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

C. Obrecht, F. Kuznik, Bernard Tourancheau, J.-J. Roux

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

Christian Obrecht*[,1,2,3], Frédéric Kuznik[2,3], Bernard Tourancheau[2,4,5], and Jean-Jacques Roux[2,3]

[1]EDF R&D, Département EnerBAT, F-77818 Moret-sur-Loing Cedex, France
[2]Université de Lyon, F-69361 Lyon Cedex 07, France
[3]INSA-Lyon, CETHIL, UMR5008, F-69621 Villeurbanne Cedex, France
[4]INSA-Lyon, CITI, INRIA, F-69621 Villeurbanne Cedex, France
[5]Université Lyon 1, F-69622 Villeurbanne Cedex, France

December 10, 2010

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.

1 Introduction

First introduced by McNamara and Zanetti in 1988 [11], 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 becomes more and more widespread [2, 6]. Several successful single-GPU implementations of the LBM using CUDA were reported during the past years [9, 19]. 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.

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

2 LBM Flow Solvers

Although first introduced as an extension to the lattice gas automata method [8], the LBM is nowadays more commonly interpreted as a discrete numerical procedure to solve the Boltzmann transport equation:

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

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, 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:

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

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

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 $\boldsymbol{\xi}_0 = \boldsymbol{0}$, 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.

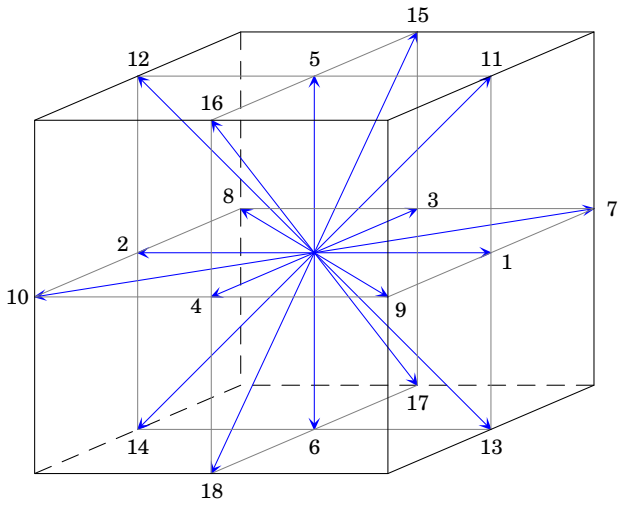


Figure 1: D3Q19 stencil

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:

$$\big|f_i(\boldsymbol{x} + \delta t\, \boldsymbol{\xi}_i, t + \delta t)\big\rangle - \big|f_i(\boldsymbol{x}, t)\big\rangle = \Omega\Big(\big|f_i(\boldsymbol{x}, t)\big\rangle\Big) \tag{4}$$

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

$$\rho = \sum_{i=0}^{N} f_i \tag{5}$$

[1] Commonly, only nearest neighbors are linked.

$$\rho \boldsymbol{u} = \sum_{i=0}^{N} f_i \boldsymbol{\xi}_i \tag{6}$$

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 [4, 5]. Equation 4 naturally breaks in two elementary steps:

$$\left|\tilde{f}_i(\boldsymbol{x},t)\right\rangle = \left|f_i(\boldsymbol{x},t)\right\rangle + \Omega\Big(\left|f_i(\boldsymbol{x},t)\right\rangle\Big) \tag{7}$$

$$\left|f_i(\boldsymbol{x}+\delta t\,\boldsymbol{\xi}_i, t+\delta t)\right\rangle = \left|\tilde{f}_i(\boldsymbol{x},t)\right\rangle \tag{8}$$

Equation 7 describes the *collision* step in which an updated particle distribution is computed, and Eq. 8 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 global synchronization 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, 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. 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 [12]. 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 as illustrated in Fig. 2 (in the two-dimensional case for the sake of simplicity).

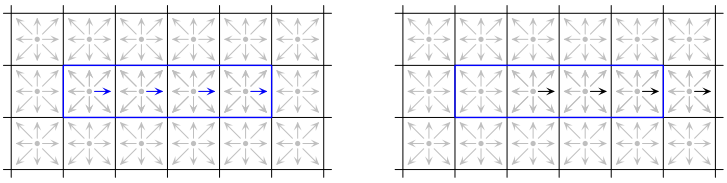
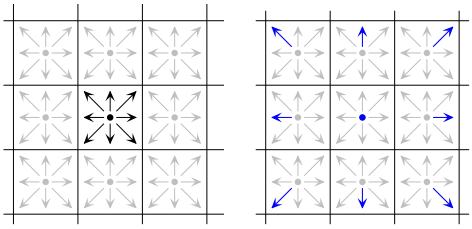
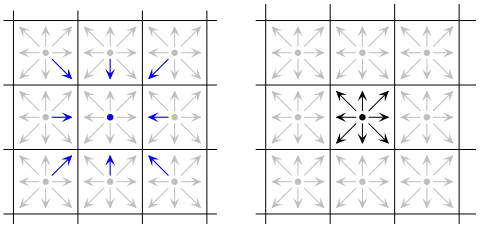


Figure 2: Misalignment issue

For CUDA devices of compute capability 1.0 or 1.1, misaligned memory transactions may have dramatic 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 transactions as possible. An alternative approach for this kind of hardware consists in using in-place propagation instead of out-of-place propagation. Figure 3 illustrates the two propagation scheme.



(a) Out-of-place propagation



(b) In-place propagation

Figure 3: Propagation schemes

With the GT200, misaligned reads are far less expensive than misaligned writes [14]. Thus, the in-place propagation approach in only slightly less efficient than the shared memory method for simulations in three dimensions [13], while being simpler to implement and exerting less pressure

on hardware.

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. As outlined by Fig. 4, each interface between sub-domains is associated to four buffers: two for in-coming populations and two for out-going populations. Pointers are switched at each time step.

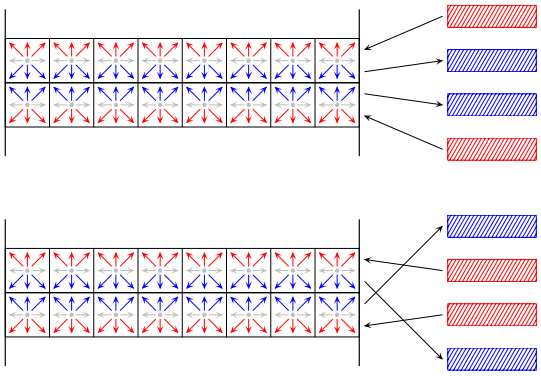


Figure 4: Inter-GPU communication scheme

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 handles at most eight sub-domains. We therefore chose to split the cavity in rectangular cuboids along one direction orthogonal to the blocks.

This 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 cavity, 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 $m = 15$.

From a software engineering perspective, the CUDA technology is a great improvement over the early days of GPGPU [7]. 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 for instance, makes difficult to follow an incremental, library oriented approach. To improve code reusability, we therefore developed the TheLMA framework [1]. TheLMA stands

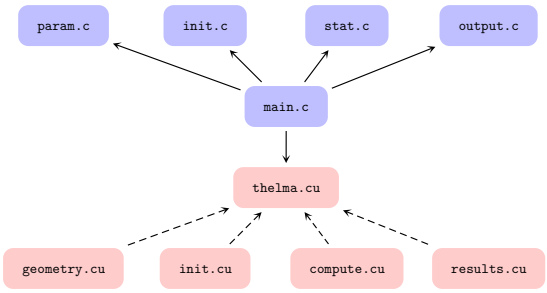


Figure 5: The TheLMA framework

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 in 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 initialisation 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.

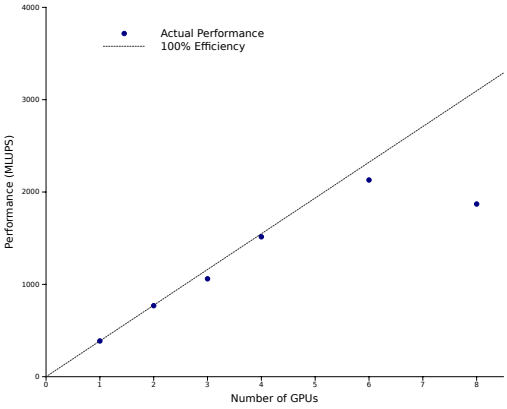
The source organisation we describe is meant to ease subsequent code changing. For instance, the implementation of a different collision model or propagation scheme would mainly require the modification of the computation module. Likewise, the setting up of a given simulation configuration principally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

5 Performance and Scalability

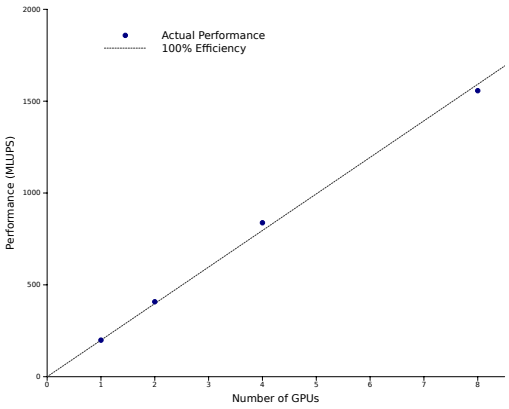
To validate our code, we implemented the lid-driven cubic cavity test case in which a constant velocity is imposed at the top lid, whereas the other walls have null velocity boundary condition. We chose to assign the x direction to the blocks and to split the cavity along the z direction. Performance for LBM solvers is usually given in MLUPS (Million Lattice node Updates Per Second). We measured performance in single precision for a 192^3 and a 256^3 lattice on a Tyan B7015 server with 8 Tesla C1060 computing devices. 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 result is comparable to the one obtained with the acknowledged Palabos code in double precision on a Blue Gene/P computer using 4,096 cores [10]. 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 optimal scalability.

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 in our case, we may use the mean of these two values, i.e. 4.58 GB/s, as a lower



(a) Performance on a 192^3 lattice



(b) Performance on a 256^3 lattice

Figure 6: Performance and scalability on a cubic lattice

Number of GPUs	1	2	3	4	6	8
Kernel duration (ms)	18.29	9.14	6.10	4.57	3.05	2.29
Exchanged data (MB)	0	2.95	5.90	8.85	14.75	20.64
Required throughput (GB/s)	0	0.32	0.97	1.93	4.84	9.03

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
Exchanged data (MB)	0	5.24	15.73	36.70
Required throughput (GB/s)	0	0.12	0.75	3.48

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

bound 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 at most comparable to this bound 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.

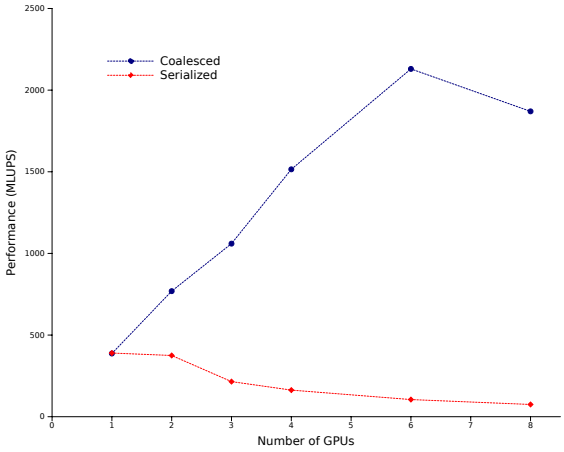


Figure 7: Performance for coalesced and serialized communication

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.

6 Inter-GPU Communication

In order to get better insight into inter-GPU communication, we implemented a benchmark program based on a stripped-off 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 $k = 5$ and $L = 192$. 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

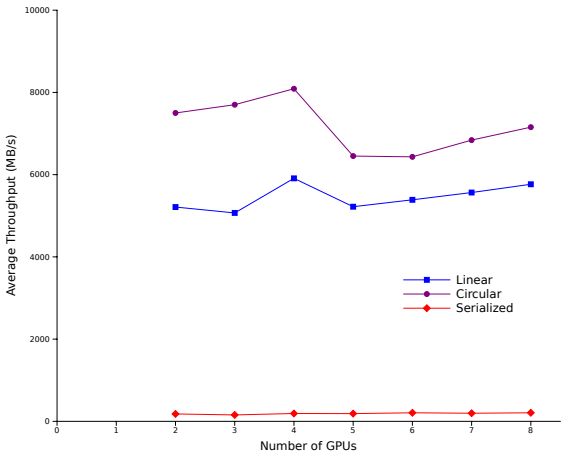


Figure 8: Inter-GPU communication throughput

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:

$$P = \ell^3 \times \frac{T_i}{2(i-1)D \times \ell^2} = \frac{T_i \times \ell}{2(i-1)D} \tag{9}$$

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.

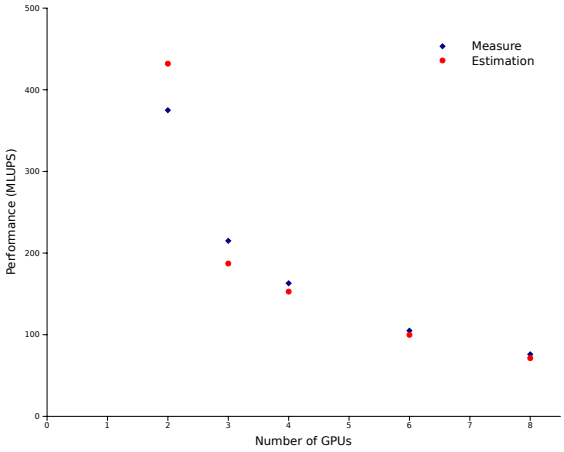


Figure 9: Performance model for serialized communication

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

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 [3], would help performing further investigation and could be a first step for adding dynamic auto-tuning in our framework.

References

[1] Thermal LBM on Many-core Architectures. www.thelma-project.info.

[2] 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, page 012037. IOP Publishing, 2009.

[3] 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.

[4] D. d'Humières. Generalized lattice-Boltzmann equations. *Rarefied gas dynamics- Theory and simulations*, pages 450–458, 1994.

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

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

[7] 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*, page 47. IEEE Computer Society, 2004.

[8] U. Frisch, B. Hasslacher, and Y. Pomeau. Lattice-Gas Automata for the Navier-Stokes Equation. *Phys. Rev. Lett.*, 56(14):1505–1508, 1986.

[9] F. Kuznik, C. Obrecht, G. Rusaouën, and J.-J. Roux. LBM Based Flow Simulation Using GPU Computing Processor. *Computers and Mathematics with Applications*, (27), June 2009.

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

[11] G. R. McNamara and G. Zanetti. Use of the Boltzmann Equation to Simulate Lattice-Gas Automata. *Phys. Rev. Lett.*, 61:2332–2335, 1988.

[12] NVIDIA. *Compute Unified Device Architecture Programming Guide version 3.1.1*, July 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*, (in press), 2010.

[14] C. Obrecht, F. Kuznik, B. Tourancheau, and J.-J. Roux. Global Memory Access Modelling for Efficient Implementation of the LBM on GPUs. In *High Performance Computing for Computational Science – VECPAR2010*. Lecture Notes in Computer Science, Springer, 2010.

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

[16] E. Riegel, T. Indinger, and N. 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 CUDA SDK Application Note*, 2009.

[18] J. Tölke. Implementation of a Lattice Boltzmann kernel using the Compute Unified Device Architecture developed by nVIDIA. *Computing and Visualization in Science*, pages 1–11, 2008.

[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.