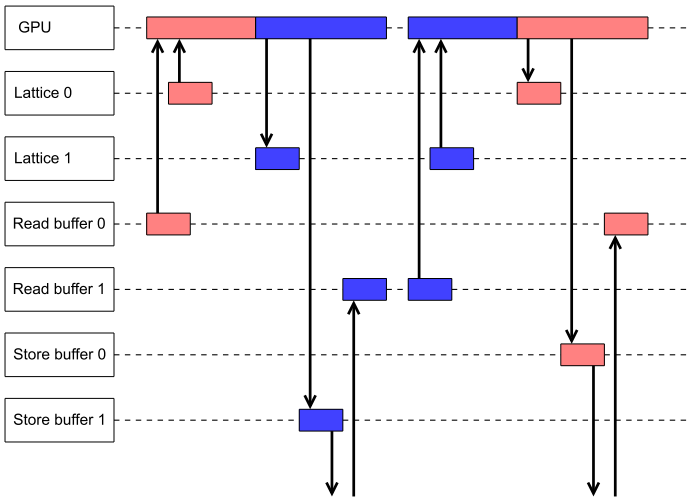


Figure 3: Inter-GPU communication scheme

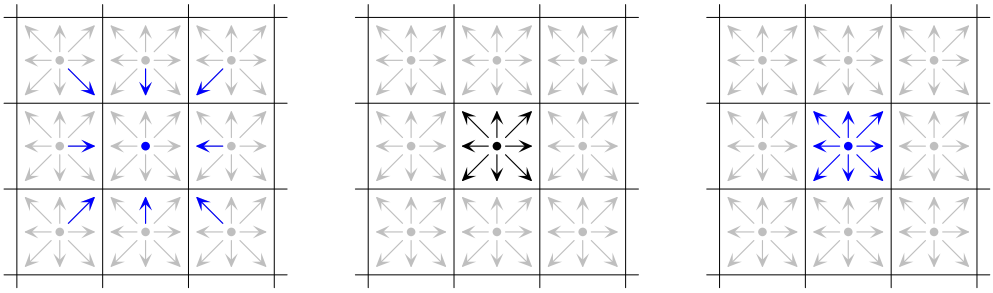
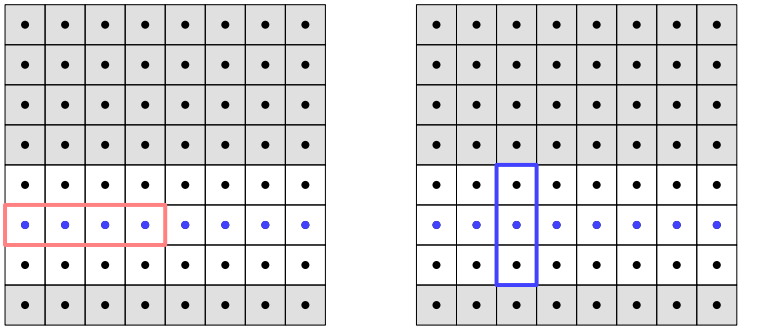


Figure 4: In-place propagation



(a) Block configuration (b) Read access pattern for temperature

Figure 5: Kernel execution pattern

option would increase the communication needs, not mentioning the overhead of kernel switching. As a matter of fact, handling temperature in a separate kernel, would require momentum to be stored and read back at each time step, thus increasing the amount of exchanged data by about 15%, for both read and store accesses. Since our isothermal multi-GPU solver is communication bound, we chose to process both parts in the same kernel.

The fluid flow component is derived from the one described in [12]. Beside other optimisations, the kernel uses in-place propagation as illustrated in Fig. 4 instead of the usual out-of-place propagation. This approach allows to minimise the cost of misaligned memory transactions [13]. Misalignment may have a dramatic impact on performance with pre-Fermi hardware, since the device's main memory is not cached.

CUDA implementations of the LBM generally assign a thread to each node in order to take advantage of the massive parallelism of the architecture. This approach often leads to the use of a two-dimensional grid of one-dimensional blocks, which allows a straightforward determination of the coordinates. The grid and block dimensions are therefore identical to the size of the computation sub-domain. The direction of the blocks corresponds to the slowest varying dimension in memory in order to enable coalesced memory accesses.

In our case, these common sense optimisation principles had to be altered. Since the implemented kernel takes care of both the fluid flow part and the thermal part, the register consumption is fairly higher than for usual isothermal LBM kernels. For compute capability 1.3, we could not achieve less than 124 registers per thread. Thus, assigning one thread to each node and using blocks spanning the entire cavity width causes register shortage with large cavities. In order to avoid this issue, we instead use small blocks containing one to four warps (i.e. 32, 64, 96 or 128 threads), each one associated to a one-dimensional zone spanning the cavity width. The kernel loops over the zone, in case of the block size being smaller than the zone size. Figure 5a outlines the chosen configuration (in two dimensions for the sake of simplicity). The blue dots represent the nodes belonging to the zone; the red frame represents the nodes being processed by the block of threads at a given step; the white background is used for the nodes whose temperature is required by the finite-difference computations in the zone.

The associated grid is two-dimensional, its size corresponding to the remaining dimensions of the sub-domain. It is worth mentioning that we assign the first rather than the second field of the `blockIdx` structure to the fastest varying dimension in memory. This option appears to improve the overlapping of computation and inter-GPU communication.

When implementing stencil computations on the GPU, reducing read redundancy is a key optimisation target [8]. We therefore chose to store the temperature of the neighbouring nodes in shared memory. In the case of boundary nodes, the surplus cells in the temperature array may be used to store shadow values determined by extrapolation. During the read phase, each thread is responsible for gathering the temperatures of all the nodes sharing the same abscissa, as outlined in Fig. 5b.

Not taking the boundaries into account, the chosen approach reduces the read redundancy in the D3Q19 case from 19 to at most[2] 9.3125. Moreover, it should be noted that this data access pattern induces no misalignement at all.

5 Results and discussion

5.1 Test case

To test our code, we simulated the well-known differentially heated cubic cavity illustrated in Fig.6. In this test case, two vertical opposite walls have imposed temperatures $\pm T_0$ whereas the four remaining walls are adiabatic. The buoyancy force $\boldsymbol{F}$ is computed using the Boussinesq approximation:

$$\boldsymbol{F} = -\rho\beta T\boldsymbol{g} \tag{9}$$

where β is the thermal expansion coefficient, and $\boldsymbol{g}$ the gravity vector of magnitude g.

The simple bounce-back scheme is used for the flow field boundary conditions. As shown by [3], the solid boundary is asymptotically located half-way between the wall nodes and the fluid nodes. This aspect must be taken into account when imposing thermal boundary conditions. In our implementation, we use halo temperatures computed by second-order extrapolations. For imposed temperature T_0, we have:

[2] Five additional temperatures for both the first and the last node are read, thus the worst case is for blocks of size 32 and the read redundancy equals $(32 \times 9 + 2 \times 5)/32$.

$$T(-1) = \frac{8}{3}T_0 - 2T(0) + \frac{1}{3}T(1) \qquad (10)$$

For adiabatic walls, we have:

$$T(-1) = \frac{21}{23}T(0) + \frac{3}{23}T(1) - \frac{1}{23}T(2) \qquad (11)$$

The wall temperature T_0 is set to $T_0 = 1$ and the Prandtl number is set to $\mathrm{Pr} = 0.71$. The parameters for the simulations are the Rayleigh number Ra and the kinematic viscosity ν, which determine the thermal diffusivity κ and the value of βg.

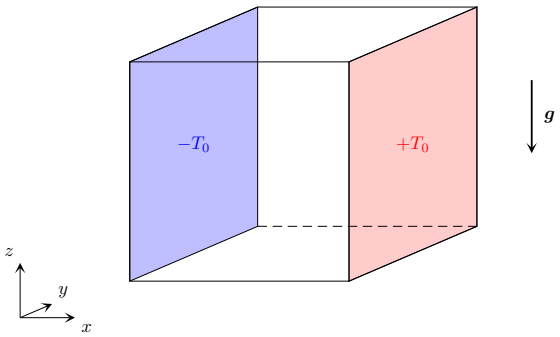


Figure 6: Differentially cubic heated cavity

We ran our program on a Tyan B7015 server fitted with eight Tesla C1060 computing devices. We could therefore perform computations on cavities as large as 512^3 in single precision.

5.2 Simulations

For validation purpose, we performed several simulations of the differentially heated cubic cavity using a 384^3 lattice and a 448^3 lattice in single precision. The kinematic viscosity was set to $\nu = 0.05$ and the Rayleigh number ranged from 10^4 to 10^7. The computations were carried out until convergence to steadiness, which is assumed to be reached when:

$$\max_{\boldsymbol{x}} \left| T(\boldsymbol{x}, t_{n+1}) - T(\boldsymbol{x}, t_n) \right| < 10^{-5} \qquad (12)$$

where $t_n = n \times 500\delta t$.

The obtained Nusselt numbers at the isothermal walls for both grid configurations are in good agreement as shown in Tab. 1, thus assuring grid independence. The simulation results are also close to previously published data [17], the maximum relative deviation being 0.84% for the coarser grid.

Using lattices of size 512^3 allowed us to run simulation for Rayleigh numbers up to 10^9 without facing numerical instabilities. From a phenomenological standpoint, although unsteady, the flow rapidly leads to a rather stable vertical stratification. We furthermore observe quasi-symmetric and quasi-periodic flow patterns near the bottom edge of the cold wall and the top edge of the hot wall. Figure 7 shows the temperature field in the symmetry plane after 10^6 iterations. Further investigations on this simulation are required and will be published in a future contribution.

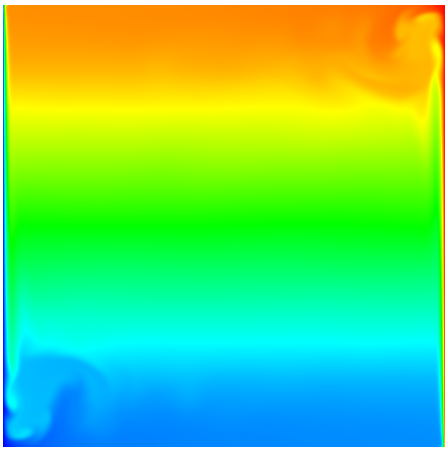


Figure 7: Symmetry plane temperature field at $\mathrm{Ra} = 10^9$

5.3 Performance

We recorded performance results of our solver for increasing block size and cavity size (see Fig. 8). The chosen performance metric is the million lattice node updates per second (MLUPS). The cavity size has to be a multiple of the block size, hence several configurations are not available. Performance obtained with a given block size appears to be correlated to the corresponding occupancy. For compute capability 1.3, global memory is split in eight 256 bytes wide

Rayleigh number	10^4	10^5	10^6	10^7
Tric *et al.* [17]	2.054	4.337	8.640	16.342
Present (384^3)				
Nusselt number	2.055	4.339	8.652	16.481
Time steps	420,000	304,000	210,500	148,500
Computation time (min)	365	245	177	123
Relative deviation	0.05%	0.05%	0.14%	0.84%
Present (448^3)				
Nusselt number	2.050	4.335	8.645	16.432
Time steps	485,000	380,000	266,000	182,000
Computation time (min)	394	309	216	148
Relative deviation	0.19%	0.07%	0.06%	0.55%

Table 1: Comparison of Nusselt numbers at the isothermal walls

memory banks [9]. Hence, the poor performance obtained for cavity size 256 and 512 is probably caused by partition camping, since the stride between corresponding nodes in distinct blocks is necessarily a multiple of the cavity size.

The maximum performance is 1,920 MLUPS, achieved for cavity size 448 and block size 64. The corresponding GPU to device memory data throughput is 46.1 GB/s per GPU, which is about 62.3% of the maximum sustained throughput.[3] The multiprocessor occupancy, which is only 13%, appears to be the limiting factor since it is lower than the minimum required to properly hide the global memory latency, i.e. 18.75% for compute capability 1.3.

To evaluate scalability, we also ran our program on a 192^3 lattice using from one to eight GPUs (see Fig. 9). Parallelisation efficiency is very satisfactory with no less than 84% for a fairly small computation domain. As for our isothermal multi-GPU LBM solver [14], our implementation allows excellent overlapping of communication and computations. Moreover, the amount of data to exchange does not exceed the capacity of the PCI-E links.

[3] Using the `bandwidthTest` program of the CUDA SDK, we estimate the GPU to device memory maximum sustained throughput to 73.3 GB/s for the Tesla C1060.

6 Conclusion

In this contribution, we present a multi-GPU implementation of a thermal LBM solver, which to the best of our knowledge was never reported before. Using appropriate hardware, our program is able to run up to eight GPUs in parallel. With the latest generation of Nvidia computing devices, it is therefore possible to perform simulations on lattices containing as much as 3.2×10^8 nodes.

Validation studies have been carried out, showing both the accuracy and the stability of the chosen thermal LBM model and the correctness of our implementation. Although slightly less efficient than the isothermal version of our solver, our program provides unrivaled performance compared to CPU implementations. Recent studies [6] have shown that optimised multi-threaded CPU implementations of isothermal LBM solver running on up-to-date hardware achieve at most 85 MLUPS, which is 22× less than our maximum performance.

We furthermore study the performance bottlenecks, showing that the limiting factor is the low occupancy. Since the multiprocessor occupancy is bound by the amount of available registers there is little room for improvements using the same hardware. Yet, a more elaborate memory access pattern

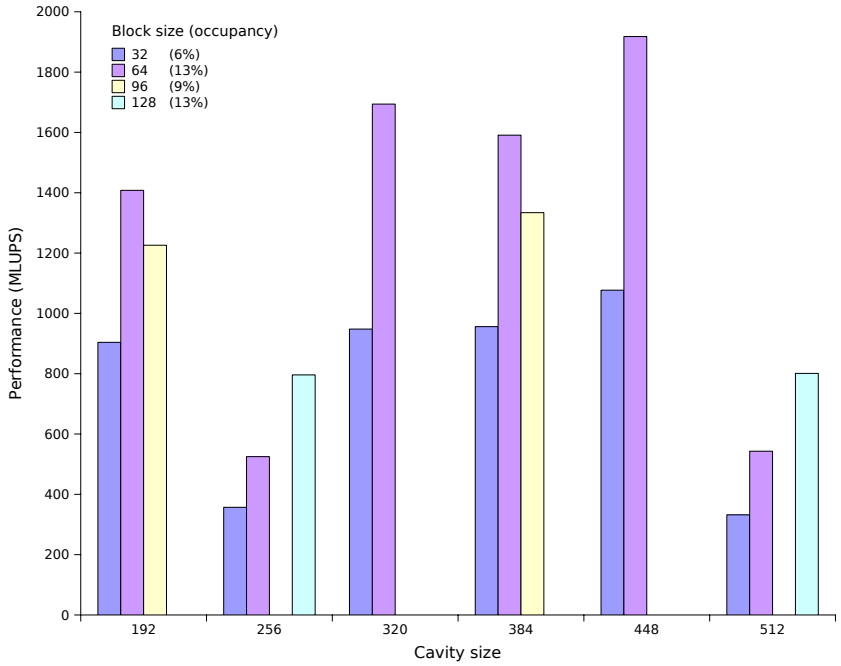


Figure 8: Performance for increasing block size and cavity size

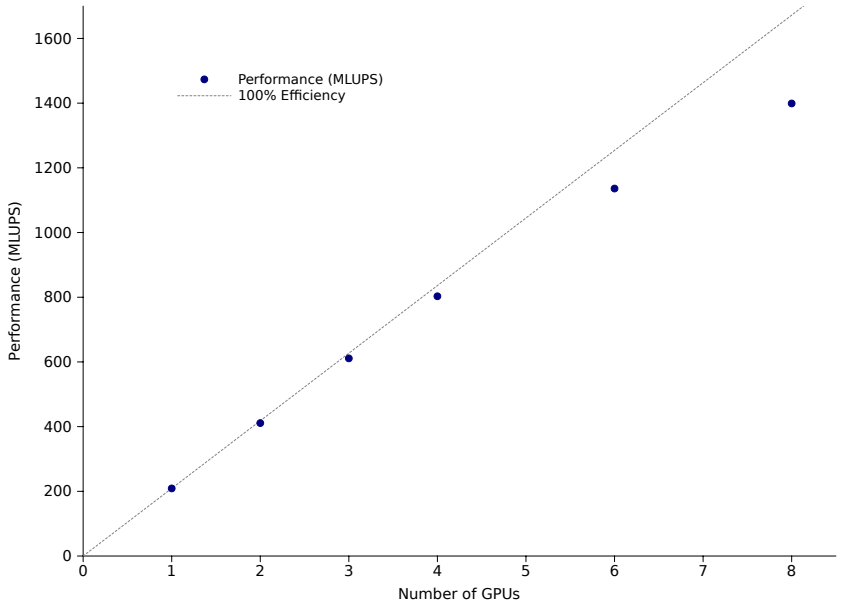


Figure 9: Parallelisation efficiency on a 192^3 cavity

could avoid the partition camping effects we could observe in some cases.

We believe our work is a significant step towards the use of GPU based LBM solvers in practice. In near future, we intend to add specific optimisations for compute capability 2.0 and 2.1 hardware, i.e. the latest CUDA capable GPUs. We also plan to extend the TheLMA framework on which our program is based to multi-node implementations.

References

[1] D. d'Humières, I. Ginzburg, M. Krafczyk, P. Lallemand, and L.S. Luo. Multiple-relaxation-time lattice Boltzmann models in three dimensions. *Philosophical Transactions of the Royal Society A*, 360:437–451, 2002.

[2] J. Dongarra, G. Peterson, S. Tomov, J. Allred, V. Natoli, and D. Richie. Exploring new architectures in accelerating CFD for Air Force applications. In *DoD HPCMP Users Group Conference*, pages 472–478. IEEE, 2008.

[3] I. Ginzbourg and P. M. Adler. Boundary flow condition analysis for the three-dimensional lattice Boltzmann model. *Journal de Physique II*, 4(2):191–214, 1994.

[4] X. He, S. Chen, and G. D. Doolen. A Novel Thermal Model for the Lattice Boltzmann Method in Incompressible Limit. *Journal of Computational Physics*, 146(1):282–300, 1998.

[5] P. Lallemand and L. S. Luo. Theory of the lattice Boltzmann method: Acoustic and thermal properties in two and three dimensions. *Physical review E*, 68(3):36706(1–25), 2003.

[6] V.W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A.D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, et al. Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. In *ACM SIGARCH Computer Architecture News*, volume 38, pages 451–460. ACM, 2010.

[7] G. R. McNamara and G. Zanetti. Use of the Boltzmann Equation to Simulate Lattice-Gas Automata. *Physical Review Letters*, 61(20):2332–2335, 1988.

[8] P. Micikevicius. 3D finite difference computation on GPUs using CUDA. In *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, pages 79–84. ACM, 2009.

[9] D. Mudigere. Data access optimized applications on the GPU using NVIDIA CUDA. Master's thesis, Technische Universität München, 2009.

[10] NVIDIA. *Compute Unified Device Architecture Programming Guide version 3.1.1*, 2010.

[11] C. Obrecht, F. Kuznik, B. Tourancheau, and J.-J. Roux. Thermal LBM on Many-core Architectures. Available on www.thelma-project.info, 2010.

[12] C. Obrecht, F. Kuznik, B. Tourancheau, and J.-J. Roux. A new approach to the lattice Boltzmann method for graphics processing units. *Computers and Mathematics with Applications*, 61(12):3628–3638, 2011.

[13] C. Obrecht, F. Kuznik, B. Tourancheau, and J.-J. Roux. Global Memory Access Modelling for Efficient Implementation of the Lattice Boltzmann Method on Graphics Processing Units. In *Lecture Notes in Computer Science 6449, High Performance Computing for Computational Science, VECPAR 2010 Revised Selected Papers*, pages 151–161. Springer, 2011.

[14] C. Obrecht, F. Kuznik, B. Tourancheau, and J.-J. Roux. Multi-GPU implementation of the lattice Boltzmann method. *Computers and Mathematics with Applications*, (doi:10.1016/j.camwa.2011.02.020), 2011.

[15] C. Obrecht, F. Kuznik, B. Tourancheau, and J.-J. Roux. The TheLMA project: Multi-GPU implementation of the lattice Boltzmann method. *International Journal of High Performance Computing Applications*, 25(3):295–303, 2011.

[16] Y. H. Qian. Simulating thermohydrodynamics with lattice BGK models. *Journal of Scientific Computing*, 8(3):231–242, 1993.

[17] E. Tric, G. Labrosse, and M. Betrouni. A first incursion into the 3D structure of natural convection of air in a differentially heated cubic

cavity, from accurate numerical solutions. *International Journal of Heat and Mass Transfer*, 43(21):4043 – 4056, 2000.

[18] J. Tölke. Implementation of a Lattice Boltzmann kernel using the Compute Unified Device Architecture developed by nVIDIA. *Computing and Visualization in Science*, 13(1):29–39, 2010.

Article H

Efficient GPU Implementation of the Linearly Interpolated Bounce-Back Boundary Condition

Computers and Mathematics with Applications, published online June 28, 2012

Accepted May 23, 2012

Abstract

Interpolated bounce-back boundary conditions for the lattice Boltzmann method (LBM) make the accurate representation of complex geometries possible. In the present work, we describe an implementation of a linearly interpolated bounce-back (LIBB) boundary condition for graphics processing units (GPUs). To validate our code, we simulated the flow past a sphere in a square channel. At low Reynolds numbers, results are in good agreement with experimental data. Moreover, we give an estimate of the critical Reynolds number for transition from steady to periodic flow. Performance recorded on a single node server with eight GPU based computing devices ranged up to 2.63×10^9 fluid node updates per second. Comparison with a simple bounce-back version of the solver shows that the impact of LIBB on performance is fairly low.

Keywords: Lattice Boltzmann method, GPU programming, CUDA, Interpolated bounce-back boundary condition, TheLMA project

1 Introduction

From a computational perspective, the lattice Boltzmann method (LBM) can be seen as a data parallel algorithm with local synchronisation constraints. It is therefore well-suited to massively parallel architectures such as graphics processing units (GPUs) as shown in the pioneering work of Fan *et al.* in 2004 [3]. Since the advent of the CUDA technology in 2007 [9], several efficient implementations of the LBM for the GPU were reported [23, 7]. Recent multi-GPU implementations [15] make the use of large computation domains possible, which otherwise would be bound by the limited amount of onboard memory. Nevertheless, several other issues, such as accurate representation of complex geometries, remain to be addressed in order to improve the practical interest of GPU LBM solvers. Implementing a LBM boundary condition for the GPU is quite challenging since it usually requires branching and, in most cases, access to specific data. With CUDA enabled GPUs, branch divergences often lead to warp serialisation[1] which alters the schedule of global memory transactions and therefore may have a significant impact on performance [10].

In this contribution, we describe the multi-GPU implementation of an extension to the simple bounce-back boundary condition. This approach introduced in 2001 by Bouzidi [1], uses interpolations to take into account the exact location of the solid boundaries. For validation purposes, we simulated the flow past a sphere in a square channel and compared our results with experimental data. The paper is organised as follows. First, we briefly introduce the LBM and present the boundary condition we implemented. Then, we outline the TheLMA framework, on which our solver is based, and describe the proposed implementation. Next, we report and discuss our simulation results and finally we present some performance measurements.

[1] CUDA threads are run concurrently in *warps* of 32. When branch divergence occurs within a warp, the divergent branches are serialised.

2 Lattice Boltzmann method

With the continuous Boltzmann equation, fluid dynamics is represented through the evolution in time of a single-particle distribution function f in phase space. As shown by He and Luo [5], lattice Boltzmann models are based on discretised versions of the Boltzmann equation in both time and phase space. In general, the LBM uses a regular orthogonal lattice of mesh size δx and constant time steps δt. The velocity space is replaced by a finite set of $N+1$ particle velocities $\{\boldsymbol{\xi}_\alpha \,|\, \alpha = 0,\ldots,N\}$. The lattice Boltzmann analogue of the distribution function f is a set of functions $\{f_\alpha \,|\, \alpha = 0,\ldots,N\}$ associated to the particle velocities. Using the former notations, the lattice Boltzmann equation (LBE), i.e. the governing equation of the LBM, is written:

$$\left|f_\alpha(\boldsymbol{x}+\boldsymbol{c}_\alpha, t+\delta t)\right\rangle - \left|f_\alpha(\boldsymbol{x},t)\right\rangle = \Omega\Big[\left|f_\alpha(\boldsymbol{x},t)\right\rangle\Big]. \quad (1)$$

where Ω is the collision operator. The mass density ρ and the momentum $\boldsymbol{j}$ of the fluid are given by:

$$\rho = \sum_\alpha f_\alpha, \qquad \boldsymbol{j} = \sum_\alpha f_\alpha \boldsymbol{\xi}_\alpha. \quad (2)$$

From an algorithmic perspective, Eq. 1 naturally breaks into two elementary steps:

$$\left|\widetilde{f}_\alpha(\boldsymbol{x},t)\right\rangle = \left|f_\alpha(\boldsymbol{x},t)\right\rangle + \Omega\Big[\left|f_\alpha(\boldsymbol{x},t)\right\rangle\Big] \quad (3)$$

$$\left|f_\alpha(\boldsymbol{x}+\boldsymbol{c}_\alpha, t+\delta t)\right\rangle = \left|\widetilde{f}_\alpha(\boldsymbol{x},t)\right\rangle \quad (4)$$

where $\boldsymbol{c}_\alpha = \delta t\, \boldsymbol{\xi}_\alpha$. Equation 3 describes the *collision* step in which updated particle populations $\widetilde{f}_\alpha$ are computed. Equation 4 describes the *propagation* step in which the updated particle populations are transferred to the neighbouring nodes. The particle velocity set is usually chosen such as to link the nodes to some of their nearest neighbours, as the three-dimensional D3Q19 stencil illustrated by Fig. 1.

For the present work, we used a D3Q19 multiple-relaxation-time (MRT) lattice Boltzmann model. As shown in [2], the MRT approach increases the numerical stability of the LBM compared to the widespread LBGK approach [19]. With MRT, collision is performed in moment space. The particle distribution is mapped to a set of moments $\{m_\alpha \,|\, \alpha = 0,\ldots,N\}$ by an orthogonal matrix M:

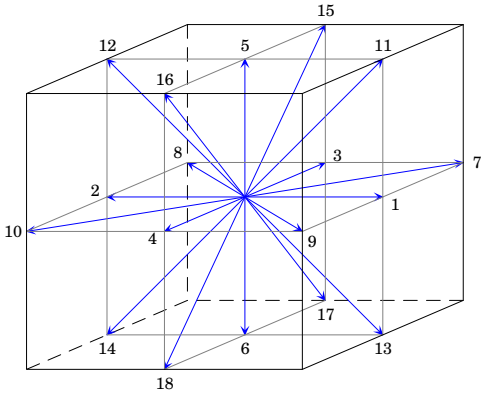


Figure 1: The D3Q19 stencil

$$\left|m_\alpha(\boldsymbol{x},t)\right\rangle = \mathsf{M}\left|f_\alpha(\boldsymbol{x},t)\right\rangle \quad (5)$$

where $\left|m_\alpha(\boldsymbol{x},t)\right\rangle$ is the moment vector. The LBE becomes:

$$\left|f_\alpha(\boldsymbol{x}+\boldsymbol{c}_\alpha, t+\delta t)\right\rangle - \left|f_\alpha(\boldsymbol{x},t)\right\rangle = -\mathsf{M}^{-1}\Lambda\Big[\left|m_\alpha(\boldsymbol{x},t)\right\rangle - \left|m_\alpha^{(\mathrm{eq})}(\boldsymbol{x},t)\right\rangle\Big] \quad (6)$$

where Λ is a diagonal collision matrix and the $m_\alpha^{(\mathrm{eq})}$ are the equilibrium values of the moments. The transformation matrix M, the collision matrix Λ, and the definition of equilibrium moments $m_\alpha^{(\mathrm{eq})}$ we used in our implementation can be found in Appendix A of [2].

3 Bounce-back boundary conditions

Lattice Boltzmann boundary conditions for solid walls basically divide up into wet node conditions and bounce-back conditions. In the former category, the boundary nodes, i.e. the nodes on which the condition is applied, are supposed to be both located on the solid boundary and part of the fluid [6]. In the later category, the boundary nodes are in general the fluid nodes next to the solid nodes and the solid boundary is located somewhere in between.

An elementary version of bounce-back is the so-called simple bounce-back (SBB). With SBB, an un-

known particle population f_α at a boundary node obeys the following equation:

$$f_\alpha(\boldsymbol{x},t) = \widetilde{f}_{\bar{\alpha}}(\boldsymbol{x},t-\delta t) \tag{7}$$

where $\bar{\alpha}$ is the direction opposite to α. Algorithmic simplicity of SSB is obvious when considering Eq. 7. The only information required for a given node is the list of unknown particle populations. Moreover, it is known that (asymptotically) the solid boundary is located halfway between the solid and the fluid nodes [4]. Simple bounce-back is therefore convenient in many situations. However, to handle complex geometries, a more elaborate approach is needed.

In 2001, Bouzidi *et al.* [1] introduced an extension to SBB based on either linear interpolation (LIBB) or quadratic interpolation, which allows the solid boundary to take any desired position. In the present work, we implemented the LIBB as formulated by Pan *et al.* [18]. Let $\boldsymbol{x}$ denote a boundary node such that $\boldsymbol{x}+\boldsymbol{c}_\alpha$ is a solid node, and q be the number such that $\boldsymbol{x}+q\boldsymbol{c}_\alpha$ is on the solid boundary. For $q<1/2$,

$$f_{\bar{\alpha}}(\boldsymbol{x},t) = (1-2q)f_\alpha(\boldsymbol{x},t) + 2q\widetilde{f}_\alpha(\boldsymbol{x},t-\delta t) \tag{8}$$

and for $q \geq 1/2$,

$$f_{\bar{\alpha}}(\boldsymbol{x},t) = \left(1-\frac{1}{2q}\right)\widetilde{f}_{\bar{\alpha}}(\boldsymbol{x},t-\delta t) + \frac{1}{2q}\widetilde{f}_\alpha(\boldsymbol{x},t-\delta t) \tag{9}$$

It should be noted that for $q=1/2$, LIBB reduces to SBB. The interpolation schemes of LIBB are illustrated in Fig. 2. In the first case, i.e. $q<1/2$, one considers fictitious particles located at D, which end up at A after bouncing back on the wall at C. The particle population at D is constructed from the pre-collision ones at E and A. In the second case, i.e. $q \geq 1/2$, the particles leaving A end up at D. The unknown incoming particle population at A is constructed from the post-propagation ones at D and E.

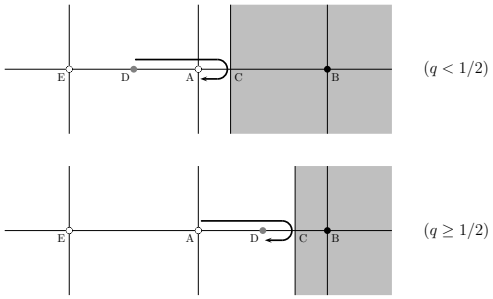


Figure 2: Interpolation schemes of LIBB

4 Implementation

4.1 The TheLMA framework

The proposed implementation of the LIBB boundary condition was carried out within the TheLMA framework [11]. The design of graphics processing units is guided by their primary use which is rather different from general purpose computations. As a consequence, general purpose computing technologies such as CUDA suffer from significant limitations. Regarding CUDA, it is worth mentioning the inability of the compilation tool-chain to link several GPU binaries. In other words, all symbols (e.g. device functions, device constants...) appearing in a kernel must be in the same compilation unit. Such a constraint, makes library oriented development irrelevant in many situations, and more specifically for LBM solvers. We therefore decided to create a framework in order to improve code reusability.

TheLMA stands for *Thermal LBM on Many-core Architectures*, thermal simulations being our main topic of interest. The framework consists of a set of modules which are designed such as to minimise code modifications when setting up a new simulations or implementing a new model. It provides native single-node multi-GPU support based on POSIX threads [14]. The core collision and propagation kernel is derived from the single-GPU code described in [12]. The execution grid of the core kernel is two-dimensional with one-dimensional blocks, each node of the lattice being associated to a thread. In order to ensure global synchronisation, two instances of the particle distribution are kept in global memory, corresponding to even and odd time steps.

For each computation sub-domain, the particle distribution is stored in a four-dimensional array. The fastest varying dimension corresponds to the direction of the blocks which allows memory transactions to be coalesced. The second fasted varying dimension corresponds to the velocity set index. Instead of using the usual out-of-place propagation, our core kernel performs in-place propagation which consists

in carrying out propagation before collision instead of after. This propagation scheme is illustrated by Fig. 3. The represented case is only two-dimensional for the sake of clarity. It was shown in [13] that this simple optimisation minimises the cost of misaligned memory transactions, which may have substantial effects on performance with pre-Fermi hardware.

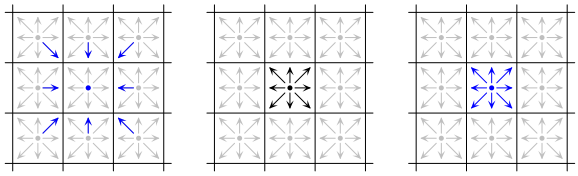


Figure 3: In-place propagation

4.2 Proposed implementation

In the TheLMA framework, geometry is represented using bit-fields. To process a node, the corresponding thread first loads a 32-bit integer. The first N bits of the integer are used to indicate whether the node in the corresponding direction is solid. This technique makes the implementation of SBB rather straightforward. Since we use in-place propagation, some of the particle populations loaded for a boundary node are invalid, but these values are discarded when applying the boundary condition. Our tests have shown that it is of little interest to avoid loading these invalid populations. As a matter of fact, it may have a positive impact to cancel invalid loads when a whole half-warp is involved, e.g. for cavity walls parallel to the blocks or within very large obstacles. Yet, the overhead of branching decisions together with surface to volume effects make the benefits negligible in practice.

Our implementation of the LIBB takes advantage of these unnecessary memory accesses. At initialisation, the distance information for the solid boundaries are computed and stored in the unused particle population array cells of the relevant solid nodes. At each time step, the distance information are retrieved by the threads processing boundary nodes during propagation. To perform interpolation, the threads need in addition to fetch some of the local updated particle populations of the former time step (see Eqs. 8 and 9). It should be mentioned that the additional data is accessed as if $q \geq 1/2$. This yields an unnecessary read access when $q < 1/2$ but requires simpler code and thus leads to a smaller kernel. The data access scheme is outlined by Fig. 4. Blue is used for the particle populations involved in collision, red for the distance information, and black for the particle populations involved in interpolation. Again, the displayed case is two-dimensional for the sake of clarity.

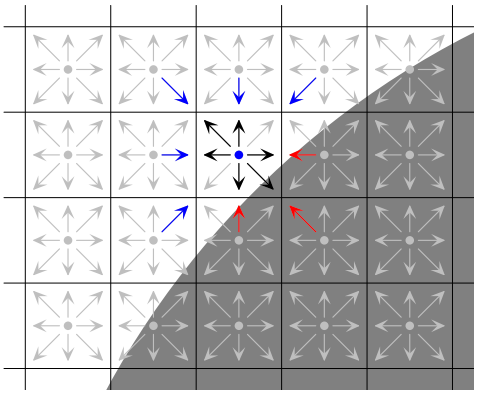


Figure 4: Implementation of the LIBB

It is worth stressing that, in practice, because of the chosen propagation scheme, the proposed implementation of LIBB only slightly increases the overall number of memory accesses compared to our implementation of SBB. The implemented initialisation and simulation kernels are summarised in Pseudo-Codes 1 and 2.

5 Simulations

5.1 Test case

For validation purposes, we studied the vortex shedding frequency for a uniform flow past a sphere in a square channel of width ℓ. We performed single precision simulations with four different configurations for Reynolds numbers ranging from $\mathrm{Re} = 270$ to $\mathrm{Re} = 350$, the Reynolds number being defined as $\mathrm{Re} = u_0 D/\nu$ where u_0 is the bulk velocity at the inflow, D is the diameter of the sphere and ν is the kinematic viscosity. This range was chosen in order to compare our results with experimental data provided by Sakamoto and Haniu [21], and by Ormières and Provansal [16].

Figure 5 outlines the simulation setup. For each configuration, the distance between the inlet and the

```
1.  if node x is solid then
2.    set flag solid for x
3.    for each direction α do
4.      if node x + c_α is fluid then
5.        compute q for x and x + c_α
6.        store q in f_ᾱ(x + c_α, 0)
7.      end if
8.    end for
9.  else
10.   for each direction α do
11.     if node x + c_α is solid then
12.       set flag α for x
13.     end if
14.   end for
15. end if
```

Pseudo-Code 1: Initialisation kernel

```
1.  read bit-field for x
2.  if node x is fluid then
3.    for each direction α do
4.      read f̃_α(x − c_α, t − δt)
5.    end for
6.    for each direction α do
7.      if flag α is set then
8.        set q to f_ᾱ(x, t)
9.        read f̃_α(x, t − δt) and f̃_ᾱ(x, t − δt)
10.       interpolate f_ᾱ(x, t)
11.     end if
12.   end for
13.   compute distribution f̃_α(x, t)
14.   store distribution f̃_α(x, t)
15. end if
```

Pseudo-Code 2: Simulation kernel

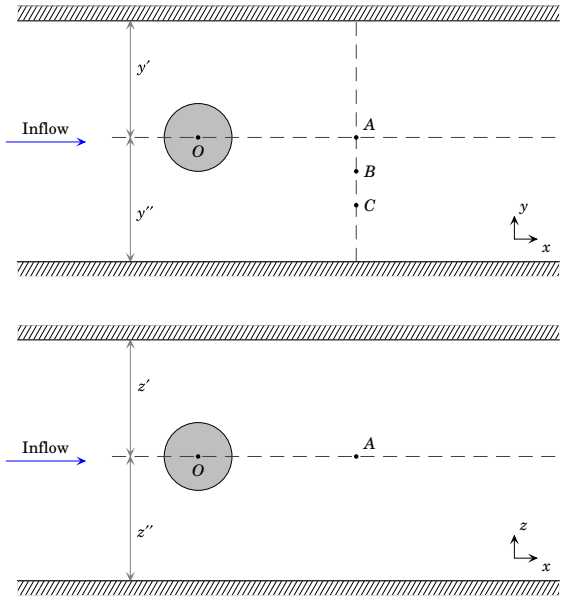


Figure 5: Simulation set-up

centre of the sphere is greater than $4D$, and the distance between the centre of the sphere and the outlet is greater than $13D$, D being the diameter of the sphere. The position of the sphere with respect to the stream cross section is symmetric in the z direction, and slightly asymmetric in the y direction,[2] in order to stabilise the flow pattern. Table 1 summarises the specifications of each configuration (for the sake of simplicity, we set $\delta x = 1$ and $\delta t = 1$).

For each configuration, the length of the channel is $L = 768$. We use SBB for the lateral walls and, as Yu *et al.* in [24], we impose the fully-developed boundary condition at the outlet, i.e.

$$\left|f_\alpha(\boldsymbol{x}, t)\right\rangle = \left|f_\alpha(\boldsymbol{x} - \boldsymbol{c}_\omega, t)\right\rangle,$$

where ω denotes the streamwise direction. The imposed velocity at the inlet is $u_0 = 0.05$, which correspond to a Mach number M ≈ 0.087, and the kinematic viscosity ν ranges from 2.5×10^{-3} to 4.3×10^{-3}.

In order to evaluate the vortex shedding frequency f, we recorded the flow velocity components at points A, B, and C such that $OA = 3.5D$ and $AB = BC = 0.5D$. The position of the control points in the streamwise direction is similar to the position of

[2] The deviation from the central position is about 1.3% of the total width

Configuration	C1	C2	C3	C4
Boundary condition	LIBB	SBB	LIBB	LIBB
Width of the channel	352	352	352	480
Diameter of the sphere	35.2	35.2	42	48

Table 1: Simulation configurations

the hot-wire probes for the experimental set-up described in [20]. The results are given in terms of Strouhal numbers, the Strouhal number being defined as $\mathrm{St} = fD/u_0$. We performed frequency analysis using fast Fourier transform on a 2^{19} sample, the overall number of time steps being 10^6. The size of the sample is about one hundred shedding periods considering the typical values of St reported for the Reynolds numbers we investigated. We considered the obtained Strouhal numbers significant when identical for the three control points and stable when sliding the sampling window from $t = 2.5 \times 10^5$ on. The results for the four configurations are gathered in Table 2.

Re	C1	C2	C3	C4
290	0.1343	0.1370	0.13779	0.1373
300	0.1370	—	0.14099	0.1392
310	0.1383	—	0.14259	0.1392
320	0.1396	—	0.14259	0.1410
330	0.1396	0.1396	0.14259	0.1410
340	0.1396	0.1396	0.14259	0.1392
350	0.1396	0.1370	0.14099	0.1373

Table 2: Strouhal numbers with respect to Reynolds number

5.2 Discussion

In their aforementioned work [16], Ormières and Provansal investigate the transition from steadiness to periodicity for the flow past a sphere. The reported value for the critical Reynolds number is $\mathrm{Re}_c = 280 \pm 5$. Our simulations are in agreement with this estimate. We performed simulations for configuration C1 at $\mathrm{Re} = 270$ and $\mathrm{Re} = 280$ and computed, for the last 5×10^5 time steps, the standard deviation σ_v of the cross-stream component v of the velocity in the y direction. In both cases, σ_v is less than 0.23% of u_0 at each tracking point and spectral analysis does not provide any relevant frequency.

Configuration C2 is identical to configuration C1 except for the boundary condition which is SBB instead of LIBB. The results for both configurations are rather close, yet for $\mathrm{Re} = 300$, $\mathrm{Re} = 310$ and $\mathrm{Re} = 320$, SBB appears to cause numerical instabilities. The obtained Strouhal numbers are neither stable in space, nor in time. The power spectral density diagram shows in general a low frequency peak, which may be caused by spectrum folding.

Configuration C4 is similar to configuration C1 in terms of blockage ratio $\beta = D/\ell = 0.1$, with higher grid resolution. The obtained Strouhal numbers for both configurations are in good agreement. The relative deviation is at most 2.2%, which assesses for the grid independence of our results.

The measurements in the experiments we chose as reference were carried out in wide tunnels, whereas the blockage ratio in our simulations is not negligible. According to Ota *et al.* [17], the Strouhal numbers for an unbounded flow St^* may be computed from the Strouhal numbers we obtained using:

$$\mathrm{St}^* = (1 - \beta\xi_{\mathrm{St}})\mathrm{St} \tag{10}$$

where ξ_{St} is a correction factor depending on the shape of the obstacle. To the best of our knowledge, the value of ξ_{St} in the case of a sphere was never mentioned in literature. In order to determine this value, we used the least square method with the fit given by Ormières and Provansal:

$$\mathrm{Ro} = -48.2 + 0.391 \times \mathrm{Re} - 3.6 \times 10^{-4} \times \mathrm{Re}^2 \tag{11}$$

where $\mathrm{Ro} = \mathrm{Re} \times \mathrm{St}$ is the Roshko number. Applying this procedure to the Roshko numbers computed for configuration C1 and for configuration C3 leads to $\xi_{\mathrm{St}} = 0.990 \pm 0.005$ for C1 and $\xi_{\mathrm{St}} = 1.000 \pm 0.005$ for C3. The blockage ratio in both configurations being different, this coincidence is in favor of the validity of our approach. As shown in Fig. 6, agreement of the corrected values with experimental data is satisfactory, the deviation from the fit being within 2.4%.

6 Performance results

We carried out our computations on a Tyan B7015 server with eight Tesla C1060 computing devices. For

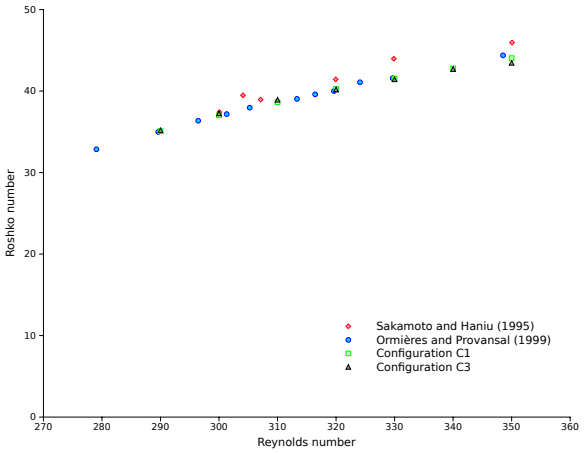


Figure 6: Variation of the Roshko number with the Reynolds number

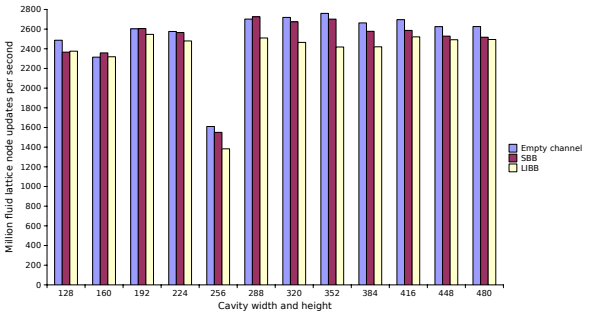


Figure 7: Performance comparison of SBB and LIBB implementations

the purpose of evaluating the efficiency of our LIBB implementation, we ran single precision simulations of a fluid flow in a square channel of size $768 \times \ell \times \ell$ with increasing ℓ, using three different configurations. For the first configuration, we used a SBB version our multi-GPU solver with an empty channel, to serve as a reference. For the second configuration, we also used the SBB version and the channel contains a sphere located at the centre of the channel, the blockage ratio being $\beta = 0.5$. The third configuration is identical to the second except for the boundary condition, which is LIBB instead of SBB. In addition, we used a modified version of our initialisation kernel to compute the number of fluid nodes, the number of boundary nodes, and the number of required additional read accesses. Figure 7 displays the obtained performance in million fluid lattice node updates per second (MFLUPS), which is a usual metric for LBM.

When comparing the first and the second configuration, we see that the branch divergences induced by the application of SBB have a significant impact on performance, with at most 4.9% loss (and even a slightly positive influence in some cases). It should be noted that this performance loss occurs even though no additional data is read from global memory. Performance for the third configuration is in general close to the one of the second configuration though inferior, the loss with respect to the first configuration being at most 14.1%. It is worth stressing that the supplementary global memory accesses required for LIBB are random individual transactions that break the regularity of the data access pattern. Moreover, with the GT200 GPU of the Tesla C1060 computing devices, i.e. with compute capability 1.3, the minimum data access is 32 bytes wide. Therefore the 0.13% boundary nodes corresponding to a blockage ratio $\beta = 0.5$ yield a 0.5% increase of the amount of data read.

With the Tesla C1060, the maximum sustained throughput for communication between GPU and device memory is 73.3 GB/s. Except for $\ell = 256$, the data throughput with the LIBB version is thus above 60% of its maximum. The quite low performance obtained with $\ell = 256$ is most probably due to partition camping effects [22]. For compute capability 1.3, global memory is split in eight 256 bytes wide memory banks [8]. With the data layout chosen for our implementation, the stride between corresponding nodes in distinct blocks is necessarily a multiple of the cavity width. For cavity width 256, concurrent blocks are therefore more likely to access the same memory bank simultaneously.

In order to investigate further the influence of our LIBB implementation on performance, we performed simulations on a $768 \times 352 \times 352$ cavity with a sphere of increasing diameter located at the center. We recorded performance for blockage ratios ranging from $\beta = 0.1$ to $\beta = 0.9$ as well as for an empty channel. Figure 8 reports the relative performance loss with respect to the relative variation of read data. The five leftmost points in the diagram correspond to blockage ratios ranging from $\beta = 0.1$ to $\beta = 0.5$, whereas the four rightmost points correspond to blockage ratios above $\beta = 0.5$.

For the two subsets, the relation between the two variables seems linear, the correlation coefficient be-

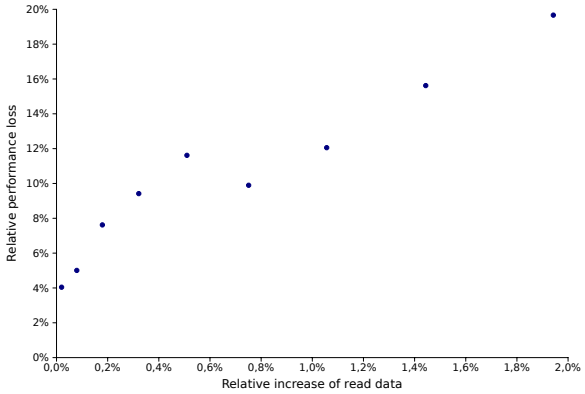


Figure 8: Performance loss with respect to increase of read data

ing 0.986 for the first one and 0.999 for the second one. The difference in terms of performance loss is due to the fact that in the first case, the sphere spans over two sub-domains whereas in the second case, it spans over four sub-domains. Moreover, it is worth mentioning that the slope of the second linear regression line is about one half of the slope of the first one, and that the y-intercept of both lines are about the same.

7 Conclusion

In the present work, we describe an implementation of the LIBB boundary condition within a multi-GPU LBM solver based on the TheLMA framework. When simulating the flow past a sphere in a channel, our solver allowed us to successfully compute a vortex shedding frequency for Reynolds numbers belonging to the regular mode flow pattern region. In addition, we could observe numerical instabilities occurring for some values of the Reynolds number when performing similar simulations with a SBB version of our code. The transition from steady to periodic flow appears to be in good agreement with the experimental value of the critical Reynolds number mentioned in literature.

The computed Strouhal numbers seem to agree with experimental results obtained for unbounded flows, provided the correction formula proposed by Ota *et al.* is valid in our case. To confirm this point, additional simulations with a wider set of blockage ratios must be carried out. The proposed approach proves to be efficient, with moderate impact on performance taking into account the sensitivity of CUDA enabled GPUs to irregular data access patterns. We are at present working on an extension of the TheLMA framework to multi-node multi-GPU hardware, providing more flexible sub-domain decomposition. This enhancement could contribute to improve the GPU load balance, and therefore reduce the overhead of boundary node processing.

References

[1] M. Bouzidi, M. Firdaouss, and P. Lallemand. Momentum transfer of a Boltzmann-lattice fluid with boundaries. *Physics of Fluids*, 13(11):3452–3459, 2001.

[2] D. d'Humières, I. Ginzburg, M. Krafczyk, P. Lallemand, and L.S. Luo. Multiple-relaxation-time lattice Boltzmann models in three dimensions. *Philosophical Transactions of the Royal Society A*, 360:437–451, 2002.

[3] Z. Fan, F. Qiu, A. Kaufman, and S. Yoakum-Stover. GPU cluster for high performance computing. In *Proceedings of the 2004 ACM/IEEE Conference on Supercomputing*, pages 47–58. IEEE, 2004.

[4] I. Ginzbourg and P. M. Adler. Boundary flow condition analysis for the three-dimensional lattice Boltzmann model. *Journal de Physique II*, 4(2):191–214, 1994.

[5] X. He and L.S. Luo. Theory of the lattice Boltzmann method: From the Boltzmann equation to the lattice Boltzmann equation. *Physical Review E*, 56(6):6811–6817, 1997.

[6] T. Inamuro, M. Yoshina, and F. Ogino. A non-slip boundary condition for lattice Boltzmann simulations. *Physic of Fluids*, 7(12):2928–2930, 1995.

[7] F. Kuznik, C. Obrecht, G. Rusaouën, and J.-J. Roux. LBM Based Flow Simulation Using GPU Computing Processor. *Computers and Mathematics with Applications*, 59(7):2380–2392, April 2010.

[8] D. Mudigere. Data access optimized applications on the GPU using NVIDIA CUDA. Master's thesis, Technische Universität München, 2009.

[9] NVIDIA. *Compute Unified Device Architecture Programming Guide version 4.0*, 2011.

[10] NVIDIA. *CUDA C Best Practices Guide version 4.0*, 2011.

[11] C. Obrecht, F. Kuznik, B. Tourancheau, and J.-J. Roux. Thermal LBM on Many-core Architectures. Available on www.thelma-project.info, 2010.

[12] C. Obrecht, F. Kuznik, B. Tourancheau, and J.-J. Roux. A new approach to the lattice Boltzmann method for graphics processing units. *Computers and Mathematics with Applications*, 61(12):3628–3638, 2011.

[13] C. Obrecht, F. Kuznik, B. Tourancheau, and J.-J. Roux. Global Memory Access Modelling for Efficient Implementation of the Lattice Boltzmann Method on Graphics Processing Units. In *Lecture Notes in Computer Science 6449, High Performance Computing for Computational Science, VECPAR 2010 Revised Selected Papers*, pages 151–161. Springer, 2011.

[14] C. Obrecht, F. Kuznik, B. Tourancheau, and J.-J. Roux. Multi-GPU implementation of the lattice Boltzmann method. *Computers and Mathematics with Applications*, (doi:10.1016/j.camwa.2011.02.020), 2011.

[15] C. Obrecht, F. Kuznik, B. Tourancheau, and J.-J. Roux. The TheLMA project: Multi-GPU implementation of the lattice Boltzmann method. *International Journal of High Performance Computing Applications*, 25(3):295–303, 2011.

[16] D. Ormières and M. Provansal. Transition to turbulence in the wake of a sphere. *Physical review letters*, 83(1):80–83, 1999.

[17] T. Ota, Y. Okamoto, and H. Yoshikawa. A correction formula for wall effects on unsteady forces of two-dimensional bluff bodies. *Journal of fluids engineering*, 116(3):414–418, 1994.

[18] C. Pan, L.-S. Luo, and C. T. Miller. An evaluation of lattice Boltzmann schemes for porous medium flow simulation. *Computers & Fluids*, 35(8-9):898–909, 2006.

[19] Y. H. Qian, D. d'Humières, and P. Lallemand. Lattice BGK models for Navier-Stokes equation. *EPL (Europhysics Letters)*, 17(6):479–484, 1992.

[20] H. Sakamoto and H. Haniu. A study on vortex shedding from spheres in a uniform flow. *ASME, Transactions, Journal of Fluids Engineering*, 112:386–392, 1990.

[21] H. Sakamoto and H. Haniu. The formation mechanism and shedding frequency of vortices from a sphere in uniform shear flow. *Journal of Fluid Mechanics*, 287:151–172, 1995.

[22] Y. Torres, A. Gonzalez-Escribano, and D.R. Llanos. Understanding the impact of CUDA tuning techniques for Fermi. In *Proceedings of the International Conference on High Performance Computing and Simulation*, pages 631–639. IEEE, 2011.

[23] J. Tölke and M. Krafczyk. TeraFLOP computing on a desktop PC with GPUs for 3D CFD. *International Journal of Computational Fluid Dynamics*, 22(7):443–456, 2008.

[24] H. Yu, L.S. Luo, and S.S. Girimaji. LES of turbulent square jet flow using an MRT lattice Boltzmann model. *Computers & Fluids*, 35(8):957–965, 2006.

Article I

Scalable Lattice Boltzmann Solvers for CUDA GPU Cluster

Submitted to *Parallel Computing*, August 22, 2012

Abstract

The lattice Boltzmann method (LBM) is an innovative and promising approach in computational fluid dynamics. From an algorithmic standpoint it reduces to a regular data parallel procedure and is therefore well-suited to high performance computations. Numerous works report efficient implementations of the LBM for the GPU, but very few mention multi-GPU versions and even fewer GPU cluster implementations. Yet, to be of practical interest, GPU LBM solvers need to be able to perform large scale simulations. In the present contribution, we describe an efficient LBM implementation for CUDA GPU clusters. Our solver consists of a set of MPI communication routines and a CUDA kernel specifically designed to handle three-dimensional partitioning of the computation domain. Performance measurement were carried out on a small cluster. We show that the results are satisfying, both in terms of data throughput and parallelisation efficiency.

Keywords: GPU clusters, CUDA, lattice Boltzmann method

1 Introduction

A single-GPU based computing device is not proper to solve large scale problems because of the limited amount of on-board memory. However, applications running on multiple GPUs have to face the PCI-E bottleneck, and great care has to be taken in design and implementation to minimise inter-GPU communication. Such constraints may be rather challenging; the well-known MAGMA [14] linear algebra library, for instance, only added support for single-node multiple GPUs with the latest version (i.e. 1.1), two years after the first public release.

The lattice Boltzmann method (LBM) is a novel approach in computational fluid dynamics (CFD), which, unlike most other CFD methods, does not consist in directly solving the Navier-Stokes equations by a numerical procedure [7]. Beside many interesting features, such as the ability to easily handle complex geometries, the LBM reduces to a regular data-parallel algorithm and therefore, is well-suited to efficient HPC implementations. As a matter of fact, numerous successful attempts to implement the LBM for the GPU have been reported in the recent years, starting with the seminal work of Li *et al.* in 2003 [8].

CUDA capable computation devices may at present manage up to 6 GB of memory. This capacity allows the GPU to process at most 8.5×10^7 nodes running a standard three-dimensional LBM solver in single-precision. Taking architectural constraints into account, the former amount is sufficient to store a 416^3 cubic lattice. Although large, such a computational domain is likely to be too coarse to perform direct numerical simulation of a fluid flow in many practical situations as, for instance, urban-scale building aeraulics or thermal modeling of electronic circuit boards.

To our knowledge, the few single-node multi-GPU LBM solvers described in literature all use a one-dimensional (1D) partition of the computation domain, which is relevant given the small number of involved devices. This option does not require any data reordering, provided the appropriate partitioning direction is chosen, thus keeping the computation kernel fairly simple. For a GPU cluster implementation, on the contrary, a kernel able to run on a three-dimensional (3D) partition seems preferable, since it would both provide more flexibility for load balancing and contribute to reduce the volume of communication.

In the present contribution, we describe an implementation of a lattice Boltzmann solver for CUDA

GPU clusters. The core computation kernel is designed so as to import and export data efficiently in each spatial direction, thus enabling the use of 3D partitions. The inter-GPU communication is managed by MPI-based routines. This work constitutes the latest extension to the TheLMA project [10], which aims at providing a comprehensive framework for efficient GPU implementations of the LBM.

The remainder of the paper is structured as follows. In Section 2, we give a description of the algorithmic aspects of the LBM as well as a short review of LBM implementations for the GPU. The third section consists of a detailed description of the implementation principles of the computation kernel and the communication routines of our solver. In the fourth section, we present some performance results on a small cluster. The last section concludes and discusses possible extensions to the present work.

2 State of the art

2.1 Lattice Boltzmann Method

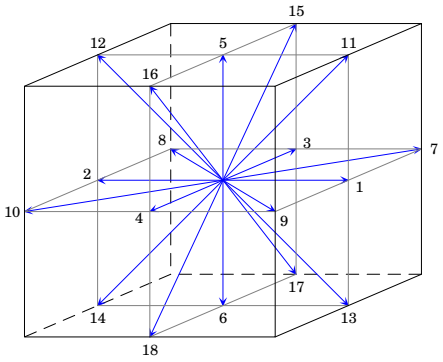


Figure 1: The D3Q19 stencil — *The blue arrows represent the propagation vectors of the stencil linking a given node to some of its nearest neighbours.*

The lattice Boltzmann method is generally carried out on a regular orthogonal mesh with a constant time step δt. Each node of the lattice holds a set of scalars $\{f_\alpha \,|\, \alpha = 0, \ldots N\}$ representing the local particle density distribution. Each particle density f_α is associated with a particle velocity ξ_α and a *propagation vector* $\boldsymbol{c}_\alpha = \delta t \cdot \xi_\alpha$. Usually the propagation vectors link a given node to one of its nearest neighbours, except for $\boldsymbol{c}_0$ which is null. For the present work, we implemented the D3Q19 propagation stencil illustrated in Fig. 1. This stencil, which contains 19 elements, is the most commonly used in practice for 3D LBM, being the best trade-off between size and isotropy. The governing equation of the LBM at node $\boldsymbol{x}$ and time t writes:

$$\left|f_\alpha(\boldsymbol{x}+\boldsymbol{c}_\alpha, t+\delta t)\right\rangle - \left|f_\alpha(\boldsymbol{x}, t)\right\rangle = \Omega\Big(\left|f_\alpha(\boldsymbol{x}, t)\right\rangle\Big), \quad (1)$$

where $\left|f_\alpha\right\rangle$ denotes the distribution vector and Ω denotes the so-called *collision operator*. The mass density ρ and the momentum $\boldsymbol{j}$ of the fluid are given by:

$$\rho = \sum_\alpha f_\alpha\,, \qquad \boldsymbol{j} = \sum_\alpha f_\alpha \xi_\alpha\,. \quad (2)$$

In our solver, we implemented the multiple-relaxation-time collision operator described in [3]. Further information on the physical and numerical aspects of the method are to be found in the aforementioned reference. From an algorithmic perspective, Eq. 1 naturally breaks in two elementary steps:

$$\left|\widetilde{f}_\alpha(\boldsymbol{x}, t)\right\rangle = \left|f_\alpha(\boldsymbol{x}, t)\right\rangle + \Omega\Big(\left|f_\alpha(\boldsymbol{x}, t)\right\rangle\Big), \quad (3)$$

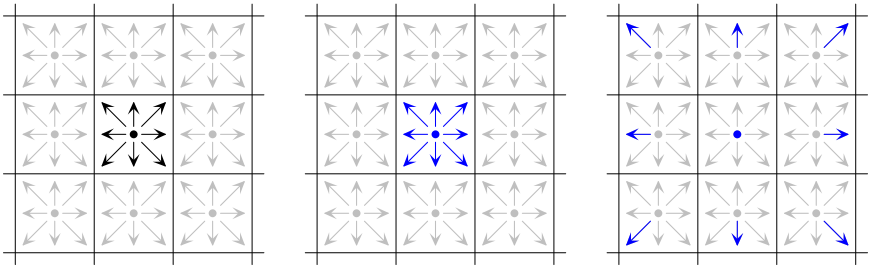
$$\left|f_\alpha(\boldsymbol{x}+\boldsymbol{c}_\alpha, t+\delta t)\right\rangle = \left|\widetilde{f}_\alpha(\boldsymbol{x}, t)\right\rangle. \quad (4)$$

Equation 3 describes the *collision* step in which an updated particle distribution is computed. Equation 4 describes the *propagation* step in which the updated particle densities are transferred to the neighbouring nodes. This two-step process is outlined by Fig. 2 (in the two-dimensional case, for the sake of clarity).

2.2 GPU implementations of the LBM

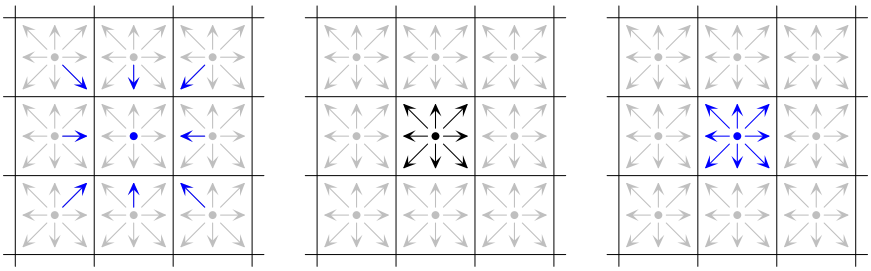
Due to substantial evolution of hardware, the pioneering work of Fan *et al.* [4] reporting a GPU cluster LBM implementation is only partially relevant today. The GPU computations were implemented using pre-CUDA techniques that are now obsolete. Yet, the proposed optimisation of the communication pattern still applies, although it was only tested on Gigabyte Ethernet; in future work, we plan to evaluate its impact using InfiniBand interconnect.

In 2008, Tölke and Krafczyk [15] described a single-GPU 3D-LBM implementation using CUDA. The authors mainly try to address the problem induced by misaligned memory accesses. As a matter



(a) Initial state (b) Post-collision state (c) Post-propagation state

Figure 2: Collision and propagation — *The collision step is represented by the transition between (a) and (b). The pre-collision particle distribution is drawn in black whereas the post-collision one is drawn in blue. The transition from (b) to (c) illustrates the propagation step in which the updated particle distribution is advected to the neighbouring nodes.*



(a) Initial state (b) Pre-collision state (c) Post-collision state

Figure 3: In-place propagation — *With the in-place propagation scheme, contrary to the out-of-place scheme outlined in Fig. 2, the updated particle distribution of the former time step is advected to the current node before collision.*

of fact, with the NVIDIA G80 GPU available at this time, only aligned and ordered memory transactions could be coalesced. The proposed solution consists in partially performing propagation in shared memory. With the GT200 generation, this approach is less relevant, since misalignment has a lower—though not negligible—impact on performance. As shown in [11], the misalignment overhead is significantly higher for store operations than for read operations. We therefore suggested in [12] to use the in-place propagation scheme outlined by Fig. 3 instead of the ordinary out-of-place propagation scheme illustrated in Fig. 2. The resulting computation kernel is simpler and leaves the shared memory free for possible extensions.

Further work led us to develop a single-node multi-GPU solver, with 1D partitioning of the computation domain [13]. Each CUDA device is managed by a specific POSIX thread. Inter-GPU communication is carried out using zero-copy transactions to page-locked host memory. Performance and scalability are satisfying with up to 2,482 million lattice node updates per second (MLUPS) and 90.5% parallelisation efficiency on a 384^3 lattice using eight Tesla C1060 computing devices in single-precision.

In their recent paper [16], Wang and Aoki describe an implementation of the LBM for CUDA GPU clusters. The partition of the computation domain may be either one-, two-, or three-dimensional. Although the authors are elusive on this point, no special care seems to be taken to optimise data transfer between device and host memory, and as a matter of fact, performance is quite low. For instance, on a 384^3 lattice with 1D partitioning, the authors report

about 862 MLUPS using eight GT200 GPUs in single-precision, i.e. about one third of the performance of our own solver on similar hardware. It should also be noted that the given data size for communication per rank, denoted M_{1D}, M_{2D}, and M_{3D}, are at least inaccurate. For the 1D and 2D cases, no account is taken of the fact that for the simple bounce-back boundary condition, no external data is required to process boundary nodes. In the 3D case, the proposed formula is erroneous.

3 Proposed implementation

3.1 Computation kernel

To take advantage of the massive hardware parallelism, our single-GPU and our single-node multi-GPU LBM solvers both assign one thread to each node of the lattice. The kernel execution set-up consists of a two-dimensional grid of one-dimensional blocks, mapping the spatial coordinates. The lattice is stored as a four-dimensional array, the direction of the blocks corresponding to the minor dimension. Two instances of the lattice are kept in device memory, one for even time steps and one for odd time steps, in order to avoid local synchronisation issues. The data layout allows the fetch and store operations issued by the warps to be coalesced. It also makes possible, using coalesced zero-copy transactions, to import and export data efficiently at the four sub-domain faces parallel to the blocks, with partial overlapping of communication and computations. For the two sub-domain faces normal to the blocks, the data is scattered across the array and needs reordering to be efficiently exchanged.

A possible solution to extend our computation kernel to support 3D partitions would be to use a specific kernel to handle the interfaces normal to the blocks. Not mentioning the overhead of kernel switching, this approach does not seem satisfying since such a kernel would only perform non-coalesced read operations. However, the minimum data access size is 32 bytes for compute capability up to 1.3, and 128 bytes above, whereas only 4 or 8 bytes would be useful. The cache memory available in devices of compute capability 2.0 and 2.1 is likely to have small impact in this case, taking into account the scattering of accessed data.

We therefore decided to design a new kernel able to perform propagation and reordering at once. With this new kernel, blocks are still one-dimensional but, instead of spanning the lattice width, contain only one warp, i.e. $W = 32$ threads for all existing CUDA capable GPUs. Each block is assigned to a tile of nodes of size $W \times W \times 1$, which imposes for the sub-domain dimensions to be a multiple of W in the x- and y-direction. For the sake of clarity, let us call *lateral densities* the particle densities crossing the tile sides parallel to the y-direction. These lateral densities are stored in an auxiliary array in device memory. At each time step, the lateral densities are first loaded from the auxiliary array into shared memory, then the kernel loops over the tile row by row to process the nodes saving the updated lateral densities in shared memory, last the updated lateral densities are written back to device memory. This process is summarised in Fig. 4. Note that we only drew a 8×8 tile in order to improve readability.

Using this new kernel, the amount of 4-byte (or 8-byte) words read or written per block and per time step is :

$$Q_{\mathrm{T}} = 2\left(19W^2 + 10W\right) = 38W^2 + 20W, \qquad (5)$$

and the amount of data read and written in device memory per time step for a $S_x \times S_y \times S_z$ sub-domain is:

$$Q_{\mathrm{S}} = \frac{S_x}{W} \times \frac{S_y}{W} \times S_z \times Q_{\mathrm{T}} = S_x S_y S_z \frac{38W + 20}{W}. \qquad (6)$$

We therefore see that this approach only increases the volume of device memory accesses by less than 2%, with respect to our former implementations [12], while greatly reducing the number of misaligned transactions. Yet, most important is the fact that it makes possible to exchange data efficiently at the interfaces normal to the blocks. The procedure simply consists in replacing, for the relevant tiles, accesses to the auxiliary array with coalesced zero-copy transactions to host memory. The maximum amount of data read and written to host memory per time step is:

$$\begin{aligned} Q_{\mathrm{H}} &= 2(S_x S_y + S_y S_z + S_z S_x) \times 5 \\ &= 10(S_x S_y + S_y S_z + S_z S_x). \end{aligned} \qquad (7)$$

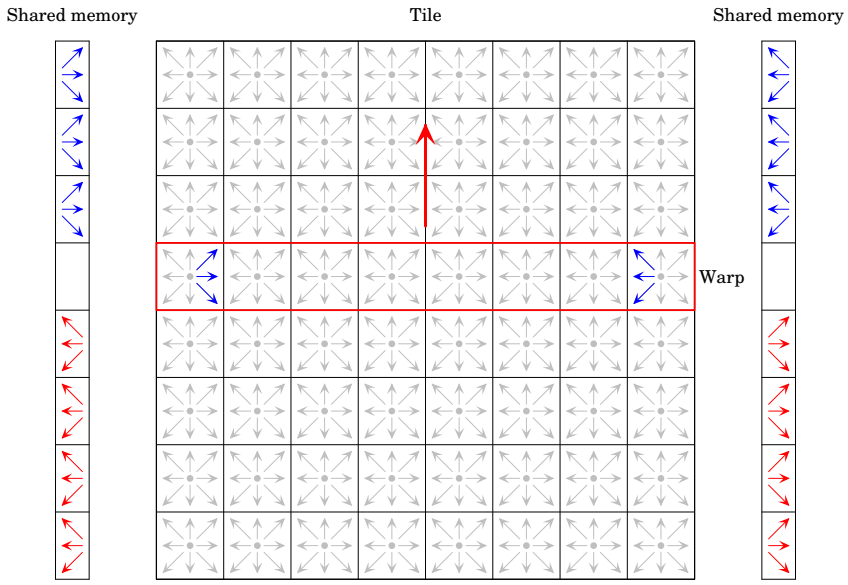


Figure 4: Processing of a tile — *Each tile of nodes is processed row by row by a CUDA block composed of a single warp. The current row is framed in red and the direction of the processing is indicated by the bold red arrow. The in-coming lateral densities are drawn in blue whereas the out-going ones are drawn in red. During the execution of the loop, these densities are stored temporarily in an auxiliary array hosted in shared memory.*

3.2 Multi-GPU solver

To enable our kernel to run across a GPU cluster, we wrote a set of MPI-based initialisation and communication routines. These routines as well as the new computation kernel were designed as components of the TheLMA framework, which was first developed for our single-node multi-GPU LBM solver. The main purpose of TheLMA is to improve code reusability. It comes with a set of generic modules providing the basic features required by a GPU LBM solver. This approach allowed us to develop our GPU cluster implementation more efficiently.

At start, the MPI process of rank 0 is responsible for loading a configuration file in JSON format. Beside general parameters, such as the Reynolds number for the flow simulation or the graphical output option flag, this configuration file mainly describes the execution set-up. Listing 1 gives an example file for a $2 \times 2 \times 1$ partition running on two nodes. The parameters for each sub-domains, such as the size or the target node and computing device, are given in the `Subdomains` array. The `Faces` and `Edges` arrays specify to which sub-domains a given sub-domain is linked, either through its faces or edges. These two arrays follow the same ordering as the propagation vector set displayed in Fig. 1. Being versatile, the JSON format is well-suited for our application. Moreover, its simplicity makes both parsing and automatic generation straightforward. This generic approach brings flexibility. It allows any LBM solver based on our framework to be tuned to the target architecture.

Once the configuration file is parsed, the MPI processes register themselves by sending their MPI processor name to the rank 0 process, which in turn assigns an appropriate sub-domain to each of them and sends back all necessary parameters. The processes then perform local initialisation, setting the assigned CUDA device and allocating the communication buffers, which fall into three categories: send buffers, receive buffers and read buffers. It is worth noting that both send buffers and read buffers consist of pinned memory allocated using the CUDA API, since they have to be made accessible by the GPU.

The steps of the main computation loop consist of a kernel execution phase and a communication phase. During the first phase, the out-going particle densities are written to the send buffers assigned to the

```
{
    "Path": "out",
    "Prefix": "ldc",
    "Re": 1E3,
    "U0": 0.1,
    "Log": true,
    "Duration": 10000,
    "Period": 100,
    "Images": true,
    "Subdomains": [
        {
              "Id": 0,
              "Host": "node00",
              "GPU": 0,
              "Offset": [0, 0, 0],
              "Size": [128, 128, 256],
              "Faces": [   1, null,    2, null, null, null],
              "Edges": [   3, null, null, null, null, null,
                        null, null, null, null, null, null]
        },
        {
              "Id": 1,
              "Host": "node00",
              "GPU": 1,
              "Offset": [128, 0, 0],
              "Size": [128, 128, 256],
              "Faces": [null,    0,    3, null, null, null],
              "Edges": [null,    2, null, null, null, null,
                        null, null, null, null, null, null]
        },
        {
              "Id": 2,
              "Host": "node01",
              "GPU": 0,
              "Offset": [0, 128, 0],
              "Size": [128, 128, 256],
              "Faces": [   3, null, null,    0, null, null],
              "Edges": [null, null,    1, null, null, null,
                        null, null, null, null, null, null]
        },
        {
              "Id": 3,
              "Host": "node01",
              "GPU": 1,
              "Offset": [128, 128, 0],
              "Size": [128, 128, 256],
              "Faces": [null,    2, null,    1, null, null],
              "Edges": [null, null, null,    0, null, null,
                        null, null, null, null, null, null]
        }
    ]
}
```

Listing 1: Configuration file

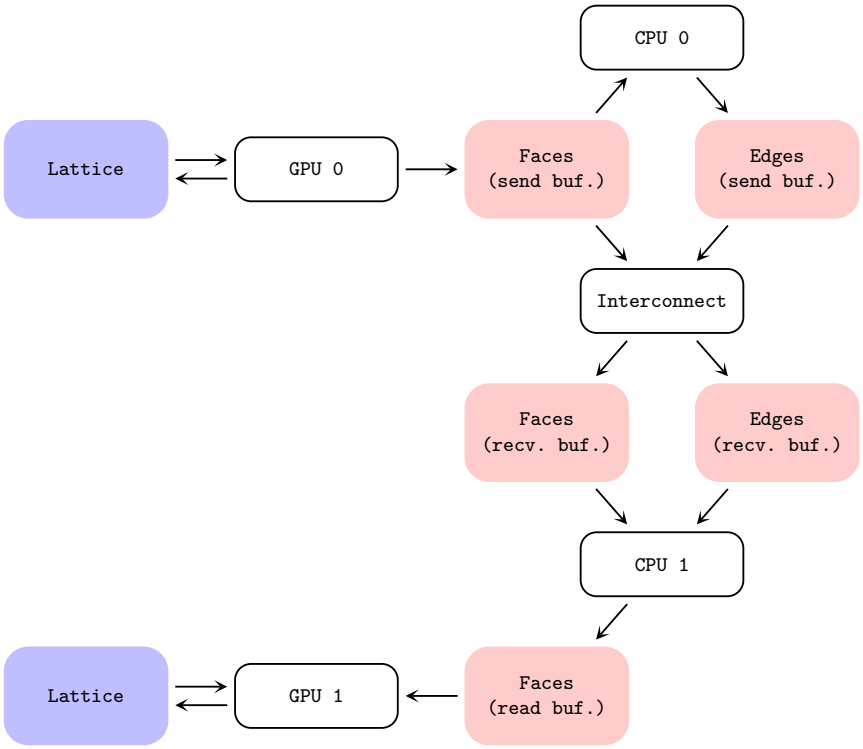


Figure 5: Communication phase — *The upper part of the graph outlines the path followed by data leaving the sub-domain handled by GPU 0. For each face of the sub-domain, the out-going densities are written by the GPU to pinned buffers in host memory. The associated MPI process then copies the relevant densities into the edge buffers and sends both face and edge buffers to the corresponding MPI processes. The lower part of the graph describes the path followed by data entering the sub-domain handled by GPU 1. Once the reception of in-coming densities for faces and edges is completed, the associated MPI process copies the relevant data for each face of the sub-domain into pinned host memory buffers, which are read by the GPU during kernel execution.*

faces *without* performing any propagation as for the densities written in device memory. During the second phase, the following operations are performed:

1. The relevant densities are copied to the send buffers assigned to the edges.
2. Asynchronous send requests followed by synchronous receive requests are issued.
3. Once message passing is completed, the particle densities contained in the receive buffers are copied to the read buffers.

This communication phase is outlined in Fig. 5. The purpose of the last operation is to perform propagation for the in-coming particle densities. As a result, the data corresponding to a face and its associated edges is gathered in a single read buffer. This approach avoids misaligned zero-copy transactions, and most important, leads to a simpler kernel since only six buffers at most have to be read. It should be mentioned that the read buffers are allocated using the *write combined* flag to optimise cache usage. According to [9, 6, 5], this setting is likely to improve performance since the memory pages are locked.

4 Performance study

We conducted experiments on an eight-node GPU cluster, each node being equipped with two hexa-core X5650 Intel Xeon CPUs, 36 GB memory, and three NVIDIA Tesla M2070 computing devices; the network interconnect uses QDR InfiniBand. To evaluate raw performance, we simulated a lid-driven cavity [1] in single-precision and recorded execution times for 10,000 time steps using various configurations. Overall performance is good, with at most 8,928 million lattice node updates per second (MLUPS) on a 768^3 lattice using all 24 GPUs. To set a comparison, Wang and Aoki in [16] report at most 7,537 MLUPS for the same problem size using four time as many GPUs. However, it should be mentioned that these results were obtained using hardware of the preceding generation.

The solver was compiled using CUDA 4.0 and OpenMPI 1.4.4. It is also worth mentioning that the computing devices had ECC support enabled. From tests we conducted on a single computing device, we expect the overall performance to be about 20% higher with ECC support disabled.

4.1 Performance model

Our first performance benchmark consisted in running our solver using eight GPUs on a cubic cavity of increasing size. The computation domain is split in a $2 \times 2 \times 2$ regular partition, the size S of the sub-domains ranging from 128 to 288. In addition, we recorded the performance for a single-GPU on a domain of size S, in order to evaluate the communication overhead and the GPU to device memory data throughput. The results are gathered in Tables 1 and 2.

Table 1 shows that the data throughput between GPU and device memory is stable, only slightly increasing with the size of the domain. (Given the data layout in device memory, the increase of the domain size is likely to reduce the amount of L2 cache misses, having therefore a positive impact on data transfer.) We may therefore conclude that the performance of our kernel is communication bound. The last column accounts for the ratio of the data throughput to the maximum sustained throughput, for which we used the value 102.7 GB/s obtained using the `bandwidthTest` program that comes with the CUDA SDK. The obtained ratios are fairly satisfying taking into account the complex data access pattern the kernel must follow.

In Tab. 2, the parallel efficiency and the non-overlapped communication time were computed using the single-GPU results. The efficiency is good with at least 87.3%. The corresponding data throughput given in the last column is slightly increasing from 8.2 GB/s to 9.9 GB/s, which implies that the proportion of communication overhead decreases when the domain size increases. The efficiency appears to benefit from surface-to-volume effects. Figure 6 displays the obtained performance results.

4.2 Scalability

In order to study scalability, both weak and strong, we considered seven different partition types with increasing number of sub-domains. Weak scalability represents the ability to solve larger problems with larger resources whereas strong scalability accounts for the ability to solve a problem faster using more resources. For weak scalability, we used cubic sub-domains of size 128, and for strong scalability, we used a computation domain of constant size 384 with

Domain size (S)	Runtime (s)	Performance (MLUPS)	Device throughput (GB/s)	Ratio to peak throughput
128	54.7	383.2	59.2	57.6%
160	100.6	407.2	62.9	61.2%
192	167.6	422.3	65.2	63.5%
224	260.3	431.8	66.7	64.9%
256	382.3	438.8	67.8	66.0%
288	538.7	443.4	68.5	66.7%

Table 1: Single-GPU performance

Domain size ($2S$)	Runtime (s)	Performance (MLUPS)	Parallel efficiency	Data transfer (s)	Throughput (GB/s)
256	62.7	2,678	87.3%	7.9	9.9
320	114.5	2,862	87.9%	13.9	8.9
384	186.9	3,030	89.7%	19.3	9.6
448	289.6	3,105	89.9%	29.3	8.2
512	418.7	3,206	91.3%	36.4	8.6
576	587.0	3,256	91.8%	48.3	8.2

Table 2: Performance for a $2 \times 2 \times 2$ regular partition

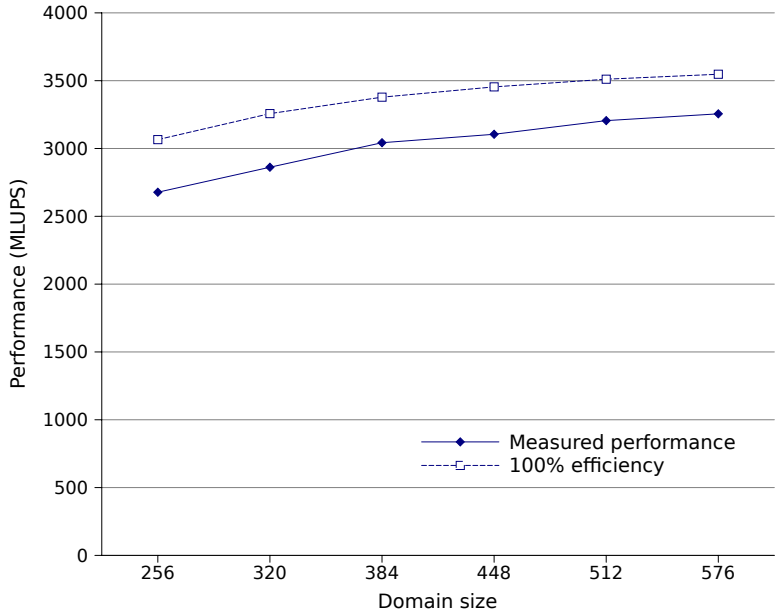


Figure 6: Performance for a $2 \times 2 \times 2$ regular partition

cuboid sub-domains. Table 3 gives all the details of the tested configurations.

For our weak scaling test, we use fixed size sub-domains so that the amount of processed nodes linearly increases with the number of GPUs. We chose a small, although realistic, sub-domain size in order to reduce as much as possible favourable surface-to-volume effects. Since the workload per GPU is fixed, perfect scaling is achieved when the runtime remains constant. The results of the test are gathered in Tab. 4. Efficiency was computed using the runtime of the smallest tested configuration. Figure 7 displays the runtime with respect to the number of GPUs. As illustrated by this diagram, the weak scalability of our solver is satisfying, taking into account that the volume of communication increases by a factor up to 11.5.

In our strong scalability test, we consider a fixed computation domain processed using an increasing number of computing devices. As a consequence the volume of the communication increases by a factor up to three, while the size of the sub-domains decreases, leading to less favourable configurations for the computation kernel. The results of the strong scaling test are given in Tab. 5. The runtime with respect to the number of GPUs is represented in Fig. 8 using a log-log diagram. As shown by the trend-line, the runtime closely obeys a power law, the correlation coefficient for the log-log regression line being below -0.999. The obtained scaling exponent is approximately -0.8, whereas perfect strong scalability corresponds to an exponent of -1. We may conclude that the strong scalability of our code is good, given the fairly small size of the computation domain.

5 Conclusion

In this paper, we describe the implementation of an efficient and scalable LBM solver for GPU clusters. Our code lies upon three main components that were developed for that purpose: a CUDA computation kernel, a set of MPI initialisation routines, and a set of MPI communication routines. The computation kernel's most important feature is the ability to efficiently exchange data in all spatial directions, making possible the use of 3D partitions of the computation domain. The initialisation routines are designed in order to distribute the workload across the cluster in a flexible way, following the specifications contained in a configuration file. The communication routines manage to pass data between sub-domains efficiently, performing reordering and partial propagation. These new components were devised as key parts of the TheLMA framework[10], whose main purpose is to facilitate the development of LBM solvers for the GPU. The obtained performance on rather affordable hardware such as small GPU clusters makes possible to carry out large scale simulations in reasonable time and at moderate cost. We believe these advances will benefit to many potential applications of the LBM. Moreover, we expect our approach to be sufficiently generic to apply to a wide range of stencil computations, and therefore to be suitable for numerous applications that operate on a regular grid.

Although performance and scalability of our solver is good, we believe there is still room for improvement. Possible enhancements include better overlapping between communication and computation, and more efficient communication between sub-domains. As for now, only transactions to the send and read buffers may overlap kernel computations. The communication phase starts once the computation phase is completed. One possible solution to improve overlapping would be to split the sub-domains in seven zones, six external zones, one for each face of the sub-domains, and one internal zone for the remainder. Processing the external zones first would allow the communication phase to start while the internal zone is still being processed.

Regarding ameliorations to the communication phase, we are considering three paths to explore. First of all, we plan to reinvest the concepts presented in [6] and [5] to improve data transfers involving page-locked buffers. Secondly, we intend to evaluate the optimisation proposed by Fan *et al.* in [4], which consists in performing data exchange in several synchronous steps, one for each face of the sub-domains, the data corresponding to the edges being transfered in two steps. Last, following [2], we plan to implement a benchmark program able to search heuristically efficient execution layouts for a given computation domain and to generate automatically the configuration file corresponding to the most efficient one.

Number of GPUs	Nodes × GPUs	Partition type	Domain (weak scalability)	Sub-domains (strong scalability)
4	2 × 2	1 × 2 × 2	128 × 256 × 256	384 × 192 × 192
6	2 × 3	1 × 3 × 2	128 × 384 × 256	384 × 128 × 192
8	4 × 2	2 × 2 × 2	256 × 256 × 256	192 × 192 × 192
12	4 × 3	2 × 3 × 2	256 × 384 × 256	192 × 128 × 192
16	8 × 2	2 × 4 × 2	256 × 512 × 256	192 × 96 × 192
18	6 × 3	2 × 3 × 3	256 × 384 × 384	192 × 128 × 128
24	8 × 3	2 × 4 × 3	256 × 512 × 384	192 × 96 × 128

Table 3: Configuration details for the scaling tests

Number of GPUs	Runtime (s)	Efficiency	Performance (MLUPS)	Perf. per GPU (MLUPS)
4	59.8	100%	1402	350.5
6	64.2	93%	1959	326.6
8	62.7	95%	2676	334.5
12	66.8	90%	3767	313.9
16	71.1	84%	4721	295.1
18	67.0	89%	5634	313.0
24	73.2	82%	6874	286.4

Table 4: Runtime and efficiency for the weak scaling test

Number of GPUs	Runtime (s)	Efficiency	Performance (MLUPS)	Perf. per GPU (MLUPS)
4	335.0	100%	1690	422.6
6	241.9	92%	2341	390.1
8	186.1	90%	3043	380.3
12	134.7	83%	4204	350.3
16	109.9	76%	5152	322.0
18	98.4	76%	5753	319.6
24	80.3	70%	7053	293.9

Table 5: Runtime and efficiency for the strong scaling test

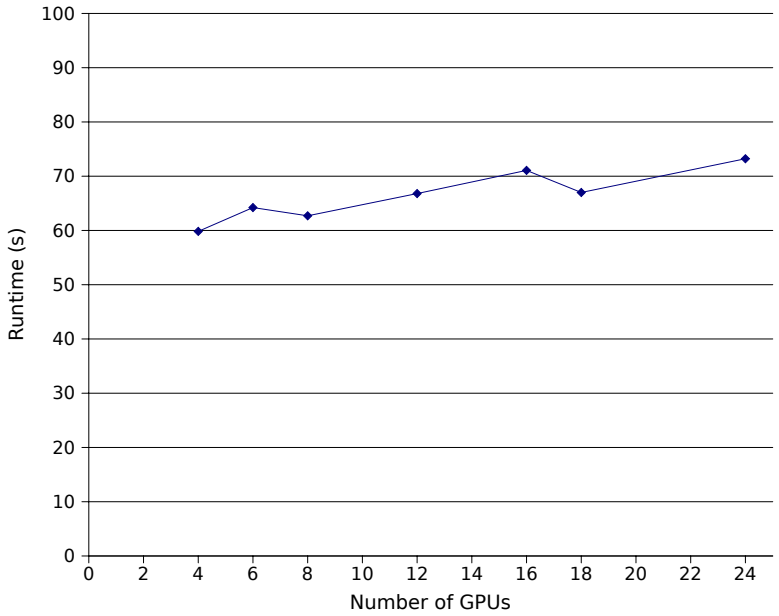


Figure 7: Runtime for the weak scaling test — *Perfect weak scaling would result in an horizontal straight line.*

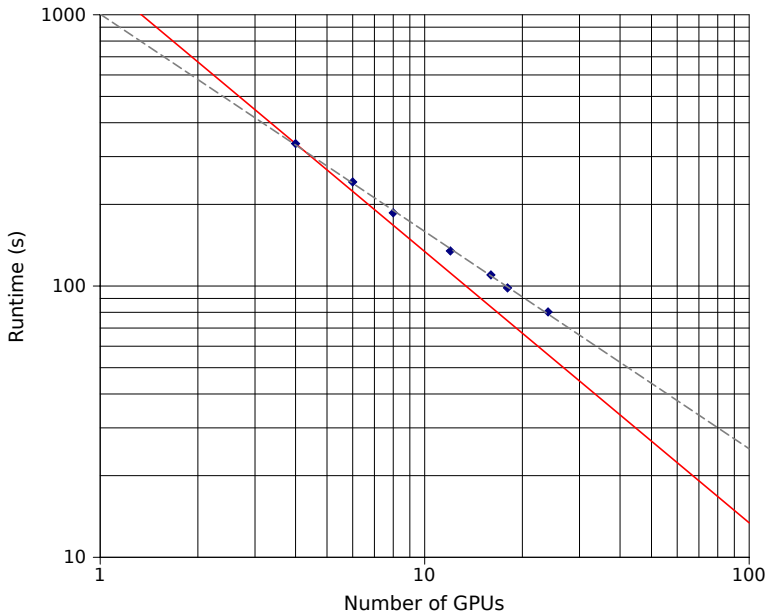


Figure 8: Runtime for the strong scaling test — *Perfect strong scaling is indicated by the solid red line*

Acknowledgments

The authors wish to thank the INRIA PlaFRIM team for allowing us to test our executables on the Mirage GPU cluster.

References

[1] S. Albensoeder and H. C. Kuhlmann. Accurate three-dimensional lid-driven cavity flow. *Journal of Computational Physics*, 206(2):536–558, 2005.

[2] R. Clint Whaley, A. Petitet, and J.J. Dongarra. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, 27(1):3–35, 2001.

[3] D. d'Humières, I. Ginzburg, M. Krafczyk, P. Lallemand, and L.S. Luo. Multiple-relaxation-time lattice Boltzmann models in three dimensions. *Philosophical Transactions of the Royal Society A*, 360:437–451, 2002.

[4] Z. Fan, F. Qiu, A. Kaufman, and S. Yoakum-Stover. GPU cluster for high performance computing. In *Proceedings of the 2004 ACM/IEEE Conference on Supercomputing*, pages 47–58. IEEE, 2004.

[5] P. Geoffray, C. Pham, and B. Tourancheau. A Software Suite for High-Performance Communications on Clusters of SMPs. *Cluster Computing*, 5(4):353–363, 2002.

[6] P. Geoffray, L. Prylli, and B. Tourancheau. BIP-SMP: High performance message passing over a cluster of commodity SMPs. In *Proceedings of the 1999 ACM/IEEE Conference on Supercomputing*, pages 20–38. ACM, 1999.

[7] X. He and L.S. Luo. Theory of the lattice Boltzmann method: From the Boltzmann equation to the lattice Boltzmann equation. *Physical Review E*, 56(6):6811–6817, 1997.

[8] W. Li, X. Wei, and A. Kaufman. Implementing lattice Boltzmann computation on graphics hardware. *The Visual Computer*, 19(7):444–456, 2003.

[9] NVIDIA. *Compute Unified Device Architecture Programming Guide version 4.0*, 2011.

[10] C. Obrecht, F. Kuznik, B. Tourancheau, and J.-J. Roux. Thermal LBM on Many-core Architectures. Available on www.thelma-project.info, 2010.

[11] C. Obrecht, F. Kuznik, B. Tourancheau, and J.-J. Roux. A new approach to the lattice Boltzmann method for graphics processing units. *Computers and Mathematics with Applications*, 61(12):3628–3638, 2011.

[12] C. Obrecht, F. Kuznik, B. Tourancheau, and J.-J. Roux. Global Memory Access Modelling for Efficient Implementation of the Lattice Boltzmann Method on Graphics Processing Units. In *Lecture Notes in Computer Science 6449, High Performance Computing for Computational Science, VECPAR 2010 Revised Selected Papers*, pages 151–161. Springer, 2011.

[13] C. Obrecht, F. Kuznik, B. Tourancheau, and J.-J. Roux. The TheLMA project: Multi-GPU implementation of the lattice Boltzmann method. *International Journal of High Performance Computing Applications*, 25(3):295–303, 2011.

[14] F. Song, S. Tomov, and J. Dongarra. Efficient Support for Matrix Computations on Heterogeneous Multi-core and Multi-GPU Architectures. Technical Report UT-CS-11-668, University of Tennessee, 2011.

[15] J. Tölke and M. Krafczyk. TeraFLOP computing on a desktop PC with GPUs for 3D CFD. *International Journal of Computational Fluid Dynamics*, 22(7):443–456, 2008.

[16] X. Wang and T. Aoki. Multi-GPU performance of incompressible flow computation by lattice Boltzmann method on GPU cluster. *Parallel Computing*, 37(9):521–535, 2011.

FOLIO ADMINISTRATIF

THÈSE SOUTENUE DEVANT L'INSTITUT NATIONAL DES SCIENCES APPLIQUÉES DE LYON

NOM : OBRECHT DATE de SOUTENANCE : 11 décembre 2012
(avec précision du nom de jeune fille, le cas échéant)
Prénoms : Christian, Charles

TITRE : High Performance Lattice Boltzmann Solvers on Massively Parallel Architectures with Applications to Building Aeraulics

NATURE : Doctorat Numéro d'ordre : 2012ISAL0134

École doctorale : MEGA (mécanique, énergétique, génie civil, acoustique)

Spécialité : Génie civil

RÉSUMÉ :

Avec l'émergence des bâtiments à haute efficacité énergétique, il est devenu indispensable de pouvoir prédire de manière fiable le comportement énergétique des bâtiments. Or, à l'heure actuelle, la prise en compte des effets thermo-aérauliques dans les modèles se cantonne le plus souvent à l'utilisation d'approches simplifiées voire empiriques qui ne sauraient atteindre la précision requise. Le recours à la simulation numérique des écoulements semble donc incontournable, mais il est limité par un coût calculatoire généralement prohibitif. L'utilisation conjointe d'approches innovantes telle que la méthode de Boltzmann sur gaz réseau (LBM) et d'outils de calcul massivement parallèles comme les processeurs graphiques (GPU) pourrait permettre de s'affranchir de ces limites. Le présent travail de recherche s'attache à en explorer les potentialités.

La méthode de Boltzmann sur gaz réseau, qui repose sur une forme discrétisée de l'équation de Boltzmann, est une approche explicite qui jouit de nombreuses qualités : précision, stabilité, prise en compte de géométries complexes, etc. Elle constitue donc une alternative intéressante à la résolution directe des équations de Navier-Stokes par une méthode numérique classique. De par ses caractéristiques algorithmiques, elle se révèle bien adaptée au calcul parallèle. L'utilisation de processeurs graphiques pour mener des calculs généralistes est de plus en plus répandue dans le domaine du calcul intensif. Ces processeurs à l'architecture massivement parallèle offrent des performances inégalées à ce jour pour un coût relativement modéré. Néanmoins, nombre de contraintes matérielles en rendent la programmation complexe et les gains en termes de performances dépendent fortement de la nature de l'algorithme considéré. Dans le cas de la LBM, les implantations GPU affichent couramment des performances supérieures de deux ordres de grandeur à celle d'une implantation CPU séquentielle faiblement optimisée.

Le présent mémoire de thèse est constitué d'un ensemble de neuf articles de revues internationales et d'actes de conférences internationales (le dernier étant en cours d'évaluation). Dans ces travaux sont abordés les problématiques liées tant à l'implantation mono-GPU de la LBM et à l'optimisation des accès en mémoire, qu'aux implantations multi-GPU et à la modélisation des communications inter-GPU et inter-nœuds. En complément, sont détaillées diverses extensions à la LBM indispensables pour envisager une utilisation en thermo-aéraulique des bâtiments. Les cas d'études utilisés pour la validation des codes permettent de juger du fort potentiel de cette approche en pratique.

MOTS-CLEFS : calcul intensif, méthode de Boltzmann sur gaz réseau, processeurs graphiques, aéraulique des bâtiments.

Laboratoire(s) de recherche : CETHIL (Centre de Thermique de Lyon)

Directeur de thèse : Jean-Jacques ROUX

Président de jury : Christian INARD

Composition du jury : Jean-Luc HUBERT, Christian INARD, Manfred KRAFCZYK, Frédéric KUZNIK, Jean ROMAN, Jean-Jacques ROUX, Bernard TOURANCHEAU