Modeling and analyzing performance for highly optimized propagation steps of the lattice Boltzmann method on sparse lattices

M. Wittmann*, T. Zeiser*, G. Hager*, and G. Wellein**

*Erlangen Regional Computing Center, University of Erlangen-Nuremberg, Germany
**Department of Computer Science, University of Erlangen-Nuremberg, Germany

October 3, 2014

1 Introduction

Performance optimization is important in the field of computational fluid dynamics (CFD) as it not only reduces the time to solution, but also reduces the required resources. It starts at the core and chip level and targets the critical resource, which represents the bottleneck. In most CFD approaches this is the memory bandwidth. A high quality implementation therefore should not only utilize the full available bandwidth, but also gain the best benefit from it, which means choosing a well adapted variant of the algorithm.

In case of well established lattice Boltzmann methods (LBM), there exist propagation step variants which achieve the lowest data traffic required for a lattice node update (byte/LUP) [9]: the two-grid one-step algorithm with non-temporal stores [6], Bailey et. al’s AA-pattern [1], and Geier’s Eso-Twist [3, 4]. The latter two, both, work with a single grid only and arrange the processing order in a clever way to work around data dependencies. The first two variants were successfully implemented in a fluid flow solver framework [8], which relies on a sparse lattice structure of the simulation domain [11]. This introduces indirect accesses compared to a full array approach, but delivers, if done correctly, excellent performance not only for flow in porous media like fixed-bed reactors or foams. The indirect addressing hinders at first sight vectorization and introduces additional data transfers for the lookup, which results in lower performance compared to a full lattice approach.

In this work we present a modified version of the two-grid one-step algorithm with non-temporal stores (OS-NT) and the AA-pattern algorithm, which implement a technique called RIA – reduced indirect addressing – as it allows to avoid under certain conditions the indirect access. This optimization is based on idea of run length encoding and enables

• a reduction of the data traffic per node update and
• allows for partial vectorization, which is normally hindered by indirect addressing.

Depending on the simulation geometry performance measurements of these optimizations indicate better linear scaling of the OS-NT algorithm. The AA-pattern reaches its saturation point with less
cores, which further enables the reduction of power consumption, by utilizing less cores. In case of using MPI/OpenMP, a side effect is that less processes/threads are required.

First in Sec. 2 we give a brief introduction into the lattice Boltzmann methods and how fluid nodes with adjacent obstacle nodes are treated in this work. The principles of OS-NT and AA-pattern are described in Sect. 3 and the interaction with indirect addressing is discussed. The reduction of the indirect addressing RIA and the partial vectorization are discussed in Sect. 4. These optimizations are implemented in a fluid flow solver and evaluated on current Intel processor based on the Westmere, SandyBridge, and IvyBridge microarchitectures in Sect. 5. Hereby the impact of the simulation geometry is shown. In Sec. 6 a roundup of the findings is given.

2 Lattice Boltzmann Methods

Lattice Boltzmann Methods are governed by the lattice Boltzmann equation (LBE) and result from a discretization of the velocity space and a numerical discretization of the spatial and time derivatives of the Boltzmann equation [5, 10]. The discretization model used is denoted as $D_d^q$, where $d$ represents the spatial and $q$ the velocity discretization. Typical implementations are $D_2Q9$, $D_3Q15$, or $D_3Q19$. Each lattice node comprises $q$ particle distribution functions (PDF) $f_i$ with $i = 0, \ldots, q - 1$, which are the central element for computation. The LBE is written as

$$f_i(x + c_i \Delta t, t + \Delta t) = f_i(x, t) + \Omega(f_i(x, t), f_i(x, t)), \quad i = 0, \ldots, q - 1,$$

(1)

with the PDFs $f_i$, the position vector $x$, the velocity vector $c_i$, and the collision operator $\Omega$. The right-hand side of (1) performs the collision of PDFs of the current time step, whereas on the left-hand side the propagation takes places, which transfers the newly computed post-collision PDFs to the neighbor nodes. This is visualized in Fig. 1 for $D_2Q9$ discretization. In Fig. 1a the central node is about to be updated. The read and collided PDFs are then propagated as depicted in Fig. 1b. In this work the two-relaxation time (TRT) collision operator from Ginzburg et al. [2] is used.

Despite the fact, that in (1) collision and propagation are separated in an efficient implementation they are fused, so that they are node-wise carried out together. This fusion introduces data dependencies which can be resolved by different strategies depending how the propagation, i.e. the propagation step, is implemented. Either two lattices are used, one acting as source and one as a destination lattice for the post-collision PDFs, or the propagation itself is arranged in a smart way to circumvent these dependencies.

Fluid nodes with adjacent solid nodes require a special treatment. The implementation used in this work normally streams PDFs to their neighbors. In case of a solid neighbor, as the top three nodes in Fig. 2a, when the center node is about to be updated, the bounce-back rule is applied. Hereby the direction of the PDF to be streamed to an obstacle node is inverted and stored at the origin node, as shown in Fig. 2b.

3 Propagation Steps With Indirect Addressing

Several strategies exist to implement the propagation step, i.e. the left-hand side of the lattice Boltzmann equation (1). An analysis of their minimal data traffic per node showed that the one-step two-grid algorithm with non-temporal stores (OS-NT), Bailey’s AA-Pattern, and Geier’s Eso-Twist exhibit the lowest amount of data required for one lattice update [9]. The first two are implemented in a fluid-flow solver discussed in this work. The flow-solver is optimized for porous media and therefore
Figure 1: Collision (a) of the PDFs of the centering node and the following propagation (b) of the post-collision PDFs for a D2Q9 model.

Figure 2: For the update of a fluid node with adjacent solid nodes (dashed) the collision (a) remains unchanged, but with the propagation the bounce-back rule is applied. The direction of PDFs, which would be streamed to a solid neighbor, is inverted and they are stored at the origin node (b).
operates not on a full lattice, but instead uses a 1-D vector for storing the PDFs and accompanied by an adjacency list which contains the neighborhood information. This means that PDFs of a certain node can directly be accessed, whereas access to neighboring PDFs must be performed indirect through a lookup in the adjacency list first. In Fig. 3a an example for a D2Q9 discretization is given, which only shows the directions north (N), south (S), east (E), and west (W) along the main axes. The PDFs of node 2, for example, are located at the second position from the beginning of each direction’s section inside the 1-D vector. The neighboring PDFs of this node are found through accessing the adjacency list first and following the entries in the section for node 2.

The choice for the sparse lattice seems at first side unnecessary complex, but saves memory in the case of porous media and allows an efficient implementation of the bounce-back rule with nearly no overhead. Therefore in the adjacency list for a PDF, which would be streamed to an obstacle node, the position after applying the bounce-back rule is stored.

OS-NT [6] works in that way that in the streaming step for each node first neighboring PDFs are read from the source lattice. In the case a sparse lattice is used, here an indirect access is required. The read PDFs are then collided and the post-collision values are written to the node’s location in the destination lattice. For performance reason the update loop over the nodes has been blocked and splitted, such that now for a certain number nodes first the relevant PDFs are loaded into a small temporary array. In the second step in \((q-1)/2\) loops the collision and the storing of two opposing directions is performed. As the stores require no indirect accesses and target consecutive addresses non-temporal stores can be used. These stores bypass the cache hierarchy and avoid thereby the write allocate. The splitting into \((q-1)/2\) store loops is required as only a small number of concurrent non-temporal store streams achieves a high bandwidth. Depending on the underlying processor architecture it can be beneficial to have even \(q-1\) separate loops where each handles one direction only.

The AA-pattern [1] consists of an even and an odd time step where in the even time step only the PDFs of the current node are loaded and stored to the same node, but to opposite directions. This part can use direct access and is trivial to vectorize. During the odd time step instead only neighboring PDFs are accessed, which requires indirect access as here nonconsecutive accesses can occur. This needs more effort for vectorization.

For determination of the data traffic per node update we assume double-precision floating-point numbers for the PDFs, 4-byte integers for each entry in the adjacency list (IDX) and a D3Q19 discretization.
| Prop. Step | Variation   | PDFs | IDXs | Blocks | data traffic/node |
|-----------|-------------|------|------|--------|------------------|
| OS-NT     | plain       | $2 \times q$ | $q - 1$ |        | 376              |
| RIA       | worst       | $2 \times q$ | $q - 1$ | 1      | 380              |
| RIA       | best        | $2 \times q$ |        |        | 304              |
| AA-pattern| plain       | $2 \times q$ |        |        | 304              |
|           | even        | $2 \times q$ |        |        | 304              |
|           | odd         | $2 \times q$ | $q - 1$ |        | 376              |
|           | even + odd  | $2 \times q$ | $(q - 1)/2$ |        | 340              |
| RIA       | odd, worst  | $2 \times q$ | $q$ | 1      | 380              |
| RIA       | even + odd worst | $2 \times q$ | $(q - 1)/2$ | $1/2$ | 342              |
| RIA       | odd, best   | $2 \times q$ |        |        | 304              |
| RIA       | even + odd best | $2 \times q$ |        |        | 304              |

Table 1: Data traffic per node update for the OS-NT and AA-pattern propagation step with no optimizations (plain) and reduced indirect access (RIA). Assumed are double-precision floating-point numbers for PDFs (8 bytes), 4-byte integers for an entry in the adjacency list (IDX) as well as an entry for the run length coded blocks.

Hereby only the transfers between the core and the memory are accounted for. A node update with OS-NT requires loading and storing $q$ PDFs and $(q - 1)$ times an indirect access, resulting in 376 byte/LUP. During the even and odd time step of the AA-pattern $q$ PDFs are loaded and stored, respectively. Additionally the odd time step requires for each node update $(q - 1)$ indirect accesses. The average data traffic over both time steps is then 340 byte/LUP. An overview is shown in Tab. 1 and more details are found in [9].

4 Reduced Indirect Addressing

RIA allows to avoid the indirect access for certain nodes. This is the case if two or more consecutive nodes share the same access pattern to their neighboring PDFs. Such consecutive nodes are denoted as blocks. Only for the first node inside a block an indirect access is necessary, whereas for all remaining nodes a direct access can be used. This is based on the assumption that the nodes are consecutively updated. For the first node of a block an indirect access is performed and the pointers to the neighboring PDFs, i.e. the entries inside the adjacency list, are remembered. All following nodes of this block need only to increment these pointers to the neighboring PDFs used by the first one.

The access pattern used is determined by the adjacency list and is known in advance. Therefrom starting from the beginning a vector with the number of nodes per block can be computed and equip the adjacency list with a run length encoding. These so called block vector is then used to identify if a node during simulation requires indirect addressing, if it is the first node of the block, or otherwise if a direct access can be performed.

The example in Fig. 3b shows the first entries of the adjacency list and the PDFs in the 1-D vector they are pointing to. Further the block vector is shown, with the first two blocks containing only one node each, as neither node one and two nor two and three share the same access pattern. The third block comprises two nodes, namely three and four, which exhibit the same pattern. When the neighbor
| Processor       | Westmere         | SandyBridge      | IvyBridge        |
|-----------------|------------------|------------------|------------------|
| Intel Xeon      | Intel Xeon       | Intel Xeon       |
| X5650           | E5-2680          | E5-2660          |
| Frequency [GHz] | 2.67             | 2.70             | 2.20             |
| Cores           | 6                | 8                | 10               |
| ISA extension   | SSE 4.2          | AVX              | AVX              |
| Copy-NT-SL-19A  | [GB/s]           |                  |                  |
| 15.9            | 33.9             | 34.0             |
| Update-19A      | [GB/s]           |                  |                  |
| 21.3            | 36.2             | 36.9             |

Table 2: Specification of the processors and platforms used for performance evaluation. Two micro-benchmarks were used to determine the attainable single socket memory bandwidth with the access pattern used by the OS-NT and AA-pattern propagation steps.

| Channel | Fixed-Bed Reactor |
|---------|-------------------|
| Dimensions | 500 × 100 × 100 | 500 × 100 × 100 |
| Nodes | 4,800,000 | 2,100,000 |
| Blocks [%] | 3 | 39 |
| Vectorizable (SSE/AVX) [%] | 98/98 | 65/58 |
| OS-NT, RIA [byte/LUP] | 306 | 333 |
| AA pattern, RIA [byte/LUP] | 305 | 319 |
| Data Size | | |
| 1-D Vector [MB] | 730 | 319 |
| Adj. List [MB] | 345 | 151 |

Table 3: Characteristics of the empty channel and fixed-bed reactor benchmark geometries.

PDFs of node three are required an indirect access is mandatory. As the block’s count is two, for accessing the neighbors of node four, the pointers into the 1-D vector obtained previously just need to be incremented by one. Depending on the fraction of nodes which can utilize direct access the required data traffic per node decreases for OS-NT and AA-pattern. The impact of RIA on this metric for these propagation steps is listed in detail in Tab. 1.

In the best case for OS-NT and AA-pattern no indirect access is necessary. This means only very few large blocks exists, which results in 304 bytes/LUP for both propagation steps. In the worst case where only blocks of size one exist an additional lookup in the block vector is required. Here OS-NT requires 380 bytes/LUP and AA-pattern 342 bytes/LUP.

For the odd time step of the AA-pattern with this optimization it is now possible to execute blocks vectorized, which leads to a partial vectorized (PV) implementation of this time step. As nodes inside a block can be treated the same SIMD instructions can be used to load, collide, and store them en block. Previously a vectorization would have included to implement manual gather and scatter operations to load PDFs from memory into the SIMD registers or write them back. The only requirement now is that the number of nodes inside a block is larger or equal to the number of the vector width, i.e. the number of operands a SIMD-operation can be applied to. This is two for SSE and four for AVX, when double-precision is used.
5 Results

As fluid flow solver the ILBDC [11] is used, which is based on the lattice Boltzmann methods and uses the TRT collision operator [2], a D3Q19 discretization model, double-precision arithmetic, and a structure of arrays data layout (SoA). For performance evaluation three platforms were utilized, which are based on Intel’s Westmere, SandyBridge, and IvyBridge microarchitecture, respectively. Details of the used platforms are given in Tab. 2. All processes of the MPI-parallel fluid flow solver were pinned to a separate core and the processors’ frequency was fixed to the base frequency listed in Tab. 2. As benchmark geometries an empty channel and a fixed-bed reactor were employed with the features listed in Tab. 3.

The roofline performance model [7] is used to estimate the achievable performance of the fluid flow solver with the two propagation steps and their optimizations. As all discussed implementations are memory bound, i.e. the performance is limited by the memory bandwidth, an estimation can be obtained by dividing the attainable memory bandwidth by the data traffic per node update. The memory bandwidth an application can reach depends on their access pattern. For OS-NT and AA-pattern two micro-benchmarks Copy-NT-19A and Update-19A were constructed, which resemble the propagation steps’ access patterns. Copy-NT-19A copies 19 arrays concurrently in chunks by using non-temporal stores and Update-19A updates 19 arrays in parallel. Both benchmarks use direct addressing only. The measured bandwidth for full sockets is listed in Tab. 2. The anticipated performance values are shown in Fig. 4 and 5 as horizontal lines.

Figure 4a and 4b summarizes the results of OS-NT and AA pattern on the evaluated platforms, respectively. The propagation step OS-NT in Fig. 4a nearly reaches the predicted performance. As the algorithm is memory-bound it scales with the bandwidth, which is doubled from Westmere to Sandy-Bridge. With the RIA optimization the performance gain should be larger according to the roofline model as it is measured. Only if the saved amount of data traffic is nearly at the maximum, as it is the case with the channel geometry, an improvement is visible. For the fixed-bed reactor where the data traffic per node can only be reduced by 10% no or even a negative impact is observed. The performance characteristics for the AA-pattern, shown in Fig. 4b, are nearly the same. Only for the channel geometry performance is increased. With partial vectorization enabled for this geometry on the SandyBridge and IvyBridge system the performance can be further raised.
In the next step, the scaling behavior over the cores of a socket is evaluated. This is only shown for the IvyBridge platform as the other two systems share the same features. The performance of the empty channel is shown in Fig. 5a. The optimized version of OS-NT with reduced indirect addressing scales better with the number of cores than the plain version and achieves an 8% better performance when utilizing the full socket. By the reduction of the data traffic per node update from 376 to 306 byte/LUP actually an improvement of 18% would be expected. In the case of the fixed-bed reactor in Fig. 5b the RIA version is slightly worse than the plain version. Here an improvement of 10% was anticipated (376 vs. 333 byte/LUP).

When using all cores of a socket the difference for AA-pattern with and without RIA is an increase in performance by 6%. With nine and ten cores AA with RIA still gains an performance improvement where the plain version has already saturated. As with OS-NT the anticipated improvement of 18% is not reached. With the usage of partial vectorization (PV) in addition to RIA the performance benefit is only minimal, but the scaling over the number of cores changes dramatically. With one core already an 70% increased performance to AA and AA with RIA versions is observed with the channel geometry where 98% of the node updates could vectorized. This is clearly a result of the vectorization as the data traffic per node is the same with and without PV. The results for the fixed-bed reactor in contrast are only slightly better then the plain version. The improvement with PV is only seen when the performance is not yet saturated and turns out to be much smaller as the vectorizable amount of nodes is only around 60%.

6 Conclusion

Reduced indirect addressing (RIA) avoids for certain nodes the indirect access and thereby reduces the data traffic per node update. This should for this memory-bound lattice Boltzmann algorithm directly translate into a performance increase. The quantity is now determined by the structure of the simulation domain as only for consecutive nodes sharing the same access pattern to their neighboring PDFs the indirect access can be skipped. The implementation of RIA for the propagation steps OS-NT and AA-pattern showed that savings of ≈ 10% on the data traffic per node update are not enough.
to see any change in performance. If in contrast a simulation geometry allows a higher reduction, like an empty channel, an performance increase of $\approx 6 \text{–} 8\%$ is observed. This alone would probably be not worth the effort of the implementation, as the the empty channel represents the best-case geometry. For the AA-pattern with RIA the odd time step can partially be vectorized. Hereby the saturated performance is not higher than with only RIA enabled, but saturation can be reached with less cores. This gives the ability to reduce the number of MPI processes or threads, e.g. when OpenMP is used. Furthermore this reduces power consumption, as unused cores can be powered down without any performance impact.

This work was financially supported by KONWIHR III and partially supported by BMBF under grant No. 01IH08003A (project SKALB).

References

[1] P. Bailey, J. Myre, S. Walsh, D. Lilja, and M. Saar. Accelerating lattice Boltzmann fluid flow simulations using graphics processors. In International Conference on Parallel Processing 2009 (ICPP’09), pages 550–557, Sept 2009. doi 10.1109/ICPP.2009.38.

[2] I. Ginzburg, F. Verhaeghe, and D. d’Humières. Two-relaxation-time lattice Boltzmann scheme: About parametrization, velocity, pressure and mixed boundary conditions. Commun. Comput. Phys., 3(2):427–428, 2008.

[3] J. Linxweiler. Ein integrierter Softwareansatz zur interaktiven Exploration und Steuerung von Strömungssimulationen auf Many-Core-Architekturen. PhD thesis, 2011. (in German).

[4] M. Schönherr, M. Geier, and M. Krafczyk. 3D GPGPU LBM implementation on non-uniform grids. In International Conference on Parallel Computational Fluid Dynamics 2011 (Parallel CFD 2011), May 2011. http://parcfd2011.bsc.es/sites/default/files/abstracts/id121-schonherr.pdf.

[5] S. Succi. The Lattice Boltzmann Equation: For Fluid Dynamics and Beyond. Numerical Mathematics and Scientific Computation. Oxford University Press on Demand, 2001.

[6] G. Wellein, T. Zeiser, S. Donath, and G. Hager. On the single processor performance of simple lattice Boltzmann kernels. Computers & Fluids, 35:910–919, 2006. doi 10.1016/j.compfluid.2005.02.008.

[7] S. Williams, A. Waterman, and D. Patterson. Rooftline: an insightful visual performance model for multicore architectures. Commun. ACM, 52(4):65–76, Apr 2009. doi 10.1145/1498765.1498785.

[8] M. Wittmann, G. Hager, T. Zeiser, and G. Wellein. An analysis of energy-optimized lattice-Boltzmann CFD simulations from the chip to the highly parallel level. Submitted as journal paper, 2013. arXiv 1304.7664.

[9] M. Wittmann, T. Zeiser, G. Hager, and G. Wellein. Comparison of different propagation steps for lattice Boltzmann methods. Computers & Mathematics with Applications, 65(6):924–935, 2013. doi 10.1016/j.camwa.2012.05.002.

[10] D. Wulf-Gladrow. Lattice-Gas Cellular Automata and Lattice Boltzmann Models: An Introduction. Number 1725. Springer, 2000.

[11] T. Zeiser, G. Hager, and G. Wellein. Benchmark analysis and application results for lattice Boltzmann simulations on NEC SX vector and Intel Nehalem systems. Parallel Processing Letters, 19(4):491–511, 2009. doi 10.1142/S0129626409000389.