Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 4 (2011) 984–993

Procedia Computer Science

International Conference on Computational Science, ICCS 2011

Free-Surface Lattice-Boltzmann Simulation on Many-Core Architectures

Martin Schreiber[a], Philipp Neumann[a], Stefan Zimmer[b], Hans-Joachim Bungartz[a]

[a] *SCCS, Technische Universität München, Germany*
[b] *IPVS, Universität Stuttgart, Germany*

Abstract

Current advances in many-core technologies demand simulation algorithms suited for the corresponding architectures while with regard to the respective increase of computational power, real-time and interactive simulations become possible and desirable. We present an OpenCL implementation of a Lattice-Boltzmann-based free-surface solver for GPU architectures. The massively parallel execution especially requires special techniques to keep the interface region consistent, which is here addressed by a novel multipass method. We further compare different memory layouts according to their performance for both a basic driven cavity implementation and the free-surface method, pointing out the capabilities of our implementation in real-time and interactive scenarios, and shortly present visualizations of the flow, obtained in real-time.

Keywords: Lattice Boltzmann, Free-Surface, Many-Core, GPU, OpenCL, Interactive, Steering, Visualization

1. Introduction

Graphics Processing Units (GPUs) have turned out to provide a suitable architecture for the simulation of physical problems [1, 2]. With many-core systems such as GPUs possessing an increased computational power compared to traditional multi-core CPUs, these massively parallel systems are particularly used for problems that require fast response times as it is the case for interactive or real-time computations. This particularly holds for interactive flow visualization [3] and simulation in computer games [4]. With regard to the aspects of real-time fluid simulation and interactivity, the Lattice-Boltzmann Method (LBM), an approach to numerically compute flows via a discrete form of the Boltzmann equation [5], appears to be a promising approach (cf. [6, 7]), as big parts of its basic algorithm can be evaluated in a strictly local (cellwise) manner, fitting perfectly to GPUs. In the following, we present a Lattice-Boltzmann based free-surface algorithm adopted for the GPU architecture to simulate a two-phase flow consisting of gas and liquid as it is the case for water and gas. As porting more complex algorithms like the respective free-surface extension to graphic cards frequently constitutes a non-trivial task, several modifications become necessary in order to fit the algorithm to the massive concurrency execution model. We discuss respective modifications in the underlying algorithm, show performance results and discuss them with special respect to the memory demand and storage patterns

Email addresses: martin.schreiber@in.tum.de (Martin Schreiber), neumanph@in.tum.de (Philipp Neumann), stefan.zimmer@ipvs.uni-stuttgart.de (Stefan Zimmer), bungartz@in.tum.de (Hans-Joachim Bungartz)

Selection and/or peer-review under responsibility of Prof. Mitsuhisa Sato and Prof. Satoshi Matsuoka
doi:10.1016/j.procs.2011.04.104

used in the simulation setups. We further hint at realistic visualization techniques allowing for real-time rendering of the running simulation and point out the real-time capability of our software.

The basic theory of the Lattice-Boltzmann method and the respective free-surface algorithm [8] is reviewed in Sec. 2. Its implementation is extensively explained in Sec. 3. Herein, special focus is put on the choice of different memory layouts, storage patterns and algorithmic difficulties that need to be particularly addressed due to the underlying GPU architecture. Performance results for both the basic Lattice-Boltzmann implementation and its free-surface extension are reported in Sec. 4. An example for realistic real-time visualizations of the free-surface is given in Sec. 5. Finally, we close the discussion and give short conlcusions and an outlook on future work in Sec. 6 and 7.

2. Free-Surface Simulations using the Lattice-Boltzmann Method

2.1. Lattice-Boltzmann Method

Since the fundamental equations of the LBM including respective notations will be used throughout this paper, we briefly review them in this section. In the following, we refer to this standard discretisation and its implementation, i. e. to the LBM without free-surface handling, as the *basic LBM*.

The Lattice-Boltzmann scheme computes the particle distribution functions (PDFs) $f_i(\boldsymbol{x}, t)$ representing the probabilities to find molecular populations with velocity $\vec{e}_i \in \mathbb{R}^D$ ($i \in \{1, ..., Q\}$) inside a cubic cell centered around $\boldsymbol{x} \in \mathbb{R}^D$ at time t. The set of discrete lattice velocities $(\vec{e}_i)_{i=1,...,Q}$ is chosen such that the velocities form a minimal isotropic set needed to recover the Navier-Stokes equations in the macroscopic limit [9]. Our implementations are based on the D3Q19 discretization providing a good balance between numerical stability and accuracy [10], however the theory of the described algorithms carries over to all other next neighbour velocity discretization schemes.

The update rules for the dimensionless particle distribution functions f_i based on a single relaxation time approximation for intermolecular collision processes [11] read

$$\begin{array}{lcll} f_i^c(\boldsymbol{x}, t) & := & f_i(\boldsymbol{x}, t) - \frac{1}{\tau}\left(f_i(\boldsymbol{x}, t) - f_i^{eq}(\rho(\boldsymbol{x}, t), \vec{u}(\boldsymbol{x}, t))\right) & \text{collision step} \\ f_i(\boldsymbol{x} + \vec{e}_i, t+1) & := & f_i^c(\boldsymbol{x}, t) & \text{propagation step} \end{array} \tag{1}$$

where τ is the fluid specific relaxation time and $f^{eq} \in \mathbb{R}^Q, \rho \in \mathbb{R}, \vec{u} \in \mathbb{R}^D$ the usual expressions for the local equilibrium distribution, fluid density and fluid velocity of the system [5]. The collision step only implies local distribution functions of a single cell whereas the propagation step copies the PDFs f_i to the adjacent cells. The inclusion of external forces such as gravity which is essential for realistic free-surface simulations can be accomplished in a local manner during the collision step, see amongst others [12]. At the boundaries, the PDFs which would have entered the computational domain from outside by the propagation step need to be reconstructed. We only consider fixed wall boundaries where we apply the standard bounce back rule [5].

2.2. Free-Surface Model

There are different ways to include an adaption of the basic LBM to free-surface flows [8, 13, 14]. For our choice, the model presented in [8] builds the general starting point for the implementation. We decided to develop the many-core implementation based on the model developed by Körner et al. [8] as it has already been validated for various types of simulations [12] and implemented in the 3D modeling and rendering software Blender [15]. Besides, a simplified GPU implementation of this model including an OpenGL shader technique has previously been studied and tested by one of the authors[1] yielding promising results with respect to the challenge of running LBM-based free-surface simulations interactively in real-time on massively-threaded GPUs. In the following, we briefly review the methodology of the underlying model as we consider it to be essential for the understanding of the GPU implementation in Sec. 3.

Free-surface simulations are characterized by the fact that the discrete cells of the flow domain dynamically change their physical state by getting filled or emptied due to the movement of the liquid that is simulated. In order to keep track of the different states of the cells, a cell type $T(\boldsymbol{x}, t)$ is assigned to each cell, allowing to distinguish between obstacle (O), fluid (F), gas (G) and interface cells (I). Cells are regarded to be of fluid or gas type, if they are either

[1] http://www.martin-schreiber.info/lattice_boltzmann_opengl.html

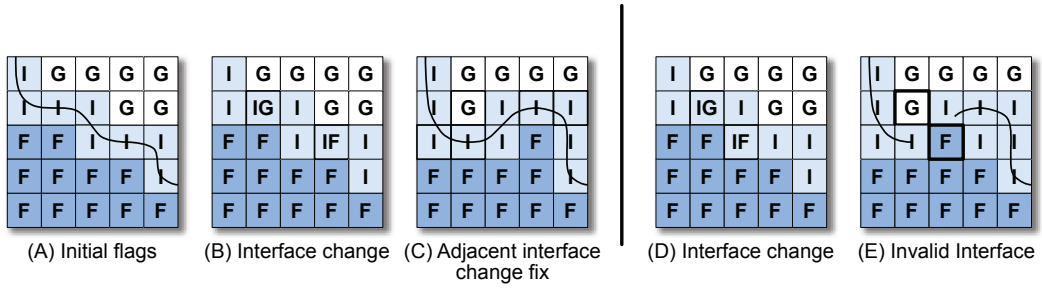

Figure 1: (A): Original interface separating gas from fluid cells. (B): Interfaces converted to gas cells (IG) and interface cells converted to fluid cells (IF). (C): Adjacent cells of IG and IF cells from (B) are converted to maintain a gapless interface. (D): State after interface conversions to IG cell with adjacent fluid IF cell. (E): Possible invalid state due to race conditions: Gas cell is directly adjacent to fluid cell.

completely filled by fluid or completely empty. Interface cells mark the border between gas and fluid phase and are partially filled with fluid, that is their fluid fraction $\epsilon = \frac{m}{\rho}$ amounts to a value $0 < \epsilon < 1$ whereas $\epsilon = 1$ for filled cells and $\epsilon = 0$ for empty cells (m denotes the fluid mass inside a lattice cell). After each timestep, the interface layer must separate all gas cells from fluid cells to maintain a consistent state. As the amount of fluid in the cells may vary over time, one keeps track of the mass flux between neighbouring cells by measuring the mass exchange $\Delta m_i(\boldsymbol{x}, t+1)$ arising from the incoming and outgoing populations $f_i(\boldsymbol{x}, t)$ and $f_{inv(i)}(\boldsymbol{x} + \vec{e}_i, t)$ at the lattice node $\boldsymbol{x}$:

$$\Delta m_i(\boldsymbol{x}, t+1) = A_i \cdot \left(f_{inv(i)}(\boldsymbol{x} + \vec{e}_i, t) - f_i(\boldsymbol{x}, t)\right), \tag{2}$$

where A_i is the fluid exchange area between both cells. Its value is approximated depending on the types of the neighbouring cells and their fluid fractions, see [12] for details. The total mass of the cell for the next timestep is thus determined by the mass of the current timestep $m(\boldsymbol{x}, t)$ and the sum over all mass fluxes $\Delta m_i(\boldsymbol{x}, t+1)$; the fluid fraction is updated respectively and a reflagging of the cell types becomes necessary before the next timestep as depicted in Fig. 1. Filled cells are flagged to F, emptied cells to G and partially filled cells to I. Afterwards, a closed layer of interface cells needs to be constructed between gas and liquid phase in order to retain a valid domain configuration (cf. Fig.1 (A)-(C)). Besides, missing PDFs in the interface cells originating from the gas phase are reconstructed using a pressure-type boundary condition:

$$f_i = f^{eq}_{inv(i)}(1, \vec{u}) + f^{eq}_i(1, \vec{u}) - f_{inv(i)} \tag{3}$$

where the velocity $\vec{u}$ is interpolated from adjacent fluid cells and further influences of the gas onto the liquid phase are neglected (see for example [8]).

3. GPU Implementation

In the following, we focus on the implementational concept of the free-surface algorithm using the Open Computing Language (OpenCL) and start with a very brief introduction to OpenCL semantics for NVIDIA GPUs in Sec. 3.1 (see [16, 17] for detailed descriptions). Since our main purpose is to develop a LBM framework which is capable of running on different architectures, we choose OpenCL to enhance the portability for further work. In the implementation optimized for our GPU, we did not observe major performance differences compared to existing CUDA implementations [18, 7]. Subsequently, we address the underlying memory access patterns (Sec. 3.3) as well as the memory layout and storage requirements (Sec. 3.2) as these points are of essential importance for performance considerations. As many GPU implementations of Lattice-Boltzmann schemes have been discussed and published in recent years [19, 20, 21, 6, 7, 22], we will not go into much detail on our basic implementation and particularly focus on the treatment and respective reflagging of the cell types in the free-surface simulations posing a major challenge with respect to many-core architectures. This topic is described in Sec. 3.4.

3.1. OpenCL

Using OpenCL semantics, the NVIDIA GPU architecture is divided logically into subsets of *local work groups* which are executed in parallel depending on the available ressources. A local work group itself represents a group

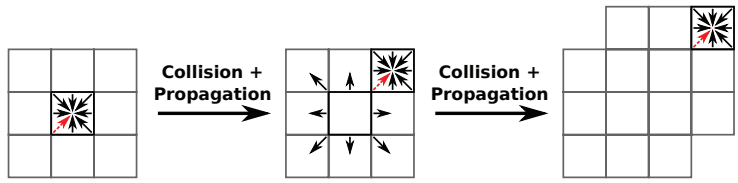

Figure 2: A-B pattern for collision and propagation operator: Both steps can be combined when different buffers are used for reading and writing.

of *work items* which are executed on *processing elements* (PEs). For concurrency and efficiency reasons, the PEs are grouped into *half warps* with 16 PEs per half warp.

The development of the algorithms and the accomplishment of the respective implementations typically aim at maximizing the *coalesced memory accesses* as this is important to maximize both the bandwidth utilization and the performance of many GPU programs. Considering compute capability 1.2 (as provided by the GPU that we used), the only restriction to get a coalesced memory access for PEs executing a kernel on a half warp is that all processing elements have to access the memory within a single memory segment to issue a single memory transaction only. Accessing words during a read and write (RAW) operation which are not inside the segment creates requests for further memory segment transactions. The overall execution time of all kernels is further reduced by the asynchronous execution of warps using the processing elements of other warps with work items waiting for the memory transaction to be finished [23]. However, this has several restrictions given by the maximum number of local work groups which can be executed in parallel due to limited ressources.

At the beginning of any GPU computations, the data relevant for the computations are stored in the *global memory* with a high access latency. Beside the global memory buffers, a separate *local memory* is available for each local work group with a latency similar to registers but with a limitation of 16 KB on the GPU hardware used in our benchmarks. A frequent utilization of this local memory is to increase the coalesced memory access if regular datasets have to be read; however, it does not always lead to performance improvements (see Sec. 4.2).

3.2. Memory Layout and Storage Requirements for the Basic Implementation

We first consider the necessary storage demand. The following datasets have to be accessed by the kernels to run the simulation: 19 floats for the PDFs, 1 integer for the cell flags, 3 floats for the velocity and 1 float for the density with the PDFs and cell flags being the only values used in the next timestep. The velocity and density values are stored for visualization purposes or further processing steps.

In our implementation, one cell is handled by one work item. The PDFs along lattice vectors pointing in one direction are stored consecutively in the global memory. To avoid the creation of any further memory buffers the respective 19 blocks of PDFs (D3Q19 discretization) are again stored consecutively. A pointer to the PDFs is used to avoid several indexing instructions for accessing different lattice vectors which leads to less PTX [24] instructions.

Similar to the storage pattern of the PDFs, the velocity field is stored, aligning the vector entries componentwise in the memory. For the density values, the cell type flags and the velocity vectors, coalesced memory access is assured at any time.

3.3. Memory access patterns for PDFs

This section gives an overview of existing memory access patterns in LBM simulations with special regard to their memory demands as this is one of the main bottlenecks in Lattice-Boltzmann simulations [25]. We address two access patterns that are used in the context of LBM computations and which are known as A-B and A-A pattern [18].

The A-B pattern uses two memory buffers for the PDFs; the respective memory access is demonstrated in Fig. 2. The first buffer is used for reading the current values of the PDFs. These values are processed as described by Eq. (1) combining collision and propagation into a single iteration over the data. As the propagation of the PDFs to adjacent cells can lead to RAW conflicts with different work items computing the timestep of adjacent cells concurrently, the second buffer is used to store the PDFs. After streaming, the memory buffer handlers are swapped and the solution of the subsequent timestep can be computed.

In order to reduce the memory demand of two complete PDF fields in Lattice-Boltzmann simulations, Bailey et al. proposed a new memory access pattern in [18], allowing for LBM computations on GPUs with only one field

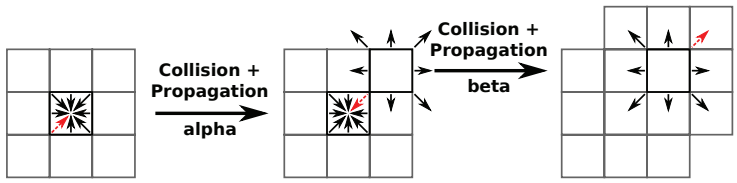

Figure 3: A-A pattern for collision and propagation operator. Read and write operations are executed on the same memory buffer.

cell type flag	abbrevation	usage
FLAG_INTERFACE	I	Interface cell
FLAG_INTERFACE_TO_FLUID	IF	The interface cell is converted to a fluid cell
FLAG_INTERFACE_TO_GAS	IG	The interface cell is converted to a gas cell
FLAG_GAS_TO_INTERFACE	GI	The gas cell is converted to an interface cell

Table 1: Phase, boundary and multipass flags used in the free-surface algorithm

of PDFs. However, in order to overcome the RAW conflicts mentioned before, different access patterns are required for odd and even timesteps which are depicted in Fig. 3. We distinguish the kernels handling the different memory accesses by denoting them *alpha* and *beta kernel*.

Similar to the A-B pattern, the A-A pattern combines the collision and propagation step into a single pass. For the *alpha kernel* the PDFs are read in the same way as it is done in the A-B pattern. After applying the local collision or boundary operator, the PDFs are stored to the memory location of the opposite lattice vector belonging to the same cell. Assuming a linear enumeration of the work items, all accesses to the PDFs are coalesced.

While the alpha kernel only works on data from the local cell, the beta kernel in turn only accesses data outside the current cell. First, the PDFs are read from the neighbours. After the collision and boundary treatment, the PDFs are written to the data storage of the cells which lie in the opposite direction of the cell from which the specific PDFs were read from.

3.4. Free-Surface LBM on GPUs

A major topic in LBM-based free-surface simulations consists in the reflagging of the cells according to their phase and boundary state. For single core or more general distributed memory implementations, double linked lists are typically used [12] to store the interface cells yielding an efficient traversal strategy for the interface. However, these implementations do not fit the GPU programming paradigm since it is particularly the reflagging part of the algorithm in which race conditions can occur when avoiding any synchronization mechanisms such as mutual exclusions; one example for race conditions occurring in the reflagging algorithm is exemplarily depicted in Fig. 1 (D) and (E).

To circumvent these problems, we propose a *multipass method* with its single passes depicted in Fig. 4; the passes that have been included due to existing race conditions are described in more detail below. In this context, further cell states (beside the phase and boundary flags defined in Sec. 2.2) are introduced and listed in Tab. 1. The actions of kernels denoted with a "*F:*" prefix are only executed if the current flag of the cell equals the flag defined after "*F:*".

OUTGOING MASS: This kernel is needed to overcome RAW conditions occurring for the A-A pattern only. In this case, the outgoing mass is determined and subtracted from the mass. For the A-B pattern, this kernel was omitted by implementing the mass tracking in the upcoming kernel.

COLL. AND PROP.: First the flag of the current cell is loaded. Consecutively, all opposite values of PDFs are loaded and the local velocity and density values are computed right after the loading. In case of an interface cell the incoming PDFs are updated according to Eq. (3) right after two opposite PDFs have been loaded.

The enumeration of the PDFs was optimized similar to the OpenGL implementation of [21]. This helps to hide the memory latency by starting some computations such as reconstructing the incoming PDFs of gas cells immediately after two opposite PDFs have been read instead of starting the computations after all PDFs have been loaded.

If the cell is flagged as obstacle cell, the bounce back method is applied. If the fluid fraction exceeds $1+\kappa$, the flag of the cell is updated to *FLAG_INTERFACE_TO_FLUID*; $\kappa = 10^{-2}$ is used as stabilization of the interface layer (see [12]). Finally the PDFs, the density, the velocity and the potentially updated flag are written back to global memory.

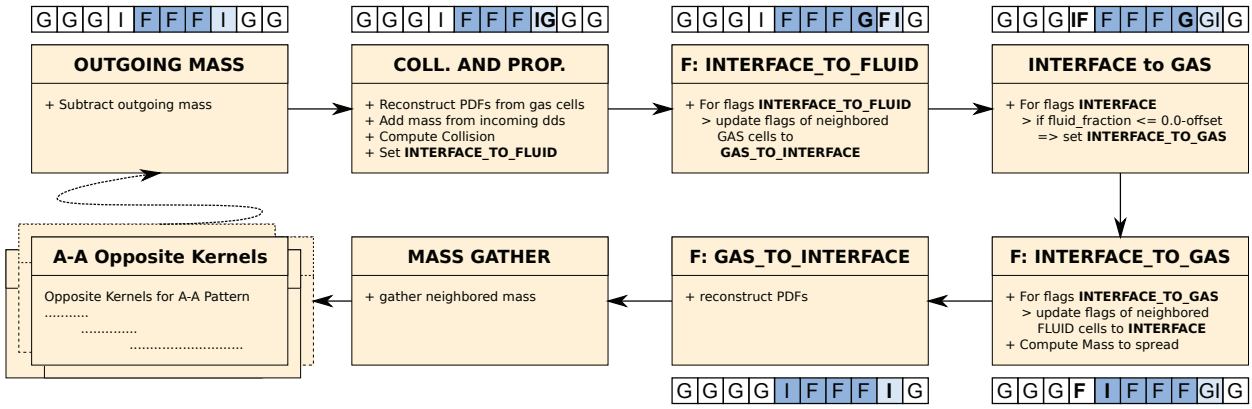

Figure 4: Overview of the LBM free-surface handling for one kernel (alpha or beta). The kernels marked with "*F: flag_name*" handle cells with the flag "*flag_name*". The flags are given for a 1D domain, assuming that both interface cells change their state.

F: INTERFACE_TO_GAS: This kernel handles interface cells which are going to be converted to gas cells. All adjacent cells of type *FLAG_FLUID* are set to *FLAG_FLUID_TO_INTERFACE*. To avoid any race conditions to previous kernels this is implemented in a separate pass. Otherwise, a race condition could occur, if an adjacent cell was converted to a fluid cell after the current cell has been converted to a gas cell. Then the interface between gas and fluid cells would not be consistent anymore.

A-A Opposite Kernels:
For the A-A pattern, the access of the PDFs is different for the alpha and beta kernel. Assuming that the previously described kernels account for memory access of type alpha, the *A-A opposite kernels* box stands for the kernels executed for the access patterns of type beta. Since the PDFs are accessed in a different manner, some of the mentioned kernels that access the PDFs need to be modified. The kernels *F: INTERFACE_TO_FLUID*, *F: INTERFACE_TO_GAS*, *INTERFACE_TO_GAS*, however, can be reused.

4. Results

In this section, we first show performance results for our basic Lattice-Boltzmann implementation. Next, the results for the free-surface implementation and its different memory patterns are given. The benchmarks have been created on a "Zotac NVIDIA GeForce GTX 285 amp!".

4.1. Results for basic LBM implementation

The basic LBM implementation was developed to *validate* and *test* the environment and also to allow a *comparison* with the free-surface implementation. A fair performance comparison of different LBM-based codes running on GPUs is difficult to establish, as – beside possible differences in the numerical methods (e.g. different velocity discretization schemes) – differences in the utilized driver versions and the underlying hardware play crucial roles. However, the following results of our basic implementation are similar to those obtained by other groups (cf. [18]). An exemplary benchmark is given in Fig. 5 for the A-A pattern running a driven cavity simulation for different domain sizes and measuring the *Mega Lattice Updates Per Second* (MLUPS). Storing the velocity and/or the density values was disabled. As expected, the implementations with a local work group size which are not a power of 2 are slower for most domain sizes. Comparing both benchmark diagrams, it is clearly visible that utilizing the *local memory is advantageous for the basic implementation* due to the increased coalesced accesses.

4.2. Free-Surface Implementation

Next, we show the performance results for the free-surface flow model using the multipass method and the different memory layouts described in Sec. 3.3. We chose the breaking-dam problem as test scenario with one third of the cubic domain initially filled with water. The performance graphs for different resolutions and work group sizes are given in Fig. 6 for several implementations and are explained in detail below.

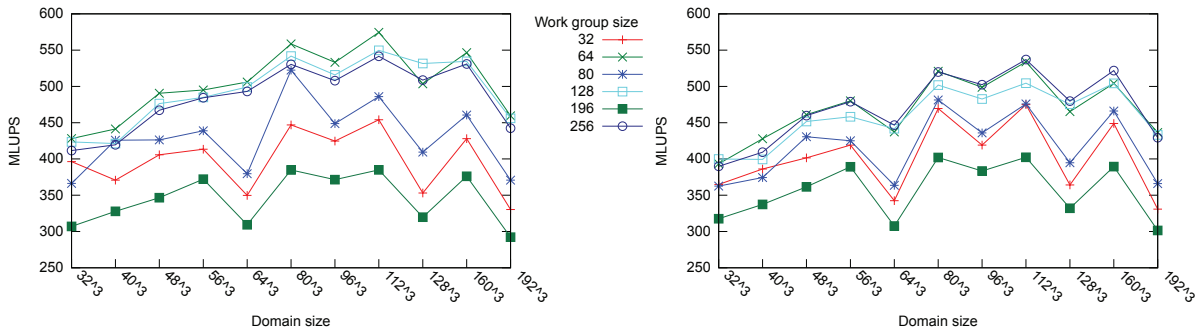

Figure 5: Driven cavity benchmark with (left image) and without (right image) utilizing the local memory.

4.2.1. A-B Pattern benchmarks for total runtime:

For this implementation two memory buffers are used due to the RAW race conditions between PDFs which even exist for the basic implementation. Since no RAW race conditions exist for the computation of the outgoing mass and the modified PDFs in the collision and propagation kernel, the general advantage of this pattern for free-surface flows is to join the kernels *OUTGOING MASS* and *COLL. AND PROP.*.

A-B pattern, version 1: In the first version of the A-B pattern, the PDFs are read with offsets from adjacent cells similar to the beta kernel avoiding an overall coalesced memory access and are written without any offset to the storage of the current cell. The advantage of this implementation is that PDFs for *reconstructed values of interface cells* can be stored *without any displacement*.

A-B pattern, version 1 utilizing the local memory: This version is similar to the version 1 but utilizes the local memory in the collision and propagation kernel leading to an increase in the coalesced memory access. In contrast to the basic implementation, utilization of *local memory for free surfaces resulted in a performance decrease* for our breaking dam test scenario. Since the kernels have to read PDFs for adjacent cells, they are not allowed to quit the kernel if the currently processed cell is a gas cell leading to lower performance.

A-B pattern, version 2: The PDFs are read without any displacements and have to be written with displacements in the memory. The main disadvantage of this implementation is that *reconstructed interface PDFs* have to be *written with a displacement* slightly slowing down the performance.

4.2.2. A-A Pattern benchmarks for total runtime:

The memory access for PDFs in this implementation is equal to the basic implementation for the A-A memory access pattern. Since the kernels *OUTGOING MASS* and *COLL. AND PROP.* cannot be joined, the execution time for one timestep compared to the A-B pattern implementations is increased. Due to the results of the A-B pattern utilizing the local memory, a similar A-A pattern implementation was disregarded.

4.2.3. Benchmarks for individual kernel runtimes:

According to Fig. 7, the alpha and beta kernels of the A-A pattern show significant differences in their runtimes for a domain resolution of 128^3 due to different amounts of register utilization and the access of data without/with displacements in the alpha/beta kernel. For the alpha kernel, this advantage gets to a disadvantage for the *OUTGOING MASS (alpha)* since the PDFs of adjacent cells now have to be accessed. The *F: GAS_TO_INTERFACE* beta kernel takes a longer time compared to the respective alpha kernel since the velocity components are read with a displacement and also the computed PDFs (based on the velocity values) are written to the adjacent cells avoiding a full coalesced access. Thus, the overall computation for one timestep takes more time compared to the A-B pattern.

In order to obtain best performance, empirical parameter studies often need to be carried out determining optimal hardware related parameters for specific hardware platforms. Particular studies relevant for our simulations are to setup individual work group sizes for each kernel or to limit the register usage for each kernel accomplished via an NVIDIA OpenCL extension [26]. For example, we chose an arbitrary local work group size of 160 for the collision and propagation kernel and a domain size of 128^3 cells. Running multiple benchmarks resulted in an average

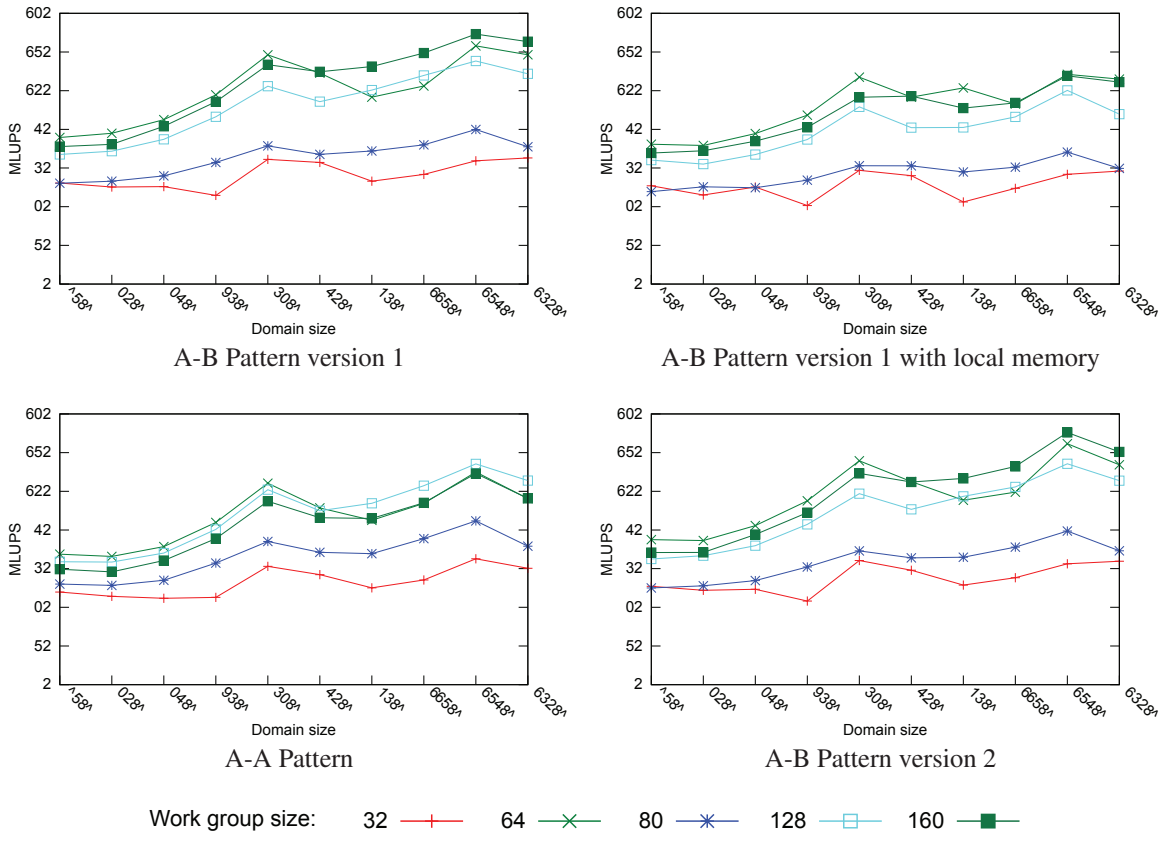

Figure 6: Free surface breaking-dam benchmark results for different work group, domain sizes and different pattern.

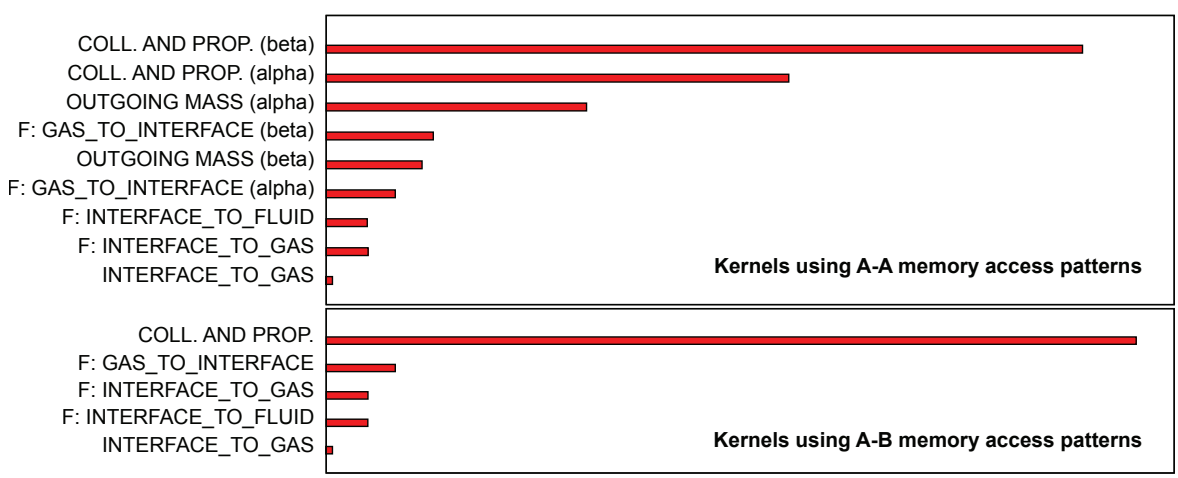

Figure 7: Relative runtime for 100 timesteps of a breaking-dam simulation with a domain resolution of 128^3. The *MASS GATHER* kernel was disabled. Top image: Relative execution time for A-A memory access patterns. The runtime for kernels which are shared among alpha and beta types are only given once. Bottom image: Relative execution for A-B memory access patterns.

Figure 8: Left: Breaking-dam simulation with volume tracing in real-time Right: Simulation, volume tracing and photon mapping in real-time

performance of 129 MLUPS. Optimizing the MLUPS by modifying the local work group size for each kernel individually, a maximum increase of 3 MLUPS could be reached in average, setting the local work group size of the kernel *INTERFACE_TO_GAS* and *COLL_AND_PROP* to 160 and those of the other kernels to 256. Since the collision and propagation kernel is computationally the most intensive part as shown in Fig. 7, individual work group size optimizations for the other kernels only lead to small performance increases. Limiting the maximum registers available for a kernel led to decreases in performance.

5. Visualization

In order to simulate free-surface water flows in real-time, different scenarios such as breaking-dam and falling drop simulations have been set up using a domain resolution of $64 \times 64 \times 32$, a dimensionless maximum gravity of 0.0156 in the collision step and limiting the maximum dimensionless velocity to 0.25. With respect to these settings, we could establish water simulations at a performance rate of approx. 113 MLUPS. With a single timestep representing 0.0027 seconds in real-time, this implies the simulation of approx. 862 timesteps per second which corresponds to 2.32 (real-time) seconds. Besides, the GPU work load has been measured to be 43% in this scenario. Thus, there are still computational ressources left on the GPU to simultaneously visualize the flow.

For this purpose, we use the values of the fluid fraction as voxel values in volumetric textures. The surface of a fluid cell rendered on screen covers more pixels compared to a rendering of high resolution volumetric datasets and thus creates different demands such as increased intersection accuracy compared to the existing algorithms, while on the other hand, approaches like empty space skipping [27] are not expected to yield improved performance for low domain resolutions. As the discussion of the underlying visualization techniques is beyond the scope of this paper, we refer to [28] and our website [29] and give two exemplary screenshots in Fig. 8 of the scenario described above.

6. Conclusions

Compared to the basic implementation, only about 20% of the performance is reached in case of the free-surface simulation, due to the required multipass method implying multiple executions of different kernels for each timestep, the increased amount of branching instructions (more cell types), the increased code size (reconstruction of PDFs, multipass method, etc.) as well as the increased demand of bandwidth (fluid fraction, fluid mass, velocity and density values).

The A-A pattern is recommended for large simulation domains with a big memory demand whereas the A-B pattern should be used if the simulation should run at top speed. Utilizing the local memory for free-surface flows in our test case led to a performance decrease and should therefore be avoided.

7. Outlook

The challenge of surface tracking and the suppression of race conditions in LBM based free-surface simulations were addressed in this paper. Both aspects were overcome by a multipass method for the reflagging of the respective

interface cells. Our performance studies point out the capability of our software in real-time simulation and visualization scenarios. Our code has been published online [29] in order to allow further performance evaluations on different platforms and thus give a better comparison between them.

For free-surface flows, the handling of the interface cells is one of the most challenging parts on GPUs. This part could be further improved by enhanced computational techniques which have been left for future work. Amongst others, for simulation domains with a small amount of filled cells an adaptive handling of fluid and interface cells would be desirable. A stack-based approach to selectively run work items only for interface or fluid cells could be used to handle these cell types adaptively.

With the computational power of GPUs further increasing and more memory available, domains with increased resolutions can soon be simulated in real-time. Then, a coloring scheme could become an efficient alternative to avoid race conditions between concurrently running local work groups.

References

[1] A. Tasora, D. Negrut, M. Anitescu, A GPU-Based Implementation of a Cone Convex Complementarity Approach for Simulating Rigid Body Dynamics With Frictional Contact, ASME Conference Proceedings 2008.

[2] D. Göddeke, S. H. Buijssen, H. Wobker, S. Turek, GPU Acceleration of an Unmodified Parallel Finite Element Navier-Stokes Solver, in: W. W. Smari, J. P. McIntire (Eds.), High Performance Computing & Simulation 2009, 2009, pp. 12–21.

[3] J. Mora, W.-S. Lee, Real-time 3d fluid interaction with a haptic user interface, in: Proceedings of the 2008 IEEE Symposium on 3D User Interfaces, 3DUI '08, IEEE Computer Society, Washington, DC, USA, 2008, pp. 75–81.

[4] H. Nguyen, Gpu gems 3, 1st Edition, Addison-Wesley Professional, 2007.

[5] S. Succi, The Lattice Boltzmann Equation for Fluid Dynamics and Beyond, Oxford University Press, New York, 2001.

[6] Z. Fan, F. Qiu, A. Kaufman, S. Yoakum-Stover, GPU Cluster for High Performance Computing, in: Proceedings of the 2004 ACM/IEEE conference on Supercomputing, SC '04, IEEE Computer Society, Washington, DC, USA, 2004.

[7] J. Tölke, M. Krafczyk, TeraFLOP computing on a desktop PC with GPUs for 3D CFD, Int. J. Comput. Fluid Dyn. 22 (2008) 443–456.

[8] C. Körner, M. Thies, T. Hofmann, N. Thürey, U. Rüde, Lattice Boltzmann Model for Free Surface Flow for Modeling Foaming, J. Stat. Phys. 121 (2005) 179–196.

[9] S. Chapman, T. Cowling, The mathematical theory of nonuniform gases, Cambridge University Press, London, 1960.

[10] R. Mei, W. Shyy, D. Yu, L.-S. Luo, Lattice Boltzmann method for 3-D flows with curved boundary, J. Comput. Phys. 161 (2000) 680–699.

[11] P. Bhatnagar, E. Gross, M. Krook, A model for collision processes in gases. I. Small amplitude processes in charged and neutral one-component systems, Phys. Rev. 94 (3) (1954) 511–525.

[12] N. Thuerey, Physically based Animation of Free Surface Flows with the Lattice Boltzmann Method, Master's thesis, Universität Erlangen-Nürnberg (Mar 2007).

[13] A. K. Gunstensen, D. H. Rothman, S. Zaleski, G. Zanetti, Lattice Boltzmann model of immiscible fluids, Phys. Rev. A 43 (8) (1991) 4320–4327.

[14] I. Ginzburg, K. Steiner, A Free-Surface Lattice Boltzmann Method for Modelling the Filling of Expanding Cavities by Bingham Fluids, Philosophical Transactions: Mathematical, Physical and Engineering Sciences 360 (2002) 453–466.

[15] Blender website, `http://www.blender.org/`.

[16] NVIDIA, NVIDIA OpenCL Best Practices Guide (2009).

[17] NVIDIA, OpenCL Programming Guide for the CUDA Architecture 2.3 (2010).

[18] P. Bailey, J. Myre, S. D. C. Walsh, D. J. Lilja, M. O. Saar, Accelerating Lattice Boltzmann Fluid Flow Simulations Using Graphics Processors, in: ICPP, 2009, pp. 550–557.

[19] J. Tölke, Implementation of a Lattice Boltzmann kernel using the Compute Unified Device Architecture developed by NVIDIA, Comput. Vis. Sci. 13 (2009) 29–39.

[20] E. Riegel, T. Indinger, N. A. Adams, Implementation of a Lattice-Boltzmann method for numerical fluid mechanics using the NVIDIA CUDA technology, Inform., Forsch. Entwickl. 23 (3-4).

[21] W. Li, Z. Fan, X. Wei, A. Kaufman, GPU-Based flow simulation with complex boundaries, Tech. rep. (2003).

[22] M. Bernaschi, L. Rossi, R. Benzi, M. Sbragaglia, S. Succi, Graphics processing unit implementation of lattice boltzmann models for flowing soft systems, Phys. Rev. E 80 (6) (2009) 066707. doi:10.1103/PhysRevE.80.066707.

[23] NVIDIA, NVIDIA CUDA Programming Guide 3.0, 2010.

[24] NVIDIA, Ptx: Parallel thread execution, `http://www.nvidia.com/content/CUDA-ptx_isa_1.4.pdf` (2009).

[25] G. Wellein, T. Zeiser, G. Hager, S. Donath, On the single processor performance of simple lattice Boltzmann kernels, Computers & Fluids 35 (8-9) (2006) 910 – 919, proceedings of the First International Conference for Mesoscopic Methods in Engineering and Science.

[26] Nv opencl compiler options,
`http://developer.download.nvidia.com/compute/cuda/3_0/toolkit/docs/opencl_extensions/cl_nv_compiler_options.t`

[27] J. Kruger, R. Westermann, Acceleration Techniques for GPU-based Volume Rendering, in: VIS '03: Proceedings of the 14th IEEE Visualization 2003 (VIS'03), IEEE Computer Society, Washington, DC, USA, 2003.

[28] M. Schreiber, GPU based Simulation and Visualization of Fluids with Free Surfaces, Diplomarbeit, Institut für Informatik, Technische Universität München (Jun. 2010).
URL `http://www5.in.tum.de/pub/schreibm_da.pdf`

[29] M. Schreiber, Source Code: GPU based Simulation and Visualization of Fluids with Free Surfaces,
`http://www.martin-schreiber.info/diplomathesis.html`.