Computation of Lattice Boltzmann in CPU and GPU heterogeneous environment

T. Rybak[1], J. Latt[2], B. Chopard[3]

[1] *University of Geneve*
rte de Drize 7, CH-1227, Battelle, Carouge, Switzerland
Email: Tomasz.Rybak@unige.ch;
[2] *University of Geneve*
rte de Drize 7, CH-1227, Battelle, Carouge, Switzerland
Email: Jonas.Latt@unige.ch; URL: http://www.unige.ch/
[3] *University of Geneve*
rte de Drize 7, CH-1227, Battelle, Carouge, Switzerland
Email: Bastien.Chopard@unige.ch; URL: http://www.unige.ch/

Keywords: lattice Boltzmann modelling, parallel computations, GPU, heterogenus environment, distributed computing

1 Abstract

Increase of accuracy of Lattice Boltzmann simulation increases memory usage by $O(N^3)$ and computational complexity by $O(N^5)$ for 3-dimensional grid [15]. Thus simulating real life situation requires using computing cluster consisting of many nodes. This rises complexity of software and increases management costs, like costs of hardware, administering it, and used electric power. At the same time using of-the-shelf computers allows for using standard and well known solutions (like MPI) for inter-node communication.

High-end Graphical Processing Unit (GPU) like NVIDIA GeForce GTX 460 contains 336 cores, each of them able to perform mathematical calculations. GPU is working with lower frequency than CPU, but it has access to fast memory and contains many cores, which means that overall it can perform computations faster than CPU. This means that single GPU can replace small cluster while being cheaper and using less power [10]. At the same time GPU is limited by the amount of memory on the board — high-end computing GPU Tesla M2070 contains 6GB of memory shared by all its cores, while similar amount is often available to a single core on CPU cluster. GPU works well in cases when many threads perform the same program; each difference, like different condition, causes thread group on GPU to divergent ([3]), which slows down the computation.

This work describes joining two Lattice Boltzman libraries: Palabos, written in C++ and using MPI, running on CPU cluster, and Sailfish, written in Python, able to run

simulation on single GPU. We divide simulated space into subdomains and run simulation in parallel on CPU (using Palabos) and GPU (using Sailfish). We simulate unsteady flow around circular cylinder as described by Kalro and Tezduyar [11]. We use Palabos on CPU to simulate sophisticated flow around the cylinder, and Sailfish on GPU to simulate flow in the rest of the space. This way we use the raw computing power of GPU on the space not containing obstacles and let CPU to deal with sophisticated conditions near obstacles. This way we use CPU ability to deal with branches ([16]) and GPU vast computing power to compute situation on the large space.

Each simulation step requires transferring the state of nodes. We use envelopes surrounding subdomains to avoid having to transmit every node state during every step. Because we only transmit necessary nodes surrounding subdomains, cost of transfer increases like $O(N^2)$ for 3-dimensional simulation. This is important in case of transferring data between computers and between GPUs because it reduces time needed to transfer data over PCI express bus and over the network.

Existing libraries using GPU for Lattice Boltzmann Modelling deal with simple situations because of limitations of memory and GPU architecture. Lattice Boltzmann on GPU can use simple domains of square shape and with stair-case approximated boundaries. Palabos is slower but more sophisticated and allows for having generic domains with curved boundaries. This work describes joining those two approaches which leads to having computing power while being able to simulate complex domains.

2 Literature review

Lattice Boltzmann Method is widely used for simulating fluids and gases because it is simple to program and can be easily parallelized. To be able to perform simulation with required level of details and with wide possible range of values of viscosity, velocity, and Reynolds number, it requires using dense grid.[15] Such a grid consists of many points and therefore requires much memory. Because each point requires performing computations in each simulation step, it means that increasing accuracy requires increasing computing power.

Usage of GPUs for computing is based on the long line of research. As noted by Ungerer et al. [16] tries of introducing parallelism to CPU were done for long time. CUDA can be seen as Simultaneous multithreading (SMT) family of solutions. Is uses interleaved multithreading, but uses warps (groups of threads, now 32) instead of single threads. It can be seen as chip multiprocessor with less fine-grained, using groups of threads instead of single threads. When one thread waits for resources (e.g. memory) GPU quickly switches to another thread to hide latency. The best effects can be acquired when there is enough

threads to hide memory access latency, and latency to access specialised hardware (to execute trigonometric functions, etc. [9], [2], [7] This makes it similar to DanSoft processor described by Ungerer et al. [16], in which two processors shared FPU. GF104 also is also similar to Cray MTA in which there were resources available to CPU. Fermi has also resources: half-warps, LSU (Load Sture Unit, memory access), SFU (Special Function Unit), Texturing unit. This means that thread can access some of them. In GF104 there is possibility to use 4 of 7 units, GF100 can use 2 of 6 unit maximally.

[13]

Programmer using GPUs have access to low-level details of hardware. At the same time companies started offering libraries helping managing those details. NVIDIA offers CUDA (Compute Unified Device Architecture) [3] and ATI offers Stream technology [1] to ease the task of using different hardware features. Because solutions offered by hardware vendors differ Khronos came with OpenCL (Open Computing Language) which can be used to program different devices using the same program.[6] [5] It is similar to OpenGL, but is used for computations, not for graphics.

Sailfish is Python based library implementing Lattice Boltzmann Method which uses GPUs to perform computations. It can use either CUDA or OpenCL as computing libraries. Because it is written in Python, it does not use CUDA directly, but through PyCUDA [12]. PyCUDA allows for changing programs on the fly, making run-time optimisations possible. Some possible optimisations of CUDA programs using PyCUDA were described in [14]; although there were used for different research area, the same rules still apply.

So viscosity is $v = \frac{1}{3}(\tau - \frac{1}{2})$, where $\tau = \frac{1}{\omega}$

$$F = f\frac{\delta x}{\delta t}$$

$$\delta t = u_{LB}\delta x$$

Strouhal number: $St = \frac{fL}{V}$ f - frequency, V - velocity, L - length (?) radius, 1 in our case use it to get frequency

resolution of the radius velocity at inlet relaxation parameter

3 Simulation description

We decided to use Kalro and Tezduyar 3D cylinder simulation [11] for testing Palabos and Sailfish. They used time step 0.05 and inflow velocity 1.0. We have set model to D3Q19, inflow velocity to 0.1, Reynolds number to 300; Sailfish example used time step 0.000434 and space step 0.0208. Both our simulations have used the same geometry setup.

The main velocity axix is Y, which is 75 units long. Box is 15 units high along axis Z and 8 units wide along axis X. Cylinder is placed along X axis, so is 8 units long. Its radius is 1 unit, so diagonal is 2 units.

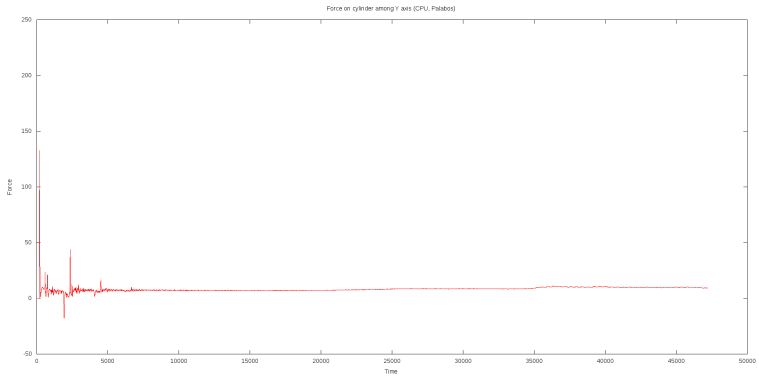

Figure 1: Force on cylinder simulated in Palabos.

Depending on available memory and computing capabilities one unit can contain different number of nodes. For Palabos one unit contains 8 nodes. Sailfish code was run on GeForce GTX460 (GF104) with 1GB of memory. This limits number of nodes that can be put simulated. As noted in documentation for simulating XxYxZ D3Qy scene one needs 2*X*Y*Z(y+4)*size of used floating data type (4 for 32-bit float, 8 for 64-bit float). So for 1GB card one can use only 6 nodes per unit for floats and 4 nodes per unit for doubles.

As can be seen on charts, Palabos simulation shows some periodicity, but it is not yet regular. We wonder whether running simulation for longer time would cause simulation to stabilise like in Kalro and Tezduyar.

Drag coefficient is

$$C = \frac{2F}{\rho v^2 A}$$

where F is force, ρ is density of fluid, v is speed, and A is area; it is projected frontal area. There is also volumetric drag coefficient, where area is square of cube root of volume: $A = (\sqrt[3]{V})^2$ We are not sure which was used in original article.

Drag coefficient in Palabos is 100 times less than in original article (0.019 vs. 2). We are not sure what causes this difference. We tried to check value of Reynolds number.

$$Re = \frac{r u_{LB}}{\frac{1}{3}(\frac{1}{\omega} - \frac{1}{2}}$$

r is resolution of the radius (8 in our simulation), u_{LB} is velocity (0.1 in our simulation), ω is relaxation parameter, retrieved from system. After computing Reynolds number from those values we got 299.96 which is very close to our chosen 300 so it is not source of difference.

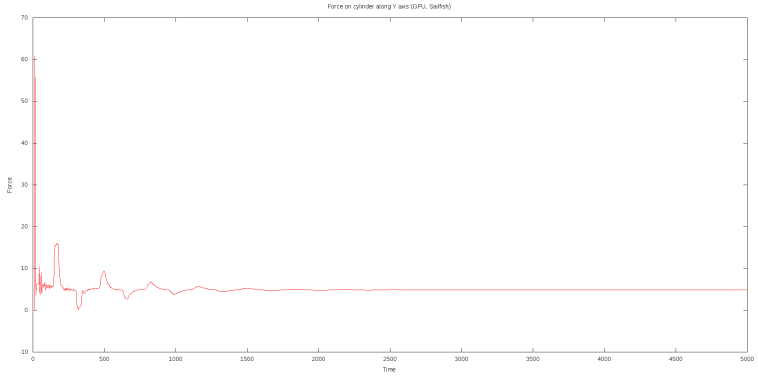

Figure 2: Force on cylinder simulated in Sailfish.

Sailfish simulation also uses Reynolds number 300. As can be seen from the charts instead of getting periodic force we got damping. We are not sure what causes this. We do not get any numerical instability which is good but we might be using not dense enough grid. We currently lack hardware to run denser simulation.

Unfortunately original article [11] did not have any information about time units in charts nor detailed units allowing to compare our results to ones done by Kalro and Tezduyar.

4 Implementation details

Modern graphics cards (GPU — Graphical Processing Unit) are not limited to generating realistically looking images on the screen. High-end GPU, like NVIDIA GeForce GTX 460, contains 336 cores, each of them able to perform mathematical calculations. GPU is working with lower frequency than CPU, but it has access to fast memory and contains many cores, which means that it can perform computations faster than CPU. This means that single GPU can replace small cluster while being cheaper and using less power. GPUs use different programming model than CPU; instead of trying to hide delays in accessing data in memory by cache and branch prediction they use so many threads that at any

given moment there is always some thread ready to run because it has data ready. Other threads can use this time to wait for data to arrive. [10]

CUDA and OpenCL use hierarchy of threads which demands that group of threads (warp) execute exactly the same instructions. Every difference, like different condition in the conditional instruction among warp threads causes drop of performance, because all other threads need to wait for that thread. This means that although CUDA can be used to speed up LBM, it is best used for the same situation.

At the same time, while GPUs contain many powerful cores, there is problem with memory. They just have little memory, so e.g. Tesla M2070 contains 512 (!) cores and only 6GB of memory. This is different from situation with CPUs. There each die has few private gigabytes. This means that CPU might be responsible for larger area. It will be slower because only one thread will need to compute many points, but all those points will be in the local memory. At the same time GPU can serve many points at the same time, but it can squeeze not many of them in the memory. This means that there is need to transfer information between main memory and device memory. This is slow, because transfer over PCI express is slow. But this gives opportunity to perform many computations for the same point. One should be able to put more computations into the thread, and it will not increase total computation time as GPU will just need to wait for data from memory. So instead of just waiting we can use productively this time and compute some more - but of course this "more" needs to make physical sense. But this means that on more powerful cards it might make more sense to use doubles instead of floats, and have better accuracy.

4.1 Needed changes in Sailfish

Sailfish does not provide outflow nodes. Those are available in Palabos. It means that we had to implement them in Sailfish.

5 Future

At the same time NVIDIA sees the problem of memory transfers. It introduced direct GPU-GPU transfers to CUDA 4.0. This means that any transfer from one card to another in multi-card setup is done directly between cards, without using CPU. This reduces time as there is no need to use CPU and to transfer from one card to the host and then from host to another card. Another novum, not yet understood, is introduction of MPI into CUDA 4.0. It might mean that CUDA will become MPI-aware, and kernels can be direct recipients of MPI messages and data, again without need to use CPU much and avoiding costly memory transfers. Using DMA helps here.

http://forums.nvidia.com/index.php?showtopic=194235&view=findpost&p=1199941 http://forums.nvidia.com/index.php?showtopic=194235&view=findpost&p=1199961 I am especially curious for the direct MPI support. Can I assume that it basically works the way MPI always works, but that you just give GPU Memory pointers to the send and recv functions? And is it a completely seperate implementation of MPI or an addition to openmpi or any other of the available solutions? We're not releasing our own MPI implementation; we're working with existing implementers to add this support. You'll hear more about it during the next few months (hopefully sooner rather than later).

While GPUs contain many powerful cores they are limited by amount of available memory; e.g. Tesla M2070 contains 512 (!) cores and only 6GB of memory. In theory GPU can perform computations for many points, but one cannot fit many points into memory available for the card. Transferring data to the GPU over PCI express is slow. At the same time this might allow for performing more computations for the same point. One should be able to put more computations into the thread, and it will not increase total computation time as GPU just needs to wait for data from memory. But this means that on more powerful cards it might make more sense to use doubles instead of floats, and have better accuracy.

6 Acknowledgements

Michał Januszewski for Sailfish Palabos team
[4] [8]

References

[1] ATI Corporation. *ATI Stream Computing OpenCL Programming Guide*, 08 2010. http://developer.amd.com/gpu/ATIStreamSDK/assets/ATI_Stream_SDK_OpenCL_Programming_Guide Accessed at 2010-10-04.

[2] NVIDIA Corporation. *Fermi Compatibility Guide for CUDA Applications Version 1.3*, 08 2010. http://developer.download.nvidia.com/compute/cuda/3_2/toolkit/docs/Fermi_Compatibility_G Accessed at 2010-10-04.

[3] NVIDIA Corporation. *NVIDIA CUDA C Programming Guide Version 3.2*, 09 2010. http://developer.download.nvidia.com/compute/cuda/3_2/toolkit/docs/CUDA_C _Programming_Guide.pdf Accessed at 2010-10-04.

[4] NVIDIA Corporation. *NVIDIA CUDA Reference Manual Version 3.2*, 08 2010. http://developer.download.nvidia.com/compute/cuda/3_2/toolkit/docs/CUDA_Toolkit_Reference_Manua Accessed at 2010-10-04.

[5] NVIDIA Corporation. *NVIDIA OpenCL Best Practices Guide*, 05 2010. http://developer.download.nvidia.com/compute/cuda/3_2/toolkit/docs/OpenCL_Best_Practices_Guide.p Accessed at 2010-10-04.

[6] NVIDIA Corporation. *NVIDIA OpenCL Programming Guide for the CUDA Architecture Version 3.2*, 08 2010. http://developer.download.nvidia.com/compute/cuda/3_2/toolkit/docs/OpenCL_Programming_Guide.p Accessed at 2010-10-04.

[7] NVIDIA Corporation. *NVIDIA's Next Generation CUDA Compute Architecture: Fermi*, 08 2010. http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIAFermiComputeArchitectureWhitepap Accessed at 2010-10-04.

[8] NVIDIA Corporation. *PTX: Parallel Thread Execution ISA Version 2.1*, 04 2010. http://developer.download.nvidia.com/compute/cuda/3_1/toolkit/docs/ptx_isa_2.1.pdf Accessed at 2010-09-03.

[9] NVIDIA Corporation. *Tuning CUDA Applications for Fermi Version 1.3*, 08 2010. http://developer.download.nvidia.com/compute/cuda/3_2/toolkit/docs/Fermi_Tuning_Guide.pdf Accessed at 2010-10-04.

[10] Kayvon Fatahalian and Mike Houston. A closer look at gpus. *Communications of ACM*, 51(10):50–57, 2008.

[11] V. Kalro and T. Tezduyar. Parallel 3d computation of unsteady flows around circular cylinders. *Parallel Computing*, 23(9):1235–1248, 1997. Parallel computing methods in applied fluid mechanics.

[12] Andreas Klöckner, Nicolas Pinto, Yunsup Lee, B. Catanzaro, Paul Ivanov, and Ahmed Fasih. PyCUDA and PyOpenCL: A Scripting-Based Approach to GPU Run-Time Code Generation. Technical Report 2009-40, Scientific Computing Group, Brown University, Providence, RI, USA, November 2009.

[13] Andreas Klöckner, Nicolas Pinto, Yunsup Lee, Bryan Catanzaro, Paul Ivanov, and Ahmed Fasih. Pycuda: Gpu run-time code generation for high-performance computing.

[14] Tomasz Rybak. Using gpu to improve performance of calculating recurrence plot. *Zeszyty Naukowe Politechniki Białostockiej. Informatyka Zeszyt 6*, pages 77–94, 2010.

[15] Michael C. Sukop and Daniel T. Thorne. *Lattice Boltzmann Modeling. An Introduction for Geoscientists and Engineers.* 2005.

[16] Theo Ungerer, Borut Robič, and Jurij Šilc. A survey of processors with explicit multithreading. *ACM Comput. Surv.*, 35:29–63, March 2003.