Bachelorarbeit
zur Erlangung des akademischen Grades
Bachelor of Science (B. Sc.)

MPI-based multi-GPU extension of the Lattice Boltzmann Method

vorgelegt von

Fabian Häusl
Geboren am: 16. Mai 1995
Matrikelnummer: 1540550

Abgabedatum: 29.10.2019

Prüfer: Prof. Dr. Stephan Gekle

Biofluid Simulation and Modeling
Fakultät für Mathematik, Physik und Informatik
Universität Bayreuth

UNIVERSITÄT
BAYREUTH

Zusammenfassung

In dieser Bachelorarbeit wird eine Methode vorgestellt, um die Lattice Boltzmann Methode durch Kopplung mit dem Message Passing Interface (MPI) auf mehreren Grafikprozessoren zu parallelisieren. Diese Aufgabe stellt sich v.a. im Hinblick auf den begrenzten Speicherplatz eines einzelnen Grafikprozessors und der Speicherintensität der Lattice Boltzmann Methode. Es wird ein konkreter Algorithmus für die Simulationssoftware FluidX3D gezeigt und validiert. Dieser weist die Flexibilität auf, mit verschiedenen Erweiterungen der Lattice Boltzmann Methode zu funktionieren: neben komplexen geometrischen Randbedingungen, Wärmeflüssen und Kondensationsprozessen ist auch die Simulation von freien Oberflächen möglich. Eine besondere Herausforderung ist dabei, die funktionsorientierte MPI-Kommunikation mit dem objektorientierten Ansatz von FluidX3D sinnvoll zu vereinen.

Im Verlauf der Arbeit werden verschiedene Optimierungen der multi-GPU Erweiterung erläutert: einerseits greifen diese auf Wissen über die Programmiersprache OpenCL und die Hardware von GPUs zurück, andererseits wird der Algorithmus selbst dahingehend erweitert, dass ein Überlapp von Berechnung und Speichertransfer stattfinden kann. Die Optimierungen werden dabei durch Laufzeitmessungen auf zwei verschiedenen Clustern mit bis zu 4 GPUs gleichzeitig bestätigt. Der multi-GPU Algrithmus erreicht fast unabhängig von der Anzahl verwendeter GPUs 95% seines theoretischen Optimums im weak-scaling, im strong-scaling ergibt sich bei 4 GPUs einer Effizienz von 77%. Es wurden bis zu 13600 MLUPs mittels 4 Radeon VII GPUs für ein würfelförmiges benchmark setup erreicht.

Abstract

This bachelor thesis presents a method to parallelise the Lattice Boltzmann method on several graphics processing units by coupling it with the Message Passing Interface (MPI). This task is mainly related to the limited on-board memory of a single graphics processing unit and the memory intensity of the Lattice Boltzmann method. A concrete algorithm for the simulation software FluidX3D is shown and validated. This has the flexibility to work with different extensions of the Lattice Boltzmann method: besides complex geometric boundary conditions, heat flux and condensation processes, the simulation of free surfaces is also possible. A special challenge is to combine the function-oriented MPI communication with the object-oriented approach of FluidX3D.

This thesis wil explain various optimizations of the multi-GPU extension: on the one hand, they rely on knowledge about the programming language OpenCL and the hardware of GPUs, on the other hand, the algorithm itself is extended in such a way that an overlap of calculation and memory transfer can take place. The optimizations will be confirmed by runtime measurements on two different clusters with up to 4 GPUs at the same time. The multi-GPU algrithm reaches 95% of its theoretical optimum in weak-scaling almost independent of the number of GPUs used. In strong-scaling the efficiency of 4 GPUs is 77%. Up to 13600 MLUPs when using 4 Radeon VII GPUs were achieved for a cubic benchmark setup.

Contents

1 Introduction

The Lattice Bolzmann Method (LBM) is a quite novel approach in computational fluid dynamics (CFD), that is gaining increasing popularity [1]. This is especially due to its properties: as opposed to classical Navier-Stokes Solvers, the LBM can easily handle complex boundary conditions and is compatible with a large amount of application-oriented extensions (e.g. heat diffusion, multiphase and multicomponent flows, immersed boundary methods for simulation of deformable boundaries [2]). Furthermore, its algorithm is highly parallelizable, so for large and runtime intensive setups speedup can easily be reached by computing on multiple CPU (central processing unit) cores or by using GPUs (graphics processing units). Because GPUs were originally developed to process geometric data in parallel and quickly, their architecture is superior to CPUs in computing power and memory bandwidth. However, since accuracy does not play such a major role in graphics processing, in contrast to CPUs they are only optimized for 32-bit floating point values. Fortunately, *float* accuracy is usually sufficient for the LBM. Another difference to CPUs is the amount of available memory: while the RAM available to a CPU can be several hundred gigabytes, the memory space on GPUs is very limited. The GPUs used for this thesis had 16 GB (Radeon VII) or 12 GB (Nvidia Tesla P100) available. Because the LBM is memory-intensive while at the same time the memory on GPUs is quite limited, this directly restricts the size of the simulation box. Multi-GPU implementations of the LBM are able to widen this limitation as well as to gain even bigger speedup.

Many multi-GPU implementations of the LBM have been realized, trying to reduce its communication overhead to a negligible size. In the weak scaling, the efficiency reaches from 69.9 % using 32 GPUs with a total performance of about 2000 MLUPs (see [3]) to 98.5 % using 64 GPUs with a total performance of about 35000 MLUPs (see [4]). However, the latter implementation - as well as [1], [5], [6], [7], [8] - is limited to a subdivision of the domain along a preferred axis. [4] does not provide a strong scalability test for cubic domains, while [1] reaches an efficiency of 91 % with 8 GPUs and a total performance of about 3000 MLUPs here. [3] is the only one of the references mentioned above that can subdivide the domain along all three spatial directions. All of them use halo nodes to organize the data to be transferred between the GPUs. While [4] runs with CUDA as well as OpenCL via the wrappers PyCUDA and PyOpenCL, all other references only work with CUDA. In order to manage the different computational devices, POSIX, MPI or ZeroMQ is used in the different implementations.

For Bayreuths new LBM software - FluidX3D - multi-GPU support is provided and evaluated together with this thesis. That therefore larger systems can be simulated is particularly important for a project in the context of the Collaborative Research Centre 1357 Microplastics (german: Sonderforschungsbereich Mikroplastik). In this project, the exchange of microplastic particles at the air-water interface is studied. Because of the buoyancy many microplastic particles might swim directly under the water surface. Processes such as the impact of a raindrop on the surface of the water cause small droplets to be thrown into the air. These could serve as carriers for the micropastic particles and thus explain a potential migration path from the hydro- to the atmosphere. In order to understand the properties of this complicated fluid mechanical mechanism, the LBM is right now extended by the immersed boundary method and a module for the simulation of free surfaces.

2 Theoretical and methodical Background

2.1 Kinetic theory - basis of the LBM

The LBM can be derived from kinetic theory, which is valid on a scale that lies between the microscopic scale (looking at the motion of individual molecules) and the macroscopic scale (considering only macroscopic quantities as e.g. density, velocity, temperature) and therefore is called *mesoscopic* scale.

The simplest case in kinetic theory is that of a dilute monoatomic gas. It is a justified assumption here that molecules only collide in pairs and have a long mean free path. This is not true for an actual fluid with far bigger density, and additional considerations have to be done: the intermolecular attracting forces, the inner degrees of freedom of multiatomic molecules (rotational and vibrational). Nevertheless the theoretical foundation of the LBM equations can be understood when we look at the kinetic theory of a dilute monoatomic gas. Here the fundamental variable is the particle distribution function $f(\vec{x}, \vec{\xi}, t)$, which lives in the six-dimensional phase space and denotes the density of particles with velocity $\vec{\xi}$ at position $\vec{x}$ at time t.

Macroscopic variables can be found as **moments** of f. The moments are weighted integrals of f over its entire velocity space, where the weights are functions of $\vec{\xi}$. The most important ones are:

$$\rho(\vec{x}, t) = \int f(\vec{x}, \vec{\xi}, t) d^3\xi \text{ (**mass density**)}, \tag{1}$$

$$\rho(\vec{x}, t)\vec{u}(\vec{x}, t) = \int \vec{\xi} f(\vec{x}, \vec{\xi}, t) d^3\xi \text{ (**momentum density**)}. \tag{2}$$

Further moments, like the **internal energy density** $e(\vec{x}, t)$, have a similar structure. Due to the collisions of the particles, their velocity distribution soon converges into an equilibrium distribution f^{eq}, if there are no external forces present. This equilibrium distribution can be found as the *Maxwell-Boltzmann distribution*

$$f^{eq}(\vec{x}, \vec{\xi}, t) = f^{eq}(\vec{x}, \vec{v}, t) = \rho \left(\frac{3}{4\pi e}\right)^{3/2} e^{-2|\vec{v}|/(4e)}, \tag{3}$$

with $\vec{v} = \vec{\xi} - \vec{u}$.

For the evolution of f in time, one has to consider (note the Einstein notation):

$$\frac{\mathrm{d}f}{\mathrm{d}t} \equiv \Omega(f) = \left(\frac{\partial f}{\partial t}\right)\frac{\mathrm{d}t}{\mathrm{d}t} + \left(\frac{\partial f}{\partial x_\alpha}\right)\frac{\mathrm{d}x_\alpha}{\mathrm{d}t} + \left(\frac{\partial f}{\partial \xi_\alpha}\right)\frac{\mathrm{d}\xi_\alpha}{\mathrm{d}t}. \tag{4}$$

Carrying out the total differentials on the right-hand side we get $\mathrm{d}t/\mathrm{d}t = 1$, $\mathrm{d}x_\alpha/\mathrm{d}t = \xi_\alpha$ and $\mathrm{d}\xi_\alpha/\mathrm{d}t = F_\alpha/\rho$. $\vec{F}$ denotes the specific body force. This leads to the *Boltzmann equation*

$$\Omega(f) = \frac{\partial f}{\partial t} + \xi_\alpha \frac{\partial f}{\partial x_\alpha} + \frac{F_\alpha}{\rho}\frac{\partial f}{\partial \xi_\alpha}. \tag{5}$$

The terms on the right hand-side describe the advection of f and how forces affect f. The source term $\Omega(f)$ describes the redistribution of f due to particle collision and is therefore called *collision operator*. Boltzmann's original collision operator has the form

$$\Omega(f) = \int W(\vec{\xi_1}, \vec{\xi_2}, \vec{\xi_3}, \vec{\xi})(f(\vec{x}, \vec{\xi_1}, t)f(\vec{x}, \vec{\xi_2}, t) - f(\vec{x}, \vec{\xi_3}, t)f(\vec{x}, \vec{\xi}, t))d\vec{\xi_1}d\vec{\xi_2}d\vec{\xi_3}. \tag{6}$$

$W(\vec{\xi_1}, \vec{\xi_2}, \vec{\xi_3}, \vec{\xi})$ describes the probability per time unit that two particles have the velocities $\vec{\xi_1}, \vec{\xi_2}$ before and $\vec{\xi_3}, \vec{\xi}$ after their collision and is dependent of the underlying model.

By integrating the Boltzmann equation and its moments over velocity space it can be shown that for media with short mean free path, the Boltzmann equation becomes the macroscopic Navier-Stokes equation (NSE). While the NSE is much simpler from a analytical point of view, the Boltzmann equation is superior when being solved numerically for complex boundaries on mesoscopic scales, especially when considering its comparatively easy implementation and parallelization. The numerical approach to the Boltzmann equation for CFD purposes will be seen now.

2.2 The main equations of the LBM

The LBM works with *discrete-velocity distribution functions* $f_i(\vec{x}, t)$, also called *populations*. Analog to the kinetic theory it describes the density of particles with velocity $\vec{c_i}$ at position $\vec{x}$ at time t, but now $\{\vec{c_i}, w_i\}$ are a small discrete set of velocities and its corresponding weighting coefficients. The first moments of f_i are the discrete version of equations 1 and 2:

$$\rho(\vec{x}, t) = \sum_i f_i(\vec{x}, t), \tag{7}$$

$$\rho(\vec{x}, t)\vec{u}(\vec{x}, t) = \sum_i \vec{c_i} f_i(\vec{x}, t). \tag{8}$$

In addition, f_i is only defined on a discrete square lattice in space with spacing Δx and on discrete times with intervals Δt. A common choice is $\Delta x = \Delta t = 1$ (*lattice units*). This also affects the Boltzmann equation 5, which in the force-free case reduces to the *lattice Boltzmann equation*

$$f_i(\vec{x} + \vec{c_i}\Delta t, t + \Delta t) = f_i(\vec{x}, t) + \Omega_i(\vec{x}, t). \tag{9}$$

For the purposes of the LBM, much simpler collision operators than in equation 6 can be used, e.g. the *BGK collision operator*[1]:

$$\Omega_i = -\frac{f_i - f_i^{eq}}{\tau}\Delta t. \tag{10}$$

The constant τ determines the speed of relaxation and is called *relaxation time*. The collision operator has to conserve mass, momentum and internal energy. In FluidX3D, available collision operators are - in ascending order of accuracy - BGK, the *two-relaxation-time* (TRT) and *multi-relaxation-time* (MRT) collistion operator.

[1] names after its inventors Bhatnagar, Gross and Krook

In the LBM, the equilibrium distribution f^{eq} in equation 3 is replaced by the discretised version of its Hermite Series expansion:

$$f_i^{eq}(\vec{x},t) = w_i\rho\left(\frac{\vec{u}\circ\vec{c_i}}{c_s^2} + \frac{(\vec{u}\circ\vec{c_i})^2}{2c_s^4} + 1 - \frac{\vec{u}\circ\vec{u}}{2c_s^2}\right). \tag{11}$$

Here $c_s^2 = (1/3)\Delta x^2/\Delta t^2$ is the *speed of sound*, which also describes the relation $p = c_s^2\rho$ between pressure p and density ρ. During a simulation, for stability reasons all occurring values of $|\vec{u}|$ should stay below c_s. f_i^{eq} is chosen such that its moments are the same as those of f_i.

The connections between the LBM and the NSE is determined by the Chapman-Enskog analysis. It yields that the results of the LBM follow the solutions of the NSE with

$$\nu = c_s^2\left(\tau - \frac{\Delta t}{2}\right) \text{ (kinematik shear viscosity) and} \tag{12}$$

$$\nu_B = \frac{2}{3}\nu \text{ (kinematic bulk viscosity).} \tag{13}$$

Realize that ν is still given in lattice units. So how is the link between the LBM's lattice units and the real world physical units? Which parameters are to choose, if e.g. for a given time t_p you want to simulate a fluid of given viscosity ν_p in a canal of given diameter l_p represented by a given number of lattice nodes N, while a given constant force F_p effects the fluid? For the given mechanical system, three independent conversion factors have to be found, that connect between lattice and physical units. Normally one first determines $C_x = \Delta x_p/\Delta x$, $C_t = \Delta t_p/\Delta t$ and $C_\rho = \rho_{0,p}/\rho_0$. They are given by:

$$C_x = \frac{l_p}{N\Delta x}, \tag{14}$$

$$C_\rho = \frac{\rho_0}{\rho_{0,p}}, \tag{15}$$

$$C_t = \frac{C_x^2}{C_\nu} = c_s^2(\tau - 0.5\Delta t)\frac{C_x^2}{\nu_p}. \tag{16}$$

It is common to set $\Delta x = \Delta t = \rho_0 = 1$ and $\tau > 0.5$ should be used for stability reasons. The first expression for C_t in equation 16 can be found by considering the unit of its associated physical value: $[t] = \text{s} = \text{m}^2/(\text{Pa s}) = [\Delta x^2/\nu]$. All other relevant conversion factors can be constructed as products of these three conversion factors by analogous unit considerations. In the further course of this thesis all occurring simulation parameters are given in lattice units.

It is worth mentioning that the *law of similarity* plays an important role in fluid dynamics: "two incompressible flow systems are dynamically similar if they have the same Reynolds number and geometry" ([2], p. 268). With l and u being typical length and velocity scales in the system, the Reynolds number is defined as

$$\text{Re} = \frac{lu}{\nu} = \frac{\rho lu}{\eta}, \tag{17}$$

where ν is the kinematic and η the dynamic viscosity of the fluid.

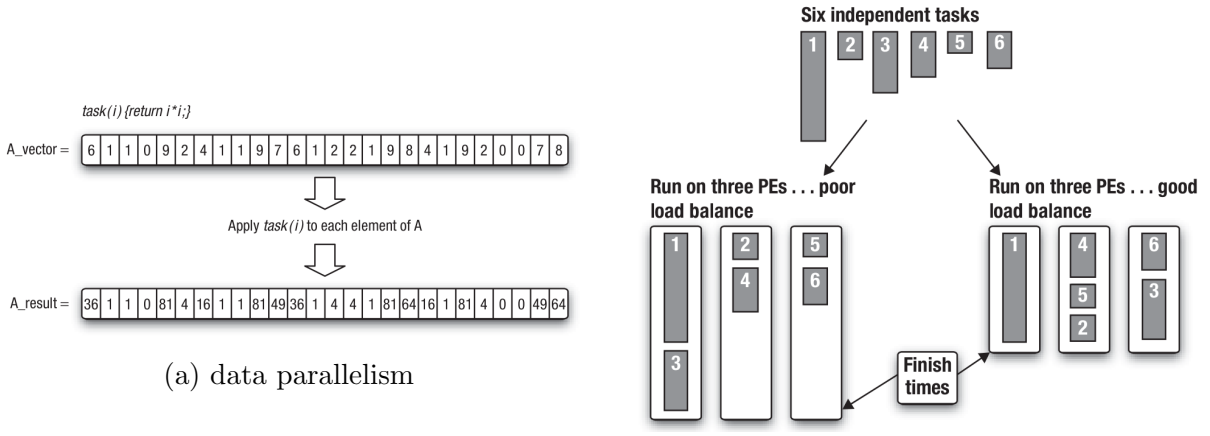
2.3 Some hardware and software details concerning GPUs, OpenCL and MPI

Before we can take a closer look at the implementation of the LBM in FluidX3D, we have to present some details about graphics processing units. This is important to understand, why FluidX3D is implemented the way it is. Runtime tests in this and the following sections were conducted on two different clusters: On the one hand on the chair's own cluster called *SMAUG*, whereupon as GPUs up to 4 Radeon VII from AMD could be used simultaneously. On the other hand on a cluster of the University of Bayreuth named *btrzx4*, which is equipped with 5 nVidia Tesla P100 GPUs.

2.3.1 OpenCL and GPUs

In FluidX3D the GPU side code and the communication between host and GPU is implemented in OpenCL. OpenCL is an industrial standard framework for programmming heterogeneous platforms consisting of a combination of CPUs and GPUs. Because working with GPUs is so different to CPUs, OpenCL even created a whole new programming model around their framework [9]. In FluidX3D the version OpenCL 1.2 is used, which is the latest compatible with the vendors AMD as well as nVidia.

There are in general two ways an algorithm can be parallelized: by taking advantage of its *task parallelism* or its *data parallelism*. In the former case, the programmer searches for tasks that can be executed independently of each other at the same time. Optimal execution time is reached, when the load of the tasks is balanced between the processing elements (PE). With the latter, on the other hand, the same calculation rules can be applied to large amounts of data simultaneously, so the parallelism lies in the data itself. This is called *single-instruction multiple-data* (SIMD) execution scheme and is the case for the LBM-algorithm itself. The two parallelism paradigms are shown in 1. Both find application when it comes to multi-GPU LBM.



(a) data parallelism

(b) task parallelism

Figure 1: Visualisation of data vs. task parallelism ([9], p. 10)

An OpenCL platform consists of a single host and one or several OpenCL devices. The OpenCL device is further divided into compute units made up of processing elements (PEs). The PEs have their hardware equivalent in the cores of the GPU or CPU.

FluidX3D consists of a host program and a collection of kernels. The host program is coded in C++ (extended by the OpenCL C++ API) and runs on the host, while the OpenCL programs (kernels) are coded in the OpenCL C programming language (basically a reduced version of C99 extended by OpenCLs memory model) and run on the OpenCL devices. In order to meet the heterogenous requirements of the platform the kernels are built from source at runtime. The host program interacts with the OpenCL devices via a command-queue. There are three types of commands:

- Kernel execution commands
- Memory commands (data transfer between host and device)
- Synchronization commands

There are two ways how commands in a queue are executed:

- In-order execution: Commands are launched in the order the host defines. Before a new command begins execution, the prior must have finished.
- Out-of-order execution: Commands are launched in the order the host defines, but don't have to complete before the next command is executed.

In the latter case, the programmer has to explicitly synchronize the queue if needed. FluidX3D uses an in-order queue in the single- as well as multi-GPU implementation. Synchronization points may also be needed to enforce the host program to wait for a queue command to finish. Otherwise the host program runs in parallel to the execution of the command-queue.

When kernel execution begins on a OpenCL device, the OpenCL runtime system creates an integer index space $I \subset \mathbb{N}^D$. The kernel enqueuement command specifies the dimensionality $D \in \{1, 2, 3\}$, the global ranges r_g^i and local ranges r_l^i $(i = 0, ..., D-1)$ of the index space. In FluidX3D only $D = 1$ is used[2]. We will therefore further refer to r_g^0 as r_g (r_l analogously). Each instance of an execution kernel (work-item) is assigned a global ID $n \in [0, r_g - 1]$ and a group ID $w \in [0, r_l - 1]$, which expresses the organisation of work-items in work-groups. A kernel is always executed by blocks of threads simultaneously, with r_l being the *thread-block size* (TBS). It must apply

$$r_g \bmod r_l = 0 \ , \tag{18}$$

so all work-groups are full. Each work-item has a local ID l within its work-group. It applies: $n = r_l w + l$, so each work-item can be identified by its global ID or by knowing the group ID and local ID. If a command only applies to a subset of the group (e.g. an if-statement), all unaffected work-items have to wait for the command to be finished. This is called *branching* and should be avoided for performance reasons. During a kernel execution, work-groups are launched asynchronously - synchronization can only be ensured by performing multiple kernel launches.

OpenCL defines five different memory regions:

[2]The three-dimenional position in simulation space is computed starting from this one-dimensional index, as will be seen later.

- Host memory: This region is visible only to the host.
- Global memory: All work-items in all work-groups have read/write access to it. On GPUs this is the video memory, where access latency is quite high[3]. Although in FluidX3D latency is completely hidden by computations, the program is its bandwidth limit due to the low arithmetic intensity of the LBM. Memory bandwidth is the typical bottleneck of LBM implementations [10].
- Constant memory: This denotes a global memory region with read-only access.
- Local memory: This region is shared only by the members of the work-group. Compared to global memory, access is a lot faster for GPUs, because local memory is represented by L1 and L2 cache on hardware side. It is not used in FluidX3D right now.
- Private memory: Every work-item has its own private memory. After caching data from global memory, computation should be done here, because the access speed is several hundred times faster (also faster than local memory access). On GPUs, private memory has its hardware equivalent in registers.

In figure 2 can be seen how they interact with the platform model of OpenCL. In order to transfer data to the host, it must first be written to the global memory. This is also the only way to preserve data between two kernel calls.

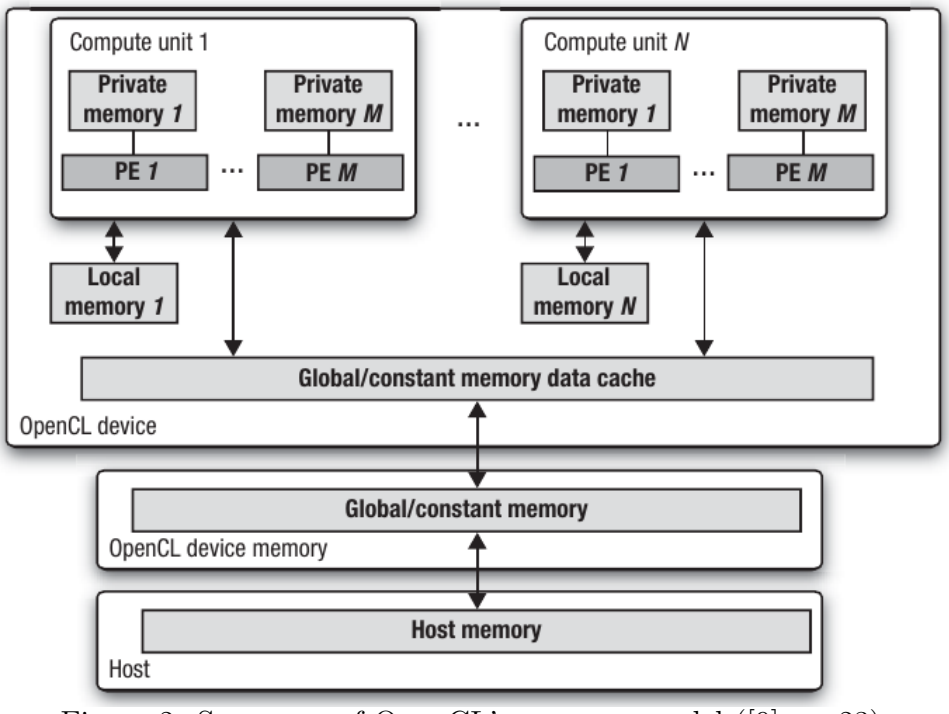


Figure 2: Summary of OpenCL's memory model ([9], p. 23).

From a memory point of view, data exchange between host and OpenCL device runs via buffer objects, which are just some contiguous block of memory made available to the kernel. Every block of global or constant memory used in a kernel corresponds to a buffer object at

[3]One memory transfer needs about 400-800 clock cycles, while computational operations like add or multiply only take 4 clock cycles.

host side.

When using GPUs, a buffer is transferred from Device to Host via the *direct memory access* (DMA) protocol, which only works on *pinned memory*[4]. Since pinned memory is a scarce resource on GPUs, normal buffer transfers need two copies: from the actual array of global memory to the pinned memory region and then from the pinned memory region to the Host (for transfers from Host to Device analogously). The pinning-process can only happen in whole pages, which typically contain 4096 bits. However, in OpenCL, the flag CL_MEM_USE_HOST_PTR can be set when creating the buffers. In the future we will then say, that *host pointer* is activated. As a result, the buffer object uses a fixed memory region as storage bits on Host side (see [11]). On Device side, it is likely (but not guaranteed) that the buffer corresponds to a region of pinned memory. This avoids an additional internal copy (see [12]). The impact on the buffer access times is quite big, as can be seen in the figure 3: without host pointer the memory bandwidth of the device itself is quite high, while the memory transfer to the host via *PCI Express* (PCIe) has much lower bandwidths. If host pointer is enabled, device side access to the buffer via PCIe is required, while the buffer for the host can be provided very quickly, since its bandwidth is mainly limited by the latency of the transfer command. If a buffer is written (read) only once during a kernel call, and is afterwards (previously) made available for the host, using a buffer with host pointer enabled is about twice as fast as using no host pointer.

OpenCL provides two routines to organize the buffer exchange between host and device. Buffers (or related parts thereof) can be accessed from the host after calling the *enqueueMapBuffer* function (read or write access depending on the arguments). While the buffer is mapped, it may not be used on the device side (e.g. by a kernel). This would lead to undefined behavior. Only a call of *enqueueUnmapMemObject* with corresponding arguments terminates the access possibilities by the host and makes them available for the devices. We call this routine *mapping*. The second possibility is to call *enqueueReadBuffer* and *enqueueWriteBuffer* (*read-write* routine). Here buffers are read into certain host-memory areas or written from host-memory areas into buffers. No undefined behavior can occur here.

When host pointer is active, both options make one internal copy from the host to the device or vice versa, and should therefore have similar runtimes. This is also the case in 4. How exactly they are implemented, however, is vendor specific, which is why in the multi-GPU version of FluidX3D both possibilities are selectable. When host pointer is not active, the mapping routine would be superior to the read-write routine when small disjointed parts of the buffer should be accessed on Host side. This is, because in the mapping routine the GPU memory is only pinned once, while in the read-write routine every call of *enqueueReadBuffer* and *enqueueWriteBuffer* would lead to new pinning of a page with the required part of the buffer within it.

2.3.2 MPI

The Message Passing Interface (MPI) is an industrial standard, that provides routines for parallel computing architectures to send messages across it. In concrete terms, this means that

[4] *Page-locked memory* is an equivalent term.

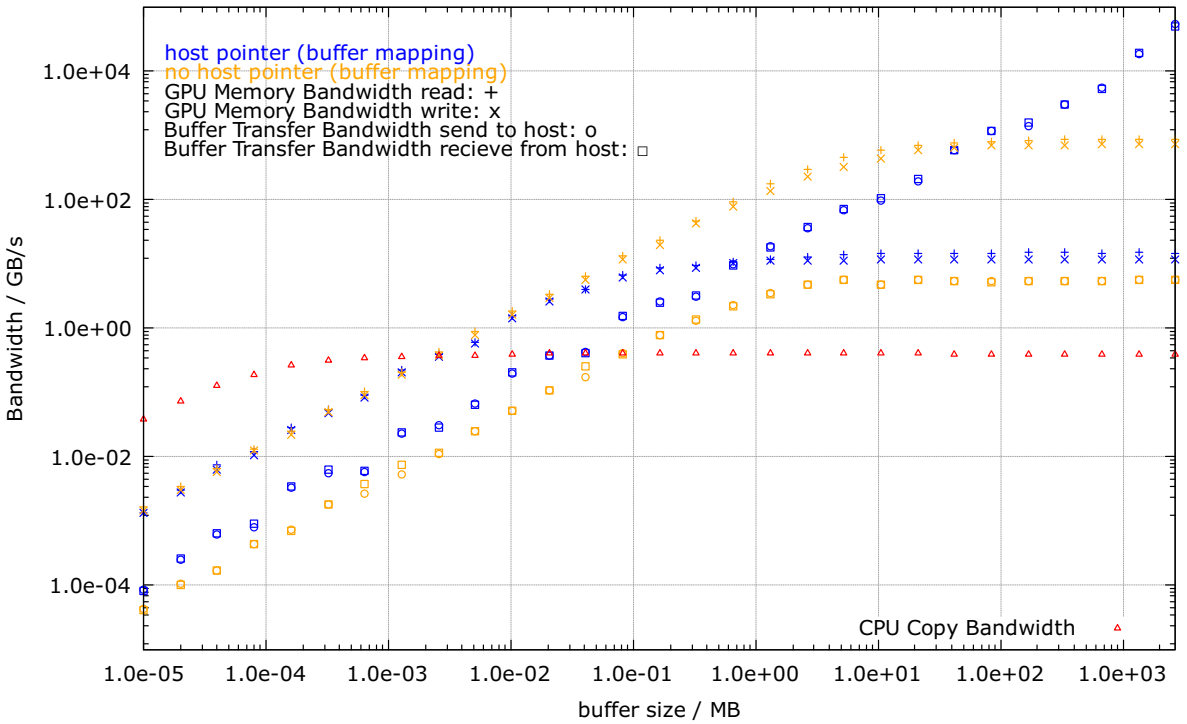


Figure 3: Runtime measurements concerning host pointer, conducted on the Cluster *SMAUG*. For small buffers, bandwidth is limited by latency. For large buffers and a process consisting of write (send) and read (recieve), the bottleneck is normally the Buffer Transfer Bandwidth, while for host pointer activated it is the GPU Memory Bandwith. Additionally, the CPU Copy Bandwidth of a normal array is depicted.

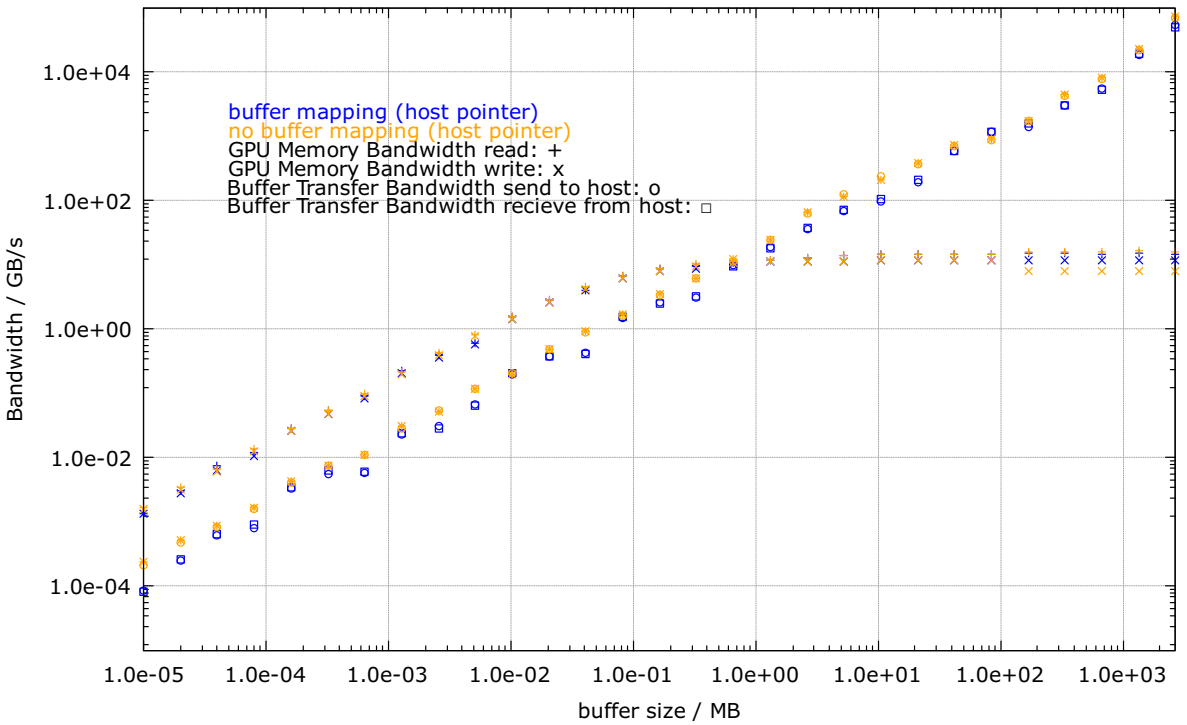
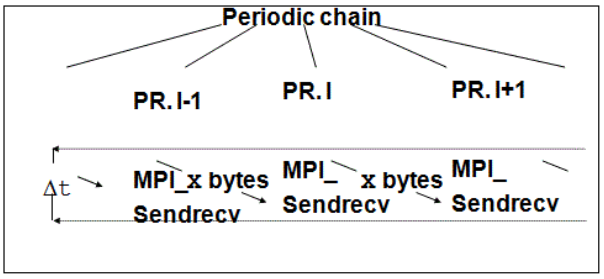


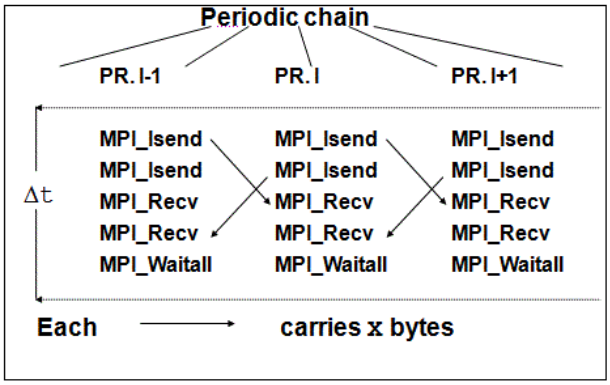
Figure 4: Runtime measurements concerning buffer mapping, conducted on the Cluster *SMAUG*. Host pointer is activated for these measurements. It can be seen that there are no significant performance differences between the map routine and the read-write routine.

data packages can be sent between CPU cores using MPI. If these are not part of the same compute node, a network must be set up between the cores. On *btrzx4*, for example, all used GPUs are on different nodes and there is an *Intel Omni-Path network* to connect the nodes. On *SMAUG*, on the other hand, all GPUs are on the same node. MPI communication can be *blocking* (e.g. *MPI_Send()*, *MPI_Recv()*, *MPI_Sendrecv()*) or *non-blocking* (e.g. *MPI_Isend()*, *MPI_Irecv()*). Non-blocking functions return immediately, which allows CPU-communication and CPU-computation to overlap, but you must call additional functions (e.g. *MPI_Waitall()*) to see whether the communication has finished. Since the multi-GPU implementation will be a grid splitting algorithm, two MPI communication patterns are in question: the *Sendrecv* Pattern and the *Exchange* Pattern. In both cases the group of processes forms a periodic chain. Applying the *Sendrecv* Pattern two times in a row, where the second time the roles of the right and left neighbour in the chain are switched, is equivalent to the *Exchange Pattern* as far as data exchange is concerned (see figure 5). But on figure 6a you can see that they can differ as it concerns the bandwidth: on the cluster *btrzx4* the *Exchange* Pattern is preferable, while on the cluster *SMAUG* the *Sendrecv* Pattern yields slightly better performance. More importantly, the bandwidth depends on the size of the message sent, similar to the GPU buffer transfers above. In addition, there are latency times that are approximately constant across all message sizes in figure 6b. As can be seen here for the cluster *btrzx4*, the latency time for messages exchanges between nodes is higher than that between cores on the same node. The latency times mainly determine the performance for message sizes of 4 kB and below, so the bandwidth (at least for the *Exchange* Pattern) is nearly the same on both clusters. For message sizes between 32 kB and 4 MB, the shared node can explain the high bandwidth on *SMAUG*. It is not yet clear why the performance drops above 4 MB.

Figure 5: *Sendrecv Pattern* vs. *Exchange Pattern*: in both patterns the cores are ordered in periodic chains. The figures were taken from [13].



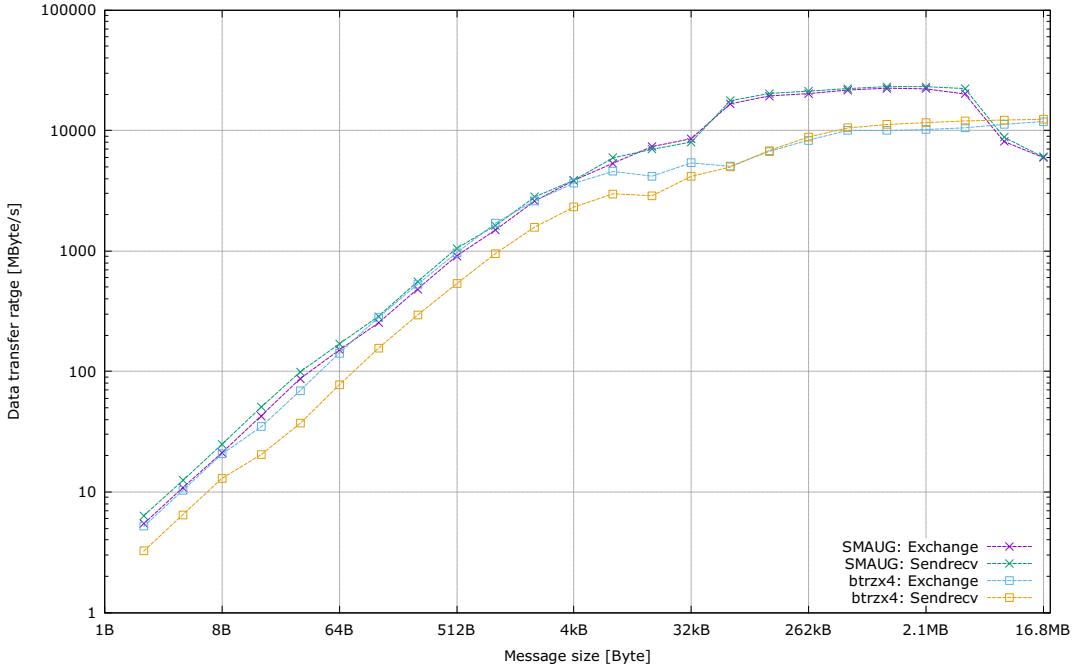
(a) *Sendrecv* Pattern: only blocking functions are called. When used two times in a row with switched neighbours, it has the same effect as the *Exchange* pattern.



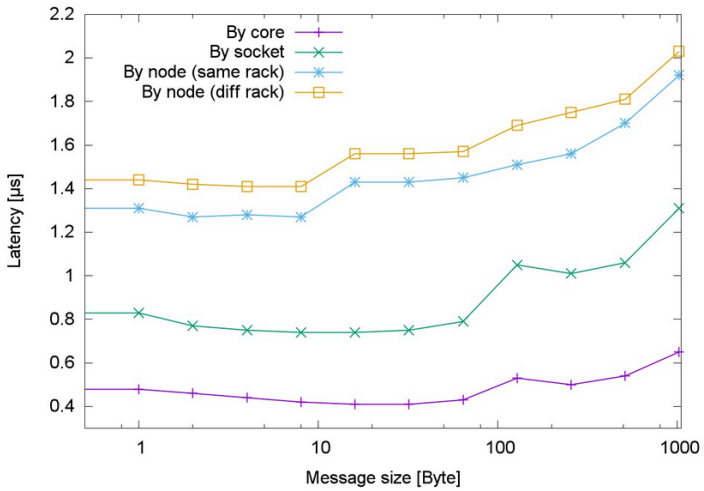
(b) *Exchange* Pattern: the communication is non-blocking, so synchronization must explicitly be ensured.

Overall, it is clear from all the benchmarks shown that unnecessary latency times can be avoided and bandwidth can be optimized by using one large buffer or message transfer operation instead of several small ones.

Figure 6: MPI bandwidth and latency depending on the message size.



(a) On MPI systems, over large ranges the data transfer rate rises exponentially with the message size. On *SMAUG*, all used cores are on the same node, while on *btrzx4* they are on different nodes. The measurements were carried out with the *Intel MPI Benchmarks* software [13]. The average over at least 125 measurements per data point is shown. All measurements for one data point differ from each other only in the sixth significant place.



(b) Point-2-point MPI latency, measured on cluster *btrzx4*. The latency time is nearly independent of the message size. The measurements were provided by [14]. Messages can be sent between processes on the same socket (by core), different sockets on the same node (by socket), and different nodes (by node).

2.4 Relevant aspects of the LBM and FluidX3D for the muli-GPU implementation

The velocity sets $\{\vec{c_i}, w_i\}$ mentioned above have to be further specified: a velocity set consisting of q different velocities that are elements of a d-dimensional vector space is denoted DdQq. FluidX3D has implemented the D2Q9, D3Q7, D3Q13, D3Q15, D3Q19 and D3Q27 velocity sets. They are depicted in figure 7 and their velocities and corresponding weighting coefficients are found in table 1.

Table 1: Properties of the velocity sets available in FluidX3D. The speed of sound is $c_s = 1/\sqrt{3}$ for all these sets. Conditions on a velocity set are conservation of mass and momentum, and "rotational isotropy" (see [2], p. 85).

velocity set	$\vec{c_i}$	number	$\lvert\vec{c_i}\rvert$	w_i
D2Q9	(0,0)	1	0	4/9
	(± 1,0),(0,± 1)	4	1	1/9
	(± 1, ± 1)	4	$\sqrt{2}$	1/36
D3Q7	(0,0,0)	1	0	1/4
	(± 1,0,0),(0,± 1,0),(0,0,± 1)	6	1	1/8
D3Q13	(0,0,0)	1	0	1/2
	(± 1,± 1,0),(± 1,0,± 1),(0,± 1,± 1)	12	$\sqrt{2}$	1/24
D3Q15	(0,0,0)	1	0	2/9
	(± 1,0,0),(0,± 1,0),(0,0,± 1)	6	1	1/9
	(± 1,± 1,± 1)	8	$\sqrt{3}$	1/72
D3Q19	(0,0,0)	1	0	1/3
	(± 1,0,0),(0,± 1,0),(0,0,± 1)	6	1	1/18
	(± 1,± 1,0),(± 1,0,± 1),(0,± 1,± 1)	12	$\sqrt{2}$	1/36
D3Q27	(0,0,0)	1	0	8/27
	(± 1,0,0),(0,± 1,0),(0,0,± 1)	6	1	2/27
	(± 1,± 1,0),(± 1,0,± 1),(0,± 1,± 1)	12	$\sqrt{2}$	1/54
	(± 1,± 1,± 1)	8	$\sqrt{3}$	1/216

We also have to give equation 9 a closer look. It says, that the collision operator Ω_i affects the populations $f_i(\vec{x}, t)$ and causes them to redistribute themselves among their neighbouring point $\vec{x} + \vec{c_i}\Delta t$ along the direction c_i. This all happens in the timeframe Δt. So the discretizations of time and space are not isolated to each other but related through the discretization of particle velocity. This allows to break the LBM down into two elementary steps: the populations move to their new lattice-position (streaming) and the populations collide (collision):

$$f_i(\vec{x} + \vec{c_i}\Delta t, t + \Delta t) = f_i^*(\vec{x}, t) \text{ (\textbf{streaming})}, \tag{19}$$

$$f_i^*(\vec{x}, t) = f_i(\vec{x}, t) + \Omega_i(\vec{x}, t) \text{ (\textbf{collision})}. \tag{20}$$

In order to take advantage of the massive parallelism of GPUs, each lattice node is assigned to one thread. Note that the streaming-step affects the neighbour populations and therefore will

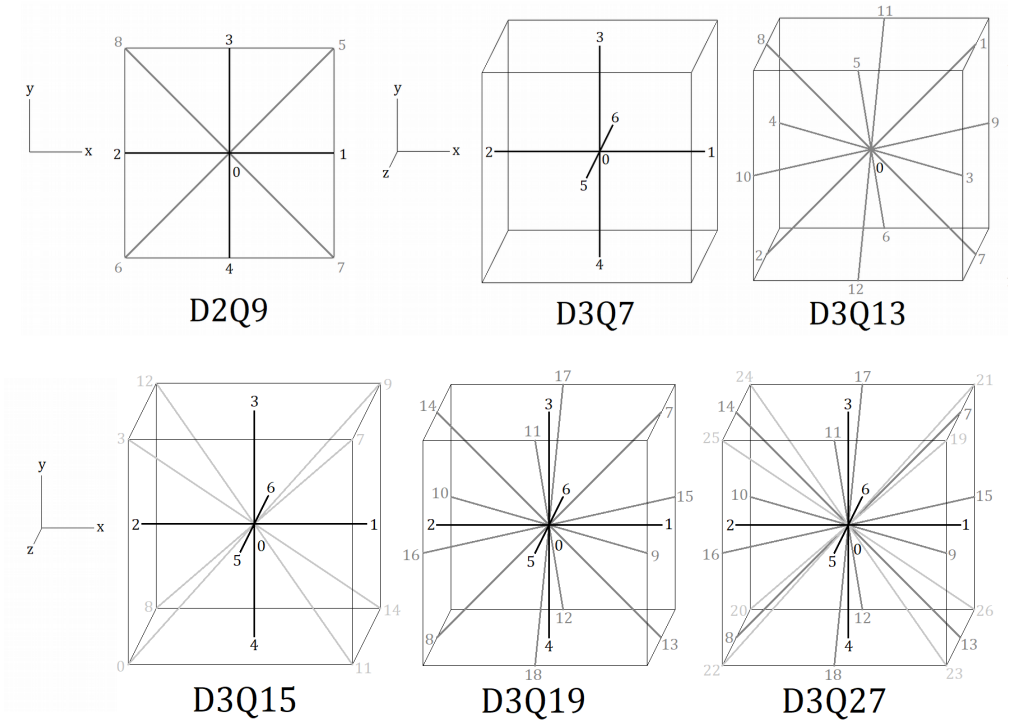
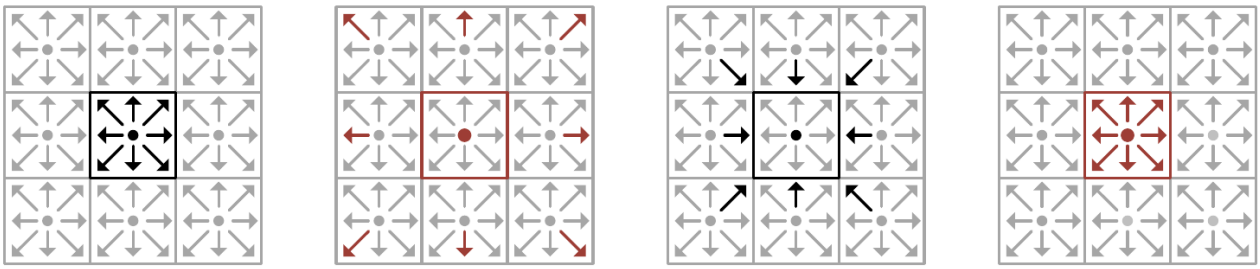


Figure 7: The six different velocity sets with indexing like in the FluidX3D-implementation. Opposite vectors $\vec{c}_i$ have consecutive numbering in order to facilitate the implementation. Larger velocity sets increase accuracy, but require more memory and computing power. Many thanks to Moritz Lehmann for providing this figure.

be challenging for a LBM multi-GPU implementation, whereas the collision-step only depends on the populations of one lattice node[5]. For the streaming itself there are several implementation possibilities: the populations could be shifted along their directions in a procedural way by only using a small temporary buffer. Of course this approach can't make use of the GPUs PEs executing in parallel. The better option here is to stream to a new distinct memory adress by reading data from $f_i^A(\vec{x})$ and writing to $f_i^B(\vec{x}+\vec{c}_i\Delta t)$. f_i^A and f_i^B have to be swapped after each time step. This approach needs twice the memory, but is parallizable: each node can be streamed independently from the others. Let's have a closer look, how this double-memory method is implemented in FluidX3D.

Great optimization efforts with regard to the streaming-step have already been taken in the single-GPU version of FluidX3D. This is, because the performance of the basic LBM is mainly limited by the time the memory-update takes. We therefore say the GPU is in its *memory bandwidth limit*, whereas it would be in its *compute limit* when the execution time of the computing operations was much longer than memory transfer time. For the streaming-step two implementations are possible, the *push-scheme* and the *pull-scheme*. In the push-scheme, each node uses the populations located on its position f^A to conduct the collision-step and then writes the resulting populations in memory of the adjacent nodes of f^B. In the pull-scheme every node in f^B fetches the populations from its adjacent nodes of f^A, that are needed to conduct the collision-step.

[5] Although the notations suggests that Ω_i only depends on population i, it actually depends on all populations on the corresponding lattice-node, because f_i^{eq} depends on $\vec{u}$.



(a) Push: Array A. (b) Push: Array B. (c) Pull: Array A. (d) Pull: Array B.

Figure 8: Visualization of the underlying arrays of f^A and f^B. The populations before streaming are colored black, the ones after streaming red. In the push-scheme, collision is performed before streaming, in the pull-scheme after streaming (from [15], p. 101).

In both schemes, the algorithm has to read from and write to global memory only once during one LB-step. All computational intersteps can be saved to the GPUs registers, where data access is cheap. This procedure is called *One-Step-Algorithm* ([15], pp. 98-100). Although the push-scheme might seem a bit more intuitive, in FluidX3D the pull-scheme has been chosen. Again, this has to do with the hardware properties of GPUs: when consecutive threads read from or write to consecutive memory addresses in video memory, the memory transfer is coalesced into one. As can be seen in figure 9, the performance penalty for misaligned reads is smaller than for misaligned writes. So when the LB-algorithm is parallelized over the fluid nodes, writing the results of the collision into global memory can happen coalesced in the pull-scheme, if additionally the *Structure of Array* (SoA) schema is used as underlying memory layout. If a lattice consists of s_x lattice nodes in the x-direction and s_y and s_z nodes in the y- and z-direction, respectively, the position p in the buffer where f_i is stored is calculated from the global ID n and the total number of nodes $s = s_x s_y s_z \equiv r_g$ using $p = s \cdot i + n$ for a kernel call. An alternative memory layout would be Array of Structures (AoS) with $p = q \cdot n + i$ (q being the number of discrete velocities in the used velocity set), which optimizes data locality and should be preferred for CPU implementations.

Since FluidX3D parallelizes over the whole lattice, but only uses a one-dimensional index space, the coordinates (x, y, z) of the lattice node to be calculated and the global ID n are connected by the following bijection:

$$n = x + s_x(y + s_y z). \tag{21}$$

Without extensions FluidX3D needs the following buffers: one float buffer each for the f-populations f^A and f^B, one float buffer each for the density ρ and velocity $\vec{u}$, and finally a unsigned char (*uchar*) buffer called *flags*, which stores information about boundary conditions. The buffers ρ and $\vec{u}$ are only needed for initialization and for enforcing fixed boundary conditions - for the majority of the fluid velocity and density are recalculated at each simulation step from the f-populations and do not have to be updated in the buffer at all. Currently, FluidX3D also implements (incompatible) extensions:

- *temperature*: simulation including the temperature-dependent density behaviour of liquids.
- *shanchen*: simulation of evaporation and condensation processes.

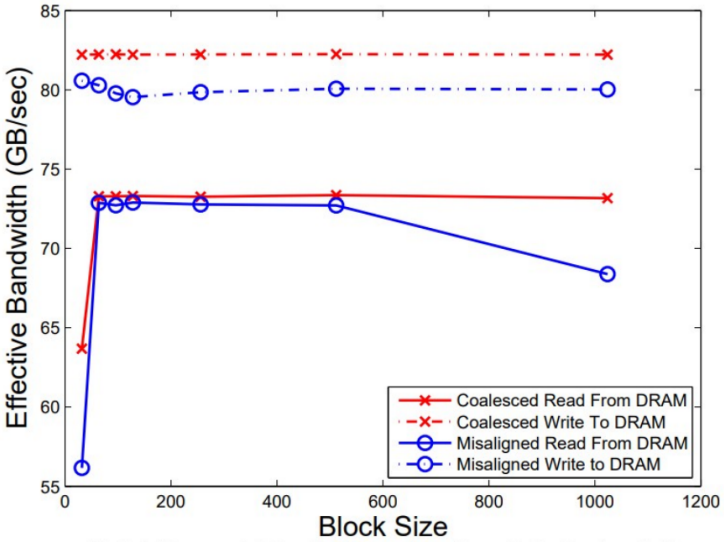


Figure 9: Performance for coalesced vs. misaligned read/write measured for the nVidia Quadro K5000M (maximal memory bandwidth according to specification sheet: 96GB/s). Performance penalty is smaller for misaligned read (from [16], p. 2570).

- *surface*: simulation of free surfaces adjacent to gas.

For implementing a multi-GPU for the LBM, the only important thing here is that some of these require additional buffers. These can be seen in the table 2.

Table 2: Overview of the buffers that FluidX3D needs for the various extensions. The extension shanchen does not need any additional buffers.

extension	additional buffer	meaning
temperature	T	temperature of each lattice node
temperature	g^A, g^B	field for thermal propagation, that uses discrete directions analog to the D3Q7 scheme
surface	φ	fill level
surface	mex	excess mass
surface mass	fluid mass	

Finally, boundary conditions should be taken into consideration here. FluidX3D without extensions has two kinds of boundary conditions. Without special adaption measures, periodic boundary conditions prevail on the surfaces of the domain, i.e. $f_i(s_x) = f_i(0)$ and analogously for all other data fields and surfaces. In the implementation of the single-GPU version of FluidX3D the periodicity is achieved by means of the arithmetic operation *modulo*. The coordinates $(x_{diag}, y_{diag}, z_{diag})$ of the diagonal neighbours of a lattice node at (x, y, z) are thus calculated as follows:

$$x_{diag} = (x + s_x \pm 1) \bmod s_x,\ y_{diag} = (y + s_y \pm 1) \bmod s_y,\ z_{diag} = (z + s_z \pm 1) \bmod s_z. \quad (22)$$

In addition, lattice nodes can be marked as *solid* in the buffer *flags*. If you apply non-slip boundary conditions, this means that at the streaming step the f-populations are calculated

through

$$f_i^*(\vec{x},t) = \begin{cases} f_{i+1}(\vec{x},t+\Delta t),\ i \text{ odd},\ \vec{x}+\vec{c_i} \text{ solid} \\ f_{i-1}(\vec{x},t+\Delta t),\ i \text{ even},\ i \neq 0,\ \vec{x}+\vec{c_i} \text{ solid} \\ f_i(\vec{x}+\vec{c_i}\Delta t, t+\Delta t),\ \text{else} \end{cases} \tag{23}$$

instead of equation 19. So the velocities are inverted at the solid boundaries.

3 Multi-GPU LBM: main idea and implementation details

3.1 Overview FluidX3D single- vs. multi-version

Firstly, an overview of FluidX3D with emphasis on initialization and communication processes will be provided. The kernel *stream_collide*, which actually executes the individual simulation steps of the LBM, is mostly treated as a black box, since only minimal interventions had to be made for the implementation of the multi-GPU version of the LBM. Figure 10 shows the program flow of FluidX3D for the single-GPU version. The class *Lattice* was created during the development of the multi-GPU version. It manages the most important simulation parameters as well as data fields, which are filled with values during initialization (figure 10, step (6)) and can be read out at any time step[6]. Here it should be emphasized once again that the kernels are not necessarily executed at the time of enqueuement - the end of their execution must be synchronized (figure 10, step (9.1)).

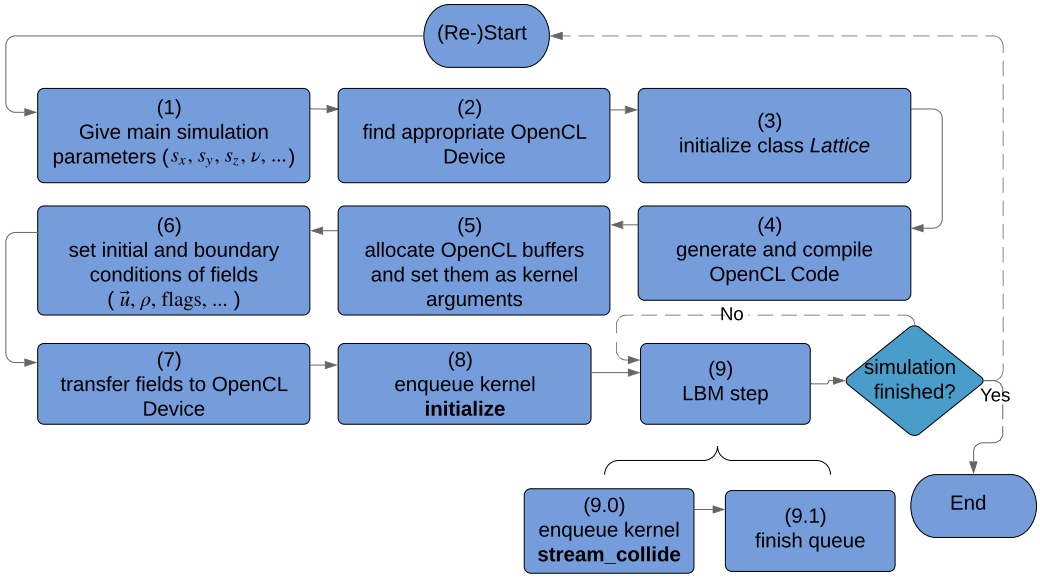


Figure 10: Initialization and communication processes of the single-GPU version of FluidX3D (without extensions). Kernels that appear in the program with this name are printed in bold type.

The most obvious difference of the multi-GPU version is that it uses multiple processor cores and GPUs. Figure 11 shows how they are organized: if n GPUs are used, FluidX3D needs $n + 1$ cores. One core (*master*) is used to coordinate the other cores (*slaves*), which hold the corresponding OpenCL context. The master is also used to initialize the data fields, which are then passed on to the slaves according to the sub-domain of the lattice for which the individual slave is responsible. Which slave is responsible for which sub-domain clearly determined by the coordinates assigned to it and by the sub-domain's sizes. The class *Lattice* automatically determines the slave coordinates and sizes of the sub-domains depending on the user input for the variables d_x, d_y and d_z (see 12, step (1.0)). Those variables determine in

[6]It does not explicitly contain f, because this can be calculated in the initialization process (by the kernel *initialize* from ρ and $\vec{u}$.)

how many sub-domains the lattice will be split along the corresponding axis. By default, the simulation box is split equally, but there is a possibility to change the sub-domain sizes along one axis[7]. It is also important that each slave is assigned a different GPU. The buffer sizes on the GPUs differ slightly from the sizes of the sub-domains, since halo nodes are introduced. The following chapter will explain this in further detail.

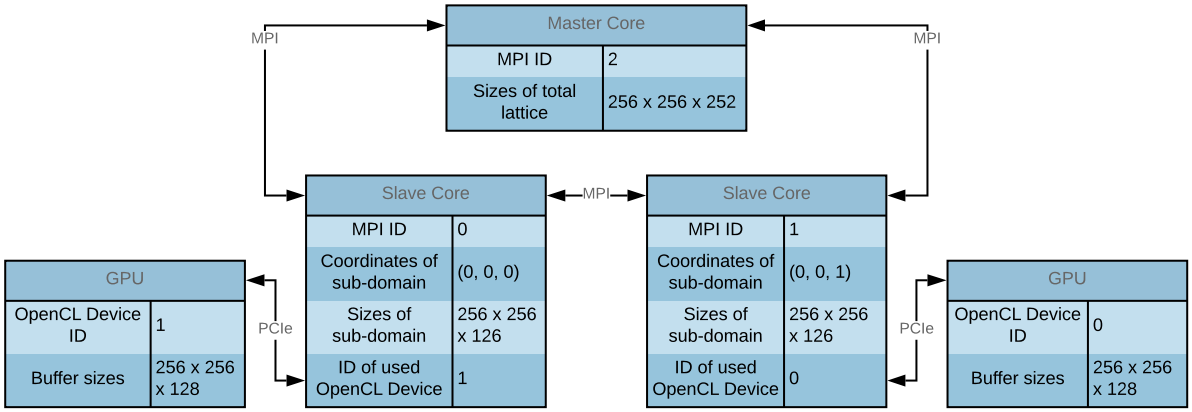


Figure 11: Communication structure of the multi-GPU version of FluidX3D. The easiest case of two GPUs splitting the lattice along z-direction is shown.

The initialization and communication processes, as they occur in the multi-GPU version of FluidX3D, are shown in Figure 12. How the LBM step (9) works, what is meant by *"template"-kernels* and what the kernel *setget_layer* does will be explained later.

3.2 Harmonization of MPI and OOP

Now a process in the course of the initialization is to be singled out first, which caused quite large difficulties on the implementation side, i.e. how MPI can brought into line with the *object-oriented programming* (OOP) approach of FluidX3D. Objects have to be dispatched over MPI, which is the case in the steps (3.1) and (6.1). In addition, no 1:1 copies of the lattice class are sent here, but rather adapted copies containing only for the slave relevant parts of the data fields and updated simulation parameters (e.g. the sizes of the respective sub-domains $s_{s,x}$, $s_{s,y}$, $s_{s,z}$ instead of the total box size s_x, s_y, s_z). Since the kernel expects stream-collide 1D buffers to be indexed according to equation 21, the *Lattice*-class therefore contains objects of another class called *Tensor3* for storing data fields. These can manage 1D arrays as effective 3D arrays and in particular easily allow copies of parts of the effective 3D array. The class *Tensor3* itself is based on the container *std::vector*. So it must be possible to send several nested objects via MPI. However, MPI itself is purely function-oriented - only basic data types can be sent and it must always be specified how many of them are sent to which slave. Basically, there are three ways to send objects via MPI:

a Functional MPI: encode all MPI transfers explicetly

b Use the C++ source library *serialization* from *Boost*.

c Own solution with modern C++.

[7]This is useful if e.g. GPUs with different memory limits or performance characteristics are used.

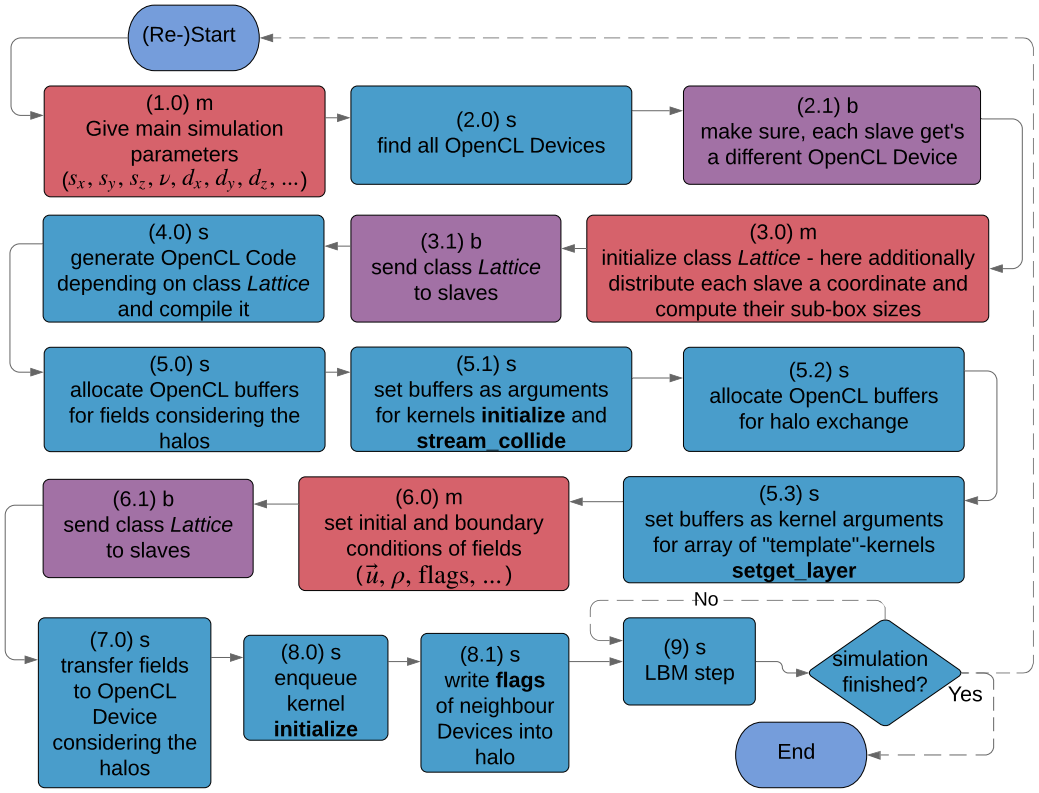


Figure 12: Initialization and communication processes of the multi-GPU version of FluidX3D (without extensions). Master processes are marked red, slave processes blue, processes involving masters and slaves purple. Kernels and buffers that appear in the program with this name are printed in bold type.

Option a) always offers the highest performance and is therefore used for communication in step (9). However, the runtime is not essential for sending *Lattice*-objects, since this only occurs during initialization. In this case, repeating code, as it is generated during functional programming, would be a disadvantage. In addition, the MPI communication protocol on the master and slave side must match exactly and therefore has a relatively large bug potential.

Boost's C++ source libraries from option b) are relatively well known, but still their use would restrict the portability of FluidX3D. The library *serialization* makes it possible to convert objects into *archives* and reconstruct them. The archives have a string representation and could easily be sent via MPI. For this only a few interventions in the respective class itself are necessary - all members of the class to be archived have to be listed in only one place in the code. The members themselves must also be compatible with *serialization*, which is already the case for all basic data types and containers from the C++ standard. However, the conversion of objects into strings is very inefficient: *text archives* have to be used instead of *binary archives*, because according to Boost only these are portable across platforms. For a double with 15 decimal digits of precision each digit would be represented as char in the string. When testing this implementation method, waiting times of several minutes were already observed during the initialization of medium-sized lattices.

Option c) is an independently developed solution that is not noticeably disadvantaged in terms of performance compared to a). Its implementation can be seen in the listings 1 and 2 in Appendix A. In the classes *Tensor3* and *Lattice* an instance of the template-class *Variant_Container* (listing 1, ll. 12-21) is created and in it inside a *std::vector* with elements of the type *std::variant* (C++17) references to class members to be sent are saved (for class *Lattice* see listing 1, ll. 84-93). The method *Variant_Container<T>::visit* (listing 1, ll. 158-169) iterates through this vector and calls for each entry *std::visit* together with the type-matching visitor consisting of lambda-functions. For each data type that can occur in the vector, a lambda-function must exist. For example, if *Lattice::mpi_send* (listing 2, ll. 4-6) is called, all contained *Tensor3*- and *std::vector*-objects will be sent recursively. Since the order of the references in the vector is clearly defined, the send order matches the recieve order of *Lattice::mpi_recv*. In listing 1 the type-matching visitor is implemented using lambda templates (C++20). In FluidX3D a version compatible with C++17 can also be selected, which writes out the templates for each used data type. The class *LBM_Data3* inheriting from *Tensor3* was introduced to be able to distinguish LBM data fields from Tensor3 objects, which only store simulation parameters, on the basis of the data type. This is important for the call *Variant_Container<T>::visit_LBM_Data3* (listing 1, ll. 70-82), in which a process analogous to the one above for all LBM data fields creates the corresponding sub-domains for the slaves from the total lattice or inserts them in the correct position.

The big advantage of option c) over a) is that when introducing further data fields (see extensions of the LBM, which are not shown in the listings) the multi-GPU code only has to be changed at one single position in the Lattice class compared to the single-GPU code - a corresponding reference has to be inserted into the *Variant_Container*-instance inside the *Lattice* class at any position. Its only disadvantage is that it requires at least compiler compatibility with C++17.

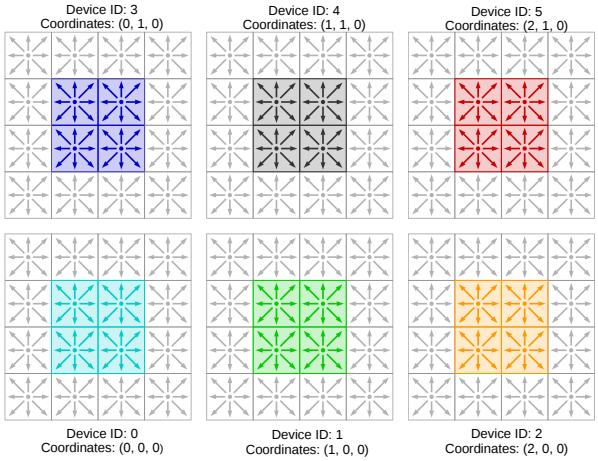
3.3 Main idea of halo transfer

A multi-GPU LBM faces the challenge that in step (9) in figure 12 f-populations of the neighbours must be accessible during streaming. However, at the surface of the sub-domains these are possibly stored on the GPU with adjacent coordinates. Therefore, a transfer of f-populations has to take place before each streaming - a process that directly influences the performance of the overall simulation and therefore has to be as efficient as possible. In a naive approach, when using the velocity set D3Q19 together with a domain split at least three times along each axis, from one GPU memory transfers to 18 different neighbouring GPUs would have to be performed. If extensions are added, some of which have to read out scalar values from neighbouring nodes in each simulation step (see table 4), there would even be transfers to 26 neighbouring GPUs. Between GPUs which are aligned diagonally to each other with respect to their coordinates, only one direction of a single f-population has to be exchanged - this would be highly inefficient due to the latency times and the bandwidth depending on the buffer or message size (see chapter 2.3) both for the transfer between device and host and for the MPI transfer between hosts. In addition, many case distinctions based on the corresponding velocity set would be necessary. Another critical point is that the streaming itself should remain unchanged. If case distinctions for marginal populations are included, branching occurs. Furthermore the question arises how to deal with periodic boundary conditions: if the neighbourhood calculation through equation 22 remains unchanged, this can lead to periodicity of the sub-domains instead of the total domain.

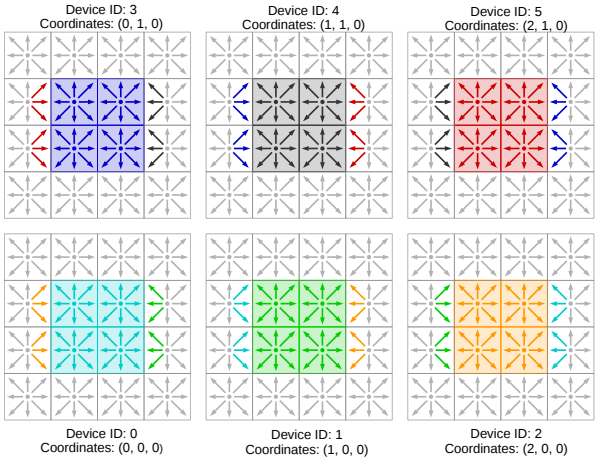
To all these problems there is an answer as shown in figure 13 for the case of 6 GPUs and the velocity set D2Q9. As can be seen in figure 13a, at each outer surface of the sub-domain that is facing another sub-domain, a one lattice node thick halo is introduced. This is implemented by increasing the size of all buffers of the sub-domains as it was exemplified in figure 11. Which lattice node is a halo-node can easily be calculated from its coordinates. During memory transfer, the layers directly behind the halo layer on both sides of the sub-domains (*sub-halo layer*) are copied into the halo layers of the adjacent GPUs (see figures 13b and 13c). Since this sub-halo layer may also include halo nodes of halo layers perpendicular to it, 4 transfers (two per direction) are sufficient in the example. For the 3D case this means a maximum of 6 transfers independent of the velocity set selected. It is sufficient to copy only f-populations, which have a component in positive direction of the neighbouring GPU (with velocity set D3Q19 5 instead of 19 f-populations). After all transfers have taken place and the buffers have been unpacked into the halo layer, the kernel *stream_collide* can be called. If all halo-nodes are returning immediately at the beginning of the kernel, the rest can be executed without changes compared to the single-GPU version: all other lattice nodes can access the required f-populations of their neighbours during streaming. Furthermore, the memory transfers, which are based on periodic neighbourhood, automatically result in periodic boundary conditions. The operation mod from equation 22 has no effect for outer surfaces consisting of halos (since they were already returned), but is retained to avoid case distinctions. For the LBM without extensions the non-slip boundary conditions work as in the single-GPU version, if the data field *flags* is transferred in a analogous manner to the neighbouring halos during the initialization process (see figure 12, step (8.1)).

Figure 14 shows what happens in detail in step (9) of figure 12. The indices 0, 1 and 2 are assigned to the axes x, y and z. Let $\mathrm{ax}(i)$ be the function that does this assignment and $\mathrm{iax}(i)$ its inversion. For each direction involved, a kernel named *setget_layer[i]* is called twice. For

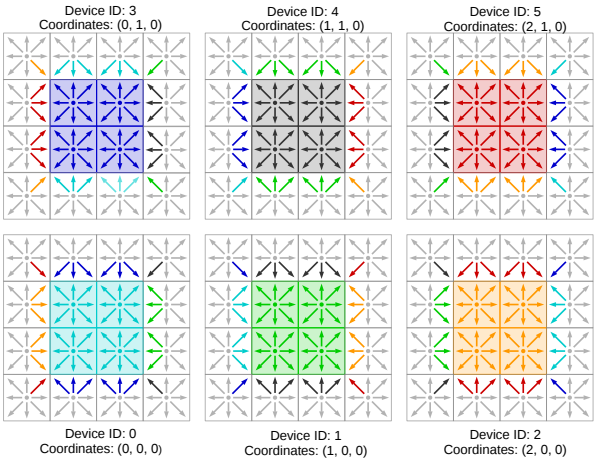
Figure 13: Visualization of the halo transfer. Coordinates are assigned to the slaves which manage the devices. The data exchange takes place via MPI and PCIe. f-populations located in the corners (edges) of the sub-domains need no special treatment: through the special scheme they are transferred thrice (twice) and end up at the right position.



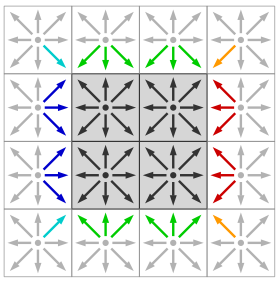
(a) f-populations before halo transfer.



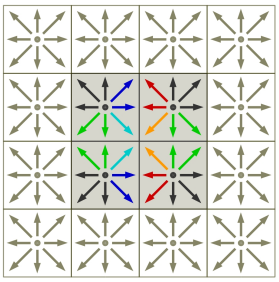
(b) f-populations after transfer in x-direction.



(c) f-populations after transfer in y-direction.



(d) Pull: Array A.



(e) Pull: Array B.

$i \in \{3, 4, 5\}$, the kernel writes for the direction $\text{iax}(i-3)$ all f-populations and scalar values (needed by the extensions) to be transferred to a single transfer buffer called *buffer_trans_send*. Which scalar values have to be transmitted for which extensions can be found in table 4. In order to handle the different occuring data types, they are cast to *uchar* before being stored. For optimal performance, the memory layout SoA is used in this buffer. In addition, the buffer divides into two halves: the first half contains all values of the sub-halo layer $\text{iax}(i) = 1$, in the other half the values of the sub-halo layer $\text{iax}(i) = s_{\text{iax}(i)} - 2$. All data can be passed from host to device in a single transfer, which promises optimal bandwidth. The two halves of the buffer are then sent one after the other via MPI to the two neighbouring hosts by using the *Sendrecv* Pattern. All cores that are in a row along the $\text{iax}(i)$-axis according to their coordinates form a periodic chain. The message simultaneously received from the neighbouring cores is stored in the corresponding half of the buffer *buffer_trans_recv*.[8] This buffer is then transferred to the device and its entries are copied to the right positions in the halo with another call of *setget_layer[i]* with $i \in \{0, 1, 2\}$.

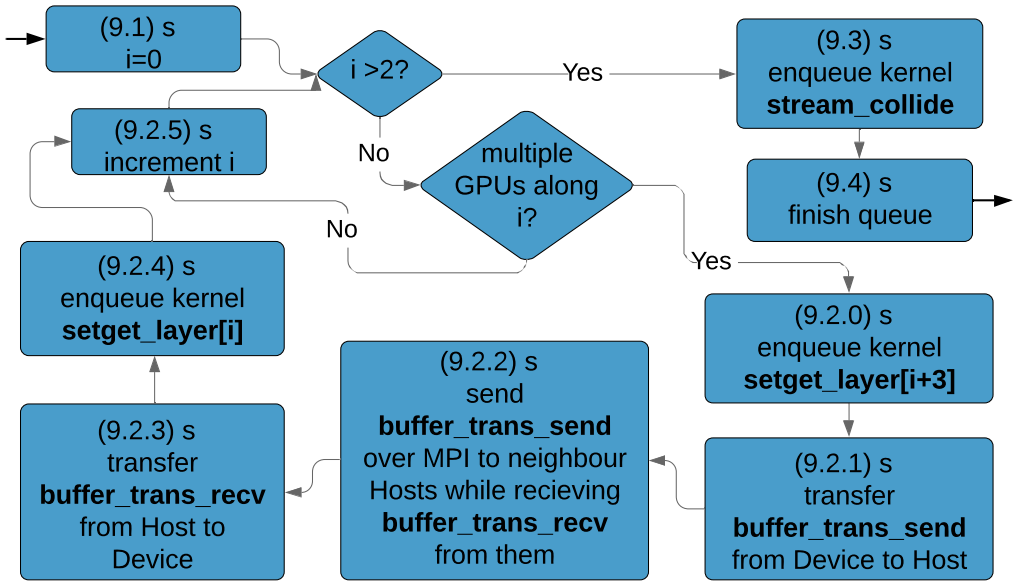


Figure 14: Detailed illustration of step (9) in figure 12. The axes x, y and z are assigned to the indices 0, 1 and 2. Kernels and buffers that appear in the program with this name are printed in bold type.

As already indicated, *setget_layer* is an array of kernels. Due to the symmetry of both processes a function could be developed, which - depending on its control parameters - either prepares a transfer buffer for the selected direction or writes one back into the halo (see listing 3, ll. 29-59). If the parameters in the loop (9.2) in figure 14 were set by the host, additional buffer transfers would be necessary. Even if the parameters were integrated into the buffer *buffer_trans_recv* by means of a fixed protocol, they would have to be read from the global memory when *setget_kernel* is called. Therefore a different way is chosen here: At runtime before compiling the OpenCL code, kernels are automatically generated which already contain the control parameters as a fixed part of their code (see listing 3, ll. 44-47). So a kind of "template"-kernel was constructed and its instances explicitly created[9]. The position of each

[8] Since memory is not shared among processors in MPI, these copy processes can not be avoided, even if the MPI cores are on the same node.

[9] In contrast to CUDA there is no real template kernel in OpenCL.

template instance in the kernel array indicates the control parameters it contains.

It is also important that the functions *assign_float* and *assign_uchar* in *setget_layer* contain an if-else-statement - but since all group-members call them with the same transfer parameter *set*, there is no branching.

Furthermore, it should be mentioned that *setget_kernel* does not parallelize over whole sub-domain during creation/readout of the transfer buffer, but over the required layers themselves. Let $s_{h,x}$ be the length of the sub-domain in x-direction, which (in contrast to $s_{s,x}$) may include the halo ($s_{h,y}$ and $s_{h,z}$ analogously). Then the equation 21 no longer applies to the indexing, but (in the case of D3Qq):

$$n = \mathrm{ax}(i) + s_{h,\mathrm{ax}(i)}\mathrm{ax}((i+1) \bmod 3) + b \cdot s_{h,ax(i)}s_{h,ax((i+1) \bmod 3)},\ i \in \{0,1,2\}. \tag{24}$$

The boolean b is 0 for the front layer and 1 for the back layer. This approach has the advantage that fewer work-groups have to be executed and less branching takes place within them than when parallelizing over the sub-domain.[10]

Let h_x be 1, if halo x-layers exists in the sub-domains, and 0 otherwise (h_y and h_z analogously). Then the number of a sub-domain's lattice nodes N_c involved in the data communication can be calculated by

$$N_c = 2(h_x s_{h,y} s_{h,z} + h_y s_{h,z} s_{h,x} + h_z s_{h,x} s_{h,y}). \tag{25}$$

It is worth mentioning that in order to exchange front layer and back layer consisting of 416 x 416 nodes each, as it can occur with large boxes (see chapter 5), $2 \cdot 416^2 \cdot 5 \cdot 4$ Byte ≈ 7 Megabyte have to be sent, if the D3Q9-scheme and no extensions are used (one float needs four Bytes; five f-populations per layer are sent). Figures 4 and 6a show, that GPU Memory Bandwidth and MPI data transfer rate are on the plateau of optimal performance for this amount of data.[11] The *Exchange* Pattern for the MPI communication would only be beneficial for layers consisting of 60 x 60 nodes and below. It is therefore not implemented in FluidX3D.

At this point it should be mentioned that many comparable multi-GPU implementations of the LBM use halo-nodes - all of them except [4] working with CUDA instead of OpenCL. But [5], [1], [4] and [7] are limited to arrange the sub-domains along a preferred axis (which here would be the z-axis), so the memory areas to be exchanged are already contiguous and therefore the call of further kernels (here *setget_kernel*) can be omitted. Only [3] can divide the lattice into sub-domains along all 3 spatial directions by using a transfer system similar to the one described here.

Why have the special scalar values from table 4 to be exchanged? Table 3 yields the applied criteria: data fields have to be exchanged in the main simulation loop if in *stream_collide*

a neighbouring positions are read out whose values can change during the simulation or

b neighbouring positions are written to.

[10] Additionally, the query of whether the work-item in question is on the desired layer is saved.

[11] The MPI transfer consists of 2 messages 3.5MB each in this example.

Except for the extension *surface*, the entries in the data field *flags* are constant during the simulation, so criterion a) is not met. For the *surface*-module already in the single-GPU version several kernels are called in the simulation loop. The procedure for the multi-GPU version can be found in table 5, where the above mentioned criteria are applied again. Further arrays of "template"-kernels analogous to *setget_layer* are necessary here, each implemented to transfer a specific combination of scalar values.

3.4 Optimization strategies

All optimizations described until now are a fixed part of the multi-GPU implementation of FluidX3D. The following optimizations can be activated by the user:

a use of host pointer

b use of vendor specific buffers (alternatively to a)

c compute-transfer overlap

d load balancing (building upon c)

3.4.1 Use of task-specific buffers

In optimization a) the flag CL_MEM_USE_HOST_PTR is set when creating the buffers *buffer_trans_send* and *buffer_trans_recv*. The use of host pointer is advantageous here, because before (after) each transfer from (to) the device the buffer is written (read out) only once. Alternatively, depending on the vendor of the GPU, special flags can be set when creating the buffer. For nVidia this is the flag CL_MEM_PINNED_NV, which promises guaranteed pinned host memory on mapping (see [17]). The memory is prevented "from being swapped out and provides improved transfer speeds" ([18]). For AMD GPUs, the use of the flag CL_MEM_USE_PERSISTENT_MEM_AMD would be conceivable. If the use of virtual memory is possible on the driver side, the buffer corresponds to a host-visible device memory, which also increases the transfer speed here (see [19]). Although both vendor specific buffers are implemented, they could not be tested because they are quite new and a corresponding driver for the operating systems of the clusters was not available.

3.4.2 Compute-transfer Overlap

The idea to overlap LBM computation and buffer transfers is not new (see [5], [1], [7] for multi-GPU along one axis and [4] for multi-GPU along all three axes). The required task parallelism must explicitly be created in the implementation. The main idea (here explained in the push-scheme) is that all nodes streaming in halo-nodes are calculated first. After filling the buffers to be exchanged, they are transferred, while the rest of the nodes are calculated at the same time. However, there are difficulties with the above mentioned transfer scheme: the individual layers cannot generally be transferred independently from each other - as shown in figure 13 for the velocity set D2Q9, the transfer buffer for the transfer of the second layer can only be filled after the transfer of the first layer. This ensures that the populations in the corner of the sub-domain are transferred two times in a row. The required kernel (here *setget_layer*)

Table 3: This overview shows which combination of kernel and extension manipulates which data fields. Manipulation can be done locally (only the memory entry specified by the global ID n is read/written), or additional neighbour entries can be manipulated. Fields in brackets are only written if the user explicitly requests this (for example, to be able to transfer the values to the host later). Fields marked with * are only read if they are set as boundary conditions.

extension	kernel	read locally only	read locally and from neighbours	write locally only	write locally and from neighbours
no extension	initialize	ρ, $\vec{u}$, flags	-	f, $\vec{u}$, flags	-
	stream_collide	ρ^*, $\vec{u}^*$	f, flags	f, (ρ, $\vec{u}$)	-
shanchen	initialize	ρ, $\vec{u}$, flags	-	f, $\vec{u}$, flags	-
	stream_collide	$\vec{u}^*$	f, flags, ρ	f, ρ, ($\vec{u}$)	-
temperature	initialize	ρ, $\vec{u}$, flags, T	-	f, $\vec{u}$, flags, g	-
	stream_collide	ρ^*, $\vec{u}^*$, T^*	f, flags, g	f, g, T, (ρ, $\vec{u}$)	-
surface	initialize	$\vec{u}$	ρ, u, flags	f, $\vec{u}$, flags, mass, mex, φ	-
	stream_collide	ρ, $\vec{u}$, mass	f, flags, mex, φ	f, flags, (ρ, $\vec{u}$)	-
	surface_1	-	flags	-	flags
	surface_2	-	ρ, $\vec{u}$, flags	f	flags
	surface_3	ρ, $\vec{u}$, mass	flags	flags, mass, mex, φ	-

Table 4: When using extensions, scalar fields have to be transferred additionally to the f-populations. $\vec{u}$ can be treated as a scalar from an algorithmic point of view.

extension	additional scalar buffers to transfer
-	-
temperature	g_c
shanchen	ρ
surface	$\vec{u}$, ρ, flags, mex, φ

Table 5: The extension *surface* requires a complex sequence of transfers and kernel calls.

step	execute kernel	transfer layers of buffer
1	-	f, flags, mex, φ
2	stream_collide	-
3	-	flags
4	surface_1	-
5	-	ρ, $\vec{u}$, flags
6	surface_2	-
7	-	flags
8	surface_3	-

must access some buffers also needed for the LBM simulation (here kernel *stream_collide*). OpenCL explicitly prohibits two kernels from using the same buffer at the same time. Possibly [3] does not mention the implementation of a compute-transfer overlap for similar reasons.

The multi-GPU implementation of FluidX3D overcomes these difficulties by assigning one of 4 *compute numbers* to each node. Having compute number 0 means that the lattice has a subdivision into sub-domains along the x-axis and at least one work-item of the own work-group is part of a layer with $x \in \{0, 1, s_{h,x} - 2, s_{h,x} - 1\}$ - i.e. part of the corresponding halo or sub-halo layer. The same applies to compute number 1 (y) and 2 (z). The work-groups are considered instead of single work-items to avoid branching. If a node meets the conditions for multiple compute numbers, the following descending priority exists: 2, 1, 0. All remaining nodes get compute number 3. For the implementation see listing 3, ll. 91-115.

Instead of a single kernel an array of "template"-kernels is now used for *stream_collide*. They are created the same way as the "template"-kernels for *setget_kernel*. Kernel *stream_collide[i]* executes the LBM simulation step only for nodes with compute number i. All other nodes return immediately. For a lattice split in sub-domains along all three axes, figure 16 shows the resulting procedure within one simulation step. It can be seen, that one group (here compute group 2; if not present descending priority 2,1,0) isn't used for overlapping communication.

In figure 15 the compute-numbers for a GPU arrangement 1 x 2 x 2 are visualized. You can nicely see that because of the consideration of whole work-groups also nodes with $y = s_{h,y} - 3$ are provided with compute number 1. Here it is also easy to notice that the computational load is not distributed equally between compute groups that overlap MPI communication (compute groups 3 and 1 in figure 15). An equal distribution would be desirable, because with cubic sub-domains MPI communication takes approximately the same time for each transfer.

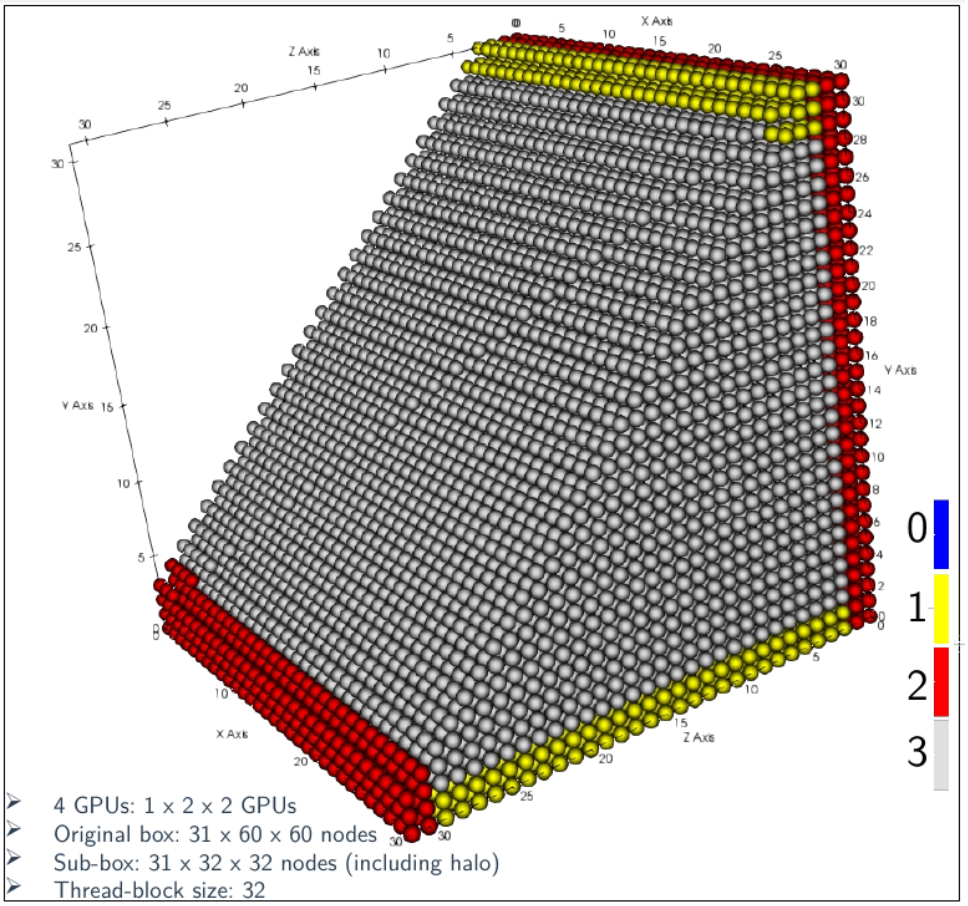


Figure 15: Visualization of the distribution of compute numbers in a sub-domain for a GPU arrangement 1 x 2 x 2. Each node is represented by a colored sphere. For a better overview only a part of the sub-lattice is shown, where compute-groups are always left intact.

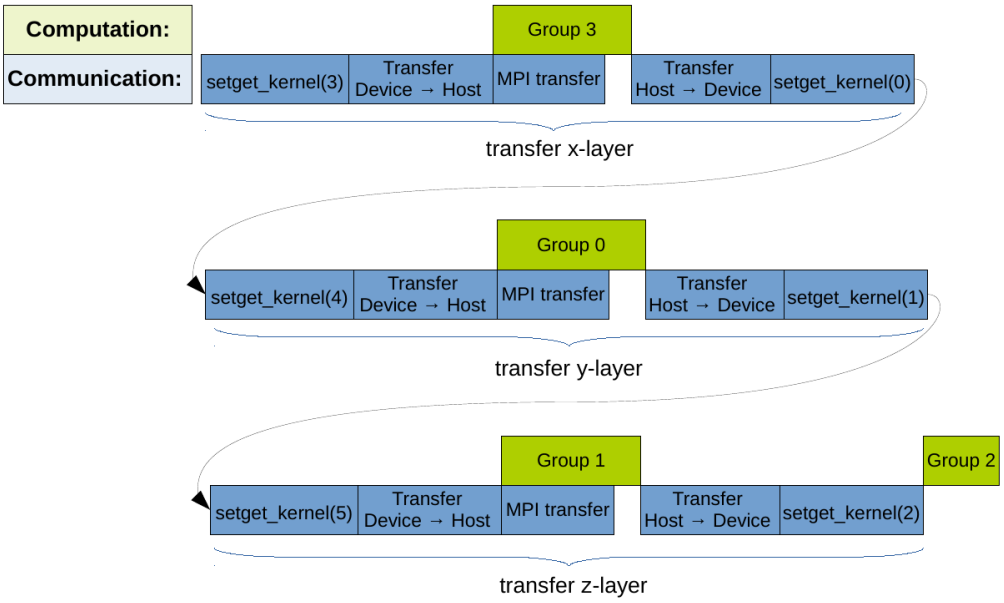
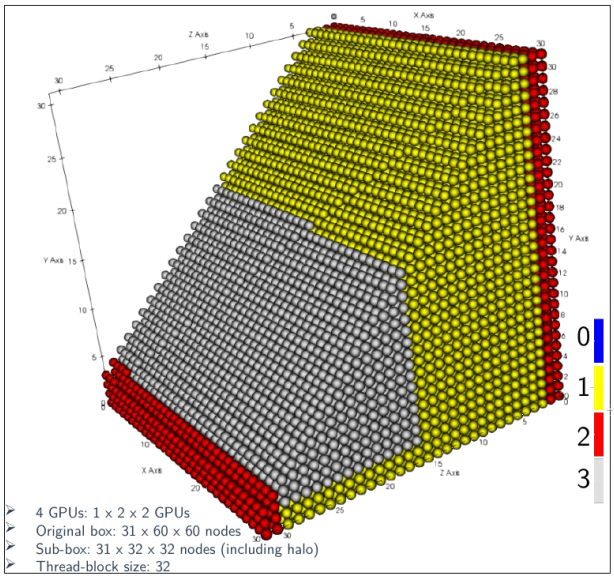


Figure 16: Procedure for a lattice split in sub-domains along all three axes, if compute-transfer overlap is activated. An overlap of Host-Device communication could not be achieved.

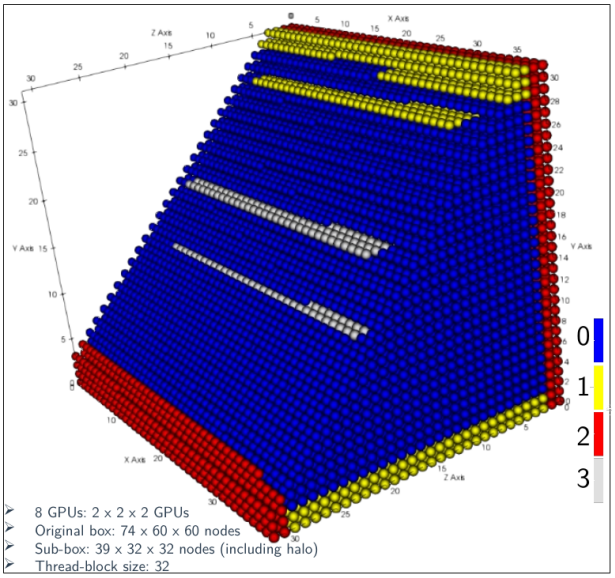
3.4.3 Load-balancing

To achieve a more equal distribution of the compute numbers, *load balancing* can be done in addition to compute-transfer overlap. The nodes that used to have compute number 3 (from now on called *inner nodes*) are distributed between all compute groups involved in the communication overlap (in figure 16 this are compute groups 3, 0 and 1). The one compute group left (in figure 16 it is compute group 2) is not assigned more nodes, so that it can be calculated as fast as possible at the end of the communication process. The division of the inner nodes is based on their z-position (*z-distribution*, see listing 3, ll. 61-91). This is shown in figure 17a for the same setup as in figure 15. Figure 17b shows that load balancing (and compute-transfer overlap in general) for lattices with subdivisions along the x-axis does not work well if $s_{h,x}$ is in an order of magnitude less than or equal to the TBS b. Because x is the linear index, work-groups containing work-items with $x \in \{0, 1, s_{h,x} - 2, s_{h,x} - 1\}$ extend far into the inner of the sub-domain. As a result, compute group 0 occurs much more often than compute groups 3 and 1 in the example. For certain $s_{h,x}$ (e.g. $s_{h,x} = b$ or $s_{h,x} = 2b$) there are no nodes with compute number 3. Therefore splitting the lattice along the z- and y-axis should be preferred.

Figure 17: Possible distributions of compute numbers in a sub-lattice, if load balancing is activated.



(a) The domain shown here has the same measures and GPU arrangement as in figure 15.



(b) Lattice with sub-domains 2 x 2 x 2. Splitting the lattice along the x-axis can decrease performance compared to splitting along the other axes.

More sophisticated balancing algorithms for distributing the inner nodes to compute numbers could be used. One more was tested: be $m \in \{1, 2, 3\}$ the number of axes with multiple GPUs. Then distribute the inner nodes by their group ID w according to $w \bmod n$ (*mod-distribution*). This would have the advantage that the compute numbers would be distributed more homogeneously over the sub-lattice[12]. However, this algorithm proved to be inefficient for AMD GPUs (see figure 26). It should also be mentioned that the division into compute-groups

[12]This is particularly relevant for the extension *surface*, where the computational effort of *stream_collide* for some parts of the domain differs drastically from the rest.

could be calculated before the main simulation and stored in a buffer, since it doesn't change during the simulation. However, it is more efficient to calculate the compute-numbers in each simulation step than to read them from the global memory.

In the current implementation of the compute-transfer overlap only the MPI communication is overlapped. An overlap with the buffer transfers Host-Device and Device-Host would be desirable and should be theoretically possible, but could not be proven in a benchmark program created for its measurement[13]. Altogether it would be conceivable to reverse the whole compute-transfer process, i.e. to transfer already calculated parts and to calculate the inner part at the end. However, this procedure does not harmonize so well with the surface module (see table 5: instead of step 1 and 2, step 2 would only overlap with a transfer of ρ, $\vec{u}$, flags). A compute-transfer overlap with possible load-balancing has also been implemented for the kernels *surface_1*, *surface_2* and *surface_3*. Due to the low computational effort of the kernels, the performance gain is balanced with the overhead of compute-transfer overlap.

4 Validation of the multi-GPU implementation

Validating the multi-GPU implementation of the LBM is conceptually simple. The simulations run deterministically, so it's sufficient to compare the results of the multi-GPU version against the results of the single-GPU version at certain simulation steps: there should be no deviation.

This kind of binary comparison has been applied to all setups in this chapter. The procedure was as follows: First, a domain size must be found that satisfies equation 18 for the single GPU version with $r_g = s_x s_y s_z$ and at the same time for all sub-domains of the multi-GPU version with $r_g = s_{h,x} s_{h,y} s_{h,z}$. Then the chosen setup is first executed in the single-GPU version and the resulting data fields $\vec{u}$ and *flags* for the whole domain are stored in files after 1000 and 9000 simulation steps. Then the same setup is executed in the multi-GPU version. The data fields $\vec{u}$ and *flags* after 1000 and 9000 steps are collected and stored by the master with the help of the *Lattice* class. The halo is removed during the collection process. Finally, the result files of the multi-GPU version are bitwise compared with those of the single-GPU version. The binary comparison is only passed if all values in the entire domain match exactly for both data fields and both simulation steps.

As test-setup, the lid-driven cavity setup was used for all simulations except when validating the extensions (see table 9) and when testing the D2Q9 scheme (poisseuille setup). In the lid-driven cavity setup all surfaces of the domain except the surface $z = s_z - 1$ have no-slip walls. On the face $z = s_z - 1$ there is the boundary condition $u_y = 0.5$. All other velocities are initialized with zeros. The viscosity of the fluid is $\nu = 0.0149$. In the poisseuille setup for D2Q9, there are no-slip walls at $y = 0$ and $y = s_y - 1$. The viscosity is $\nu = 1/6$ and there is a constant volume force in x-direction with $F_x = \nu / \sqrt{0.5 \cdot s_y - 1}$. Along a line with $x = \text{const.}$ a quadratic velocity profile should result.

All computational options, where the multi-GPU version could differ from the single-GPU version, were tested, beginning with the different velocity sets (D2Q9, D3Q7, D3Q13, D3Q15,

[13] Different GPUs and drivers were tried.

D3Q19, D3Q27). The way, how the lattice was divided into sub-domains for the multi-GPU version (denoted by d_x, d_y, d_z) was varied as well as the size of the lattice (denoted by s_x, s_y, s_z). Table 6 lists the tests for D2Q9, table 7 tests conducted for the D3Qq-schemes.

Table 6: Parameters for the D2Q9-scheme, that were validated via binary comparison. All parameters have been applied to the poisseuille setup. These parameters were constant during all simulations below: $d_z = 1$, $s_z = 1$, TBS = 256. These options were activated during all simulations below: compute-transfer overlap, buffer-mapping, load-balancing.

d_x	d_y	s_x	s_y
2	1	444	256
1	2	256	252
2	2	384	256
2	2	252	1020

Table 7: Parameters that were validated via binary comparison for the following schemes: D3Q7, D3Q13, D3Q15, D3Q19, D3Q27. All parameters have been applied to the lid-driven cavity setup. These parameters were constant during all simulations below: TBS = 256. These options were activated during all simulations below: compute-transfer overlap, buffer-mapping, load-balancing.

d_x	d_y	d_z	s_x	s_y	s_z
2	1	1	50	32	32
1	2	1	32	60	32
1	1	2	32	32	70
2	2	1	28	28	32
2	1	2	60	32	60
1	2	2	64	50	60

For all following validations as well as performance measurements, the D3Q19 scheme is used, since it is the most common one. FluidX3D was tested using different TBS. The corresponding validation tests are equal to the ones listed in table 7, only this time always the D3Q19 scheme was used and instead TBS took the following values: 32, 64, 128, 256.

The multi-GPU LBM implementation of FluidX3D has different computation options which impact the performance. Table 8 shows their validation.

Finally, the different LBM extensions (no extension, *temperature*, *shanchen*, *surface*) have been validated. For each of them, a different setup had to be used. Table 9 shows, which other parameters were varied. For the module *surface*, conducting an additional compute-transfer overlap for the kernels $surface_i$, $i \in \{1, 2, 3\}$ (further referred to as *surface overlap*) was validated here as well.

Domains with no-slip walls on all faces were chosen to test the *temperature* module (temperature-setup). Lattice nodes with $y = 1$ have constant temperature $T = 1.5$, nodes with $y = s_y - 2$

Table 8: Validation of the computation options via binary comparison. These parameters were constant during all simulations below: TBS = 256. These options were activated during all simulations below: D3Q19. Tests with compute-transfer overlap active, buffer mapping deactivated, load-balancing active and compute-transfer overlap surface deactivated are denoted 1010.

d_x	d_y	d_z	s_x	s_y	s_z	tested compute options	setup
2	1	1	124	64	64	0000, 0100, 1100, 1110	lid-driven cavity
1	2	1	64	100	64	0000, 0100, 1100, 1110	lid-driven cavity
1	1	2	80	40	124	0000, 0100, 1100, 1110	lid-driven cavity
2	2	1	124	60	64	0000, 0100, 1100, 1110	lid-driven cavity
2	1	2	60	32	124	0000, 0100, 1100, 1110	lid-driven cavity
1	2	2	64	124	60	0000, 0100, 1100, 1110	lid-driven cavity
2	2	2	124	128	136	0000, 0100, 1100, 1110	lid-driven cavity
2	2	2	36	192	60	1110, 1111	drop
2	2	2	128	124	128	1110, 1111	drop
2	2	2	444	444	448	1110, 1111	drop

Table 9: Extensions validated via binary comparison: no extension, *temperature*, *shanchen*, *surface*. These parameters were constant during all simulations below: $d_x = d_y = d_z = 2$, TBS = 256. These options were activated during all simulations below: D3Q19, compute-transfer overlap, buffer-mapping, load-balancing.

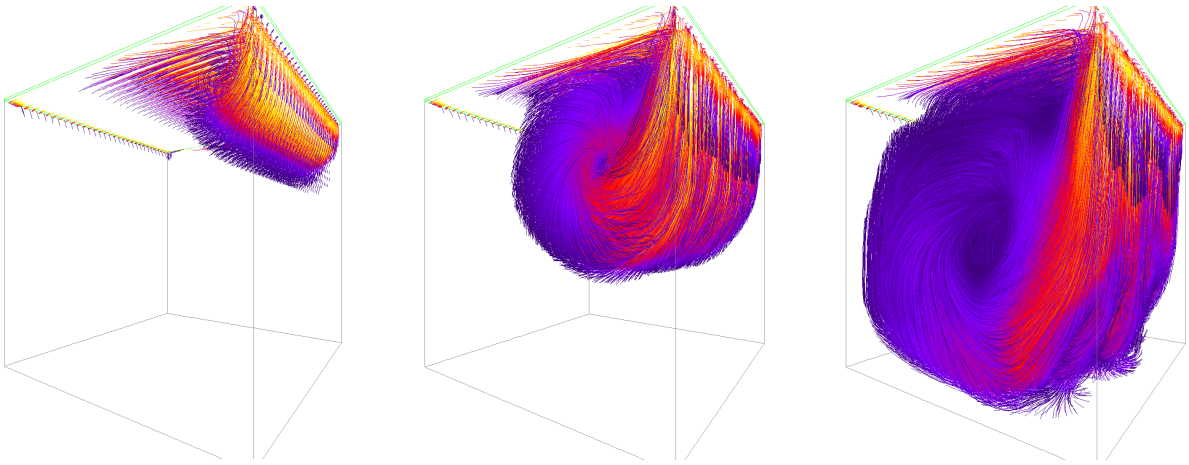
s_x	s_y	s_z
36	192	60
128	124	12
444	444	448

have constant temperature $T = 0.5$. All other nodes have an initial temperature of $T = 1.0$. The viscosity is $\nu = 0.02$.

In the setup for *shanchen* (shanchen-setup) there is a cylinder of length $l = 50$ lattice nodes with radius $r = 25$ lattice nodes aligned along the y-axis in the center of the domain. Its density is $\rho = 2$. All other lattice nodes have a density of $\rho = 0.1$. On all surfaces of the domain there are no-slip walls. The viscosity is $\nu = 0.01$, furthermore there is a constant volume force in z-direction with $F_z = -0.0001$.

The drop-setup was chosen to validate the surface module: at initialization there is a drop with radius $r = 19$ in the center of the domain, where it is shifted down in z-direction by $r + 2$ and has an initial velocity in z-direction with $u_z = -0.2$. In addition, the box is filled with resting liquid up to a filling level of $h = 40$. All other lattice nodes are of type gas. All surfaces of the domain except $z = s_z - 1$ have no-slip walls. The viscosity is $\nu = 0.02$, there is a constant volume force in z-direction with $F_z = -0.0002$.

The figures 18, 19 and 20 show the time evolution of the setups lid-driven cavity, temperature and shanchen for concrete box sizes. They are rendered with FluidX3D's own graphics at runtime, which is currently only possible in the single-GPU implementation and a Windows operating system.

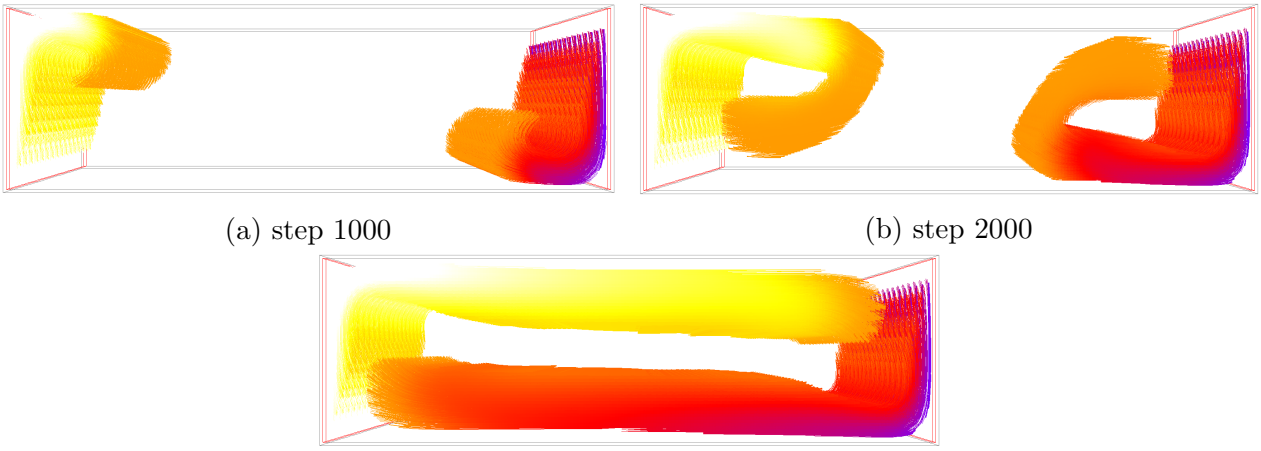


(a) step 500 (b) step 2000 (c) step 5000

Figure 18: Lid-driven cavity simulation at different simulation steps for a box 128 x 128 x 128. The colors of the streamlines depict the velocity. Many thanks to Moritz Lehmann for providing the pictures.

To visualize the drop setups in figure 21 box sizes were chosen which were not possible with the single-GPU implementation so far due to memory limitations. They were rendered with Paraview. Here you can see why a multi-GPU LBM is necessary: even with these domain sizes small droplets stick to the faces of the simulation box. The ring-shaped waves also break at the faces of the simulation box after approx. 7000 steps. A further enlargement of the lattice would reduce these unwanted effects.

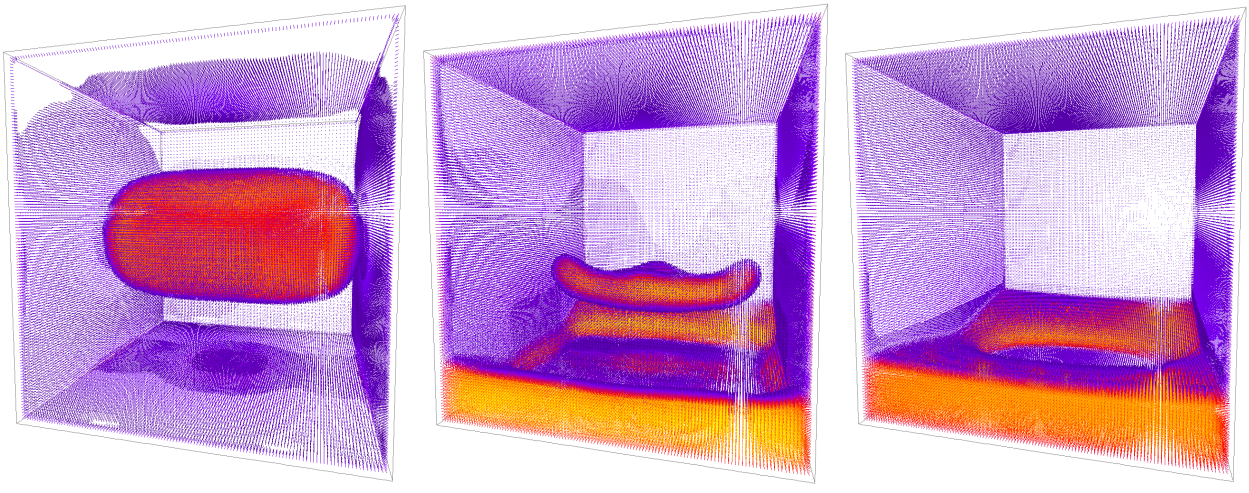
The different collision operators had not to be validated, as they do not interfere with data communication. For all validations and performance measurements, the TRT collision operator



(a) step 1000 (b) step 2000

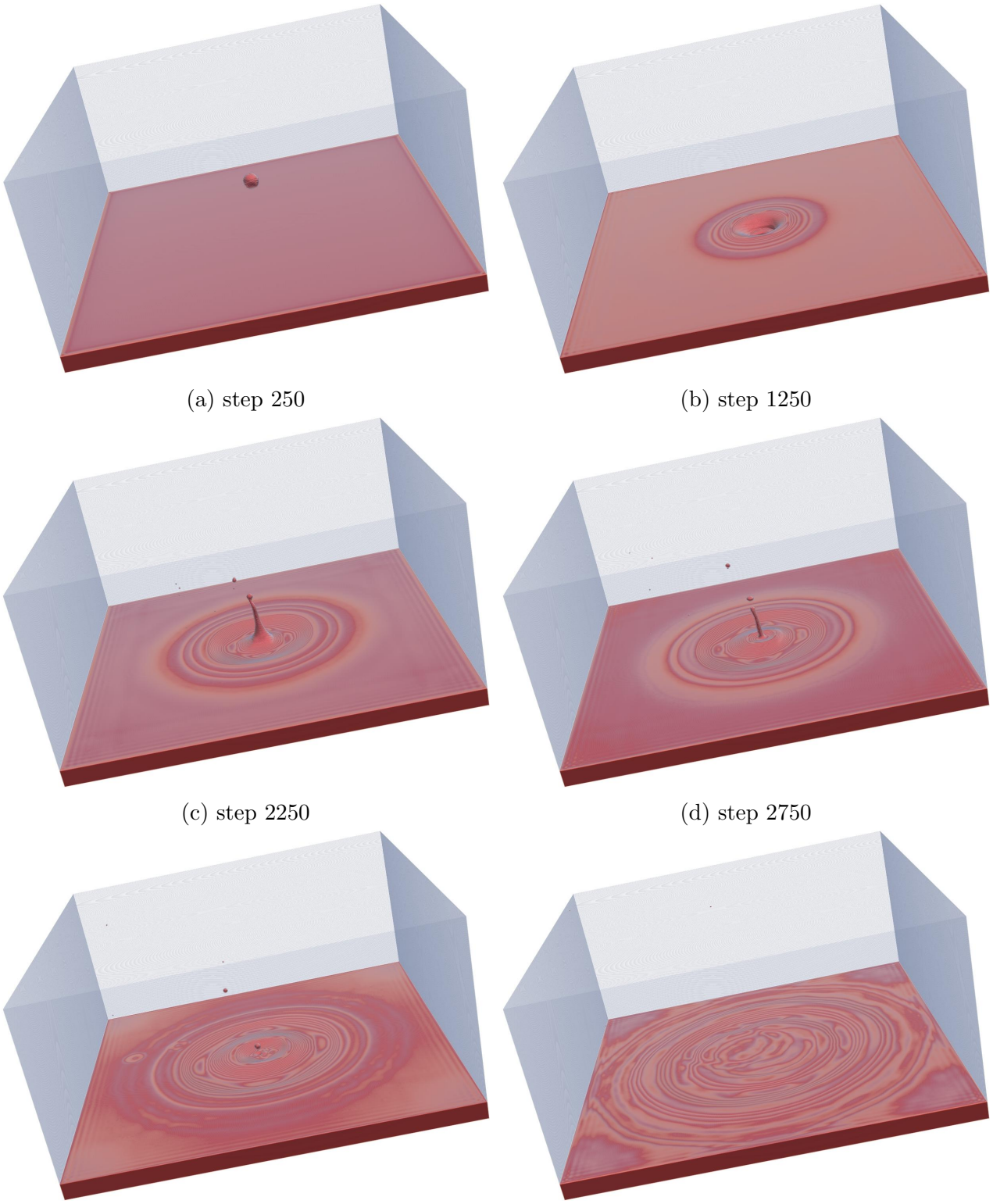
(c) step 5000

Figure 19: Setup for testing the temperature module. The boundary conditions induce a convection flow. Sizes of the lattice: 32 x 196 x 60. The colors of the streamlines depict the temperature. Many thanks to Moritz Lehmann for providing the pictures.



(a) step 100 (b) step 600 (c) step 2000

Figure 20: Setup for testing the shanchen module. The condensing process of a cylindric drop can be observed. Sizes of the lattice: 128 x 124 x 128. The colors depict the density. Many thanks to Moritz Lehmann for providing the pictures.



(a) step 250

(b) step 1250

(c) step 2250

(d) step 2750

(e) step 3250

(f) step 7000

Figure 21: Setup for testing the surface module. When the drop hits the surface of the fluid, it generates a back-jet. Sizes of the lattice: 912 x 912 x 448. The colors depict the fill level $\varphi \in [0, 1]$ of the lattice nodes.

was used.

5 Performance measurements

For LBM-simulations performance is typically computed by dividing the number of nodes $n = s_x s_y s_z$ of a lattice by the time t one simulation step takes. If the resulting value is scaled by 10^{-6}, performance is given in units of *mega lattice updates per second* (MLUPs). First, time-resolved performance measurements were made. Figure 22 shows two single-GPU measurements of different setups. Since the computational effort in the drop-setup depends essentially on the size of the fluid surface, the performance increases with running simulation time, while the water surface calms down after impact of the drop. Oscillations in the performance can also be observed in the long-term course. For the future performance measurements, the *benchmark setup* was used instead. The benchmark setup simply consists of a domain completely filled with fluid, with no external forces acting on it and for which no additional modules are activated. As a result, the performance remains constant over wide ranges. Only after about 75 s it decreases, because the clock frequency of the GPU is slowed down due to *thermal throttling*.

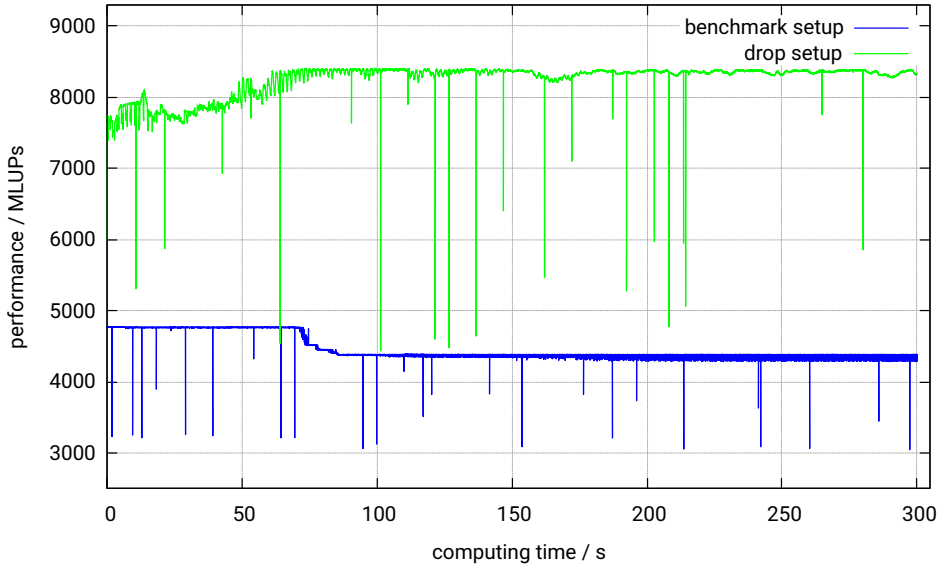
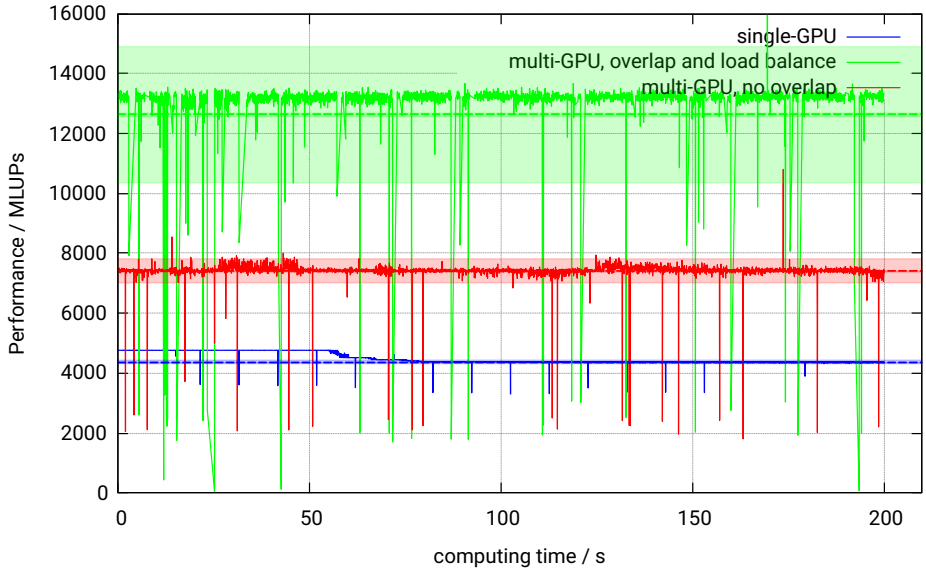


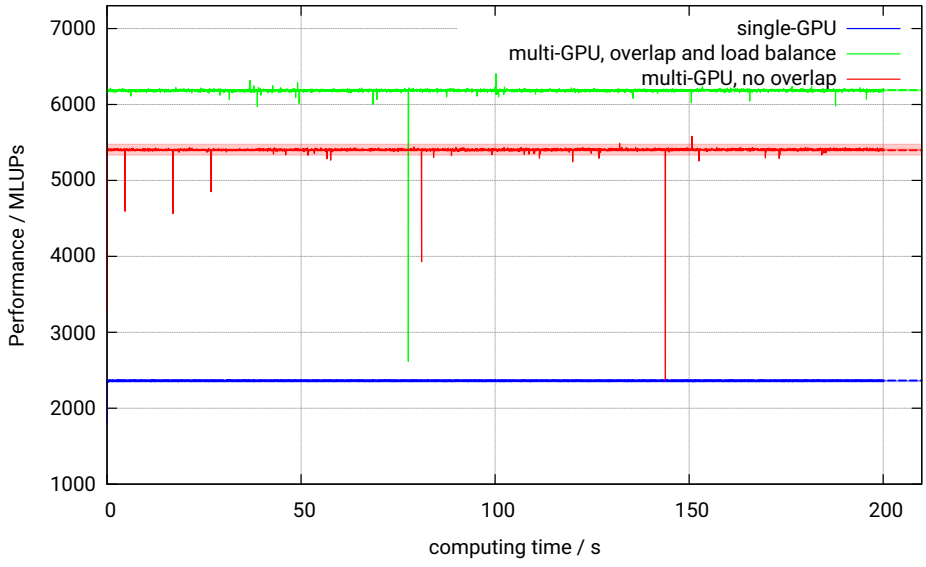
Figure 22: single-GPU time resolved performance characteristics: benchmark vs. surface setup (*SMAUG*), sizes: 448 x 448 x 442, GPU arrangement: 1 x 2 x 2.

In figure 23a the time resolved performance of the benchmark setup of the single-GPU version is compared to two types of the multi-GPU versions (one with compute-transfer overlap and load balancing, one without these optimizations). All were measured on the cluster *SMAUG*. In addition to the eye-catching performance increase due to the multi-GPU version and the optimizations, the density of peak-like performance drops increases along with the complexity of the transfer protocol. The same measurements are shown on the cluster *btrzx4* (due to a more limited memory size for a smaller lattice). Here, performance drops are much rarer, and there is no initial decrease in the clock rate. The phenomenological differences could be attributed either to vendor-specific hardware properties or different driver characteristics.

For all further measurements, the performance is determined as the mean value over the time interval 100 - 200 s (drawn as a dashed line in figure 23). Let d be the standard deviation of the performance over said interval and m the number of contained values. The error of the mean value is then calculated by $d/\sqrt{m}$.



(a) *SMAUG*, lattice sizes: 448 x 448 x 444.



(b) *btrzx4*, lattice sizes: 256 x 256 x 252.

Figure 23: multi-GPU vs. single-GPU time resolved performance characteristics. Arrangement multi-GPU: 1 x 2 x 2. The dashed line represents the mean performance over the time interval 100 - 200 s, the transparent areas the corresponding standard deviation.

Figure 24 shows the performance for different, approximately cubic box sizes and different work-groups sizes (≡ TBS). It is noticeable that with the exception of TBS 32, all TBS are approximately equivalent in terms of their performance. For large lattices, TBS 256 seems to be advantageous in the multi-GPU measurement.

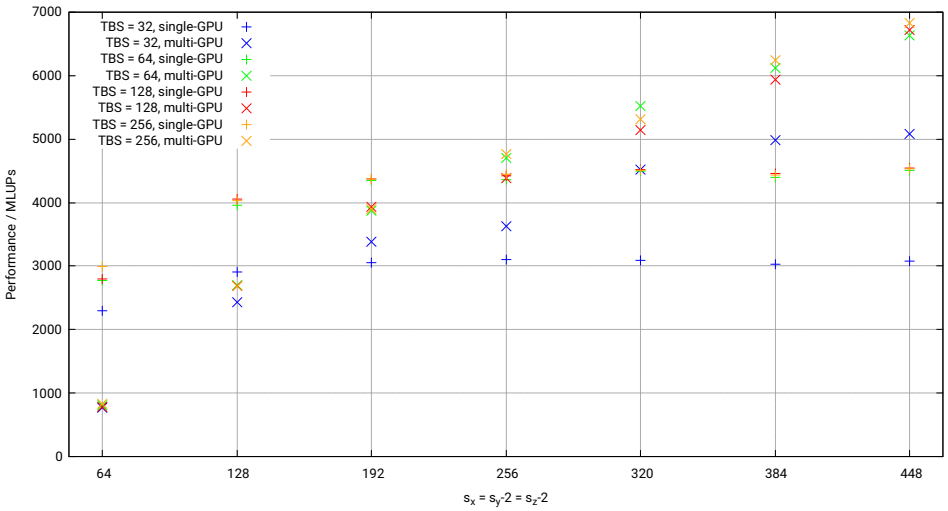


Figure 24: performance characteristics: tread-block size. GPU arrangement for multi-LBM: 1 x 2 x 2, no compute-transfer overlap, cluster: *SMAUG*.

In figure 25 you can see the performance effect of host pointer. For the cluster *SMAUG* there is no performance difference between the mapping routine and the read-write routine, if host pointer is active. This was already observed in figure 4. For the cluster *btrzx4*, host pointer apparently has only an advantage in combination with buffer mapping. The differences may be due to the vendor specific compiler on GPU-side. Therefore, buffer mapping and host pointer are now the default settings of the multi-GPU version of FluidX3D.

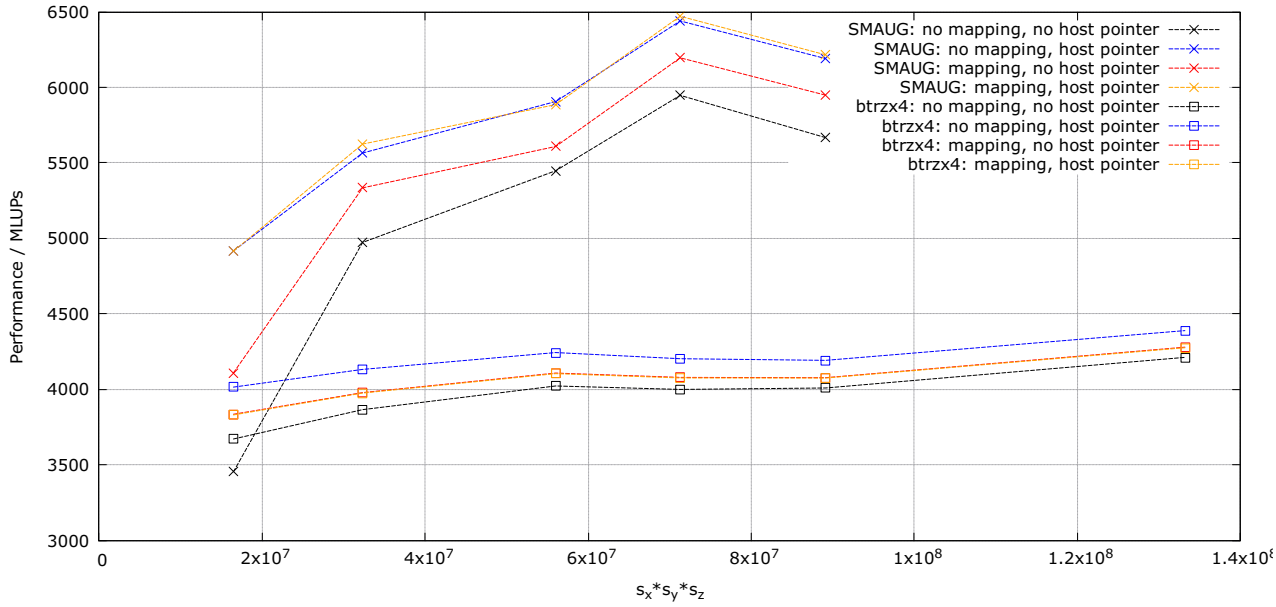


Figure 25: Performance characteristics: buffer mapping and host pointer. GPU arrangement: 1 x 1 x 2, cubic boxes, compute-transfer overlap and load balancing activated. For *btrzx4*, the red and orange graph are nearly overlapping.

Figure 26 shows the performance improvement gained by compute-transfer overlap and load balancing. The improvement is bigger for the cluster *SMAUG* - the MPI communication seems

here more of an bottleneck than on the cluster *btrzx4*. On *SMAUG*, determining the compute groups by the mod-distribution caused a performance loss. On *btrzx4* this method yields equivalent perfomance to determining them via the z-position. In order to meet both systems, the latter method is preferred in FluidX3D.

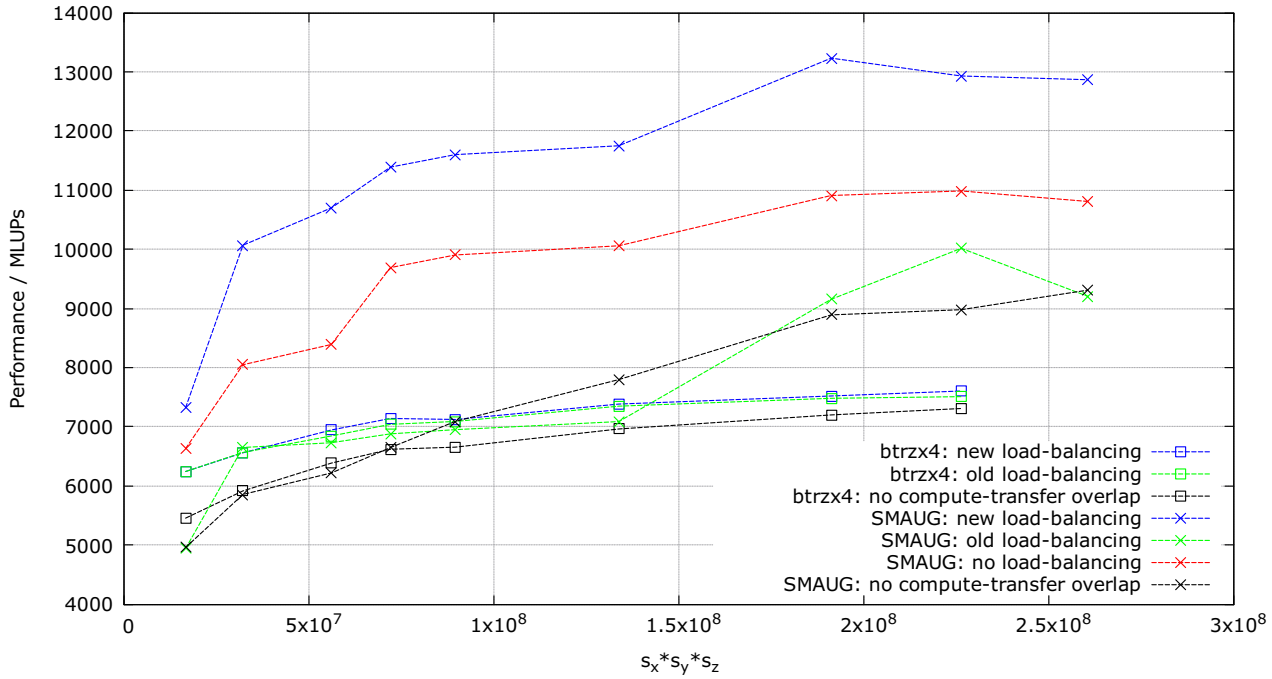
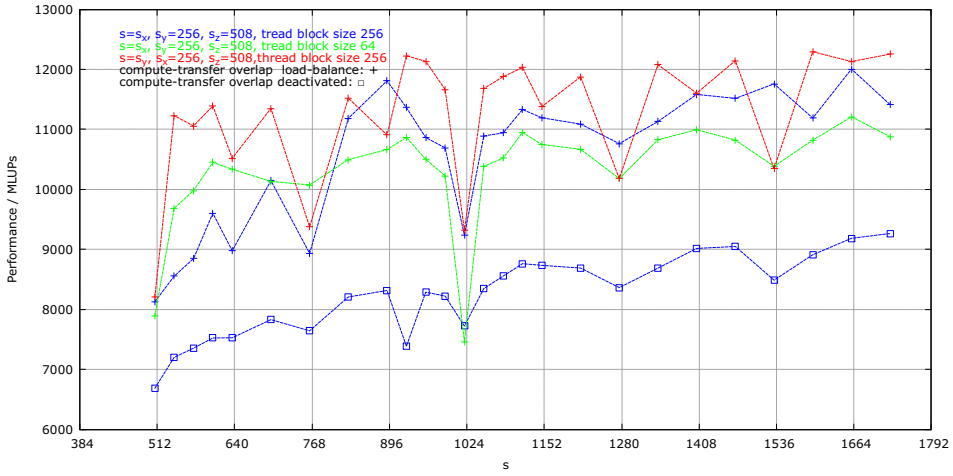


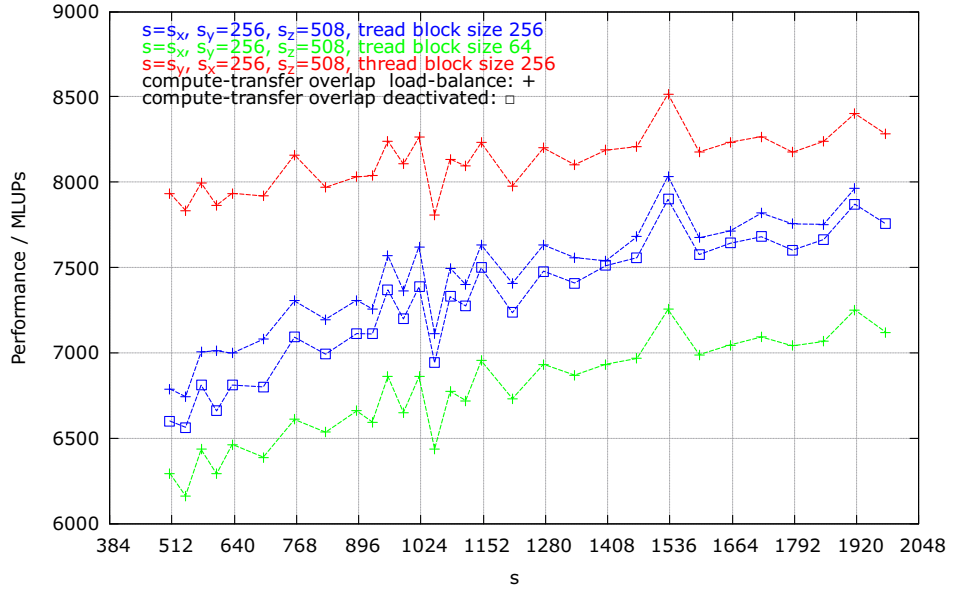
Figure 26: Performance characteristics: old (mod-distribution) vs. new (z-distribution) load-balancing. GPU arrangement: 1 x 2 x 2, cubic boxes.

The next step is to investigate how large the performance gain from load-balancing is for different constellations. In figure 27a $s_y = 256$ and $s_z = 508$ was kept constant, while s_x was varied. The box is being split along the x- and z-axis, while along the $y-$axis there is no splitting. Only for the red graph, the roles of x and y are changed. Overall it can be observed that the load-balancing does not work as well when splitting along the x-axis as when splitting along the y-axis. The reason for this is - as explained above - that x is the linear index. For $s_x \in \mathbb{S} = \{508, 636, 764, 1020\}$, TBS $= 256$ and a division along the x-axis the inner nodes are all assigned to compute group 2 (for example for $s_x = 1020$ follows: $s_{h,x} = 512 = 2 \cdot 256$). In fact, in figure 27a there are performance drops at the corresponding places (see blue graph with symbol +). As expected, for TBS=64 these are not visible at the positions $s_x = 636$ and $s_x = 1020$ (see green graph with symbol +). However, the fact that there are performance drops at $s_x = 508$ and $s_x = 1020$ with TBS=64 cannot be explained by the load-balancing, nor that there are also drops at $s \in \mathbb{S}$ if the domain is not split at all along the $x-$axis (see red graph with symbol +).

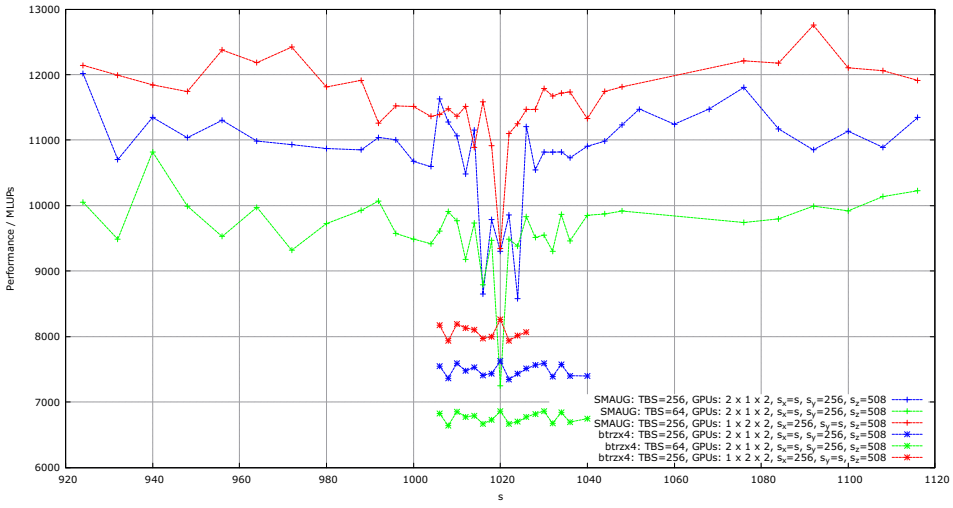
One possible explanation is that for certain combinations of TBS and (sub-)lattice size there is a loss in performance. Figure 28 confirms this, for this effect also occurs for the single-GPU version. Since the effect cannot be seen in figure 27b for the cluster *btrzx4*, it is GPU specific. However, this would also mean that performance losses due to unfavorable distribution of the inner nodes in figure 27a could not be resolved. Nevertheless, it becomes clear that a distribu-



(a)



(b)



(c)

Figure 27: multi-GPU performance characteristics: load-balancing. Two sizes $s \in \{s_x, s_y, s_z\}$ have been hold fixed for each measurement series, while the third one has been varied.

tion along the axis of the linear index is to be avoided in principle. In addition, there is now a greater difference in performance between the TBS than before in figure 25. One reason for this could be that compute-transfer overlap is now activated - since the performance is now closer to the optimum, the overhead caused by a smaller TBS is more significant. It is therefore recommended to use TBS 256 if possible.

Figure 28: Performance characteristics (*SMAUG*): regular performance drops. For the lattice, $s_x = s_y = 448$ was chosen, while s_z was varied. The drops occur in both graphs for the same (sub-)lattice size, if the halo is taken into account. For the multi-GPU measurement host pointer, buffer mapping, compute-transfer overlap and load-balancing were activated.

Now the behaviour of FluidX3D in weak-scaling and strong-scaling shall be investigated. For the weak-scaling a cubic simulation box is examined in single-GPU mode, while in multi-GPU mode n (with n being the number of used GPUs) of such cubic boxes are lined up along the z-axis. In table 10 the performance for large lattices is shown as well as the efficiency compared to the theoretical optimum. With 4 GPUs, an efficiency of 81.8 % could be reached on *SMAUG*, for *btrzx4* an efficiency of 94.8 %.

In strong-scaling, always cubic lattices are used and subdivided according to the number of GPUs. Figure 30a shows that a performance increase due to the use of multiple GPUs only occurs when the domain sizes are chosen big enough. In the measurement series this is for the first time the case with a domain of 96 x 92 x 96 for *btrzx4* and with a domain of 256 x 252 x 258 for *SMAUG*. Table 11 can explain the big difference between the clusters: The cluster *btrzx4* does the MPI transfer much faster. In the example, on both clusters the MPI-communication is overlapped - but for smaller lattices (where the *stream_collide*'s duration is much shorter) this is not the case for *SMAUG*. Regarding the transfer speed of PCIe both clusters are nearly

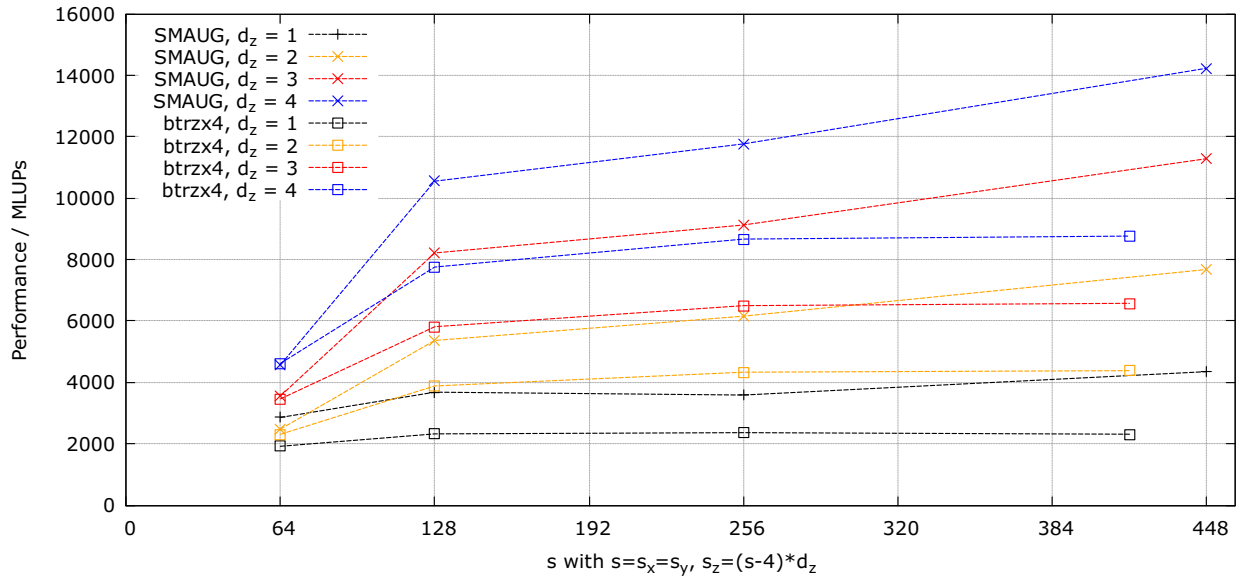


Figure 29: Weak scaling. All possible optimizations for the multi-GPU version are activated.

Table 10: Performance and efficiency for the weak scalability test. n denotes the number of GPUs used. For the cluster *btrzx4* the box size 416 x 416 x 412 was used, for the cluster *SMAUG* the box size 448 x 448 x 444. Statistical errors are below 0.1% of the respective measured value.

n	Perf. / MLUPs (*SMAUG*)	Eff. (*SMAUG*)	Perf. / MLUPs (*btrzx4*)	Eff. (*btrzx4*)
1	4351	1.0000	2311	1.0000
2	7677	0.8822	4383	0.9483
3	11284	0.8644	6574	0.9482
4	14229	0.8175	8764	0.9481

equivalent: the duration of all processes involving PCIe (4x *setget_kernel*[14], 2x transfer Device → Host, 2x transfer Host → Device) is 3.01 x 10^{-3} s for *SMAUG* and 2.81 x 10^{-3} s for *btrzx4*. The high standard deviation for all activities on *SMAUG* is conspicuous. A glance at the data (see 31 for *setget_kernel(4)* and *y-transfer MPI*) shows two reasons for this: on the one hand, there are occasional kernel durations of up to 0.6 seconds, which are associated with the performance drops from figure 23. As already suspected above, they are probably related to hardware-specific properties or special driver characteristics. They occur for both *setget_kernel* and *stream_collide*. For table 11 these are a total of 20 failures of 0.1 s or longer. On the other hand, the duration of the MPI communication fluctuates strongly: It oscillates between 0.3 x 10^{-3} and 0.9 x 10^{-3} seconds. Additionally, the MPI communication reaches durations of up to 0.1 seconds for several times. For the transfer in y-direction, in 838 out of 9000 cases MPI communication is not completely overlapped by computation, for the transfer in z-direction in 7837 cases. This is in strong contrast to 6a, where the MPI transfer on *SMAUG* was fast and stable. The reason for these discrepancies could not yet be found.

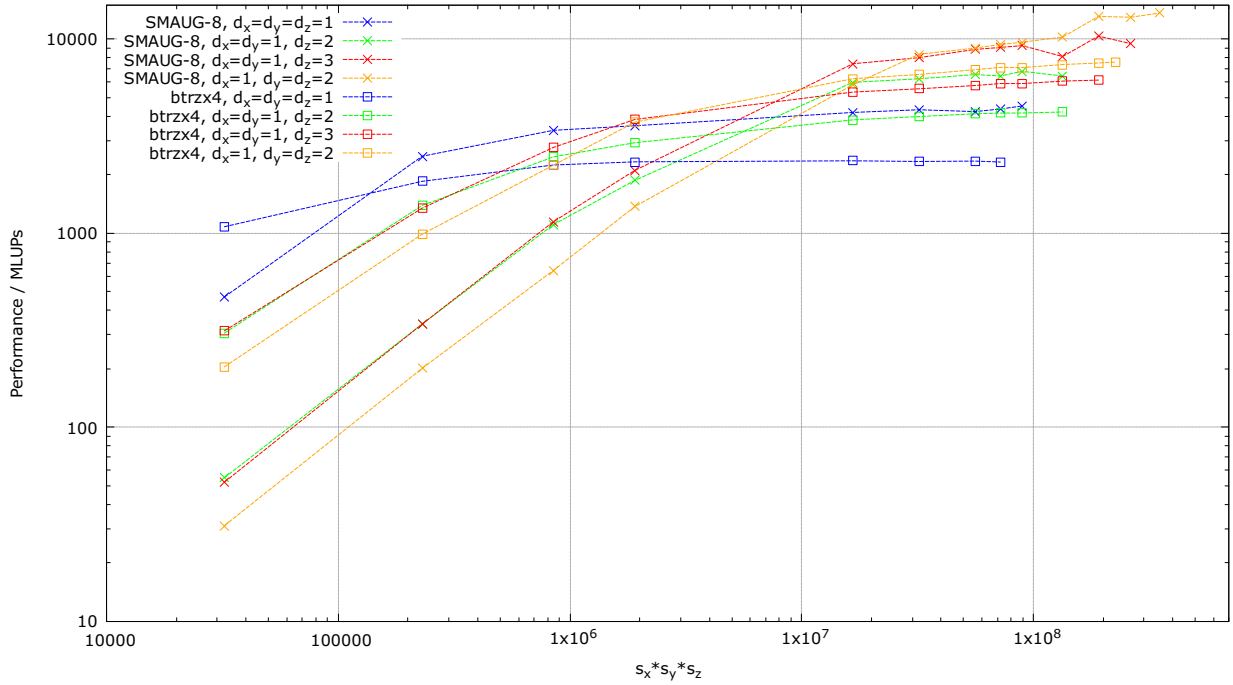
Table 11: Execution times of the simulations sub-steps (mean over 9000 steps). A lattice 416 x 828 x 828 with sub-domains 1 x 2 x 2 was used. All optimizations were enabled. Note how the MPI Communication is overlapped completely.

sub-step	activity	time / s (*SMAUG*)	stddev / s (*SMAUG*)	time / s (*btrzx4*)	stddev / s (*btrzx4*)
1	*setget_kernel(4)*	9.18 x 10^{-4}	8.96 x 10^{-3}	1.91 x 10^{-4}	1.44 x 10^{-6}
2	y-transf. Device → Host	1.25 x 10^{-5}	4.61 x 10^{-6}	5.43 x 10^{-4}	2.12 x 10^{-5}
3a	y-transf. MPI	5.17 x 10^{-3}	6.13 x 10^{-3}	1.17 x 10^{-3}	6.17 x 10^{-5}
3b	*stream_collide(3)*	9.89 x 10^{-3}	1.52 x 10^{-2}	1.58 x 10^{-2}	5.81 x 10^{-5}
4	y-transf. Host → Device	1.30 x 10^{-5}	4.40 x 10^{-5}	6.04 x 10^{-4}	1.78 x 10^{-6}
5	*setget_kernel(1)*	7.23 x 10^{-4}	5.93 x 10^{-4}	2.27 x 10^{-4}	1.78 x 10^{-6}
6	*setget_kernel(5)*	6.51 x 10^{-4}	1.48 x 10^{-4}	4.95 x 10^{-5}	1.23 x 10^{-6}
7	z-transf. Device → Host	1.16 x 10^{-5}	2.46 x 10^{-6}	5.42 x 10^{-4}	2.01 x 10^{-6}
8a	z-transf. MPI	9.85 x 10^{-3}	6.22 x 10^{-3}	1.16 x 10^{-3}	3.70 x 10^{-5}
8b	*stream_collide(1)*	1.09 x 10^{-2}	1.31 x 10^{-2}	1.59 x 10^{-2}	4.42 x 10^{-5}
9	z-transf. Host → Device	7.73 x 10^{-6}	1.41 x 10^{-5}	6.04 x 10^{-4}	1.89 x 10^{-6}
10	*setget_kernel(2)*	6.78 x 10^{-4}	3.52 x 10^{-3}	5.09 x 10^{-5}	1.12 x 10^{-6}
11	*stream_collide(2)*	7.96 x 10^{-4}	3.35 x 10^{-4}	8.80 x 10^{-4}	1.67 x 10^{-6}

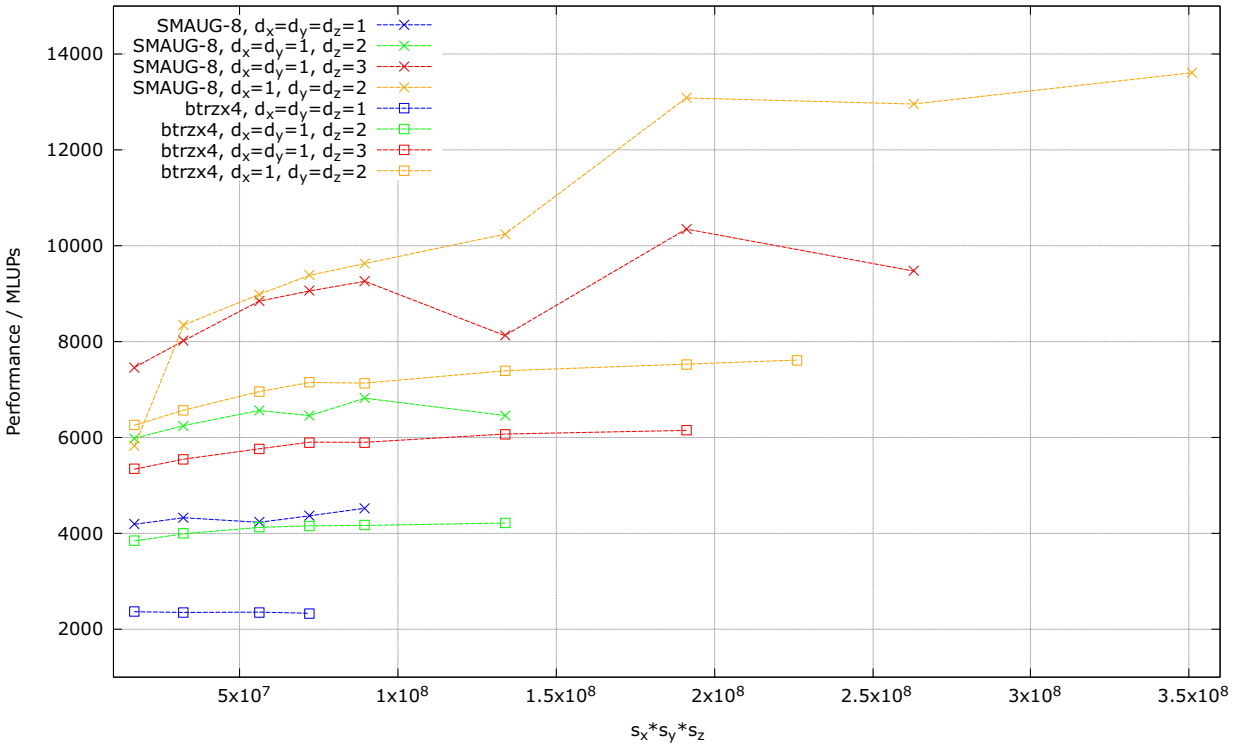
Table 10 shows the performance and efficiency for the largest lattices that were compatible with the memory limitations of the single-version. With 4 GPUs, an efficiency of 53.3 % could be reached on *SMAUG*, for *btrzx4* an efficiency of 76.8 %. On *SMAUG* the performance increases drastically in ranges where no single-GPU version can be used for comparison. The maximum performance for a cubic lattice 704 x 704 x 708 is here (13600 ± 70) MLUPs.

[14]When host pointer is active, writing to global memory may involve the PCIe.

Figure 30: Strong scaling. Cubic boxes are used. All possible optimizations for the multi-GPU version are activated.



(a) Double-logarithmic plot over a wide measuring range.



(b) Zoom to the range where the multi-GPU version is superior to the single-GPU version. Some series of measurements are stopping at smaller box sizes than others due to memory limitations.

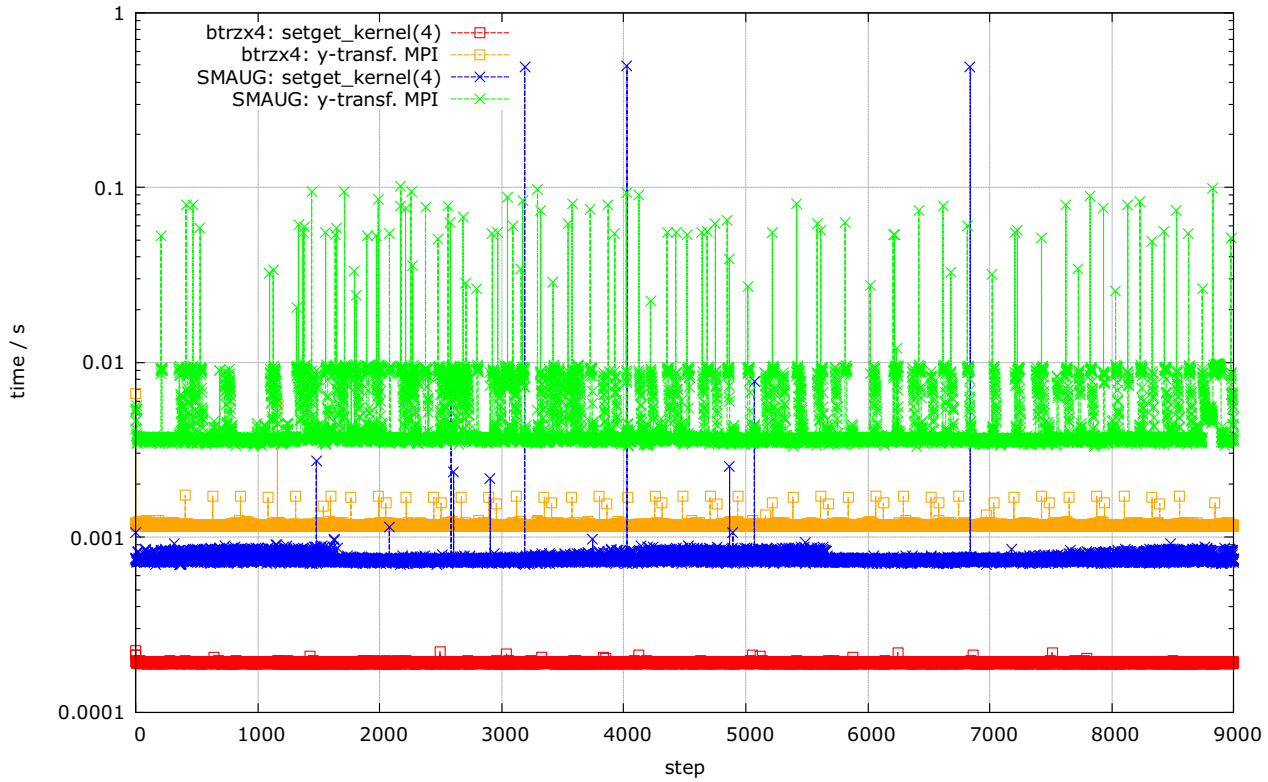


Figure 31: Selected activities from table 11 are shown time-resolved. On *btrzx4*, the performance for both *setget_kernel(4)* and *y-transfer MPI* is nearly constant. On *SMAUG*, on the other hand, the bandwidth of the MPI-communication oscillates between 0.3 x 10^{-3} and 0.9 x 10^{-3} seconds. Additionally, there are several drastic performance drops both in *setget_kernel(4)* and *y-transfer MPI*.

Table 12: Performance and efficiency of selected measuring points in the strong scalability test. For the cluster *btrzx4* the box sizes 416 x 412 x 420 was used, for the cluster *SMAUG* the box sizes 448 x 444 x 450. Statistical errors are less than 0.1% of the respective measured value.

n	Perf. / MLUPs (*SMAUG*)	Eff. (*SMAUG*)	Perf. / MLUPs (*btrzx4*)	Perf. (*btrzx4*)
1	4513	1.0000	2322	1.0000
2	6810	0.7544	4147	0.8929
3	9250	0.6832	5892	0.8457
4	9618	0.5328	7137	0.7683

6 Conclusion

The multi-GPU implementation presented in this thesis was precisely tailored to the requirements of GPUs and MPI: via host pointer it uses spezialized buffers for a fast transfer between host and device; the memory layout SoA ensures coalesced data access when writing to / reading from the transfer buffers; by "template"-kernels unnecessary readout from global memory can be saved; *setget_kernel* parallelizes over layers instead of sub-domains, so viewer thread blocks must be launched; data to be transferred is combined into a single buffer for optimal bandwidth; branching is avoided where possible; the MPI-communication is overlapped by computations; special characteristics of the used GPUs are taken into account (z-distribution instead of mod-distribution; host pointer in combination with the read-write routine); the transfer scheme needs 6 MPI communications instead of 26 in the naive approach. This pays off through a nearly linear weak-scaling on *btrzx4* and a performance up to 13600 MLUPs on *SMAUG*. Two possible approaches to a further increase in performance are possible: on the one hand, an overlap of PCIe communication between Host and Device would be desirable and theoretically possible; on the other hand, a faster and more stable MPI communication on the *SMAUG* cluster should be aimed at.

Besides its high-performance properties this multi-GPU implementation proves its flexibility: it works with several extensions; any arrangement of sub-domains is possible; it is possible to send objects over MPI; only a single reference must be set in the *Lattice*-class to make new data fields available to all MPI slaves. This makes it easy to use for future development.

7 References

[1] Christian Obrecht et al. "The TheLMA project: Multi-GPU implementation of the lattice Boltzmann method". In: *The International Journal of High Performance Computing Applications* 25.3 (2011), pp. 295–303.

[2] Timm Krüger et al. *The Lattice Boltzmann Method*. Springer, 2017.

[3] Tianluo Chen et al. "Multi-GPU solution to the lattice Boltzmann method: An application in multiscale digital rock simulation for shale formation". In: *Concurrency and Computation: Practice and Experience* 30.19 (2018), e4530.

[4] Michal Januszewski and Marcin Kostur. "Sailfish: A flexible multi-GPU implementation of the lattice Boltzmann method". In: *Computer Physics Communications* 185.9 (2014), pp. 2350–2368.

[5] Christian Obrecht et al. "Multi-GPU implementation of the lattice Boltzmann method". In: *Computers & Mathematics with Applications* 65.2 (2013), pp. 252–261.

[6] Changsheng Huang et al. "Multi-GPU based Lattice Boltzmann method for hemodynamic simulation in patient-specific cerebral aneurysm". In: *Communications in Computational Physics* 17.4 (2015), pp. 960–974.

[7] Changsheng Huang et al. "Implementation of multi-GPU based lattice Boltzmann method for flow through porous media". In: *Advances in Applied Mathematics and Mechanics* 7.1 (2015), pp. 1–12.

[8] Julien Duchateau et al. "A progressive mesh method for physical simulations using lattice boltzmann method on single-node multi-gpu architectures". In: *International Journal of Distributed Parallel Systems* 6.5 (2015).

[9] Aaftab Munshi et al. *OpenCL programming guide*. Pearson Education, 2011.

[10] Nicolas Delbosc et al. "Optimized implementation of the Lattice Boltzmann Method on a graphics processing unit towards real-time fluid simulation". In: *Computers & Mathematics with Applications* 67.2 (2014), pp. 462–475.

[11] The Khronos Group Inc. *clCreateBuffer*. Accessed: 16/10/2019. 2009. URL: `https://www.khronos.org/registry/OpenCL/sdk/1.0/docs/man/xhtml/clCreateBuffer.html`.

[12] *How OpenCL memory transfer functions work*. Accessed: 16/10/2019. 2019. URL: `https://stackoverflow.com/questions/57854782/how-opencl-memory-transfer-functions-work/57857059#57857059`.

[13] Intel Corporation. *Intel MPI Benchmarks User Guide*. Accessed: 25/10/2019. 2018. URL: `https://software.intel.com/en-us/imb-user-guide`.

[14] Ingo Schelter. *btrzx4 (2018)*. Accessed: 11/09/2019. 2018. URL: `https://www.bzhpc.uni-bayreuth.de/de/keylab/Cluster/btrzx4_page/index.html`.

[15] Markus Wittmann. "Hardware-effiziente, hochparallele Implementierungen von Lattice-Boltzmann-Verfahren für komplexe Geometrien". In: (2016).

[16] Mark J Mawson and Alistair J Revell. "Memory transfer optimization for a lattice Boltzmann solver on Kepler architecture nVidia GPUs". In: *Computer Physics Communications* 185.10 (2014), pp. 2566–2574.

[17] Joshi Nikhil. *S8837 OpenCL at Nvidia - recent improvement and plans*. Accessed: 07/09/2019. 2018. URL: http://on-demand.gputechconf.com/gtc/2018/presentation/s8837-opencl-nvidia-recent-improvements-future-plans.pdf.

[18] Department of Computer Science, University of Virginia. *Choosing Between Pinned and Non-Pinned Memory*. Accessed: 07/09/2019. URL: https://www.cs.virginia.edu/~mwb7w/cuda_support/pinned_tradeoff.html.

[19] Inc. Advanced Micro Devices. *AMD APP SDK - OpenCL Optimization Guide*. Accessed: 07/09/2019. 2015. URL: http://developer.amd.com/wordpress/media/2013/12/AMD_OpenCL_Programming_Optimization_Guide2.pdf.

8 Acknowledgements

Above all, I would like to thank Prof. Dr. Stephan Gekle for his help and for taking so much of his time to supervise this thesis. He managed to be demanding and at the same time to create an atmosphere of openness and trust in which all ideas and problems could be discussed. I would also like to thank my supervisor Moritz Lehmann: he has transferred his enthusiasm and vision for this project to me and supported me actively and creatively, whenever questions or difficulties occurred. He let me willingly make changes to his highly optimized code where it was necessary for me and has forgiven me for many a blemish, merge conflict or even bug during the development of FluidX3D. I could profit a lot from his detailed knowledge about hardware and his abilities as a "living compiler". I would also like to thank the whole team "Biofluid Simulation and Modeling - Theoretical Physics VI", especially my office colleagues Sanwardhini, Simon, Konstantin, Lukas and Mars: for a relaxed working atmosphere, support with small problems, some distractions over a cup of coffee and of course for the legendary "Pizza, Beer and Games" parties. You are very nice people. I would also like to thank Markus Hilt: he ordered the Radeon VII GPUs, installed several drivers and tried out two new operating systems for the cluster *SMAUG*. He often had spontaneous time for me, and after spending a few minutes in his office, my IT problems were always solved. Furthermore I'd like to thank my whole family: My grandmother Trudi was a great help to me in the first semesters without a washing machine. Now she still supports me with conversations, delicious food and prayers. Great thanks goes also to my father and mother and my three sisters for their support. Without you I would not have had the courage to quit my teacher training for Catholic Religion and Latin and instead follow this fascinating path of Physics. My life would be very poor without you.

A Code examples from FluidX3D

Some slightly adapted code examples from FluidX3Dare shown here. Note that only the case D3Q19 without extensions and without C++17-compatibility is presented. The single- and multi-GPU versions are shown side by side (all multi-GPU specific code is encapsulated in the define MULTIPLE_DEVICES).

```
template <class T>
class Tensor3;
template <class T>
class LBM_Data3 : public Tensor3<T> { // mark Tensor3-Objects containing lbm fields by using a different class
  using Tensor3<T>::Tensor3; // inherit constructors
};
class Lattice;

#ifdef MULTIPLE_DEVICES
template<class... Ts> struct overloaded : Ts... { using Ts::operator()...; };
template<class... Ts> overloaded(Ts...) -> overloaded<Ts...>;
template <typename T>
class Variant_Container {
public:
  const vector<T> references;
  Variant_Container(initializer_list<T> l) : references(l) {}
  Variant_Container(const Variant_Container& t) { *this = t; }; // copy-constructor has no effect so classes with Variant_Container as members are able to use default copy-constructor
  const Variant_Container& operator=(const Variant_Container& t) const { return *this; }; // operator= has no effect for similar reasons
  void visit(const bool send, const uint mpi_id) const; // send / recieve class over MPI
  void visit_LBM_Data3(const bool set, Lattice& lat, uint xfirst, uint xlast, uint yfirst, uint ylast, uint zfirst, uint zlast) const; // set sub-subdomai lat into domain *this / get copy lat containing only sub-domain of *this
};
#endif // MULTIPLE_DEVICES

template <class T>
class Tensor3 {
protected:
  uint sizex, sizey, sizez, sized; // sizes of Tensor3: sd denotes the data depth at one Tensor3 position
  uint length; // total number of data entries
  uint size; // size = sx*sy*sz
  vector<T> vec; // contains data
#ifdef MULTIPLE_DEVICES
  Variant_Container<std::variant<uint*, vector<T>*>> reference_collection { &sizex, &sizey, &sizez, &sized, &length, &size, &vec };
#endif // MULTIPLE_DEVICES
public:
  // contructors and destructor
  Tensor3() = default;
  Tensor3(uint sx, uint sy, uint sz, uint datadepth=1);
  Tensor3(const Tensor3& t) = default;
  ~Tensor3() = default;
  // operators
  Tensor3& operator=(const Tensor3& t) = default;
  T& operator[](uint i); // n = sx*(sy*z+y)+x + d*sx*sy*sz
  const T& operator[](uint i) const { return (const_cast<Tensor3<T>*>(this))->operator[](i); }; // const-version
  template <class U>
  friend ostream& operator<<(ostream& os, const Tensor3<U>& t);
  // get and set
  T* get_ptr() { return vec.data(); }; // returns the begin of the internally maintained 1D-array
  const T* get_ptr() const { return (const_cast<Tensor3<T>*>(this))->get_ptr(); }; // const-version
  uint sx() const { return sizex; };
  uint sy() const { return sizey; };
  uint sz() const { return sizez; };
  uint sd() const { return sized; };
  uint len() const { return length; }; // length == size*sized
  uint s() const { return size; };
  Tensor3<T>& resize(uint sx, uint sy, uint sz, uint datadepth=1);
  T& at(uint x, uint y, uint z, uint d=0); // alternative to operator[]
  const T& at(uint x, uint y, uint z, uint d=0) const { return (const_cast<Tensor3<T>*>(this))->at(x,y,z,d); }; // const-version
  T& at(uint n, uint d=0); // ==at(x, 0, 0, d)
  const T& at(uint n, uint d=0) const { return (const_cast<Tensor3<T>*>(this))->at(n,d); }; // const-version
  void copy_into(Tensor3<T> & t, uint xfirst, uint xlast, uint yfirst, uint ylast, uint zfirst, uint zlast);
  void set_values(const T* arr, uint xfirst, uint xlast, uint yfirst, uint ylast, uint zfirst, uint zlast);
  void set_values(const T* arr); // set all data entries of the Tensor3-Object via an existing array
  void set_values(const T* arr, uint d); // set all data entries of depth d via an existing array
  void set_values(const T& v); // set all data entries of the Tensor3-Object to the same value
  void set_values(const T& v, uint d); // set all data entries of depth d to the same value
  void write_vtk(const string path);
  void write_dat(const string path);
#ifdef MULTIPLE_DEVICES
  void mpi_send(const uint mpi_id_to);
  void mpi_recv(const uint mpi_id_from);
#endif // MULTIPLE_DEVICES
};

class Lattice { // class containing all lattice parameters and some lbm data fields
private:
  void test_box_sizes(); // test, if each simulation box is divisible by THREAD_BLOCK_SIZE
  void fill_positions(); // fills device_positions
  void fill_offsets(); // fills box_offsets
```

```
79  #ifdef MULTIPLE_DEVICES
80    void get_box(uint mpi_id, Lattice& lat); // fill lat with parameters and data for the simulation box corresponding to its
          ↪ mpi_id
81    void set_box(uint mpi_id, Lattice& lat); // fill the part of the simulation box of (*this) corresponding to mpi_id with data
          ↪  from lat
82    template<class T>
83    friend class Variant_Container;
84    Variant_Container<std::variant<uint*, bool*, float*, Tensor3<uint>*, Tensor3<uchar>*, Tensor3<float>*, Tensor3<short>*,
          ↪ LBM_Data3<uint>*, LBM_Data3<uchar>*, LBM_Data3<float>*, LBM_Data3<short>*>> reference_collection {
85      &step_count,
86      &rho, &u, &flags,
87      &config.device_order, &config.box_sizes, &config.device_positions, &config.box_offsets,
88      &config.size_x, &config.size_y, &config.size_z, &config.point_number,
89      &config.mem_sx, &config.mem_sy, &config.mem_sz, &config.mem_number,
90      &config.total_x, &config.total_y, &config.total_z, &config.total_point_number,
91      &config.devices_x, &config.devices_y, &config.devices_z,
92      &config.tau, &config.w, &config.viscosity, &config.Re, &config.Fx, &config.Fy, &config.Fz, &config.set
93    };
94  #endif // MULTIPLE_DEVICES
95    struct Config {
96      Tensor3<uint> device_order; // contains the corresponding mpi_id at each spatial position
97      Tensor3<uint> box_sizes; // contains the box size of each Device at the position of its mpi_id
98      Tensor3<uint> device_positions; // contains the spatial position of each Device at the position of its mpi_id
99      Tensor3<uint> box_offsets; // communicate between global and local lattice points: corresponding offsets at the position
            ↪ of its mpi_id
100     uint size_x, size_y, size_z, point_number; // local lattice size, number of lattice points of one Device
101     uint mem_sx, mem_sy, mem_sz, mem_number; // measures of the local lattice (including halo if any)
102     uint total_x, total_y, total_z, total_point_number; // global lattice size, number of lattice points of whole simulation
            ↪ box
103     uint devices_x, devices_y, devices_z; // denotes in how many sub-domains the domain is split along each axis
104     float tau, w, viscosity, Re; // further simulation parameters
105     float Fx, Fy, Fz; // constant volume force
106     uint set; // velocity set
107   };
108   struct Config config;
109 public:
110   uint step_count = 0;
111   LBM_Data3<float> rho, u; // density and velocity fields
112   LBM_Data3<uchar> flags; // flags for each lattice node (cl_bool is not supported as kernel argument), from 0 to 7: [wall
          ↪ boundary, equilibrium boundary, temperature boundary]
113   Lattice() = default;
114   Lattice(const uint sx, const uint sy, const uint sz, const float viscosity, const float Fx=0.0f, const float Fy=0.0f, const
          ↪ float Fz=0.0f, const uint dx=1, const uint dy=1, const uint dz=1);
115   Lattice(const Lattice& l) = default;
116   ~Lattice() = default;
117   Lattice& operator=(const Lattice& l) = default;
118   uint size_x() const { return config.size_x; }; uint mem_sx() const { return config.mem_sx; }; uint total_x() const { return
          ↪ config.total_x; }; // getter
119   uint size_y() const { return config.size_y; }; uint mem_sy() const { return config.mem_sy; }; uint total_y() const { return
          ↪ config.total_y; };
120   uint size_z() const { return config.size_z; }; uint mem_sz() const { return config.mem_sz; }; uint total_z() const { return
          ↪ config.total_z; };
121   uint devices_x() const { return config.devices_x; }; uint devices_y() const { return config.devices_y; }; uint devices_z()
          ↪ const { return config.devices_z; };
122   uint point_number() const { return config.point_number; }; uint mem_number() const { return config.mem_number; }; uint
          ↪ total_point_number() const { return config.total_point_number; };
123   float tau() const { return config.tau; }; float w() const { return config.w; }; float viscosity() const { return config.
          ↪ viscosity; }; float Re() const { return config.Re; };
124   float Fx() const { return config.Fx; }; float Fy() const { return config.Fy; }; float Fz() const { return config.Fz; }; uint
          ↪  set() const { return config.set; };
125   const Tensor3<uint>& device_order() const { return config.device_order; }; const Tensor3<uint>& box_sizes() const { return
          ↪ config.box_sizes; };
126   const Tensor3<uint>& device_positions() const { return config.device_positions; }; const Tensor3<uint>& box_offsets() const
          ↪ { return config.box_offsets; };
127 #ifdef MULTIPLE_DEVICES
128   bool halo_x() const { return config.devices_x>1; }; bool halo_y() const { return config.devices_y>1; }; bool halo_z() const
          ↪ { return config.devices_z>1; };
129   void mpi_distribute_lattices(); // MPI communication
130   void mpi_collect_lattices();
131   template<class T>
132   void mpi_collect(Tensor3<T> &t);
133   void mpi_send(const uint mpi_id_to);
134   void mpi_recv(const uint mpi_id_from);
135 #endif // MULTIPLE_DEVICES
136 };
137
138 // ######################################## implementation of class Variant_Container ########################################
139
140 #ifdef MULTIPLE_DEVICES
141 template <class T>
142 static void vector_MPI(const bool send, const uint mpi_id, vector<T>* _v) {
143   uint size;
144   if(send) {
145     size = (uint)_v->size();
146     if((long unsigned int)size*sizeof(T)>(long unsigned int)std::numeric_limits<int>::max()) {cout << "Maximum amount of data
          ↪ elements for MPI_Send exceeded." << endl; wait(); exit(-1); } // if needed, MPI_Send could be splited in smaller
          ↪ portions dynamically...
147     MPI_Send(&size, 1, MPI_UNSIGNED, mpi_id, 0, MPI_COMM_WORLD);
148     MPI_Send(_v->data(), size*sizeof(T), MPI_BYTE, mpi_id, 0, MPI_COMM_WORLD);
149   } else {
150     MPI_Recv(&size, 1, MPI_UNSIGNED, mpi_id, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
151     _v->resize(size);
152     T* arr = new T[size];
153     MPI_Recv(arr, size*sizeof(T), MPI_BYTE, mpi_id, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
154     _v->insert(_v->begin(), arr, arr+size);
155     delete[] arr;
156   }
```

```
157 }
158 template <class T>
159 void Variant_Container<T>::visit(const bool send, const uint mpi_id) const {
160   for(auto & ref : references) {
161     std::visit(overloaded {
162       [send,mpi_id]<class U>(U* _u) { if(send) MPI_Send(_u, sizeof(U), MPI_BYTE, mpi_id, 0, MPI_COMM_WORLD);
163       /*********************************/ else MPI_Recv(_u, sizeof(U), MPI_BYTE, mpi_id, 0, MPI_COMM_WORLD,  MPI_STATUS_IGNORE
          ↪ );  },
164       [send,mpi_id]<class U>(vector<U>* _v) { vector_MPI(send,mpi_id,_v); },
165       [send,mpi_id]<class U>(Tensor3<U>*   _t) { if(send) _t->mpi_send(mpi_id); else _t->mpi_recv(mpi_id); },
166       [send,mpi_id]<class U>(LBM_Data3<U>* _t) { if(send) _t->mpi_send(mpi_id); else _t->mpi_recv(mpi_id); }
167     }, ref);
168   }
169 }
170 template <class T>
171 void Variant_Container<T>::visit_LBM_Data3(const bool set, Lattice& lat, uint xfirst, uint xlast, uint yfirst, uint ylast,
      ↪ uint zfirst, uint zlast) const {
172   for(uint i=0; i<references.size(); i++) {
173     auto & ref = references[i];
174     std::visit(overloaded {
175       [i,set,&lat,xfirst,xlast,yfirst,ylast,zfirst,zlast]<class U>(LBM_Data3<U>* _t) {
176         if(set)  _t->set_values(get<LBM_Data3<U>*>(lat.reference_collection.references[i])->get_ptr(),xfirst,xlast,yfirst,
              ↪ ylast,zfirst,zlast);
177         else     _t->copy_into (get<LBM_Data3<U>*>(lat.reference_collection.references[i])->resize(lat.size_x(), lat.size_y(),
              ↪  lat.size_z(), _t->sd()),xfirst,xlast,yfirst,ylast,zfirst,zlast);
178       },
179       []<class U>(U _u) { return; }
180     }, ref);
181   }
182 }
183 #endif // MULTIPLE_DEVICES
184
185 // ######################################## implementation of class Tensor3 ########################################
186
187 #ifdef MULTIPLE_DEVICES
188 template <class T>
189 void Tensor3<T>::mpi_send(const uint mpi_id_to) {
190   reference_collection.visit(true, mpi_id_to);
191 }
192 template <class T>
193 void Tensor3<T>::mpi_recv(const uint mpi_id_from) {
194   reference_collection.visit(false, mpi_id_from);
195 }
196 #endif // MULTIPLE_DEVICES
```

Listing 1: Classes *Lattice* and *Tensor3* (only implementations of some of their methods are shown) and how the MPI send-recieve process via class *Variant_Container* works. Minor changes compared to the original file *lattice.hpp* have been made.

```
1  // ######################################## implementation of class Lattice ########################################
2
3  #ifdef MULTIPLE_DEVICES
4  void Lattice::mpi_send(const uint mpi_id_to) {
5    reference_collection.visit(true, mpi_id_to);
6  }
7  void Lattice::mpi_recv(const uint mpi_id_from) {
8    reference_collection.visit(false, mpi_id_from);
9  }
10 void Lattice::mpi_distribute_lattices() {
11   int mpi_id, n_mpi;
12   MPI_Comm_rank(MPI_COMM_WORLD, &mpi_id); // get my rank
13   MPI_Comm_size(MPI_COMM_WORLD, &n_mpi); // get the total number of processors
14   if(mpi_id==n_mpi-1) { // master
15     Lattice temp;
16     for(int i=0; i<n_mpi-1; i++) {
17       get_box(i, temp);
18       temp.mpi_send(i);
19     }
20   } else { // slaves
21     this->mpi_recv(n_mpi-1);
22   }
23 }
24 void Lattice::mpi_collect_lattices() {
25   int mpi_id, n_mpi;
26   MPI_Comm_rank(MPI_COMM_WORLD, &mpi_id); // get my rank
27   MPI_Comm_size(MPI_COMM_WORLD, &n_mpi); // get the total number of processors
28   if(mpi_id==n_mpi-1) { // master
29     Lattice temp;
30     for(int i=0; i<n_mpi-1; i++) {
31       temp.mpi_recv(i);
32       set_box(i,temp);
33     }
34   } else { // slaves
35     this->mpi_send(n_mpi-1);
36   }
37 }
38 #endif // MULTIPLE_DEVICES
```

Listing 2: Some methods of Class *Lattice* from file *lattice.cpp* that have to do with the MPI communication.

```
string get_opencl_code() { return
#ifdef MULTIPLE_DEVICES
R(
uint __attribute__((always_inline)) determine_surf_size(const uint* t_s, const uint c) { // calculate surf_size
  return t_s[(c+1)%3]*t_s[(c+2)%3];
}
void __attribute__((always_inline)) assign_float(global float* buff_value, global float* field_value, const bool set) {
  if(set) *field_value = *buff_value ;
  else    *buff_value  = *field_value;
}
void __attribute__((always_inline)) assign_uchar(global uchar* buff_value, global uchar* field_value, const bool set) {
  if(set) *field_value = *buff_value ;
  else  *buff_value  = *field_value;
}
void __attribute__((always_inline)) fill_xyz(const uint n, const bool set, const uint c, const uint surf_size, const bool offset, const uint* s_xyz, uint* xyz) { // calculate x,y and z
  const bool get = !set;
  xyz[c      ] = offset ? s_xyz[c] - 1 - get : 0 + get; // e.g. for c=0, xyz[0] in {0,1,def_sx-2,def_sx-1}
  xyz[(c+1)%3] = n % s_xyz[(c+1)%3]; // n = xyz[(c+1)%3] + s_xyz[(c+1)%3] * xyz[(c+2)%3] + get*surf_size;
  xyz[(c+2)%3] = offset ? (n - surf_size) / s_xyz[(c+1)%3] : n / s_xyz[(c+1)%3];
}
void __attribute__((always_inline)) fill_n(uint* setget_n, uint* lbm_n, const uint n, const uint* xyz, const uint surf_size, const bool offset) {
  *setget_n = n - offset*surf_size; // indices of the second half and of the first half are the same, in the buffer they are separated by b * surf_size * def_*_data_depth
  *lbm_n = xyz[0] + def_sx * (xyz[1] + def_sy * xyz[2]); // in stream_collide, the indices are computed: n = x+(y+z*sy)*sx
}
void __attribute__((always_inline)) fill_c_set(uint* c, bool* set, const uint c_setget) {
  if(c_setget>2) { *c=c_setget-3; *set=false; } // c in {0,1,2} with 0=x, 1=y, 2=z
  else /*******/ { *c=c_setget  ; *set=true ; }
}
void __attribute__((always_inline)) setget_layer(const uchar c_setget, global uchar* buff_recv, global uchar* buff_send, global fpXX* fc) {
  const uint surfDir[2*def_n_dim*def_n_trans_per_surf] = { // D3Q19
    2, 8,10,14,16, // x=0
    1, 7, 9,13,15, // x=sx-1
    4, 8,12,13,18, // y=0
    3, 7,11,14,17, // y=sy-1
    6,10,12,15,17, // z=0
    5, 9,11,16,18  // z=sz-1
  };
  const uint n = get_global_id(0);
  const uint t_s[3] = {def_sx, def_sy, def_sz};
  uint t_xyz[3], c;
  bool set;
  fill_c_set(&c,&set,c_setget);
  const uint surf_size = determine_surf_size(t_s, c);
  if(n >= 2*surf_size) return; // return indices that only fill up THREAD_BLOCK_SIZE
  const bool b = (n >= surf_size); // the layer in the back is represented by the second half of the buffer and the second half of the global index range
  fill_xyz(n, set, c, surf_size, b, t_s, t_xyz);
  uint setget_n, lbm_n;
  fill_n(&setget_n, &lbm_n, n, t_xyz, surf_size, b);
  global uchar* buff = set ? buff_recv : buff_send; // use buff_recv or buff_send depending on which mode is selected
  global fpXX* fcPtr     = (global fpXX*) (((global uchar*) buff)   +b*surf_size*def_layer_data_depth          );
  const uint* dir = &surfDir[(2*c+(set-b)*(set-b))*def_n_trans_per_surf]; // (set-b)*(set-b) can be 0 or 1
  if(set) {
    #pragma unroll
    for(uint i=0; i<def_n_trans_per_surf; i++) store(fc,index(lbm_n,dir[i]), fcPtr[i*surf_size+setget_n]);
  } else {
    #pragma unroll
    for(uint i=0; i<def_n_trans_per_surf; i++) fcPtr[i*surf_size+setget_n] = load(fc,index(lbm_n,dir[i]));
  }
}
bool __attribute__((always_inline)) work_groups_turn(const uint work_group, const uchar control) {
  if(def_load_balancing) {
    const uint n_first = work_group*def_thread_block_size, n_last=n_first+def_thread_block_size-1;
    bool z_surface=false, y_surface=false, x_surface=false;
    uint t_first, t_last;
    const uint z_first = n_first/(def_sx*def_sy), z_last = n_last/(def_sx*def_sy);
    if(def_halo_z) {
      z_surface = (z_first<2 || z_last>=def_sz-2); // true if group is part of the first or last two layers in z-direction && z-halo activated
    }
    if(def_halo_y || def_halo_x) {
      t_first = n_first%(def_sx*def_sy); t_last = n_last%(def_sx*def_sy); // disassemble 1D index to 3D coordinates (n -> x,y,z)
    }
    if(def_halo_y) {
      const uint y_first = t_first/def_sx, y_last = t_last/def_sx;
      y_surface = (y_first<2 || y_last>=def_sy-2 || y_first>y_last || def_sx*def_sy < def_thread_block_size); // true if group is part of the first or last two layers in y-direction && y-halo activated
    }
    if(def_halo_x) {
      const uint x_first = t_first%def_sx, x_last = t_last%def_sx;
      x_surface = (x_first<2 || x_last>=def_sx-2 || x_first>x_last || def_sx < def_thread_block_size); // true if group is part of the first or last two layers in x-direction && x-halo activated
    }
    const bool inner = !(z_surface || y_surface || x_surface );
    const uchar balance = def_halo_x + def_halo_y + def_halo_z; // number between 1-3

    const bool balance_y = def_halo_y && def_halo_z && inner && z_first<def_sz/balance; // z-distribution
    const bool balance_x = def_halo_x && balance>1 && inner && ((balance==2 && z_first<def_sz/balance) || balance==3 && z_first>def_sz/balance && z_first>(def_sz/balance)*2);

```

```
86      /**/ if(control==2 && z_surface) return true; // true if part of z_surface
87      else if(control==1 && ((y_surface && !z_surface ) || balance_y)) return true; // true if part of y_surface and not
            ↪ computed already || selected by load balance
88      else if(control==0 && ((x_surface && !(y_surface || z_surface)) || balance_x)) return true; // true if part of x_surface
            ↪ and not computed already
89      else if(control==3 && inner && !(balance_y || balance_x)) return true; // true if not part of activated surface
90      else return false;
91    } else { // no load balancing
92      const uint n_first = work_group*def_thread_block_size, n_last=n_first+def_thread_block_size-1;
93      bool z_surface=false, y_surface=false, x_surface=false;
94      uint t_first, t_last;
95      if(def_halo_z) {
96        const uint z_first = n_first/(def_sx*def_sy), z_last = n_last/(def_sx*def_sy);
97        z_surface = (z_first<2 || z_last>=def_sz-2); // true if group is part of the first or last two layers in z-direction &&
              ↪ z-halo activated
98        if(control==2 && z_surface) return true;
99      }
100     if(def_halo_y || def_halo_x) {
101       t_first = n_first%(def_sx*def_sy); t_last = n_last%(def_sx*def_sy); // disassemble 1D index to 3D coordinates (n -> x,y,
              ↪ z)
102     }
103     if(def_halo_y) {
104       const uint y_first = t_first/def_sx, y_last = t_last/def_sx;
105       y_surface = (y_first<2 || y_last>=def_sy-2 || y_first>y_last || def_sx*def_sy < def_thread_block_size); // true if group
              ↪  is part of the first or last two layers in y-direction && y-halo activated
106       if(control==1 && y_surface && !z_surface) return true;
107     }
108     if(def_halo_x) {
109       const uint x_first = t_first%def_sx, x_last = t_last%def_sx;
110       x_surface = (x_first<2 || x_last>=def_sx-2 || x_first>x_last || def_sx < def_thread_block_size); // true if group is
              ↪ part of the first or last two layers in x-direction && x-halo activated
111       if(control==0 && x_surface && !(y_surface || z_surface)) return true;
112     }
113     if(control==3 && !(z_surface || y_surface || x_surface )) return true; // true if not part of activated surface
114     return false;
115   }
116 }
117 void __attribute__((always_inline)) stream_collide(const uchar control, const global fpXX* fc, global fpXX* fs, global float*
        ↪ rho, global float* u, const global uchar* flags) {
118 )+
119 #else // MULTIPLE_DEVICES
120 R(
121 kernel void stream_collide(const global fpXX* fc, global fpXX* fs, global float* rho, global float* u, const global uchar*
        ↪ flags) {
122 )+
123 #endif // MULTIPLE_DEVICES
124 R(
125   const uint n = get_global_id(0); // n = x+(y+z*sy)*sx
126 )+
127 #ifdef MULTIPLE_DEVICES
128 R(
129   if(def_compute_transfer_overlap) {
130     const uint work_group = get_group_id(0);
131     if(!work_groups_turn(work_group, control)) return;
132   }
133   const uint tz = n/(def_sx*def_sy); // disassemble 1D index to 3D coordinates (n -> x,y,z)
134   const uint tt = n%(def_sx*def_sy); // n = x+(y+z*sy)*sx
135   const uint ty = tt/def_sx;
136   const uint tx = tt%def_sx;
137   if(((tx==0 || tx==def_sx-1) && def_halo_x) || ((ty==0 || ty==def_sy-1) && def_halo_y) || ((tz==0 || tz==def_sz-1) &&
        ↪ def_halo_z)) return; // node is halo node
138 #endif // MULTIPLE_DEVICES
139 R(
140   // proceed with normal stream_collide
141 );}
142
143 #ifdef MULTIPLE_DEVICES
144 string get_setget_code(const unsigned int i) { return // generate 6 different setget_*-kernels: 0=set_x, 1=set_y, 2=set_z, 3=
        ↪ get_x, 4=get_y, 5=get_z
145 "\n kernel void setget_layer_"+to_string(i)+"(global uchar* buff_recv, global uchar* buff_send, global fpXX* fc) { \n"
146 "\n setget_layer("+to_string(i)+"u, buff_recv, buff_send, fc); } \n"
147 ;}
148 string get_lbm_code(const unsigned int i) { return // generate 4 different stream_collide-kernels: 0=compute_x_surface, 1=
        ↪ compute_y_surface, 2=compute_z_surface, 3=compute_inner
149 "\n kernel void stream_collide_"+to_string(i)+"(const global fpXX* fc, global fpXX* fs, global float* rho, global float* u,
        ↪ const global uchar* flags) { \n"
150 "\n stream_collide("+to_string(i)+"u, fc, fs, rho, u, flags); }\n"
151 ;}
152 #endif // MULTIPLE_DEVICES
```

Listing 3: Some content of *kernel.cpp* relevant for the buffer transfers. The rest of kernel *stream_collide* is left out after line 139. All variables beginning with *def* are constants, that are set at runtime (before compiling the OpenCL Code). The macro R(...) converts its content to string. The words *global* and *kernel* are keywords of the OpenCL C language. The function call *get_global_id(0)* returns the global ID of the thread, the function call *get_group_id(0)* returns the group ID.

B Eidesstattliche Erklärung

Hiermit versichere ich, die vorliegende Arbeit selbstständig verfasst, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt und alle Zitate deutlich kenntlich gemacht zu haben. Die Arbeit wurde nicht bereits in gleicher oder vergleichbarer Form zur Erlangung eines akademischen Grades eingereicht. Des Weiteren versichere ich, dass die digitale und die gedruckte Version inhaltlich identisch sind.

Bayreuth, den 28.10.2019

Fabian Häusl