Performance Analysis of the Lattice Boltzmann Method on x86-64 Architectures

Jan Treibig, Simon Hausmann, Ulrich Ruede *

Zusammenfassung

The Lattice Boltzmann method (LBM) is a well established algorithm to simulate fluid flow. The complexity of todays 3D simulation problems resulting in long computation times together with the fact that a standard implementation of the LBM only achieves a small fraction of the potential of a modern CPU is the motivation for this performance analysis. We show in our paper, that it is crucial to combine new CPU architectural features as software prefetching and SIMD instruction set extensions, with the established cache blocking techniques to utilize the computational power of modern CPUs.

1 Introduction

The Lattice Boltzmann method is an alternative approach to CFD based on a cellular automata approach. The core algorithm is well suited for implementation on a computer. From the optimizer point of view the balance between arithmetical and memory demands make it a challenging target [Kow04]. The computations consist exclusively of floating point arithmetic. While a standard implementation is mainly memory bandwidth and latency limited after applying common optimization techniques and using different data layouts things are not that clear anymore. In the core loop body also a lot of computation is done, causing arithmetical limitations. Computer architectures continue to improve at a fast pace. Still there is a large gap between theoretical peak performance and achievable performance. The memory subsystem could not keep up with the huge improvements in raw computational power. But also with an purely computational bound algorithm it is still difficult to reach the theoretical peak performance on a modern CPU. Recent developments in the dominating x86 based architectures include instruction set extensions, enabling SIMD Operations, and hardware and software data prefetching. The SIMD instructions enable the CPU to apply better optimizations while data prefetching hides the latency of memory access. The newest incarnation of x86 is the x86-64 architecture introduced by AMD. It doubles the number of registers introducing 64 bit wide general purpose registers (GPR). We focused our efforts on this architecture. While many of these features are implemented for many years the compilers still do not utilize these techniques to their full extent. Therefore we decided to implement the code in assembler bypassing possible limitations of the compiler.

*Lehrstuhl für Systemsimulation, Institut für Informatik, Friedrich-Alexander Universität Erlangen-Nürnberg D-91058 Erlangen, Germany

2 Arithmetical Analysis

In the following a brief summary of the Lattice Boltzmann method is given. A detailed description along with a deep theoretical background can be found in [WG00]. The computational domain is mapped onto a regular grid with cells, the lattice. Each cell represents a volume element of fluid particles. The motion of the particles in the fluid is described only indirectly by distribution functions. A cell is divided into discrete directions of velocity, and for each a particle distribution function is defined, representing the motion of particles into that direction. These functions form the state of the cell. As discrete time advances the flow of particles is simulated by moving the values of the distribution functions to the neighboring cells and calculating the collision with particles from other directions. We implemented a 3D Lattice Boltzmann Solver using a 3DQ19 model. The test problem used is the lid driven cavity. It was used to verify the solver and do the performance measurements. The arithmetical kernel of the algorithm is the so called collide step. The evaluation depends on the previous calculation of the mass density and the velocity as well as the local equilibrium distribution function. The new particle distribution functions do not depend on each other. With regards to the x86-64 instruction set architecture it is possible to combine always two particle distribution function updates into one sequence of instructions by the use of SSE2 *packed operations*. This reduces the amount of instructions needed to formulate the algorithm and results in faster execution. An important property of the Athlon-64 processors is that they have one unit for floating point addition and one for multiplication, which can be execute in parallel. For highest performance it is therefore sensible to position arithmetic operations in the stream of instructions in a way that independent multiplications and additions can be scheduled for parallel execution. Unfortunately the calculation for the update of the particle distribution functions is largely sequential. Due to direct dependencies many instructions cannot be scheduled in parallel. With the ability to perform a multiplication and addition in parallel in one clock cycle an Athlon-64 for example with a clock frequency of 2.4 GHz has a theoretical peak performance P_{theo} of 4.8 GFlop/s. The same applies to Pentium 4/Xeon processors, with the capability of two Flops per cycle (SSE2). A simple synthetic benchmark that uses aggressive scheduling shows that it is actually possible to measure up to 94.7 % of this value on an Athlon-64 , which is roughly 4.55 GFlop/s. So the upper limit reduces to the "technical" maximum P_{tech}. With regards to the Lattice Boltzmann algorithm such a sequence however cannot be achieved as there are more additions than multiplications. The minimum total amount of floating point operations per cell is approximately 156, with 90 additions/subtractions, 65 multiplications and one division. The ratio of additions to multiplications reduces the approximation for the theoretically reachable maximum GFlop/srate. If n_{a} is the number of additions and n_{m} the number of multiplications, $1/2(n_{\mathrm{a}} + n_{\mathrm{m}})$ cycles are needed if two operations per cycle can be executed. However, when $n_{\mathrm{a}} \neq n_{\mathrm{m}}$, $\max(n_{\mathrm{a}}; n_{\mathrm{m}}) - \min(n_{\mathrm{a}}; n_{\mathrm{m}})$ additional cycles are required. As a result, the maximum reachable peak performance for the LBM P_{LBM} in comparison to P_{tech} is given by $\mathrm{P}_{\mathrm{LBM}} = \frac{n_{\mathrm{a}}+n_{\mathrm{m}}}{2 \cdot \max(n_{\mathrm{a}};n_{\mathrm{m}})} \cdot P_{\mathrm{tech}}$ That corresponds to 87 % of P_{tech}, which is approximately 4.0 GFlop/s on the analyzed machine. With regards to the Lattice Boltzmann algorithm, performance is often measured in *M*ega *L*attice *S*ite *Up*dates per second, or like in this case for the performance of only fluid cells FluidMLSUPS. With

the reduced GFlop/s rate an upper bound can also be approximated in MLSUPS:

$$P_{\mathrm{LBM}} \approx 4.0\ \mathrm{GFlop/s} = \frac{4.0\ \mathrm{GFlop/s}}{156\ {}^{\mathrm{Flops}}/_{\mathrm{cell}}} \approx 26\ \mathrm{FluidMLSUPS}$$

A more pessimistic boundary, based on the observations in the previous section that there are no calculations that permit parallel usage of the multiplication and the addition unit, would be to assume only 50 % of the peak performance, resulting in about 2.3 GFlop/s or 14 FluidMLSUPS.

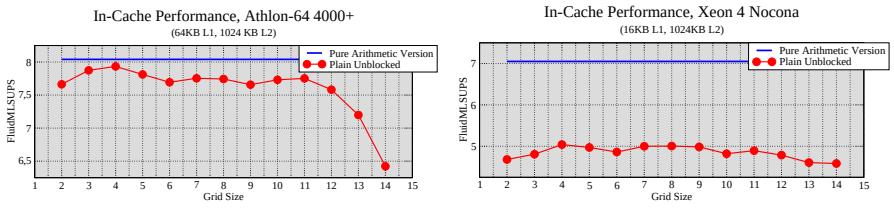
3 Memory Performance Analysis

There are different approaches to improve the re-use of data stored in the CPU caches. [Don04] presents a technique based on changes in the data layout. [Wil03] and [Igl03] instead change the way the grid is traversed, commonly called *cache blocking*. We have implemented the best performing cache blocking from [Igl03], *4-way cache blocking*. The basic idea is to divide the domain into little cubes, perform multiple time steps inside one and move on to the next cube. The cubes are supposed to fit in the CPU caches. The Lattice Boltzmann algorithm has two distinct patterns of accessing memory. The streaming step causes values to be gathered or pushed from the neighboring cells. What appears to be a direct neighborhood in a grid by just an increment or decrement of the x, y or z coordinate may result in a big distance from the current cell in the actual linear addressed memory. On the other hand reading the current 19 cell values (*collide-stream*) or writing them (*stream-collide*) results in a linear memory access pattern, as the cells are processed in order. So in the algorithm either the cell read operations are scattered and the results are written in linear fashion or the other way around. [Cor04] and [Dev04] explain that modern Intel and AMD processors attempt to detect regular read access patterns and then start loading memory further ahead into cachelines, the *hardware prefetch mechanism*. The idea is to fetch the data needed in the next loop iteration while the processor is busy with arithmetic operations in the current one. In stream-collide order in the Lattice Boltzmann algorithm the scattered read operations are likely to prevent the CPU from seeing a pattern simple enough for the hardware prefetcher. In that case software prefetch instructions were used to to achieve a similar memory bus utilization as with sequential reads in collide-stream but retain the linear writing pattern for the results.

4 Results

All measurements were done on a AMD Athlon-64 4000+ (2.4 GHz) and a Intel Xeon Nocona (3.4 GHz). As a first step for comparison the core LBM calculation sequence was extracted from the solver and on assembler level all instructions that result in memory access were removed. The remaining code of course does not calculate an actual fluid flow, but it consists of exactly the calculation sequence for doing so, minus memory access though.

This version was compared to the complete fluid solver in *stream-collide* order with domain sizes where all allocated memory fits into at least the level 2 cache, with a size of 1 Megabyte on all machines. Another *important* property was that the domain consisted entirely of fluid cells, as we are interested in comparing the arithmetic performance of the fluid flow calculation, not how much time it takes for the processor to move data around with the no-slip boundary condition handling.
Figure 1 left shows the results on the Athlon-64. The results indicate a fairly constant performance inside the cache, except when the grid size grows near to the total size of the L2 cache. A size of $14^3 \cdot 19$ (values per cell) $\cdot\ 2$ (grids) $\cdot\ 8$ (bytes per double) fills already 83 % of the L2 cache. Given how near the *in-cache* unblocked plain solver is to the pure arithmetic version it appears that the caches can deliver data reasonably fast on that machine. However the measured $\approx 8\ \mathrm{FluidMLSUPS}$ are quite in distance to what the processor could achieve in theory. The processor itself may be able to produce something in the range of 14 to 26 FluidMLSUPS, depending on how good the instruction scheduler can feed the floating point units.
As with the Athlon-64 the Xeon Nocona shows a reasonably constant *in-cache* performance, too (Fig. 1 right). A difference to the Athlon-64 however is the significant distance between the measured arithmetic fluid update performance and the *in-cache* performance, almost 2 MLSUPS. One possible reason for this difference is the higher latency of the Xeon's L2 cache, it may make stalls in the long pipeline of the Pentium core more expensive than on the Athlon-64. In this section the results of the cache optimizations are shown, the

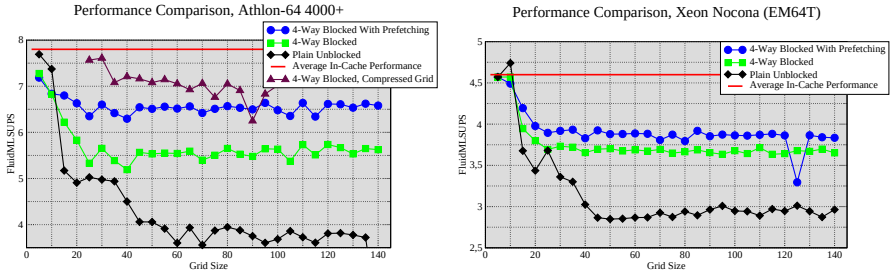


(a) Athlon-64 4000+ (b) Xeon Nocona

Abbildung 1: Comparison of in-cache version against measured arithmetic performance.

4-way blocking as well as the software prefetching on top of it. As with the in-cache measurements in the previous section the domain consisted of fluid cells only and stream-collide order was chosen.
Figure 2 left shows the results on the Athlon-64 . As already shown in [Igl03] the 4-way blocking provides a significant speed-up over the unblocked version. The software prefetching cuts execution time in average by 15 %, resulting in one more MLSUPS. However the 4-way blocked version of [Igl03] with a compressed grid and a block size of 8^3 still provides the best performance.
The key insight becomes apparent when comparing these results with the average in-cache

performance: The block techniques effectively reduce the high latency of main memory, $\approx 80 - 90\ \%$ of the fluid in-cache performance is achieved on this machine.
The measurements on the Xeon Nocona are shown in Fig. 2 right. Again an improvement of the 4-way blocking can be seen, however the difference to the unblocked version is slightly less than on the Athlon-64. The software prefetching on top of the blocked code brought only little improvement. Again the cache optimized code is at approximately 84 % of the in-cache results, showing the efficiency of the optimization.



(a) Athlon-64 4000+ (b) Xeon Nocona

Abbildung 2: Comparison of cache blocking techniques with *in-cache* performance

5 Conclusions

Under theoretical consideration the algorithm is limited by the data rate the memory can deliver, as shown in [Don04] and [Kow04]. However comparing the efficiency of the cache optimized code with the speed obtained with *in-cache* fitting fluid domains produced interesting results. In combination with *software prefetching* the 4-way cache blocked code was able to achieve 80-90 % of the FluidMLSUPS rates measured in the L2 cache. This indicates that the cache optimization was efficient enough to hide large portions of the high latency and slow data rate of main memory. In the next step the performance results from the *in-cache* measurements were put into relation with a version of the code that had all memory access instructions removed, in order to see how much the latency of the caches influence the execution time. The comparison on the Athlon-64 showed little difference in performance, the caches did not appear to be a limitation. On the Xeon Nocona a slight drop was measured though. Given the efficiency of the cache optimization and the comparison of *in-cache* to “pure arithmetic” it appears the remaining limiting factor is somewhere in the path from instruction decoding through scheduling up to the floating point execution units. For an estimation the raw floating point processing power can be reduced due to an unbalance of multiplications to additions . When comparing that with the measured performance a big gap remains. Less than 50 % of the estimated peak performance was

reached, on all platforms. It remains to be determined which factors exactly in the coding of the Lattice Boltzmann algorithm have the biggest impact on the arithmetic performance. Despite the out-of-order execution capabilities of Athlon and Pentium processors the actual placement of instructions in the machine code remains to have a big influence on the performance. Compilers tend to be good in instruction scheduling but appear to have problems in transforming the complex equations to *packed* instructions. The combination of both may possibly improve performance even more. One insight of our work also was that todays architectures with out of order instruction scheduling and hardware prefetching often behave intransparent and unpredictable to the programmer. Especially the Intel Xeon seems to have internal bottlenecks in his implementation, which are not obvious to the programmer.

Literatur

[Cor04] Intel Corporation. IA-32 Intel Architecture Optimization Reference Manual, 2004. ftp://download.intel.com/design/Pentium4/manuals/24896611.pdf . 3

[Dev04] Advanced Micro Devices. Software Optimization Guide for AMD Athlon(tm) and AMD Opteron(tm) Processors, November 2004. http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/25112.PDF . 3

[Don04] Stefan Donath. On Optimized Implementations of the Lattice Boltzmann Method on Contemporary High Performance Architectures. Lehrstuhl für Informatik 10 , Institut für Informatik, University of Erlangen-Nuremberg, August 2004. Bachelor thesis. 3, 5

[Igl03] K. Iglberger. Cache Optimizations for the Lattice Boltzmann Method in 3D. Lehrstuhl für Informatik 10 , Institut für Informatik, University of Erlangen-Nuremberg, September 2003. Bachelor thesis. 3, 4, 5

[Kow04] M. Kowarschik. *Data Locality Optimizations for Iterative Numerical Algorithms and Cellular Automata on Hierarchical Memory Architectures*. PhD thesis, Lehrstuhl für Informatik 10 , Institut für Informatik, Universität Erlangen-Nürnberg, July 2004. ISBN 3-936150-39-7. 1, 5

[WG00] Dieter A. Wolf-Gladrow. *Lattice-Gas Cellular Automata and Lattice Boltzmann Models*. Springer, 2000. 2

[Wil03] J. Wilke. Cache Optimizations for the Lattice Boltzmann Method in 2D. Lehrstuhl für Informatik 10 , Institut für Informatik, University of Erlangen-Nuremberg, February 2003. http://www10.informatik.uni-erlangen.de. 3