Global Memory Access Modelling for Efficient Implementation of the LBM on GPUs

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

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

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

[1]Centre de Thermique de Lyon (UMR 5008 CNRS, INSA-Lyon, UCB Lyon1), 69621 Villeurbanne Cedex, France
[2]Laboratoire de l'Informatique du Parallélisme (UMR 5668 CNRS, ENS-Lyon, INRIA, UCB Lyon 1), 69364 Lyon Cedex 07, France

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

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.

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

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 segment 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]

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

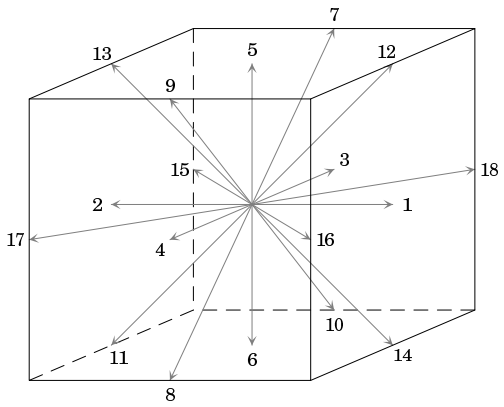

Figure 1: The D3Q19 stencil

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.

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.

[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.

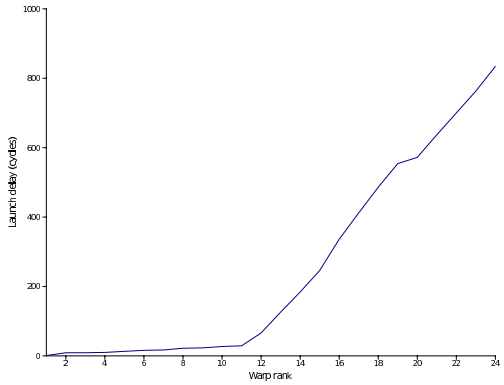

Figure 2: Launch delay in respect of warp rank

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.

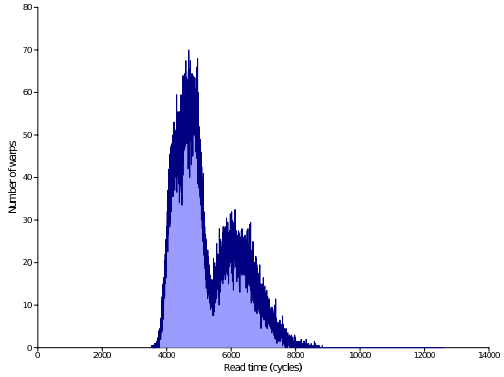

Figure 3: Read time for $N = 19$

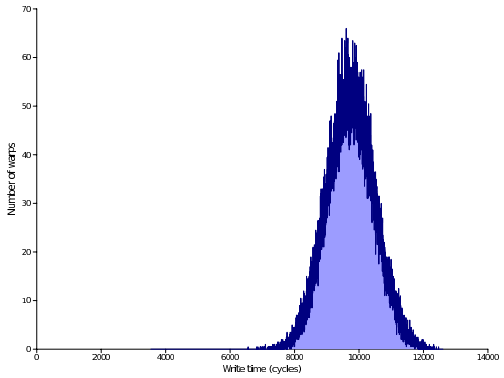

Figure 4: Write time for $N = 19$

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:

$$T_0 \approx 565(N-32)+15{,}164$$

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

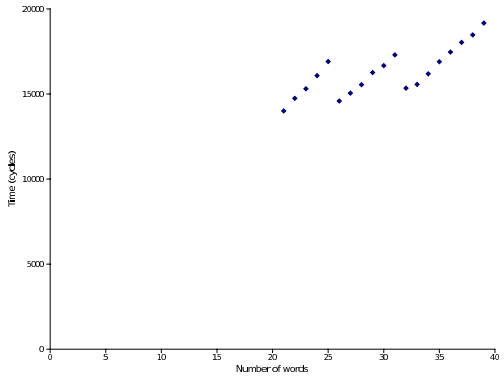

Figure 5: First stage duration

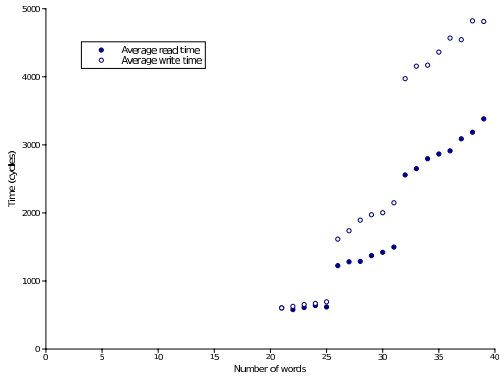

Figure 6: Timings in second stage

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 $N = 19$.

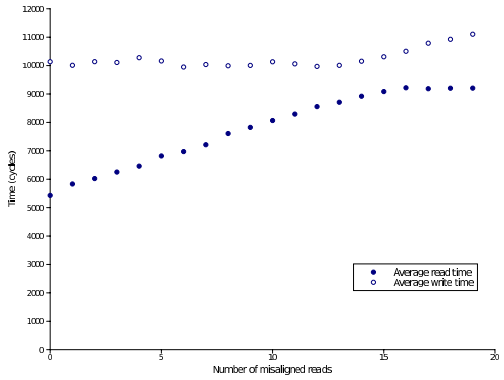

Figure 7: Misaligned reads

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.

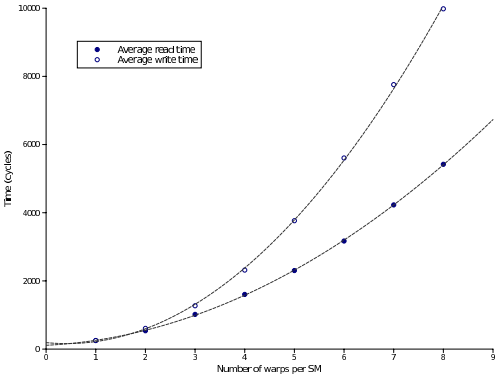

Figure 8: Occupancy impact

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.

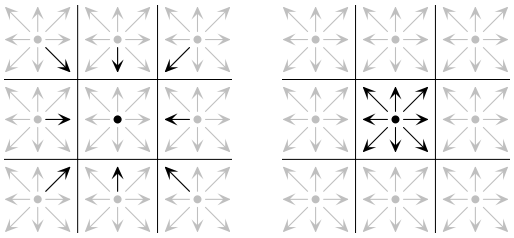

Figure 9: Reversed propagation scheme

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 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.

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)

[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.