Journal Title
XX(X):1–11
©The Author(s) 0000
Reprints and permission:
sagepub.co.uk/journalsPermissions.nav
DOI: 10.1177/ToBeAssigned
www.sagepub.com/
SAGE

Designing a GPU accelerated petaflop capable lattice Boltzmann solver: read aligned data layouts and asynchronous communication

Fredrik Robertsén[1], Jan Westerholm[1] and Keijo Mattila[2]

Abstract

The lattice Boltzmann method is a well-established numerical approach for complex fluid flow simulations. Recently general-purpose graphics processing units have become available as high-performance computing resources at large-scale. We report on designing and implementing a lattice Boltzmann solver for multi-GPU systems that achieves 1.79 PFLOPS performance on 16384 GPUs. To achieve this performance, we introduce a GPU compatible version of the so-called bundle data layout and eliminate the halo sites in order to improve data access alignment. Furthermore, we make use of the possibility to overlap data transfer between the host CPU and the device GPU with computing on the GPU. As a benchmark case, we simulate flow in porous media and measure both strong and weak scaling performance with the emphasis being on large scale simulations using realistic input data.

Keywords

Lattice Boltzmann, GPU, Titan, asynchronous communication, memory alignment, data layout, large scale I/O, load balance

Introduction

The lattice Boltzmann method (LBM) is a mesoscopic approach for the numerical simulation of fluid flow phenomena, for example single-phase flows in capillaries or binary-fluid flows in porous media. It is based on a statistical-mechanical modelling of transport phenomena and it operates by evolving discrete-velocity distribution functions in time via alternating collision-propagation steps. For a general introduction to LBM we refer to (Aidun and Clausen 2010, Benzi et al. 1992). The standard LBM is well suited for parallel computing as it involves explicit time stepping and local data dependencies. In the field of high-performance computing, much research has already been carried out where these properties have been efficiently utilized on contemporary hardware including CPUs as well as GPUs. In particular, the weak scalability of LBM has been well demonstrated both on modern CPUs (see e.g. (Godenschwager et al. 2013)) as well as on GPUs (Gray et al. 2011).

In this article we study the possibilities of efficiently utilizing the computational resources of multi-GPU systems for fluid flow simulations with LBM. Specifically, we report on progress in optimizing the run time of multi-GPU LBM simulations by designing the GPU kernels to efficiently access global memory on GPUs, and utilizing the bus between the host and the GPU for asynchronous data transfer in order to overlap communication with computation. We begin by giving a brief introduction to the lattice Boltzmann method. Section "GPU solver design" covers the memory addressing method used as well as the different data layouts utilized. In section "Scalable multi-GPU design" we cover the design choices affecting the parallel scalability of a solver, we discuss the critical steps for implementing asynchronous data communication between GPUs, and we show how a scalable parallel I/O scheme can be implemented and how the computational load can be evenly distributed between the compute nodes. Finally,

[1]Åbo Akademi University, Faculty of Science and Engineering, Åbo, Finland
[2] University of Jyväskylä, Department of Physics and Nanoscience Center, Jyväskylä, Finland and Tampere University of Technology, Department of Physics, Tampere, Finland

Corresponding author:
Fredrik Robertsén, Åbo Akademi University, Faculty of Science and Engineering, Vattenborgsvägen 5, Åbo, Finland
Email: fredrik.robertsen@abo.fi

in section "Results", we report on the computational performance resulting from the above design choices as well as the parallel scalability on a multi-petaflop machine.

This article is an extended version of Robertsén et al. (2015). New contributions are the vectorized collision and bundle data layouts, the comparison of two LBM solvers (two-lattice and AA), and an extended set of results with a new peak performance.

Lattice Boltzmann Method

The lattice Boltzmann method is a time-stepping method typically operating on a regular 2D or 3D lattice. The computational geometry can be represented by simply dividing the lattice sites into fluid and solid phases. During each time step, the single-particle distribution functions at a fluid site undergo a collision procedure after which they are propagated to the neighboring fluid sites. This ordering of the collision and propagation steps corresponds to the so-called push scheme (Wellein et al. 2006). Here we implement the D3Q19 scheme (Qian et al. 1992) with the two-relaxation-time (TRT) collision operator (Ginzburg et al. 2008) and the halfway-bounce back boundary condition (Cornubert et al. 1991).

Input data

As our input data we utilize synthetic X-ray tomography images of a Fontainebleau sandstone sample. These images are available from Institute for Computational Physics at the University of Stuttgart (Hilfer and Zauner 2011) in a range of resolutions from 128^3 voxels to 32768^3 voxels. The sample has a porosity around 13%. Overall the sample is very homogeneous, i.e. the spatial distribution of pores in the sample is uniform. Figure 1 shows a cross section of the sample used: the gray color indicates solid phase (i.e. the micrograins) of the geometry while black represents pores.

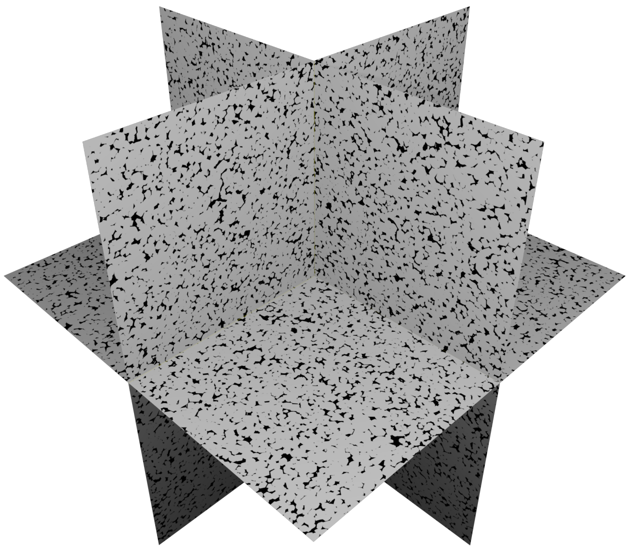
Figure 1. Cross sections of the input sample used.

GPU solver design

GPU Kernels

In general, a GPU kernel may be either computationally or memory bandwidth bound. For example, the NVIDIA K20X (NVI 2013) is specified at a peak double precision floating-point operation speed of 1.312 TFLOPS and 250 GB/s memory bandwidth. Assuming no memory bandwidth overhead from error correcting code memory we simply divide these two numbers to obtain the estimate that roughly 40 floating-point operations per double precision value are needed to reach equilibrium between peak performance and memory bandwidth.

The propagation part of an LBM implementation contains essentially no arithmetic operations while the collision part depends on the chosen collision scheme, typically involving between 10 and 15 arithmetic operations per distribution function. This simple argument indicates that the main factor limiting the performance is the available memory bandwidth on the GPU. Hence, using two separate kernels, a propagation kernel and a collision kernel, both accessing the same data in turn, is not optimal from a performance point of view. To get maximum performance from the GPUs we therefore choose to do both the collision and the propagation in the same kernel accessing fluid data only once, with one thread of the kernel being responsible for updating one lattice site.

In order to parallelize fused collision and propagation on the GPU we chose to implement it in two different ways: the two-lattice algorithm (Wittmann et al. 2013) and the AA algorithm (Bailey et al. 2009). In the first case two memory locations are allocated for each distribution function: during one time step we read from the first one and write to the second, then during the next time step vice versa. In the AA algorithm propagation is done only every other time step, and on these occasions the propagation is done both before and after the relaxation. During the other time step only relaxations are performed. Both algorithms are well-suited for GPU computing, or multithreaded computing in general, as they allow a random update order for the lattice sites without data race conditions.

Memory Addressing

In a typical porous media fluid flow simulation, a large fraction of the lattice sites belongs to the solid phase where no computation takes place. Therefore, in order to save memory, we choose to allocate memory only for the fluid lattice sites. This implementation choice can be realized with a fully indirect memory addressing scheme (Schulz et al. 2002), where for each distribution function we read a pre-calculated 32 bit integer containing the address for the propagation of the distribution function.

Additionally the indirect addressing scheme enables us to do away with the halo sites often used to encase the local computational domain. Instead of propagating out into the halo sites we enforce the bounce-back boundary conditions at the edges of each local domain. The bounce-back will put the outgoing distribution functions into memory locations for incoming distribution functions but data races will not occur as outgoing values are always communicated before incoming values are stored. A benefit of this scheme without the halo sites is that some of the memory access operations can be orchestrated in a way that are perfectly aligned and only usable data will be read in. With the two lattice algorithm this means we can have all read operations perfectly coalesced but the write operations would still involve scattered accesses, alternatively we can turn this around and have coalesced writes but scattered reads. Using the AA algorithm every other time step, the one not doing propagation, will access perfectly aligned memory for all read and write operations but for the other time step, the one doing two propagations, all accesses will be to scattered memory locations.

Data layouts The optimal data access patterns for Nvidia GPUs is that the data access by the computational kernels is done in a coalesced fashion: threads within the same warp should access data from the same cache line. This can be accomplished for instance by adopting the propagation or stream optimized data layout where values representing the same velocity component for different lattice sites are placed in contiguous memory locations. In the collision optimized data layout, all velocity components associated with a given lattice site are placed in adjacent memory locations. This latter type of layout, an array of structures, is the opposite layout to what is optimal for the GPU. The array of structures access pattern generally results in a warp of threads needing one cache line to be fetched for each thread and only one value from that cache line can be used.

We also tested the bundle layout proposed by Mattila et al. (Mattila et al. 2008). While it is still not optimal for the GPU there is some data within a cache line that can be utilized by multiple threads within a warp. Thus, from a performance point of view, it should be more efficient than the collision optimized layout while still being slower than the stream optimized version.

However, with some modifications, both the collision optimized as well as the bundle data layout can be made compliant with the data access patterns of the GPU. In the collision optimized layout we have a structure for each lattice site. However, we can modify the site structure such that instead of just containing one lattice site we extend it to contain the values for 16 sites since 16 double precision values equals one cache line on current Nvidia GPU architectures. We arrange these values to create vectors of 16 elements where all values represent the same velocity component for the 16 different lattice sites, as shown in figure 2. This way we can still maintain data coalescing to the same degree as in the stream optimized layout. The same techniques can also be used to vectorize the bundle layout: when creating the bundles, instead of using only values for a single site, we use vectors with 16 values representing the same fluid component for different lattice sites. Figure 2 illustrates the modification to the bundle data layout. This way we get the benefits from coalesced data accesses but also the possibility for a lower cache miss rate than the bundle layout can offer.

Scalable multi-GPU design

In a multi-GPU environment we divide the original simulation domain into a set of local sub-domains and arrange for the GPUs to communicate their edge layers to their spatial neighbors through their host CPU which, in turn, utilize e.g. Message Passing Interface (MPI) to execute the data transfer. With the D3Q19 scheme the boundary layers are, in principle, only one site deep. Our aim is to overlap communication with computation as much as possible. This can be achieved using asynchronous communication on two separate levels: firstly between the compute nodes and secondly between the host CPU and the device GPU.

Dividing the local computational domain for asynchronous communication while Maintaining Memory Alignment

Our asynchronous communication approach involves dividing the computations on the GPU into two parts.

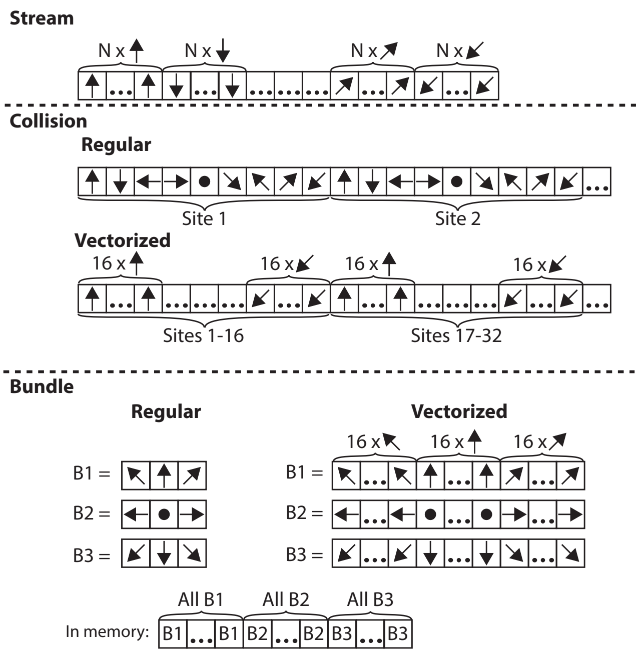

Figure 2. D2Q9 visualization of the data layouts used. N is the number of lattice sites.

First the edge sites of the local domain from which data needs to be transferred to neighboring compute nodes are updated. Then the communication is initiated, after which the update of the lattice sites immediately proceeds to the interior of the local domain. In the LBM context this very general procedure has been previously presented, e.g., in (Pohl et al. 2004, Schulz et al. 2002).

The main task is to design the kernel that handles the edge data to be communicated to other GPUs. As the hardware architecture of modern GPUs is a separate unit connected via a PCI-e bus (PCI-SIG 2009), we have to transfer communication data over this bus, and the efficient use of this bus dictates that data should be transferred in large blocks. The values to be communicated are however scattered in the GPU memory and must be packed before and unpacked after transfers. A kernel executing this will have scattered data access patterns. As the ratio of communication to computation is proportional to the surface to volume ratio of the local domain, the runtime of this edge value gather kernel is expected to be relatively small (in our measurements less than 0.1% of the total GPU run time for a fully loaded GPU).

However, there is a performance issue with the straightforward way of first handling the lattice sites at the edge of the local domain and then handling the interior sites of the local domain. The kernel performing the collision and the propagation for the edge sites will underutilize memory bandwidth since it often ends up using only one or two values from each memory segment fetched corresponding to the edge sites. The problem will also affect the performance of the kernel for the interior sites since now the memory accesses for this kernel will be misaligned, too. In order to maintain aligned memory accesses and to fully utilize the available memory bandwidth, instead of processing only one lattice site at the edge of the domain, the edge kernel should process at least 128 bytes of lattice data, or 16 lattice sites in our case when using double precision floating point values. This data should be aligned in such a way that it all falls into the same 128 byte cache line and accessed using only one operation. Using aligned 16 lattice site segments will then allow both the edge and the interior kernel to maintain proper memory coalescing, every other time step for the AA algorithm or all reads for the two lattice algorithm.

Asynchronous Data Transfers

Within a compute node, the GPU and the CPU can be occupied with different tasks at the same time. This enables us to perform different tasks asynchronously on the CPU and GPU. One use of this is to speed up the communication part of the program, an example of which is presented by Gray et al. (Gray et al. 2011). Here we instead adopt the approach discussed in the previous subsection. The local computational domains are updated in two parts on the GPU. Technically this is achieved by having two CUDA streams: the edge site computations together with the memory transfers and the interior site computations are carried out by separate CUDA streams as shown in fig. 3. The CUDA stream with the edge site computations is first executed. Once the execution reaches the memory transfers, the execution of the second CUDA stream starts in parallel, that is, the communication and the computation are overlapped. The host code waits for the memory transfer to complete before communicating the data values with the neighboring hosts using MPI. Once the MPI communication is completed the new values received are transferred to the GPU; ideally the GPU is still performing the computations for the interior sites.

Obviously the benefit of dividing the computational domain into edge and interior sites is that the communication between CPUs may finish before the entire time step has been completed. Even if the communication is not done when the computation for the interior domain finishes, we still manage to hide a part of the communication equal to the computation time. Our approach to use aligned lattice site segments

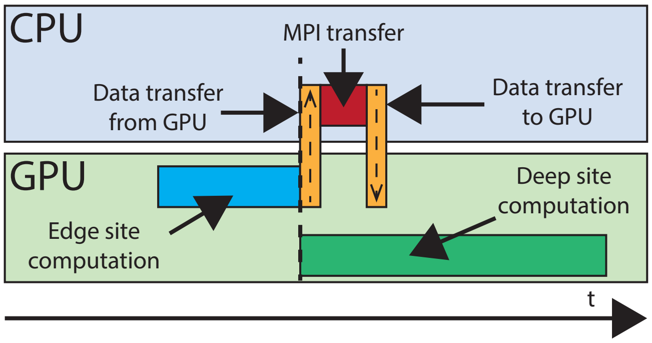

Figure 3. Asynchronous data transfer time line.

at the edges will of course somewhat diminish the amount of computation that can be used to hide communication.

Scalable I/O

In our simulation cases we utilize the 3D images coined A16, A8, A4, A2, and A1 which contain 1024^3, 2048^3, 4096^3, 8192^3, and 16384^3 voxels, respectively. The porosity of the sandstone is approximately 13 %. In the simulations one voxel corresponds to a lattice site. The voxels of an image are stored in a single file and each process thus needs to find the values associated with its own subdomain from the file. Our distribution schemes are based on each compute node being given a unique cuboid block of the larger input file and there are no lattice sites associated with more than one compute node. The file size for the largest image used (A1) is 4TB. In this case we were able to simulate flow in half of the sample only due to memory constraints on the GPU. The parallel input and output of files are carried out using the MPI-I/O (Mes 2012). With MPI-I/O one can use the built-in MPI data types to describe the pattern with which an MPI rank will access data from the input file. In particular, the access to cuboid subdomains can be described with simple subarrays. Each rank is then assigned a specific view into the given input file by utilizing the above MPI data types, after which the data can be accessed by reading the non-contiguous data representing the cuboid in the file as a single, apparently contiguous segment.

When using collective I/O operations MPI-IO may optimize the file accesses. It can merge requests from multiple compute nodes into bigger requests as well as perform data sieving by reading large chunks of data and extracting only what is needed. Finally it can improve prefetching and caching behavior.

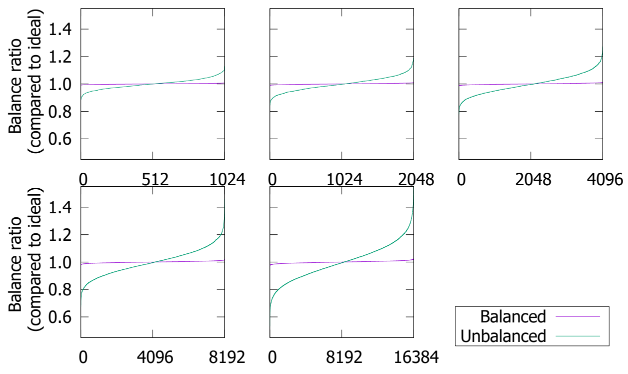

Figure 4. Sorted load distribution of the A4 sample for varying compute node counts.

Load balance

The internal load balance of the sample geometry we use is such that the input data can easily be divided into equally sized blocks and they will all contain roughly the same number of fluid sites. However, this only works as long as the number of domains is relatively small. Problems arise when distributing the same sample over thousands of compute nodes. The green line in figure 4 illustrates the distribution of the load when using this primitive scheme to divide the computational domain over multiple compute nodes. To address this issue we implemented a simple recursive bisection method with cuboid shaped domains. The load balancer works by considering only the fluid sites within the simulation domain, then dividing the domain into two parts with about equal amounts of fluid sites and recursively repeating the processes for the new domains. Furthermore, in order to minimize the amount of communication data, we maintained the domains as cubic shaped as possible. As the aspect ratio of a cuboid gets larger, not only the amount of data to be communicated but also the number of compute nodes involved in the communication per subdomain increase.

The load balance results for the recursive bisection scheme are illustrated by the magenta line in figure 4. It is clear that the recursive bisection leads to superior load balance but it is also clear that as we scale up, even this method suffers from some imbalance between the domains. For 16384 nodes we see a strong reduction in the imbalance between the largest and smallest workload, i.e. for the cartesian and recursive bisection schemes the ratio between maximum and minimum workload for subdomains is 3.12 and 1.06, respectively. One downside of the recursive bisection scheme compared to the cartesian scheme is that in the cartesian scheme each compute node needs to communicate with at most 18 other compute nodes whereas in the recursive bisection scheme there is no fixed upper limit for the number of neighbors involved in the communication.

Results

Our large-scale test machine was Titan (ORNL) with 18688 nodes, one AMD Opteron 6274 (AMD) and one NVIDIA K20X GPU (NVI 2013) per node, and a Gemini (Cra 2010) network by Cray. The interconnect is laid out as a three dimensional torus. In each intersection point in the torus, two compute nodes are connected, and each torus node is connected to its nearest neighbors with point to point links. In addition to handling the data destined for the current node the network also need to relay data destined for other nodes whose communication is passing through the current node. All links in the torus are however not equal: the design of the network interface at each node is such that there are two links in both the positive and negative X and Z directions but only a single positive and a single negative link in the Y direction. Effectively there is only half the bandwidth available in the Y direction.

Bland et al (Bland et al. 2012) ran tests on the available MPI bandwidth per node, using XK6 nodes on Jaguar, and showed that there is roughly half the available bandwidth available for bidirectional transfers in the Y direction compared to the X and Z directions: 5.4 GB/s for the Y direction compared to 10.6GB/s and 10.5GB/s in the X and Z direction. The only difference between the XK6 nodes on Jaguar compared to the XK7 nodes on Titan is that the GPU in them has been upgraded to an Nvidia K20 series GPU (Cra 2011). A downside of the torus interconnect is that as the machine grows, the average number of hops between two arbitrary points in the machine will also increase. This in combination with the fact that we have almost no control over where in the machine our processes are spawned can create some extreme case scenarios with respect to the communication.

On Titan we used the CUDA 5.5 toolkit and gcc version 4.8.2 compiler to build the programs. All results presented are the average of all the nodes in a run, unless stated otherwise.

Previous work

Previous large scale lattice Boltzmann simulations on Titan have been performed by among other (Gray et al. 2015) and (McClure et al. 2014). In both cases the simulations were multi-phase fluid flows. Gray et al. reports a performance of about 0.37 PFLOP when using 8192 nodes on Titan: these implementations utilized the shift algorithm and asynchronous data communication. The method we present for asynchronous communication is a less complicated variant that achieves the same goal. We also excluded the halo sites used for communication which improves the memory alignment of data accesses. McClure et al reports a total performance of 244754 million fluid lattice site updates per second (MFLUPS) using 4096 nodes. Their solver implementation has no asynchronous communication as well as wno load balancing between compute nodes. Our solver implements a simple recursive bisection load balancing scheme that significantly reduces the imbalance of the workload between nodes.

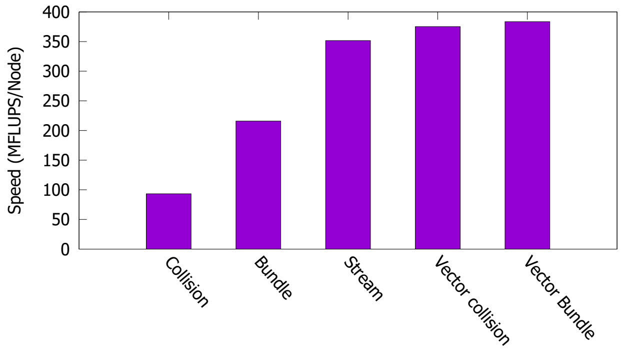

Figure 5. Performance of different data layouts using an AA algorithm based solver.

Implementation details

Algorithm baseline performance We tested both the two lattice and AA algorithm. From a memory usage standpoint the AA algorithm is a better choice. Comparing the performance, using the A4 sample and running on 1024 nodes on Titan, we measured that the AA algorithm is also 1.34 times faster than the two lattice scheme. With this sample and node count the AA algorithm runs at 384 MFLUPS per node while the two lattice scheme only manages 286.4 MFLUPS per node.

Data layout Figure 5 illustrates the performance achieved with the different data layouts using 64 compute nodes and the A8 sample. As expected, the collision optimized layout delivers the worst performance since its access pattern does not conform to the GPUs data access requirements. The bundle layout is more than twice as fast as the collision layout but as it still does not fully conform to the GPUs memory access requirements it falls behind the stream layout.

The vectorized versions of both the collision and bundle layout give a better performance than the basic stream layout. Compared to the stream layout both vectorized layouts perform fewer global memory operations with the vectorized bundle performing the fewest. This indicates that the vectorized layouts have a lower cache miss rate than the basic stream layout.

Asynchronous Data Communication The asynchronous data communication between the CPU and the GPU was tested by comparing the observed performance results against those obtained with synchronous communication. For these tests the synchronous version was set up to first complete all the computation before the communication starts, and once the communication has completed the solver proceeds to the next time step. We ran these two cases using our two lattice solver on 1024 Titan nodes using the A4 image and obtained a sustained performance of 255 and 289 MFLUPS for the synchronous and asynchronous data transfer, respectively. The speed-up from using asynchronous GPU memory transfers is thus 1.13.

Memory access alignment Our benchmarking with the two-lattice based solver showed a speed-up in performance by a factor of 1.14 when using aligned segments around the edge sites, omitting the halo sites and enforcing the bounce back boundary conditions at the edges of the local domain as presented in section "Dividing the local computational domain for asynchronous communication while Maintaining Memory Alignment".

I/O performance

The default setup of the Lustre file system (Ora 2011) on Titan uses a stripe count of four (ORLC 1), which implies that any file will be striped over four Object Storage Targets (Ora 2011). Using this configuration, a file I/O performance of 546 MB/s and 274 MB/s were observed with 128 and 1024 nodes, respectively. By using a stripe count of 160, initially the maximum value (ORLC 2), a file I/O performance of 5507.2 MB/s with 1024 nodes and 6146.4 MB/s with 8192 nodes were measured. After a system upgrade we were able to increase the stripe count to 1008 which delivered a performance of 177 GB/s and 314 GB/s when reading the A2 sample using 8192 and 16384 nodes, respectively. Reading the A1 sample with this configuration and using 16384 nodes took 6.5 seconds. With the default settings this operation would have taken about two hours based on the speed from 1024 nodes. These results show how

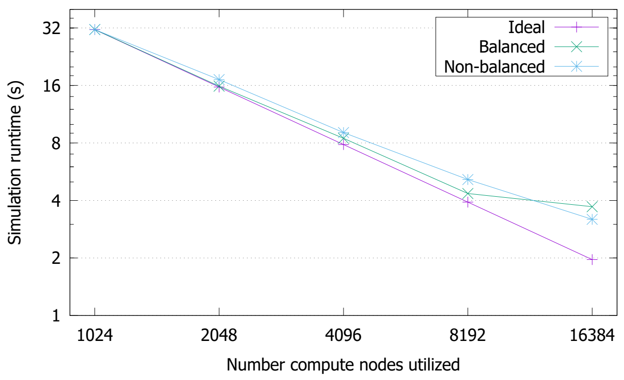

Figure 6. Strong scaling: runtime for implementations based on the two-lattice algorithm and with load balanced and non-load balanced domain decompositions.

things which might seem trivial at small scales can become bottlenecks when scaling up. It should also be noted that we experienced large fluctuations in the achievable bandwidth with large files and a large number of compute nodes participating.

Load balance

The impact of our load balance scheme is shown in figure 6 where strong scaling results for the A4 sample going from 1024 compute nodes to 16384 compute nodes using our two lattice solver are plotted. Our load balanced case follows the ideal scaling more closely than the non-load balanced case up until 8192 compute nodes at which point the load balanced solver is 1.18 times faster than the non-load balanced version.

At 16384 compute nodes the performance of the load balanced solver drops off sharply, however, there is no drop off in the performance of the non-load balanced. The likely cause of the performance degradation of the load balanced solver is that it is becoming communication bound. The non-load balanced version has a simpler communication pattern since each node only needs to communicate with at most 18 other nodes whereas the load balanced version has a higher number of nodes that need to participate in the communication for each subdomain.

Scaling

For all our remaining scaling runs we utilized the load balancing scheme presented previously and the AA algorithm solver with the vectorized bundle layout. The results of these particular scaling runs have also been discussed in ref (Mattila et al. 2016), but only in a cursory manner.

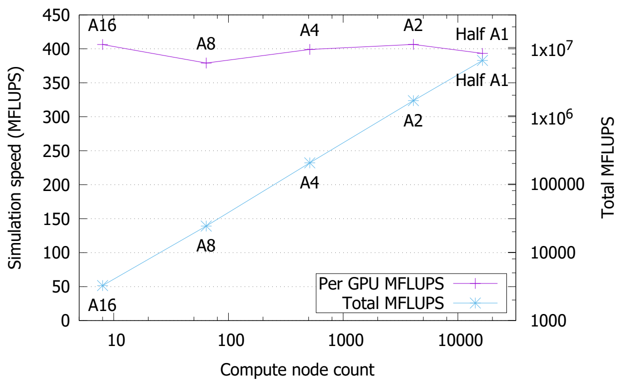

Figure 7. Weak scaling for different resolutions of input sample using an implementation based on the AA algorithm.

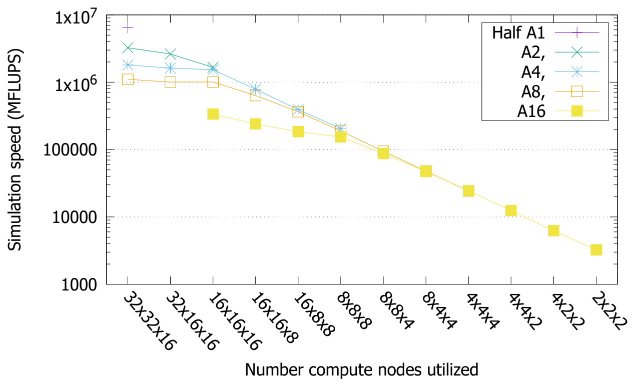

Figure 8. Strong scaling for different resolutions of input sample using an implementation based on the AA algorithm.

Starting with the weak scaling results, shown in figure 7, these are run with the maximum subdomains that we can fit on a single GPU. We see that overall a near perfect weak scaling is achieved. For the runs with half the A1 sample there is a slight drop compared to the previous results. We also observe a slight increase in the per node performance for each increase in the sample resolution up to the A2 sample. This is due to the fact that at higher resolutions the pores within the sample contain more fluid sites and with more adjacent fluid sites there is less divergence in the memory access operations for a thread warp.

Moving to strong scaling, the results shown in figure 8, both the A8 and A4 samples run into problems scaling past 4096 compute nodes. The A2 sample fares slightly better and does not show as clear a performance falloff as the A8 and A4 samples when going past 4096 compute nodes, but still it is not able to keep up to the ideal scaling for that sample. Scaling the A16 sample past 512 compute nodes we run into similar problems as with the A8 sample when going past 4096 compute nodes.

Each sample run starts with roughly 1.7E+7 fluid sites per GPU. As the number of compute nodes are doubled, the number of fluid sites per GPU is halved. In an ideal situation we expect the performance of the code to stay the same up to the point when the GPU becomes undersaturated. Based on performance measurements for a single GPU, using the porous media sample, we know that the GPU is capable of maintaining the same performance achieved at 1.7e+7 when scaling down to 2.7e+5 fluid sites. In fact, going to a domain with 3.4E+4 fluid sites, the performance only drops to 84% of the peak.

The A16 sample clearly diverges from ideal scaling when going below a per GPU domain size of roughly 2.7e+5 fluid sites. At 1.4E+5 fluid sites per GPU it is only able to maintain 44% of the peak performance. For the A8 sample the significant performance drop appears when going below 5.5E+5 fluid sites, and at 2.7E+5 the solver only manages 32% of the peak performance. The A4 already sees a steep drop in the performance when going below a per GPU domain size of 2.2E+6 fluid sites, and at 1.1E+6 fluid sites per GPU it only maintains 49% of its peak performance.

These performance drops encountered all happen at significantly higher per GPU load compared to when the single GPU starts to loose performance. The fact that all samples encounter issues at different loads also suggests that the performance issues do not stem from the performance of the GPU kernel code. The fact that the A8, A4 and, to some degree, the A2 samples all seem to encounter problems at the same compute node counts further strengthens the view that the performance issues are not due to the speed of the GPUs. Combined with the behavior we saw when comparing the load balanced to the non-load balanced scaling, we believe that it is the communication that is the cause of the scaling issues at and past 8192 compute nodes for the majority of the samples. Our load balancing scheme does put more pressure on the communication network since it does not guarantee that the number of nodes any given node needs to communicate with is fixed but varies from domain to domain. The fact that we do not have any direct control over where the processes are placed in the machine further degrades the communication performance since there is no guarantee that compute nodes that are physically close get computational domains that are logically close. Additional communication latency is added due to the GPU being a separate part of the system and all communication to and from it needs to go over the PCI-e bus through the host systems CPU to reach the network adapter.

We see that the A2 sample fares better when scaling up to 8192 nodes than the two smaller samples indicating that the communication does not entirely dictate the simulation speed for that case. Thus we are able to weakly scale up to utilizing the entire machine with our current solver code, but strongly scaling anything but the largest samples yields very small speedups when going past 4096 compute nodes.

At 16384 nodes using half the A1 sample, the total amount of data moving between GPUs through host and the network was 2.9 TB/s with each node sending on average 8.2 MB per time step. It should be noted that due to how the asynchronous communication was implemented there are times during each time step when no communication occurs. Due to the load balancing scheme we applied, the worst case scenario was that one node needed to communicate with 24 of its neighbors. The load balancing scheme in combination with the input data also caused the amount of data transferred for each node to be imbalanced. The worst case was a node sending a total of 14 MB of data with the largest message sent over 4.5 MB but over 50% of all messages sent under 0.5 MB in size.

To calculate the floating point performance of our code we took the MFLUPS performance we achieved with 16384 nodes using the subsample of the A1 simulation geometry, 393.4 MFLUPS, and multiplied it with the number of floating point operations required per lattice site update which, according to the Nvidias profiling tool (NVI 2014), is 279. For the fastest case this gives a total performance of 1.79 PFLOPS running on 16384 compute nodes. Thus our solver achieves slightly over 10% of the linpack performance reported for Titan (17.6 PFLOPS) when running on 87.7% of the full machine.

The amount of memory bandwidth utilized was measured using the Nvidia profiler to calculate the bandwidth needed per site update. From the profiler we get the total bandwidth per site update that includes both read and write bandwidth as well as the overhead caused by running with the ECC functionality enabled. For the AA algorithm we measured 545 bytes per site update as an average for odd and even time steps. Using the performance we achieved when running on 16384 GPUs, 393.4 MFLUPS, we need 209.3 GB/s of bandwidth from each GPU or around 84% of the theoretical peak bandwidth from each GPU. The profiler reported a 100% global load and store efficiency for the even time steps and 70% efficiency for the odd time steps.

Conclusions

We have shown that the lattice Boltzmann method can efficiently utilize the computational resources offered by general-purpose graphics processing units for systems up to 16384 compute nodes. With the utilized simulation geometries representing porous materials, a sustained performance beyond 393 million lattice site updates per second per GPU can be achieved in multi-GPU computing using our biggest input sample, leading to a floating point performance of 1.79 PFLOPS using double precision values. This performance has been achieved by overlapping GPU-to-GPU communication by computation on the GPU using asynchronous data transfers between the CPU and the GPU, and the compute nodes, as well as adapting the data layout of the solver to fit the GPUs memory access requirements. We identified the bottleneck for the single GPU performance to be the amount of memory bandwidth available on the GPU card. For the strong scaling case the bottleneck became the communication network of the machine.

Acknowledgements

The authors would like to thank the Institute for Computational Physics at Universität Stuttgart for the use of their tomography data.

This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

Data preprocessing and initial testing was done using resources provided by CSC - IT Center for Science Ltd.

Funding

We acknowledge the financial support from the European Community's Seventh Framework programmes ICT-2011.9.13 (CRESTA) and NMP.2013.1.4-1 (SimPhoNy) under Grant Agreements no. 287703 and no. 604005, respectively.

References

C. Aidun and J. Clausen. Lattice-Boltzmann method for complex flows. *Annual Review of Fluid Mechanics*, 42:439, 2010.

AMD. Amd opteron^{TM} 6200 series processors. Web page. Retrived 2015-09-14 `http://www.amd.com/en-us/products/server/opteron/6000/6200`.

P. Bailey, J. Myre, S. Walsh, D. Lilja, and M. Saar. Accelerating lattice Boltzmann fluid flow simulations using graphics processors. In *2009 International Conference on Parallel Processing*, pages 550–557, Vienna, Austria, 22–25th September, 2009. doi: 10.1109/ICPP.2009.38.

R. Benzi, S. Succi, and M. Vergassola. The lattice Boltzmann equation: theory and applications. *Physic Reports*, 222(3):145, 1992.

A. Bland, J. Wells, O. Messer, O. Hernandez, and J. Rogers. Titan: Early experience with the cray xk6 at oak ridge national laboratory. In *Proceedings of Cray User Group Conference (CUG 2012)*, 2012.

R. Cornubert, D. d'Humìes, and D. Levermore. A Knudsen layer theory for lattice gases. *Physica D: Nonlinear Phenomena*, 47(1-2):241–259, January 1991.

The Gemini Network. Cray Inc., 2010.

Cray XK7. Cray Inc., 2011.

Oak Ridge Leadership Computing Facility. Lustre basics. Web page, a. Retrived 2015-09-14 `https://www.olcf.ornl.gov/kb_articles/lustre-basics/`.

Oak Ridge Leadership Computing Facility. Atlas updates. Web page, b. Retrived 2015-09-14 `https://www.olcf.ornl.gov/kb_articles/atlas-updates/`.

I. Ginzburg, F. Verhaeghe, and D. d'Humières. Two-relaxation-time lattice Boltzmann scheme: About parametrization, velocity, pressure and mixed boundary conditions. *Communications in Computational Physics*, 3(2):427, 2008.

C. Godenschwager, F. Schornbaum, M. Bauer, H. Köstler, and U. Rüde. A framework for hybrid parallel flow simulations with a trillion cells in complex geometries. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 35:1–35:12, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2378-9. doi: 10.1145/2503210.2503273. URL `http://doi.acm.org/10.1145/2503210.2503273`.

A. Gray, A. Hart, A. Richardson, and K. Stratford. Lattice Boltzmann for large-scale gpu systems. In *PARCO*, pages 167–174, 2011.

A. Gray, A. Hart, O. Henrich, and K. Stratford. Scaling soft matter physics to thousands of graphics processing units in parallel. *International Journal of High Performance Computing Applications*, 29(3): 274–283, 2015. doi: 10.1177/1094342015576848.

R. Hilfer and T. Zauner. High-precision synthetic computed tomography of reconstructed porous media. *Physical Review E*, 84:062301, Dec 2011.

Oak Ridge National Laboratory. Introducing titan — the world's #1 open science supercomputer. Retrived: 2015-09-14 `https://www.olcf.ornl.gov/titan/`.

K. Mattila, J. Hyväluoma, J. Timonen, and T. Rossi. Comparison of implementations of the lattice-boltzmann method. *Computers & Mathematics with Applications*, 55(7):1514 – 1524, 2008. Mesoscopic Methods in Engineering and Science.

J.E. McClure, Hao Wang, J.F. Prins, C.T. Miller, and Wu chun Feng. Petascale application of a coupled cpu-gpu algorithm for simulation and analysis of multiphase flow solutions in porous medium systems. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pages 583–592, May 2014. doi: 10.1109/IPDPS.2014.67.

MPI: A Message-Passing Interface Standard Version 3.0. Message Passing Interface Forum, 2012.

NVIDIA® TESLA® GPU ACCELERATORS. NVIDIA, October 2013.

Profiler Users's Guide. NVIDIA, August 2014.

Lustre File System^{TM} Operations Manual for Lustre. Oracle, January 2011.

PCI-SIG. *PCI Express® Base Specification Revision 2.1.* PCI-SIG, 2009. `http://www.pcisig.com/specifications/pciexpress/base`.

T. Pohl, F. Deserno, N. Thurey, U. Rude, P. Lammers, G. Wellein, and T. Zeiser. Performance evaluation of parallel large-scale lattice Boltzmann applications on three supercomputing architectures. In *Proceedings of the 2004 ACM/IEEE Conference on Supercomputing*, SC '04, pages 21–34, 2004. ISBN 0-7695-2153-3.

Y. H. Qian, D. D'Humières, and P. Lallemand. Lattice BGK models for Navier-Stokes equation. *Europhysics Letters*, 17(6):479, 1992.

F. Robertsén, J. Westerholm, and K. Mattila. Lattice boltzmann simulations at petascale on multi-gpu systems with asynchronous data transfer and strictly enforced memory read alignment. In *Parallel, Distributed and Network-Based Processing (PDP),* *2015 23rd Euromicro International Conference on*, pages 604–609, March 2015.

M. Schulz, M. Krafczyk, J. Tölke, and E. Rank. Parallelization strategies and efficiency of CFD computations in complex geometries using lattice Boltzmann methods on high-performance computers. In *High Performance Scientific And Engineering Computing*, volume 21 of *Lecture Notes in Computational Science and Engineering*, pages 115–122. 2002.

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

M. Wittmann, T. Zeiser, G. Hager, and G. Wellein. Comparison of different propagation steps for lattice Boltzmann methods. *Computers & Mathematics with Applications*, 65(6):924–935, 2013.

Author Biographies

Fredrik Robertsén is a Phd student at Åbo Akademi University studying in the software engineering laboratory. He received his master's degree from Åb bo Akademi in 2013. His current work centers on exploring modern hardware and software systems and how these can be used to create efficient and scalable lattice Boltzmann codes.

Jan Westerholm is a professor in high performance computing with industrial applications at the Faculty of Science and Engineering at Åbo Akademi University. He received his master's degree from Helsinki University of Technology and his PhD in physics from Princeton University. His research areas include parallel computing, code optimization and accelerator programming with applications in stochastic optimization, computational physics, biology and geographical information systems.

Keijo Mattila obtained PhD in scientific computing (University of Jyväskylä, Finland, 2010) and did a two years post-doc period (2011–2013) at the Federal university of Santa Catarina, Florianópolis, Brazil. His main research interests include mathematical modelling, computational physics, numerical methods, and high-performance computing. A particular research topic is the Lattice Boltzmann method, its development and application to complex transport phenomena. Currently he is employed by the University of Jyväskylä and is an external researcher at the Tampere university of technology, Finland.