Enabling temporal blocking for a lattice Boltzmann flow solver through multicore-aware wavefront parallelization

Johannes Habich, T. Zeiser, G. Hager, G. Wellein
Erlangen Regional Computing Center (RRZE), 91052 Erlangen, Germany
email: hpc@rrze.uni-erlangen.de

Abstract

This report presents first results of a pipelined wavefront parallelization approach applied to the lattice Boltzmann method in 3D. Threads executing successive time steps on a fixed spatial domain are scheduled to the same multicore processor chip having a shared cache. The efficient reuse of data in cache by successive threads implements a multicore-aware temporal blocking parallelization in a rather simple way.

Keywords: lattice Boltzmann, CFD, temporal blocking, multicore, wavefront

1 Introduction

Recently Datta et al. [1] have shown that substantial effort has to be put into optimizations for stencil based methods, such as the Jacobi iteration in 3D, to reach an optimal implementation on state-of-the-art multicore architectures. Spatial blocking techniques, as used in that work, divide the computational domain into more compact blocks and therefore can enhance locality making better use of the caches. Performance can often further be increased by doing multiple time steps on one block. This technique is called temporal blocking. With every additional time step performed, however, the block shrinks in each of the dimensions, leading to rather shorter loops and increased overhead which may severely limit the benefit of temporal blocking. Furthermore, specific platform-dependent tuning parameters, e.g. the block size in every dimension, have to be carefully adjusted. Hardware independent, so called cache oblivious algorithms, e.g. Frigo et. al [2], come at the cost of many data TLB misses as shown in [3] for a 3D lattice Boltzmann flow solver due to irregular access patterns.

This paper presents first results of a multicore-aware lattice Boltzmann method based solver in 3D using a pipelined wavefront parallelization approach. Owing to its explicit nature and a cellular automata like update rule, the basic lattice Boltzmann algorithm is easy to implement, optimize and analyze. Compared to, e.g. Jacobi solvers, the larger stencil, e.g. the D3Q19 stencil used here, require the use of even smaller block sizes for traditional temporal blocking techniques. To overcome this limitation a wavefront based temporal blocking technique is proposed, similar to the one described in [9] for an Jacobi solver. Parallelization of the algorithm does not partition the domain right away, but executes several threads on the same domain with a certain spatial displacement. The displacement must be large enough to sustain the computational correctness depending on the stencil used to avoid race conditions among the threads. All threads of a so-called wavesocket are bound to a single multicore chip with a shared cache, which reduces the access to memory to one load for the initial data and to one store at the end of the temporal iterations.

2 Multicore testbeds

The key architectural features of the three compute nodes evaluated in this report are presented in Tab. 1. Performance data was obtained for domains which exceed the cache size by far. Since stencil computations are data intensive, the attainable main memory bandwidths as measured with optimized stream benchmark [10] runs are shown as well. “Optimized” refers to the use of non-temporal stores which are important to get unbiased results on x86 architectures as described in [9].

Modern multicore processor have different numbers of processor cores, which can communicate in many different and sophisticated ways. To express which cache level is shared by how many cores, a convenient terminology is

Proceedings of Parallel CFD 2009, 21st International Conference on Parallel Computational Fluid Dynamics, May 18-22, 2009, Moffett Field, CA, USA. pages 178-182

		Woodcrest	Dunnington	Nehalem
Type		Xeon 5160 @3.0 GHz	Xeon 7460 @2.66 GHz	Xeon @2.66 GHz
L1 group	size [kB]	32	32	32
	TRIAD GB/s	3.7	3.0	11.6
L2 group	L2 size [MB]	4	3	0.25
	shared by # cores	2	2	1
	TRIAD GB/s	3.7	3.5	see L1
L3 group / socket	L3 size [MB]	-	16	8
	shared by # cores	-	6	4
	TRIAD GB/s	-	3.5	16.6
System	# sockets	2	4	2
	raw bw [GB/s]	21.3	34.0	51.2
	TRIAD GB/s	6.7	13.2	32.7

Table 1: Overview on cache group structure and STREAM TRIAD performance for the systems in the test-bed. Non-temporal stores were used throughout, so all bandwidth numbers denote actual bus traffic. STREAM array size was 20,000,000 elements.

introduced: The group of cores of a processor that share a certain cache is called a *cache group.* Thus, there are L1 groups (which consist of a single core on all current multicore designs), L2 groups, etc.. There can be multiple instances of any cache group on a multicore chip. Note that the focus lies on the cache levels available for data only, ignoring all instruction caches.

This report focuses on Intel based multi-socket platforms using state-of-the-art dualcore ("Woodcrest"), hexacore ("Dunnington") and quadcore ("Nehalem") variants. As the lattice Boltzmann kernel on a single core can already saturate most of the memory bandwidth on those systems, temporal blocking is the most promising method of optimization. For all tests the Intel Fortran compilers in version 11.0.069 were used.

3 Lattice Boltzmann method

The lattice Boltzmann method (LBM) has evolved over the last two decades and is today widely accepted in academia and industry for solving incompressible flow problems. Coming from a simplified gas-kinetic description, i.e. a velocity-discrete Boltzmann equation with appropriate collision term, it satisfies the Navier-Stokes equations in the macroscopic limit with second-order of accuracy [4,5]. Here the D3Q19 discretization model, i.e. 19 discrete velocities in three spatial dimensions, with the BGK collision operator is used.

The evolution of the single particle distribution function f_i is described by the following equation ($i = 0 \dots 18$):

$$f_i(\vec{x}+\vec{e}_i\Delta t, t+\Delta t) = f_i^{\text{coll}}(\vec{x},t) = f_i(\vec{x},t) - \frac{1}{\tau}\left[f_i(\vec{x},t) - f_i^{eq}\left(\rho(\vec{x},t),\vec{u}(\vec{x},t)\right)\right]$$

$$\text{with } f_i^{eq}\left(\rho(\vec{x},t),\vec{u}(\vec{x},t)\right) = \rho(\vec{x},t)w_i\left[1 + \frac{3}{c^2}\vec{e}_i\cdot\vec{u}(\vec{x},t) + \frac{9}{2c^4}\left(\vec{e}_i\cdot\vec{u}(\vec{x},t)\right)^2 - \frac{3}{2c^2}\vec{u}(\vec{x},t)\cdot\vec{u}(\vec{x},t)\right]$$

where f_i^{coll} denotes the "intermediate" state after collision but before propagation. ρ and $\vec{u}$ are the macroscopic quantities and are obtained as 0^{th} and 1^{st} order moments of f_i with regard to the discrete velocity $\vec{e}_i$, i.e. $\rho(\vec{x},t) = \sum_0^{18} f_i(\vec{x},t)$ and $\rho(\vec{x},t)\vec{u}(\vec{x},t) = \sum_0^{18}\vec{e}_i f_i(\vec{x},t)$. The Taylor-expanded version of the Maxwell-Boltzmann equilibrium distribution function [5,6] f_i^{eq} is given by Eq. 1, w_i are direction-dependent constants [6] and $c = \frac{\Delta x}{\Delta t}$ with the lattice spacing Δx and the lattice time step Δt. The equation of state of an ideal gas provides the pressure p, $p(\vec{x},t) = c_s^2\rho(\vec{x},t)$, with c_s as the speed of sound. The fluid's kinematic viscosity is determined by the dimensionless collision frequency $\frac{1}{\tau}$ according to $\nu = \frac{1}{6}(2\tau-1)\Delta x c$ with $\tau > 0.5$ due to stability reasons [4–6].

For solid wall boundaries, the boundary conditions are realized by the bounce-back rule [4, 5], i.e. if a distribution is about to be propagated into a solid cell, the distribution function returns to the original cell but with reversed momentum. Bounce-back generally assumes that the wall is in the middle between the two cell centers. In the halfway formulation this leads to:

$$f_{\bar{i}}(\vec{x}, t + \Delta t) = f_i^{\text{coll}}(\vec{x}, t)$$

with $\vec{e}_{\bar{i}} = -\vec{e}_i$ and $f_i^{\text{coll}}(\vec{x}, t)$ being the right hand side of Eq. 1. The fullway bounce-back on the other hand is given by:

$$f_{\bar{i}}(\vec{x}, t + \Delta t) = f_i^{\text{coll}}(\vec{x}, t - \Delta t).$$

In order to apply the halfway bounce-back, the propagation is first done for fluid cells only and afterwards the distributions pointing to obstacles are handled by simply reversing the distribution in direction inside the fluid cell. In contrast, for the fullway bounce-back the fluid cells propagate the distributions into all surrounding cells. In the next time step, values stored at the obstacles' positions are reversed in direction and propagated back to the originating cells. This implementation uses the fullway bounce-back, as it can more easily be adapted to the pipelined parallel wavefront approach. Future work will also employ the halfway bounce-back.

3.1 Implementation

The basic LBM in 3D can be implemented with three nested loops traversing the computational domain and updating each lattice cell. As a starting point we use a mature and well optimized LBM kernel in 3D as described in [7, 13, 14]. In our discussion we use Fortran indexing, i.e. in multi-dimensional arrays the first/inner most index is consecutive in main memory, and focus on memory bandwidth bound problems.

A single LBM "sweep" through the complete domain is usually wrapped into an iteration loop, which performs several time steps. Often a two grid approach is implemented, i.e. one array holds the original data and one the updated data, allowing for simple implementation and parallelization. After each outer iteration, the grids are simply interchanged. Using the structure-of-arrays layout "SOA", F(i,j,k,Q,t), the overall data transfer on cache based architectures for a single lattice site update is $19 * 8 * 3$ Byte (for more details we refer to [7]). Together with the STREAM bandwidth numbers (cf. Tab. 1) one can easily estimate the maximum performance of the compute systems under consideration in terms of the fluid cell update rate given in million fluid cell updates per second (FluidMLUPs). The attainable STREAM bandwidth for the Woodcrest L2 cache group is measured as 3.7 GB/s (6.3 GB/s for the node respectively), therefore the upper limit is 8 FluidMLUPs (14 FluidMLUPs). For the Dunnington the limit is 7 FluidMLUPs for the L2 and L3 cache groups (28 FluidMLUPs for the node respectively), and the estimates for the Nehalem are 36 FluidMLUPs for the L3 cache group (71 FluidMLUPs for the node respectively). The code was parallelized by applying OpenMP `parallel do` work-sharing directives to the outermost (k) loop. ccNUMA data locality was ensured by parallelizing the initialization loops as well.

As can be seen from Fig. 2 for Woodcrest and Dunnington the results are in very good agreement with our estimations, both get 85 % of the attainable STREAM bandwidth. The Nehalem however performs only with 77 % of the estimated performance, which is due to the relatively short running inner (i) loop and the sub-optimal data layout, for this particular new hardware design. Benchmarks show that the F(i,Q,j,k,t) layout is able to get up to 85 % of performance as well. A detailed discussion however would be beyond the scope of this report.

Pipeline parallel wavefront implementation

The pipeline parallel wavefront approach does not perform several timesteps after another for a compact block of the domain. In contrast it traverses the whole computational domain assigned to one cache-group. For each cache group, one so-called wavesocket is launched. A wavesocket comprises as many independent threads as there are cores in the group. Note that it is essential to employ proper thread/core affinity to ensure that each wavesocket is always scheduled to the same cache group. In this preliminary report we focus on a wavesocket with two threads, i.e. two successive time steps are performed by the two threads, where the second update should be done on data available in the shared cache.

The basic idea is illustrated in Fig. 1 where *Thread 0* does the collision for all fluid cells in plane k and propagates the new distribution functions to three adjacent planes $(k-1, k, k+1)$ residing in the second array (see arrows down in Fig. 1).

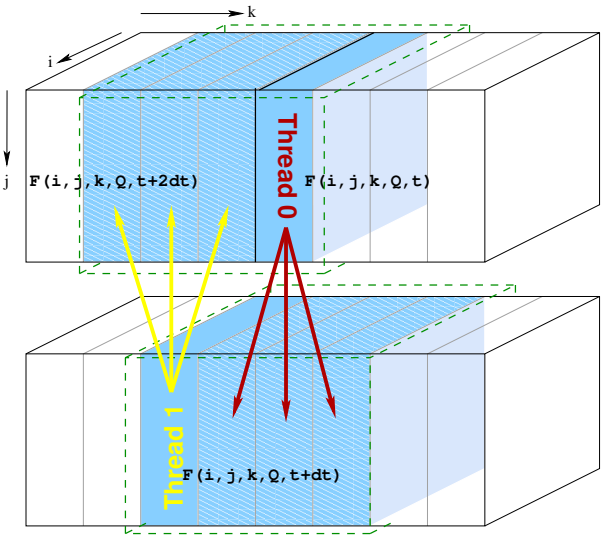

Figure 1: Time blocking through pipeline parallel processing by a two-thread wavefront group. Dashed boxes indicate (i, j) layers that must be kept in cache for optimal reuse of cache lines.

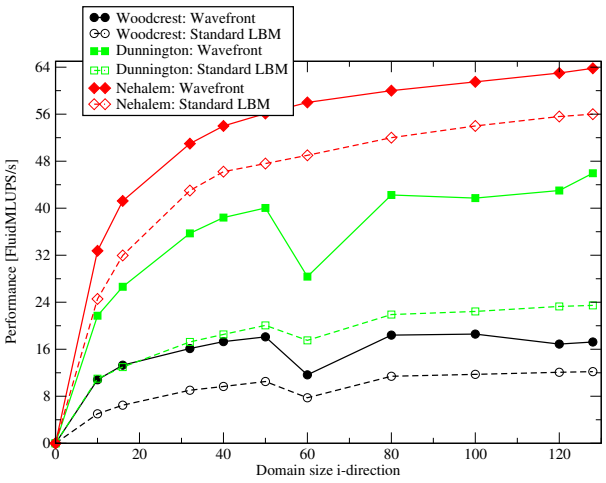

Figure 2: Performance comparison of the wavefront optimized LBM implementation with the standard implementation. All systems have been used with a maximum number of threads, i.e. 4 threads on Woodcrest, 16 threads on Nehalem (SMT on) and 24 threads on Dunnington. For the computational domain 600 was chosen in k-direction and 32 / 48 / 64 in j-direction for Woodcrest / Nehalem / Dunnington.

At the same time *Thread 1* can perform the collision of the updated $(t+dt)$ values in plane $k-2$ which do not have to be loaded from main memory again as long as the shared cache is large enough to hold the data in the lower dashed box of Fig. 1. Finally *Thread 1* propagates them (see arrows up in Fig. 1) back to the original array which then holds data for time step t ahead of the first wavefront (*Thread 0*) at larger k and the distribution functions for time step $t+2dt$ in the planes already visited by *Thread 0* before (at smaller k). It is obvious that in this approach the use of a second array spanning the complete computational domain is obsolete. It can be replaced by a temporary array holding 4 k planes, reducing the memory footprint of the LBM code by almost a factor of two.

If the shared cache is large enough to hold the two dashed boxes in Fig. 1, i.e. the complete temporary array and 4 k planes of the original array, then the overall data transfer for performing *two updates* on a single cell is just 19 load and 19 store operations, i.e. $2*19*8$ Byte per cell. This compares to $2*3*19*8$ Bytes per cell if both arrays are traversed twice as done in the original implementation. Thus, a maximum speed up of three may show up for this simple and straightforward wavefront implementation. Please note that the performance gain is larger than the number of time steps "blocked", because this implementation avoids the "Read for Ownership" (RFO) if writing back to the second array without reading it before. Here we store to the same array which has been loaded before and there is no need for the RFO if the cache is sufficiently large.

3.2 Results

In order to obtain accurate and comprehensive results, care has to be taken to properly pin the threads of common wavesockets to common cachegroups [12]. The results shown in Fig. 2 are a selection of various benchmark scenarios.

The wavefront parallelized version of the LBM leads to a 35 % higher performance on the Woodcrest node. Performance on the Nehalem system improves, however, only by 15 %. Due to the substantially better system balance of the Nehalem, the baseline performance is already high. The Dunnington, in contrast to the Nehalem, has a rather low system balance and is considered as a "bandwidth-starved" system design. However, it may be a somewhat prototypical for multicore chips to come. STREAM measurements show that already two of the six cores of a socket can saturate the full memory bandwidth. This leaves a lot of potential for temporal blocking techniques and so Dunnington shows the best gain from the pipeline parallel wavefront optimization, nearly doubling performance as compared to the standard implementation. For all systems, the performance gap is expected to grow with larger domain sizes.However, additional spatial blocking must be applied to a wavesocket's lattice partition to efficiently support

larger domains than those shown in Fig. 2. To avoid short inner loops blocking in j-direction is in order. It is also straightforward [9] to run more than two threads, e.g. 4 or 6 on the Dunnington system, in a single wavesocket further reducing the overall memory transfer. Furthermore, no cache optimizations for the kernel have been implemented so far. Work on all these topics is currently being done.

4 Conclusions and Acknowledgements

It was shown that pipelined wavefront parallelization efficiently implements temporal blocking for a large stencil based flow solver on Intel multicore CPUs. Especially highly bandwidth-starved systems show a substantial gain in terms of performance if this technique is applied.

We are indebted to Intel Germany for providing the "Nehalem" compute node through an early access program. This work was carried out within the Bavarian framework of KONWIHR. Financial support from BMBF through project SKALB (grant 01IH08003A) is gratefully acknowledged.

References

[1] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf and K. Yelick: *Stencil Computation Optimization and Auto-tuning on State-of-the-Art Multicore Architectures.* In: ACM/IEEE (Ed.): Proceedings of the ACM/IEEE SC 2008 Conference (Supercomputing Conference '08, Austin, TX, Nov 15–21, 2008).

[2] M. Frigo, C. E. Leiserson, H. Prokop and S. Ramachandran: *Cache-Oblivious Algorithms.* In: 40th Annual Symposium on Foundations of Computer Science, FOCS 99, Oct 17–18, 1999, New York, NY.

[3] T. Zeiser, G. Wellein, A. Nitsure, K. Iglberger, U. Rüde and G. Hager: *Introducing a parallel cache oblivious blocking approach for the lattice Boltzmann method.* Progress in CFD, Vol. 8, No. 1–4, pp. 179–188, 2008.

[4] S. Chen and G. D. Doolen: *Lattice Bolzmann method for fluid flows.* Annual Review of Fluid Mechanics Vol. 30 pp. 239–364, 1998.

[5] S. Succi: *The Lattice Boltzmann Equation - For Fluid Dynamics and Beyond.* Clarendon Press, 2001.

[6] X. He and L.-S. Luo: *Theory of the lattice Boltzmann method: From the Boltzmann equation to the lattice Boltzmann equation.* Phys. Rev. E, Vol. 56, pp. 6811-6817, 1997.

[7] G. Wellein, T. Zeiser, G. Hager, and S. Donath: *On the single processor performance of simple lattice Boltzmann kernels.* Computers & Fluids, Vol. 35, No. 8–9 pp. 910-919, 2006.

[8] D. J. Kerbyson and A. Hoisie: *Analysis of Wavefront Algorithms on Large-scale Two-level Heterogeneous Processing Systems.* In Proc. Workshop on Unique Chips and Systems (UCAS2), IEEE Int. Symposium on Performance Analysis of Systems and Software (ISPASS), Austin, TX, 2006.

[9] G. Wellein, G. Hager, T. Zeiser, M. Wittmann and H. Fehske: *Efficient temporal blocking for stencil computations by multicore-aware wavefront parallelization.* Submitted to IEEE International Computer Software and Applications Conference (COMPSAC 2009), Seattle, 2009

[10] J.D. McCalpin: *STREAM: Sustainable Memory Bandwidth in High Performance Computers.* `http://www.cs.virginia.edu/stream/`

[11] K. J. Barker, K. Davis, A. Hoisie, D. J. Kerbyson, M. Lang, S. Pakin and J. C. Sancho: *A Performance Evaluation of the Nehalem Quad-Core Processor for Scientific Computing.* Parallel Processing Letters , Vol. 18 , No. 4, pp. 453–469 (2008).

[12] M. Meier: *Thread pinning by overloading pthread_create().* `http://www.mulder.franken.de/workstuff/pthread-overload.c`

[13] J. Habich: *Improving computational efficiency of lattice Boltzmann methods on complex geometries.* Bachelor's thesis, Chair of System Simulation, University of Erlangen-Nuremberg, Germany, 2006

[14] S. Donath, T. Zeiser, G. Hager, J. Habich and G. Wellein: *Optimizing performance of the lattice Boltzmann method for complex structures on cache-based architectures.* In: F. Huelsemann, M. Kowarschik, U. Ruede (Eds.): Frontiers in Simulation: Simulation Techniques - 18th Symposium in Erlangen, September 2005 (ASIM), pp. 728-735, SCS Publishing House, Erlangen, 2005.