TUCS

Fredrik Robertsén

The Lattice Boltzmann Method, a Petaflop and Beyond

Turku Centre *for* Computer Science

TUCS Dissertations
No 231, April 2018

The Lattice Boltzmann Method, a Petaflop and Beyond

Fredrik Robertsén

To be presented, with the permission of the Division of Natural Sciences and Technology of the University of Åbo Akademi for public criticism in Auditorium XXI on April 20, 2018, at 13.

University of Åbo Akademi
Department of Information Technologies
Vattenborgsvägen 3 20500 Åbo Finland

2018

Supervisors

Professor Jan Westerholm
Department of Information Technologies
Åbo Akademi University
Vattenborgsvägen 3 20500 Åbo
Finland

Docent Keijo Mattila
Faculty of Mathematics and Science
University of Jyväskylä
Survontie 9 C
Finland

Reviewers

Professor Erik Lindahl
Department of Biochemistry and Biophysics
Stockholm University
Universitetsvägen 10 A, 106 91 Stockholm
Sweden

Dr. Derek Groen
Department of Simulation and Modelling
Brunel University London
Kingston Ln, London, Uxbridge UB8 3PH
United Kingdom

Opponent

Professor Erik Lindahl
Department of Biochemistry and Biophysics
Stockholm University
Universitetsvägen 10 A, 106 91 Stockholm
Sweden

ISBN 978-952-12-3692-1
ISSN 1239-1883

Abstract

With computer simulations real world phenomena can be analyzed in great detail. Computational fluid dynamics, for example, allows simulation of fluid flow phenomena that might not otherwise be observable or researchers might not have the resources to observe. Researchers want to analyze larger and more complex systems in a shorter time to allow them to get more work done with their resources. To achieve this the simulation codes need to be able to efficiently use large computer systems.

This thesis focuses on the lattice Boltzmann method (LBM) and the usage of computational accelerators to run LB simulations, but also covers some optimization and performance results for regular CPU based systems. The higher memory bandwidth of the computational accelerators has a significant impact on the performance of the LB code, allowing the accelerators to easily outperform contemporary CPU systems.

The hardware architectures of HPC systems used for these kinds of simulations are briefly presented, as well as what programming methods can be used for these systems. This thesis examines how the usage of the OpenACC programming standard makes it easier to create GPU programs. The benefit of OpenACC compared to CUDA is that it allows the user to add directives into the code. These directives control what part of the execution will be offloaded to the accelerator. In this thesis, there is a description of how OpenACC directives can be applied to an LB solver. We also include a comparison of the performance of that OpenACC solver with a native CUDA solver, both implemented using the same optimization techniques.

For large-scale GPU accelerated systems it is important that any program running on them is able to efficiently utilize the resources. In this thesis, we examine the performance achievable on large-scale GPU accelerated systems running lattice Boltzmann simulations. Specific attention is given to the scalability of our GPU accelerated LB solver on the Titan supercomputer, running on 16384 GPUs in parallel. The highlight of these simulations shows that porous media fluid flow simulations on this system can achieve over 1 petaflops of sustainable computational performance. Basic implementation details such as data layouts and algorithms used are also covered, and the impact they have on the performance is discussed. The results from the large scales simulations show that, even with rather homogeneous porous media samples the workload can become unevenly distributed among the computing units. In this thesis we demonstrate that even a simple recursive bisection scheme, a particular domain decomposition scheme, can effectively improve the load balance for the porous media case used.

The solver used on Titan is implemented using asynchronous communication. This is done to allow the GPUs to continue working uninterrupted while the communication takes place. This thesis discusses how asynchronous communication is handled on GPU systems and the steps needed to allow asynchronous communication while still maintaining memory access patterns that are well suited to the GPU.

Finally, newer processors are deriving more and more of their computational power from SIMD vectorization. The thesis examines the effects vectorization has on an LB solver running on a regular Intel Xeon processor and the manycore Xeon Phi processors. This includes an in-depth analysis of the key optimization methods applied to the code for the Xeon Phi processor. Most of the key optimizations center around how the fast, on-package memory is used. Design choices, such as the data layout and the addition of manual prefetching instructions into the program increases how efficiently the memory bandwidth can be utilized.

Sammanfattning

Denna avhandling fokuserar på lattice Boltzmann metoden (LBM) och hur beräkningsacceleratorer används för att uppnå en hög prestanda för denna typ av simuleringar. Optimeringstekniker för LB simuleringar som körs på normala CPU-baserade system diskuteras också. Den högre minnesbandbredden tillgänglig på beräkningsacceleratorer leder till att kod som kör på dessa lätt klarar av att leverera en högre prestanda än samtida CPU-baserade system.

I avhandlingen presenteras kort hårdvaruarkitekturen som används på dessa typer av högpresterande datorsystem och vilka programmeringstekniker som kan användas för dessa system. En programmeringsteknik som diskuteras mera ingående är OpenACC standarden. OpenACC gör det lättare för programmeraren att skapa kod för beräkningsacceleratorer genom att lägga in direktiv i programkoden. Direktiven beskriver för kompilatorn vilka delar av koden som skall avlastas till beräkningsacceleratorn och hur denna avlastning sker. Denna avhandling demonstrerar hur OpenACC direktiv läggs till ett LB program och jämför prestandan denna version kan uppnå med prestandan av ett CUDA program implementerat med samma metoder.

Stora accelererade datorsystem som använder grafikprocessorer börjar bli allt vanligare. Eftersom skalbarhet är viktigt för stora system, undersöker vi hur vårt accelererade LB simuleringsprogram kan köras på Titan superdatorn på 16384 grafikprocessorer. Resultaten från dessa simuleringar visar att LB simuleringar för poröst material klarar av att prestera över 1 biljard flyttalsoperationer per sekund. Hur grundläggande implementationsdetaljer så som hur data ordnas i minnet och olika typer av algoritmer påverkar prestandan diskuteras också. Dessa storskaliga simuleringar visar att även om dessa porösa material ofta har en homogen struktur orsakar strukturen en obalans i belastningen mellan beräkningsnoder då simuleringarna distribueras över tusentals beräkningsnoder. För att minska denna obalans implementerade vi en rudimentär algoritm för att bättre distribuera belastningen i systemet.

Programmet som kördes på Titan superdatorn var implementerat så att kommunikationen mellan beräkningsnoder kunde genomföras asynkront. Denna asynkrona kommunikation låter grafikprocessorerna fortsätta jobba samtidigt som data kommuniceras med andra relevanta beräkningsnoder i systemet. I avhandlingen diskuteras hur asynkron kommunikation hanteras på system med grafikprocessorer. Samtidigt diskuteras hur uppdelningen av beräkningsdomänen lellan grafikprocessorerna skall göras för att låta programmet fortfarande läsa och skriva data i ett mönster som är optimalt för grafikprocessorer.

Nyare processorer härleder mera och mera av sin beräkningskapacitet från SIMD vektorinstruktioner. Avhandlingen undersöker effekten denna typ av vektorinstruktioner har på ett LB program som körs på både vanliga Xeon processorer samt på Xeon Phi mångkärnsacceleratorer. Detta inkluderar en ingående analys av vilka optimeringstekniker som använts på Xeon Phi systemet för att få maximal prestanda. Speciellt viktigt var hur det snabba minnet används. Hur data ordnas i minnet och hur data manuellt skall läsas in en tid före det behövs är grundläggande för att utnyttja minnesbandbredden i dessa system effektivt.

Acknowledgements

First off, I would like to thank both professor Jan Westerholm and doctor Keijo Mattila, your help has been invaluable both in preparing this thesis as well as the articles presented here. Secondly, I would like to thank professor Erik Lindahl and doctor Derek Groen for as acting as one of the reviewers, additionally I am grateful that professor Lindahl agreed to act as my opponent for the thesis defense. Everyone who worked at the HPC lab during my time have also been very supportive and deserves a mention.

Finally, I would like to thank all my friends and family that have put up with me being antisocial and not having time to do things since I had to "work on my thesis".

Esbo, March 2018
Fredrik Robertsén

List of original publications

I F. Robertsén, K. Mattila, and J. Westerholm. Lattice Boltzmann method on GPUs: a comparison between OpenACC and CUDA. Submitted 2016

II 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 *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 604–609. IEEE, mar 2015

III F. Robertsén, J. Westerholm, and K. Mattila. Designing a graphics processing unit accelerated petaflop capable lattice Boltzmann solver: Read aligned data layouts and asynchronous communication. *The International Journal of High Performance Computing Applications*, 31(3):246–255, aug 2016

IV F. Robertsén, K. Mattila, and J. Westerholm. High-performance SIMD implementation of the lattice-Boltzmann method on the Xeon Phi processor. *Concurrency and Computation: Practice and Experience*, 2018. Minor revision Jan 2018

List of co-authored publications not included in the thesis

- K. Mattila, T. Puurtinen, J. Hyväluoma, R. Surmas, M. Myllys, T. Turpeinen, F. Robertsén, J. Westerholm, and J. Timonen. A prospect for computing in porous materials research: Very large fluid flow simulations. *Journal of Computational Science*, 12:62 – 76, 2016

Contents

1 Introduction **1**
1.1 Author's contributions . . . 3

2 High performance computing **5**
2.1 Hardware trends of modern compute clusters and supercomputers . . . 5
2.1.1 CPUs . . . 6
2.1.2 Nvidia GPUs . . . 8
2.1.3 Xeon Phi . . . 10
2.1.4 Interconnects . . . 11
2.2 Programing models . . . 11
2.2.1 OpenMP . . . 12
2.2.2 Vectorization . . . 13
2.2.3 CUDA programming . . . 14
2.2.4 OpenACC . . . 16

3 The lattice Boltzmann method **19**
3.1 Lattice Boltzmann theory . . . 19
3.2 Algorithms and implementation techniques . . . 22
3.2.1 Lattice site addressing . . . 23
3.2.2 Two-step algorithm . . . 25
3.2.3 Two-lattice algorithm . . . 26
3.2.4 Swap algorithm . . . 26
3.2.5 AA algorithm . . . 28
3.2.6 Algorithm summary and applicability . . . 29
3.3 Simple estimate of the performance potential of the LBM . . . 30
3.4 Arranging the lattice data . . . 32
3.4.1 Data layouts . . . 33

4 Simplified GPU programming with OpenACC **37**
4.1 Lattice Boltzmann GPU implementation . . . 38
4.2 Lattice Boltzmann OpenACC implementation . . . 39
4.3 Performance . . . 41

4.4 Conclusion 44

5 Large-scale parallel computing using the lattice Boltzmann method 45

5.1 Input sample 47
5.2 GPU data layout and algorithm performance 48
5.3 I/O 49
5.4 Load balance 52
5.5 Scalability 55
5.5.1 Weak scaling 56
5.5.2 Strong scaling 57
5.5.3 Scaling difficulties 58
5.6 System utilization 59
5.7 Scalability conclusion 60

6 Asynchronous communication 61

6.1 Dividing the computational domain 63
6.2 Removal of halo sites 64
6.3 Performance of asynchronous communication 65
6.4 Asynchronous communication conclusion 66

7 Xeon Phi and vectorized lattice Boltzmann 67

7.1 Vectorized lattice Boltzmann 68
7.2 Algorithm performance 71
7.3 Thread count and vector length 71
7.4 Prefetching 73
7.5 Data layout 74
7.6 MCDRAM and NUMA modes 75
7.7 Streaming stores 77
7.8 Conclusion 78

8 The lattice Boltzmann method, a petaflop and beyond 81

9 Conclusion 85

Paper I 95

Chapter 1

Introduction

Computational fluid dynamics (CFD) [6], the process of mathematical model-based computer simulation of fluid flows, enables researchers to analyze fluid flow phenomena in great detail. CFD allows hypothetical systems and systems that cannot be observed otherwise to be analyzed. For instance, if an airplane manufacturer wants to know how certain modifications to the wings of an airplane will perform, there is no need to build a scale model of the wings for all the modifications they want to try. Instead the manufacturer can design computer models of the wings and feed them into a CFD program that will simulate how the air moves around the wings. Not only is this easier and cheaper, since no physical manufacturing is needed, it also enables the engineers working on the problem to try countless small modifications to the wings without having to build a new physical model each time. Even though CFD greatly simplifies the design process, no matter how large the computing resources we have are, a simulation can only be as realistic as the underlying mathematical model. Before putting the wing into production there is still some real wind tunnel tests that need to be carried out.

With the need to run multiple simulations, the speed that these simulations can be run at becomes a key factor. Usually, these simulations require powerful workstation computers or clusters of computers working together for the solution to be available within a reasonable time. To achieve the results requires that the resources used are utilized to the fullest and that the simulation code is optimized for the computer system used.

There are fields where the researchers and engineers are trying to do the most with the resources they have at their disposal. A good example comes from the racing world of F1, where to keep costs down, the teams' computing resources are limited to a 25 teraflop system by the rules [7, 8]. That is, they can run a system that can do $25 * 10^{12}$ double-precision floating-point operations per second. This forces the teams and application developers to optimize their codes and the way the resources are used in order to maximize the number of simulations can be done with the limited resources available.

At the other end of the spectrum we have the supercomputer systems used for scientific research, which have grown more and more powerful each generation. The challenge with these systems is to be able to scale the simulation to run on a significant part of the machine, and still run the simulation at a satisfactory performance. Currently, as of November 2017, the largest system in the world is just below 100 petaflops [9]. With faster systems being planned and built, some centers are aiming for exaflops machines, that is machines capable of 1000 petaflops, by 2020 [10]. Scaling programs to these new machines is by no means a trivial task. With the rise of accelerated computing and more customized architectures, even being able to utilize these machines might require large modifications to existing codes.

Fluid dynamics are also used in more time-critical situations. One such example comes from the University College London group that is attempting to use the lattice Boltzmann method to simulate the blood flow in the human brain [11]. The goal is to improve the way aneurysms are treated. For this, they need to be able to run their simulations in conjunction with the treatment, requiring the simulations to be done in a short amount of time.

This thesis covers our work on a highly scalable lattice Boltzmann CFD solver. It describes the work that allowed the solver to efficiently use computational accelerators to significantly increase the performance of the solver, as well as the work carried out that allowed the solver to efficiently use what was, at the time, the second most powerful supercomputer in the world.

The thesis is structured as follows: in chapter 2.1 we present the hardware of modern supercomputers and clusters and discuss how these have evolved over the past years. Chapter 2.2 covers the different programming methods used within the thesis, and gives some basic examples of these methods. The basic theory behind the lattice Boltzmann method is presented in chapter 3 and continues with the algorithms used to turn the theory into program code. A discussion about the factors likely to limit the performance of the solver is also included in the chapter.

Chapter 4 presents the first original research paper and discusses how the programming of GPUs can be made easier using the OpenACC programming standard. Chapter 5 covers how well the GPU-accelerated LB solver could scale on the Titan supercomputer and what to take into consideration when scaling an LB solver to such large machines. Chapter 6 covers how we implemented asynchronous communication and how that improves the performance of the GPU-accelerated LB solver. The use of Xeon Phi many-core accelerators, as well as vectorization of the LB solver, is discussed in chapter 7. Finally, chapter 8 discusses considerations that should be made for large scale LB simulations and how these simulations can be carried out on current supercomputer systems.

1.1 Author's contributions

All the articles in this thesis were made in cooperation with Dr. Keijo Mattila and Professor Jan Westerholm. The author's responsibility was developing the ideas for the articles, implementing and optimizing any code needed and carrying out simulations to gather data for the article. Dr. Mattila contributed with the theoretical background for the lattice Boltzmann method. Both Dr. Mattila and Prof. Westerholm helped in preparing the articles for publication.

Chapter 2

High performance computing

High performance computing refers to the use of parallel computing resources to run programs at a faster speed than a single workstation computer is capable of [12]. These parallel computing resources come in the form of clusters of computers connected with a high-speed network or supercomputers, which are effectively more purpose-built clusters. This chapter covers the hardware used in current HPC systems, as well as the programming methods for these systems.

2.1 Hardware trends of modern compute clusters and supercomputers

The TOP500 list [13] is used to rank the performance of supercomputers around the world. The performance is measured by how many floating-point operations the machines can perform per second while running the Linpack [14] benchmark. All systems currently on the list are based on some form of distributed system with multiple compute nodes connected to each other over a fast network. Most of the systems employ regular server CPUs for their computational power. The most widely deployed ones are Intel's commodity server CPUs, but also processors from AMD, IBM, Sun, or for some machines even completely customized chips are used.

On the CPU side, there has been a steady increase in the core count with each generation. Currently, the most ubiquitous are the 12 core CPUs as of November 2017 [13], while the maximum core count purchasable for an Intel server CPU is 28 cores and AMD offers up to 32 cores. As the core count grows, the frequency tends to drop for each core added [15]. With the lower frequency and higher core counts, parallelism of the code being run becomes a more crucial factor. Any part of the code that cannot be fully parallelized will have its performance hurt by the lower clock frequency. Fortunately, the CPUs still retain higher boost clock frequencies when only a few cores are occupied, which helps serial sections of the code.

The most recent revolution in the HPC world has been the use of computational accelerators, which are used to add additional floating-point performance to the compute nodes. Accelerators usually come in the form of general purpose graphics processors (GPGPU) [16] and manycore processors [17], but also some custom floating-point accelerators are used [18]. These accelerators are specifically designed to deliver substantial amounts of floating-point performance. Since these rely on large amounts of parallelism, and in some cases larger vector units for their performance, it does limit their usage for normal datacenter tasks like for instance, serving web pages.

2.1.1 CPUs

Since the introduction of the first dual core CPU system to the TOP500 list in 2002 [19], the Power4 CPU from IBM, there has been a steady increase in the core counts of the processors used in supercomputers. Adding the fact that the most common configuration currently is to use two or more CPUs per node, the parallelism of a single node has been increasing significantly. With a higher core count, there has also been a slow decrease in the frequency of the processors to keep the power consumption of the processor manageable. This puts a larger burden on the programmers to be able to efficiently parallelize their code to utilize the resources within a node.

The memory bandwidth available for each socket in a system has been increasing with newer revisions of the DDR memory standard and the increase in the frequency of the memory. The number of memory channels connected to each socket also increases the available bandwidth for the CPU. With the introduction of the Skylake CPUs [20], Intel went from 4 to 6 memory channels per socket. The AMDs Zen [21] architecture uses 8 channels per socket, providing up to 341 GB/s of theoretical memory bandwidth for a dual socket system. This is a substantial increase compared to 136GB/s for a dual socket Haswell [22] system and 256 GB/s for a dual socket Skylake system. From the Ivy bridge [23] generation up to the newest Skylake generation of Intel CPUs each socket can support 768 GB or, with some special CPU variants supporting up to 1.5 TB of memory, and AMDs Zen generation CPUs supporting up to 2 TB per socket.

The modern CPU core has evolved to be good at running a wide variety of different codes. As such, the cores of a modern CPU have large and complex cache hierarchies, with three or even four levels of cache between the CPU and the main memory [24]. These caches are significantly faster than the main memory of the system and are used as, among other things,

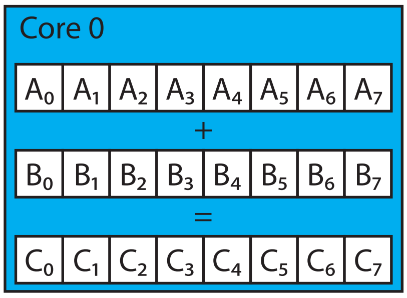

Figure 2.1: An illustration of how SIMD vectorization is used to parallelize an operation. Each vector, A and B, contains 8 values, which are added elementwise into C with one instruction

temporary storage and a buffer between CPU and memory. To reduce the observed latency for reading from the main memory, data is prefetched into the cache hierarchy. The prefetching works by the processor trying to identify the access pattern used by the program and predicting what data will be needed next and moving that into the cache structure ahead of time.

In addition to containing multiple processing cores, modern CPUs use simultaneous multithreading (SMT) [25] to allow multiple threads to run in the same core at the same time. The idea is to have multiple threads that can use any resources that the other threads do not currently use. This allows for a better utilization of the resources within a core. Mainstream Intel and AMD processors currently both support 2-way SMT. The modern CPU cores are also able to perform out of order execution, where the order that the instructions executed are issued is based on when the operands for the operations are available, instead of the order they are in the program code.

To further increase the performance and parallelism of the CPUs, vector instructions in the form of SIMD (Single Instruction Multiple Data) instructions are used [26]. These allow a single core to operate on multiple data values with one instruction as illustrated in figure 2.1. The current generation mainstream SIMD instructions support 256-bit vectors using the AVX2 instruction set [27]. With AVX2, the core can process 4 double-precision floating-point values at once. The current generation of server and HPC specific processors use 512-bit vectors, with the AVX-512 instruction set [28]. The AVX-512 instruction set enables those processors to process 8 double-precision floating-point values with one instruction. To use these instructions, the computation performed needs to be expressed in a form that allows the use of these instructions.

There is some support from the compiler to automatically vectorize the code, but this relies on the compiler to be able to identify which parts of the computation can be vectorized and to guarantee that vectorizing these parts does not change the result of the computation. Guaranteeing vectorization usually involves rewriting the program to explicitly use the vector instructions or at least adding hints for the compiler as to which sections should be vectorized and what data dependencies exist within the program.

2.1.2 Nvidia GPUs

A more recent addition to the HPC field has been the introduction of computational accelerators, hardware designed specifically to accelerate the processing of floating-point operations. As a concept this is nothing new. The original x87 FPU was already a floating-point accelerator. However, modern accelerators in HPC are massively parallel processors that are especially designed for floating-point performance. Currently, the most widely used accelerators in use in the HPC world are GPGPUs, with the clear majority being Nvidia Tesla accelerators.

The GPGPUs have their roots in the processing of real time computer graphics for computer games, rendering the 3D graphics shown on the screen in real time. The rendering methods used for real time computer graphics are massively parallel operations that rely heavily on floating-point arithmetic. Combined with the growing screen resolution and improved graphics quality, this has led to the GPU evolving into a massively parallel processor specialized in floating-point math. The first GPGPU widely used within the HPC community had a few hundred simple CUDA cores, but as on the CPU side, the core count has grown significantly as newer GPUs are released, with current GPUs having over 3000 simple CUDA cores [29].

Even though the GPUs have many cores, these are not the same kind of cores that are found in regular CPUs, the CUDA cores are far simpler. The cache structure on the GPU is also simpler and shallower and mostly used as a coalescing buffer [16]. The GPU cores come with some limitations. The cores cannot all operate independently, which means that they are unable to each run a unique instruction every clock cycle. On the GPU, the cores are arranged into larger groups of cores referred to as a streaming multiprocessor. The Pascal generation uses 64 cores per multiprocessor that can perform single precision operations and 32 cores that can perform double precision operations [29].

The parallelism within the GPU is based on the SIMT (Single-Instruction, Multiple-Thread) paradigm [16], where multiple threads are executing the same instruction at the same time. This is a comparable way to the SIMD vectorization of regular CPUs. The difference from SIMD is that in the SIMT case there are multiple cores performing the operation for the data within the vector, instead of one core running the instruction. In the case

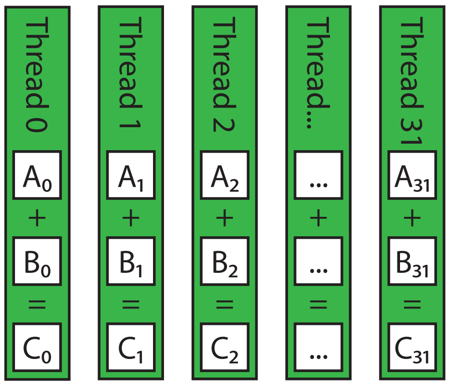

Figure 2.2: An illustration of how multiple threads in a GPU execute the same instruction in different values. Each thread adds together one value, A and B, and stores the result in C

of the Nvidia GPUs, the threads are executed in groups of 32 [16], referred to as a warp. This warp is then executed on the cores within a multiprocessor. All cores that are working on the same warp are executing the same instruction but on different data. If there is divergence in the execution between the threads, the different branches will be serialized. The serialization reduces the performance, since it still requires all cores within the warp to participate [16].

Another feature that the GPUs carried over from the graphics world is that they offer greater memory bandwidth than a normal CPU based system. The recent Nvidia GPUs can deliver over 700GB/s [29] of theoretical memory throughput. The memory bandwidth comes at the price of space, with most of the GPUs used for HPC today having between 12 and 16 GB of on board memory per GPU.

One of the major drawbacks of the GPGPU accelerators is that they are added to the compute nodes as PCI-e or SXM2 [30] add-in cards. Since the GPUs are add-in cards and cannot host themselves, they need a host system to function, effectively a compute node with a normal CPU. As add-in cards, the bus they are connected to limits the amount of bandwidth available for communication between host and GPU, and between different GPUs in the system. In the case of PCI-e cards, the maximum theoretical bandwidth is limited to 16 GB/s [31]. To improve this situation, Nvidia introduced a high-speed interconnect, NVLink [29] capable of 20-25 GB/s of bandwidth per port, which can be used to communicate between GPUs in the system. The NVLink interconnect can also be used to communicate with the host CPU in certain systems where the host CPU supports an NVLink connection [32].

2.1.3 Xeon Phi

The Xeon Phi line of manycore accelerators follows a similar design as the GPUs. They simplify the cores and pack more of them on to the chip. Unlike the GPU, these do not have thousands of cores. The first generation, codenamed Knights corner (KNC) [33], had up to 61 cores. The second-generation Xeon Phi processors, codenamed Knights landing (KNL) [34], have core counts between 64 and 72 cores. To further increase the number of threads that can run the Xeon Phi, it supports up to 4-way SMT. Instead of relying on a large core count for their performance, these use 512-bit wide vectors to achieve a higher floating-point arithmetic performance than regular CPUs can achieve.

The first generation Xeon Phis did not see wide adoption due to the difficulties of getting performance from the programs running on it [35, 36]. The second generation improved on a lot of the issues compared with the first generation. Instead of using a proprietary instruction set like the KNC generation, the KNL generation uses a subset of the AVX-512 instruction set [37]. It can also run code from older vector instruction sets, and in fact the KNL processor can directly run binaries built for regular Xeon CPUs. The KNL generation also moves away from being an add-in card that requires a host system, like the KNC generation, into a self-hosted system where the KNL processor is the only processor in the system [37]. Another substantial change is the move away from a ring bus used as the core-to-core interconnect on the KNC generation, to a mesh network. This mesh network improves the core-to-core communication. Within each node in the mesh network sits two cores that share the same L2 cache and the same connection to the network. The memory controllers, both the high speed MCDRAM and regular DDR memory, are connected to the same mesh.

As with the GPUs, the Xeon Phi processors employ a small amount of fast memory to accelerate memory bound computation. The KNC generation uses GDDR5 memory, the same memory as used by contemporary GPUs. The latest KNL generation uses 16 GB of MCDRAM capable of delivering over 450GB/s [34] of memory bandwidth. A KNC processor is also able to support standard DDR4 memory to enable it to run programs requiring more than 16GB of memory. The MCDRAM on these can be used in multiple modes. One alternative is to use it as a large last level cache that is transparent from the view of the programmer. Data from the MCDRAM cache is accessed at a higher bandwidth than what is available from the DDR4 memory space. Another mode is to use it as a separate memory space, using special allocation functions and only placing certain structures of the program in this faster memory.

2.1.4 Interconnects

Current HPC systems use many technologies that exist in a regular datacenter. The difference between HPC systems and those serving for instance web pages is that the HPC systems are balanced differently. HPC system focus on delivering the maximum floating-point performance in a specific power and size envelope. HPC system also tend to use more advanced networks to connect the nodes to each other. The node-to-node interconnects used for HPC systems are set up to deliver higher bandwidth than those usually found in regular data centers, and are designed to be capable of delivering low latency communication.

The network topology, the way the nodes are arranged and connected to each other in HPC systems, is often done to maximize the amount of bandwidth available between two given nodes in the system. Common topologies include fat tree topologies, cube, and torus topologies but also more exotic layouts such as the dragonfly topology [38]. While the November 2017 TOP500 list does include a little over 200 systems using commodity 10 gigabit ethernet, the top systems all use more exotic interconnects. InfiniBand [39], being the most widely used HPC focused network on the list, and Omnipath [40] networks can both be used to build a cluster with off the shelf components, and are available from multiple server vendors.

Another interconnect option for HPC systems is to use one of the custom interconnects available from some more HPC-focused server vendors. These are available from vendors such as Cray, with their Aries [41] network used in the XC line of supercomputers and the Tofu interconnect used by Fujitsu for the K computer [42]. These custom interconnects attempt to deliver even greater bandwidth and lower latencies to all the nodes, even when scaling up the systems to thousands of compute nodes.

2.2 Programing models

The recent evolution in modern supercomputer systems is the addition of more and more parallelism within a node, instead of scaling up the number of compute nodes, leading to the programming of these systems becoming more complex [43, 44]. Traditionally, HPC codes have relied heavily on the message passing model of parallelism, usually implemented using MPI [45]. This model is based on processes that all have their own memory space and run code serially. The processes communicate with each other by sending and receiving messages. With this model, parallel execution is achieved by running multiple processes. The drawback of this model is that as the number of cores in the system increases, so does the number of processes, making it harder to manage. The amount of communication needed also increases, which can affect scaling. Memory usage is another big drawback of this approach. Since all processes have their own memory space they all need their own copies of any shared memory structures.

MPI cannot be used on GPUs, since the GPUs require other parallelism paradigms. To reduce the amount of MPI communication needed and to enable the use of computational accelerators, HPC programs are moving to a hybrid parallelism model [46, 47, 36]. This means that programs employ MPI together with some other form of parallelism. This second level of parallelism can be accelerator specific languages, to enable the use of accelerators, but also shared memory parallelism [48] within a compute node. This allows the program to cut down on the number of MPI processes needed to efficiently use the resources available.

2.2.1 OpenMP

OpenMP [49] is one of the more popular APIs for shared memory parallelism and most of the mainstream compilers have support for it [50]. OpenMP gives the ability to implement varying degrees of parallelization, such as a simple data parallel for loop, where the iterations of the loop are executed in parallel. OpenMP is also able to do more complex task-based parallelism where threads are spawned with the goal of completing a section of code in parallel with either the main thread or other tasks.

In OpenMP, parallelism is described using simple compiler directives inserted by the programmer into the program code. The use of directives makes it easy to modify existing code without major changes to the code, in the case the algorithms and methods used can be parallelized in that way. Since OpenMP only uses compiler directives to parallelize the code it is still possible to compile the code as a single threaded application. Sections of code we want to execute in parallel are described as basic blocks starting with a `#pragma omp parallel` directive. Within the block, execution is performed in parallel by all active threads, and at the end of the section there is an implicit synchronization guaranteeing that all threads have finished. After the block, only one thread will be left to continue the execution of the program until another parallel section is encountered.

Listing 2.1 illustrates how simple data parallelism can be expressed using OpenMP, in this case the addition of two arrays. The goal in parallelizing this example is that the addition within the loop will be executed in parallel. Memory allocation and initialization has been excluded for the sake of simplicity. The `#pragma omp parallel` directive has been combined with a `#pragma omp for` directive to indicate to the compiler that the iterations of the loop should be parallelized. The work within the loop is then distributed according to the schedule used. In this case the default static scheduling is used, which statically divides the loop iterations by the number of treads. More complex scheduling, such as various kinds of dynamic scheduling, is also available. Dynamic scheduling enables the load of the loop to be more evenly distributed, in case the iterations require a different amount of work. This scheduling gives rise to some additional computational overhead because the distribution of work among the threads itself requires some effort. When parallelizing a loop using OpenMP, it is up to the programmer to guarantee that there is no dependency between the iterations of the loop. Any dependencies could cause race conditions between the threads and thus affect the correctness of the code.

Listing 2.1: Code example that shows how the addition of two arrays can be parallelized with OpenMP

```
double *a, *b, *c;
#pragma omp parallel for
for (int i = 0; i < count; ++i){
        c[i] = a[i] + b[i];
}
```

2.2.2 Vectorization

Vectorization using the single instruction multiple data (SIMD) paradigm refers to the concept of having one instruction, apply the same operation to multiple values at the same time. The length of these vectors varies with CPU type and generation. The widest vectors used in modern x86 CPUs are 512-bit in Xeon Phi manycore processors [37] and the Skylake generation of regular Intel Xeon processors [37]. The 512-bit vectors give the processor the ability to process 8 double-precision or 16 single-precision values with a single instruction.

The vector instructions form a separate instruction set from the regular scalar instructions of the CPU. To use the vector functionality, the compiler needs to generate vector instructions for the code. Ideally the compiler can identify which sections of the code can be vectorized. However, automatic vectorization relies on the compiler being able to identify which sections can be vectorized and that the compiler can guarantee that vectorizing the code will not change the result, something that is not always possible to do. In cases where the compiler is unable to vectorize the code, the programmer may need to modify the code in such a way that the compiler can vectorize it.

One approach to guarantee vectorization is to use OpenMP directives to control what parts of the program are vectorized. Since version 4.0 [51], vectorization can be described in the same way as multithread parallelism is expressed using OpenMP. A `#pragma omp simd` statement can be used to indicate to the compiler that the code should be vectorized. The compiler directives can also be used to circumvent the checks done by the compiler to guarantee correct results of the vectorized code. This is useful in the case where the programmer knows that vectorizing the code will not affect the result but the compiler is unable to detect this. One drawback here is that the vectorization still relies on the compiler to generate the vector instructions, and as such it might not be possible to vectorize computation that requires transformations the compiler is unable to perform.

Finally, vectorized code can be manually generated using intrinsic functions, as shown in listing 2.2. These functions are wrappers to specific machine instructions. Using these guarantees that the execution will be vectorized. Using intrinsic functions has the drawback of making the code seem complex, since the regular operations between operands are replaced with function calls. The intrinsic functions also limit the vectorization to one specific instruction set. One way of improving this is to use a vector math library or a compiler which defines the regular arithmetic operations for the vector types as well. With these vector libraries, vectorized code can be written in much the same way as scalar code, but the code is guaranteed to be vectorized.

Listing 2.2: A code example showing how the addition of two arrays can be vectorized using AVX vector instructions

```
double *a, *b, *c;
for (int i = 0; i < count; i = i+vecLen){
        __m512d aV = _mm256_load_pd(&a[i]);
        __m512d bV = _mm256_load_pd(&b[i]);
        __m512d cV = _mm256_add_pd(aV, bV);
        _mm256_store_pd(&c[i], cV);
}
```

2.2.3 CUDA programming

CUDA was introduced by Nvidia in 2007 [16] as a programming language that allows the use of Nvidia graphics processors for general purpose computing tasks. CUDA gives the ability to easily express the parallelism needed to efficiently offload the computation to a GPU from Fortran and C/C++ code. CUDA is often considered a low-level language when it comes to GPU programming, since it gives the programmer great control over how and what the GPU executes and fine control over what data is transferred to and from the GPU and how it is transferred.

The code that will be run on the GPU needs to be described in terms of CUDA kernels. The kernel contains the code that will be executed by each thread running on the GPU. When converting code that has previously been parallelized with the help of OpenMP, the kernel code is the same code which was in the loop body that was parallelized with OpenMP. The loop iteration variable is replaced with a thread id, computed from the thread block and grid ID numbers.

With the GPU having a separate memory space from the host node, any data used by the kernel running on the GPU also needs to be separately allocated on the GPU. Data used by the kernel must be transferred to the GPU before the computation on the GPU starts, and back to the host once the data is needed by the host computation. With the recent introduction of unified virtual addressing (UVA) [29], the problem of moving data to and from the GPU can be simplified. With UVA, using special allocation functions, we can allocate space in such a way, that we can transparently use the same pointer in both CPU and GPU code. Using this allocation method, the CUDA driver will ensure that the data is currently residing in the correct memory space. Due to the PCI-e bus being slow, and needless shuffling of data back and forth between device and host will significantly impact the performance of any computation offloaded to the GPU. For best performance programs should carry out as much computation as possible with the data already on the GPU before the data is moved back to the host.

Listing 2.3 shows how the code from the array addition example from listing 2.1 can be converted to run on the GPU using CUDA. The major difference from the OpenMP example is the addition of memory transfers between the host system and the GPU device, first copying data from the A and B pointers to the device before the execution, and then copying C back to the host after the kernel has run. The memory allocation for the device memory is also shown in this case since CUDA runtime functions are needed to allocate memory on the GPU. The kernel code shown in listing 2.4 contains the same code that was previously the main loop body. The `blockIdx`, `blockDim` and `threadIdx` variables are predefined in the programming language, and are used to compute the thread index for the current thread.

Listing 2.3: CUDA host code for allocating memory and launching the CUDA kernel on the GPU. In this example the thread block size is set to 256 threads, this is then used to calculate how many thread blocks are needed to create at least the number of threads that there are values in the arrays.

```
double *a, *b, *c;
double *aDev, *bDev, *cDev;
cudaMalloc(aDev, sizeof(float)*count);
...
cudaMemcpy(aDev, a, count);
cudaMemcpy(bDev, b, count);
add<<<256, 1+(count/256)>>> (aDev,bDev,cDev, count)
cudaMemcpy(c, cDev, count);
```

Listing 2.4: CUDA kernel code for adding two arrays. If the thread Id number is higher than the number of elements in the arrays the thread returns immediately to avoid out of bounds data accesses. Otherwise the addition operation between the elements from the two arrays is performed.

```
__global__ void add(double *a, double *b, double *c, int count)
{
        int i = (blockIdx.x * blockDim.x) + threadIdx.x;
        if(i >= count)
                return;
        c[i] = a[i] + b[i];
}
```

When writing the kernels, special consideration must be given to code structure and memory access patterns. On Nvidia GPUs, the threads of a computational kernel are run in groups of 32 threads at the same time, referred to as a warp. On the GPU, all threads within a warp are executing the same instruction at the same time. Ideally, all threads execute the same instruction but on different data, otherwise the execution of the different branches within the warp will be serialized. This does not mean code executed on the GPU cannot include divergence. However, execution of branches will be serialized, which can affect the performance.

Memory accesses follow the same patterns as the other instructions executed by the GPU. All threads within a warp execute a memory access operation at the same time but they have the possibility to access different data. For the best performance, the threads within a warp should access memory in a coalesced fashion. That is, threads within the same warp should be accessing data from a contiguous memory space. While earlier GPU architectures had strict limits on how threads from a warp could access data within a coalesced block, that has now been relaxed and the threads can access data in any order [52]. The optimal sizes for the coalesced blocks that should be accessed has also been relaxed, with the Kepler and newer generations having a load granularity of 32 bytes.

2.2.4 OpenACC

OpenACC [53] aims at giving the programmers the ability to offload computation to accelerators using simple compiler directives like those used by OpenMP. These directives are used to describe what computations should be performed on the accelerator. It also handles the data movement between the accelerator device and the host using similar compiler directives.

Code that contains loops already parallelized with OpenMP parallel directives is a suitable candidate to be offloaded to the GPU using OpenACC. Existing OpenMP loops can easily be transformed to OpenACC offloaded code. The nontrivial part is the data movement. The data used by the computation needs to be identified and moved to the GPU before the computation can start. As with CUDA, once the data has been moved to the GPU, it should be kept there for as long as possible, in order not to make the entire simulation bound by the bandwidth of the PCI-e bus.

Listing 2.5 shows how the previously used array addition example can be offloaded to the GPU using OpenACC directives. The `#pragma acc kernels` directive instructs the compiler that the loop should be executed on the GPU, and the `datain` and `dataout` directives instruct the compiler how to move data between the host and the device. OpenACC will perform dependency checks to ensure that the code can be parallelized with no side effects. These checks can, however, be overridden, but it is then up to the programmer to guarantee that there is no dependency between loop iterations.

Listing 2.5: Code example that shows how the addition of two arrays can be offloaded to the GPU using OpenACC

```
double *a, *b, *c;
#pragma acc kernels loop datain(a,b) dataout(c)
for (int i = 0; i < count; ++i){
        c[i] = a[i] + b[i];
}
```

Even though OpenACC allows computation to be easily offloaded to the GPU, it will still not transform the code for optimal execution on the GPU, that task still resides with the programmer. In practice, there is still a need to optimize the code to run on the GPU. For instance, the programmer should make sure that the data access pattern is optimal for the GPU and that the code follows the execution model of the GPU. In general, OpenACC code should be written using the same basic ideas as those presented in the CUDA programming section.

Chapter 3

The lattice Boltzmann method

The lattice Boltzmann method [54] is a method for computationally simulating fluid dynamics. It has been used for simulations of fluid phenomena at many scales, from microscopic porous media flows [5], to blood flows in vascular systems [55] and large even aerodynamics dimulations [56]. It can be used for both single and multicomponent fluid simulations, and allows even more complex phenomena with particle suspensions and liquid crystals to be simulated [57]. The method is well suited for parallel computation and has been demonstrated to work well on distributed clusters utilizing regular commodity CPUs [58], as well as on more specialized hardware with GPUs [59].

This chapter covers the theory behind the lattice Boltzmann method as well as the practical implementation aspects such as the different algorithms that can be used for implementing the method. The chapter also includes a simple estimation of the performance achievable with the different algorithms and a discussion on how the data associated with the lattice sites can be arranged in memory.

3.1 Lattice Boltzmann theory

The lattice Boltzmann method works by discretizing the entire simulation domain with a regular lattice, with the kinetic model equation for the fluid approximated only at the lattice sites. Within each lattice site, particle velocity space is further discretized into a finite set of velocities. The 2D example shown in figure 3.1 uses 9 discrete velocity vectors per lattice site. This is referred to as a D2Q9 discrete velocity set [60], all except the center one of these point to a neighboring lattice site. The DdQq notation is commonly used to annotate a discrete velocity set, with d signifying the

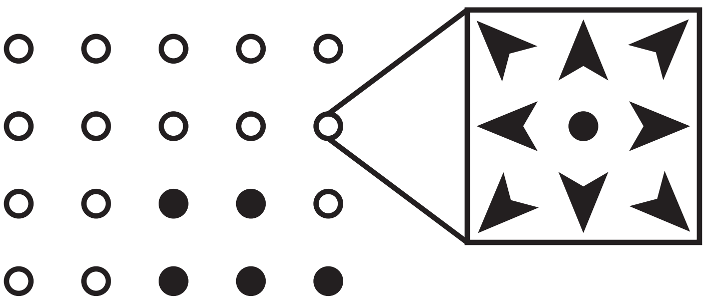
Figure 3.1: 2D illustration of the discrete representation of the domain and the velocity vectors in the D2Q9 discrete velocity set within each lattice site

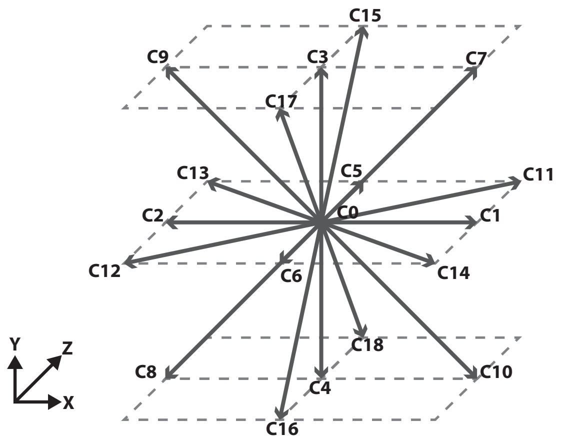

Figure 3.2: An illustration of the D3Q19 velocity set.

number of dimensions and q the number of discrete velocity vectors used. In our simulations, we use the D3Q19 discrete velocity set [60].

In the lattice Boltzmann method, the dynamic variable, $f_i(\vec{r}, t)$, is the single particle distribution function. This describes the probability of finding a particle at location $\vec{r}$ at time t with velocity $\vec{c}_i$. The D3Q19 velocity set, illustrated in figure 3.2, $\vec{c}_i$ are defined as

$$\vec{c}_i = \begin{cases} (0,0,0)c_r, \\ (\pm 1,0,0)c_r, \quad (0 \pm 1, 0)c_r, \quad (0,0 \pm 1)c_r, \\ (\pm 1, \pm 1, 0)c_r, \quad (\pm 1, 0, \pm 1)c_r, \quad (0, \pm 1, \pm 1)c_r, \end{cases} \tag{3.1}$$

with $c_r = \Delta r/\Delta t$, Δr is the spacing of the lattice and Δt the discrete time step. The dynamics of the system is described by the lattice Boltzmann equation

$$f_i(\vec{r} + \Delta t \vec{c}_i, t + \Delta t) = f_i(\vec{r}, t) + \Delta t \Omega_i\big(\vec{f}(\vec{r}, t)\big), i = 0, 1, ..., q - 1. \tag{3.2}$$

Each time step is decomposed into two operations, the propagation and relaxation operations. The relaxation is defined as

$$f_i(\vec{r}, t^*) = f_i(\vec{r}, t) + \Delta t \Omega_i\big(\vec{f}(\vec{r}, t)\big) \tag{3.3}$$

with Ω being the collision operator and $f_i(\vec{r}, t^*)$ denoting the post relaxation values, and the propagation by

$$f_i(\vec{r} + \Delta t \vec{c}_i, t + \Delta t) = f_i(\vec{r}, t^*). \tag{3.4}$$

The macroscopic fluid density ρ for a lattice site is computed from

$$\rho(\vec{r}, t) = \sum_{i=0}^{q-1} f_i(\vec{r}, t). \tag{3.5}$$

The macroscopic flow velocity $\vec{u}(\vec{r}, t)$ is computed by using

$$\vec{u}(\vec{r}, t) = \frac{1}{\rho(\vec{r}, t)} \sum_{i=0}^{q-1} \vec{c}_i f_i(\vec{r}, t). \tag{3.6}$$

A commonly used collision operator is the Bhatnagar-Gross-Krook collision operator (BGK) [60]:

$$\Omega_i = -\frac{1}{\tau}\Big(f_i - f_i^{eq}\Big) \tag{3.7}$$

where τ is the relaxation time related to the kinematic viscosity $v = c_s^2(\tau - \Delta t/2)$, and f_i^{eq} is the equilibrium function defined as

$$f_i^{eq} = w_i \rho\Big(1 + \frac{\vec{c_i}\vec{u}}{c_s^2} + \frac{(\vec{c_i}\vec{u})^2}{2c_s^4} - \frac{\vec{u}^2}{2c_s^2}\Big) \tag{3.8}$$

The weights, w_i, are defined as

$$w_i = \begin{cases} 1/3, & i = 0, \\ 1/18, & i = 1, 2, 3, 4, 5, 6, \\ 1/36, & i = 6, 7, ..., 17, 18. \end{cases} \tag{3.9}$$

For isothermal flows, c_s is the speed of sound in the lattice and, for D3Q19, is defined as $c_r/\sqrt{3}$. For porous media simulations, the more complex two relaxation time (TRT) collision operator [61] is widely used. For the TRT operator, Ω_i is defined as

$$\Omega_i = -\frac{\lambda_e}{2}(f_i^{neq} + f_{-i}^{neq}) - \frac{\lambda_o}{2}(f_i^{neq} - f_{-i}^{neq}), \tag{3.10}$$

where $f_i^{neq} = f_i - f_i^{eq}, \vec{c}_{-i} = -\vec{c}_i$, and the even parameter $\lambda_e = 1/\tau_e$, τ_e is determined by the value of kinematic viscosity, $v = c_s^2(\tau_e - \Delta t/2)$, and the odd relaxation parameter we assign using $\lambda_o = \lambda_o^*/\Delta t = 8(2 - \lambda_e^*)/(8 - \lambda_e^*)$.

Lattice sites that represent solid geometry are not assigned any distribution values. For values that would be propagated from these solid sites, a boundary condition needs to be applied. The simple boundary conditions commonly used are the bounce back and bounce forward boundary condition. In this work, we use the bounce back boundary condition [62]. In practice, this boundary condition involves turning around any distribution value moving into a solid site into the opposite direction, and placing them back into the same lattice site they originated from.

3.2 Algorithms and implementation techniques

The propagation step of the LBM algorithm introduces a dependency between the neighboring sites in the lattice. When moving the distribution values from the current lattice site to the neighboring sites, the distribution value needs a location to be stored on the neighboring lattice site. The natural place is the location corresponding to the same distribution value on the neighboring sites. But then those distribution values need to be moved to the neighbor of the current neighbor and so on, until the propagation reaches the edge of the domain.

There are various algorithms that address this inherent dependency between lattice sites. These approaches all have varying degrees of suitability for parallel computation, different memory space requirements, and varying performance on different architectures. Traditionally parallelism has been described using the message passing paradigm. Each core is handled as a separate process, which sets up its own small domain and runs the simulation for this domain only, with parallelism handled by communicating with its neighboring processes. However, on modern supercomputers, this approach quickly becomes unsustainable with machines that have hundreds of thousands of cores. Also, assigning a separate process to each core will increase the amount of communication needed and the memory usage of the program, since some data structures are often replicated on each process. Some MPI overhead also grows as the number of cores used grows [63]. While it is possible to run MPI with many ranks for large systems, some form of multi-level parallelism is often useful.

For accelerated systems, especially with GPU accelerators, it is not possible to handle every core within the processor as a separate process. The GPU programs need to be structured in a way to allow the processor to iterate over the lattice sites in any order, as there is no method of controlling the order that parallel threads on the accelerators are executed. Also, using shared memory parallelism, such as OpenMP, having the ability to iterate over the lattice sites in any order simplifies the way the code can be parallelized on regular CPUs. Finally, since modern CPUs derive a lot of computational performance from SIMD type vectorization, using these vectors means the solver will have to be able to do multiple site updates at the same time in a SIMD fashion. A better approach then is to distribute the program over multiple nodes using the MPI paradigm, with one MPI-process per node or accelerator. The parallelism within the nodes is then handled by another level of parallelism, be that shared memory parallelism with OpenMP or offloading the computation to an accelerator using CUDA or OpenACC.

3.2.1 Lattice site addressing

When applying the propagation to the distribution values in the lattice sites, there needs to be some way to compute where the neighboring lattice sites reside in memory, as data from the current lattice site will be moved to the neighboring sites. The simplest way to handle this would be to allocate a three-dimensional array, the size of the bounding box of the simulation geometry used. The location of the neighboring lattice sites can then be computed by looking at the coordinate of the current lattice site and adding the corresponding offset for the current distribution value. This approach would be a direct addressing scheme for the lattice sites.

The direct addressing scheme, while simple to create and use, has some drawbacks, the need to allocate space for all the lattice sites within the bounding box of the geometry being the biggest. Since the samples used for this work consists of a porous material, a significant percentage of the volume consists of solid lattice sites, for instance the main sample has only 13 % fluid sites. Using the direct addressing scheme would force the solver to allocate a lot of memory that will not be used for the fluid simulation, and it would severely limit the amount of simulation data that is able to fit into a compute node.

Additionally, allocating space for the solid sites has an adverse effect on the performance of the solver. The performance impact comes from the way memory accesses through the cache structure work on a modern processor. On a hardware level, if the program requests a single value, that memory access is not done for an individual value. In case the value requested does not currently reside in the cache, the processor will always fetch an entire cache line instead of a single value. In the case where there are unused, solid, sites in the lattice, when a cache line containing data associated with such a site is requested by the processor, this unusable data is moved into the cache structure. Fetching data into the cache that is not used by the computation is wasting bandwidth, since there will be no computation done with it.

To avoid the performance degradation associated with unused lattice sites and to save memory space, the LB solver used here is implemented using indirect addressing. In practice, indirect addressing requires data to be allocated only for fluid site. This is done by precomputing the indices used for the propagation targets for all the distribution values before the simulation starts. During the simulation, these indices are read from an array containing the propagation target indices for all lattice sites. With the indirect propagation, we can implement almost any conceivable data layout for the distribution values, and the lattice data can be arranged in any order.

The downside of the indirect scheme is that it requires some additional memory space for the indexing values, as well as consuming extra memory bandwidth when reading the indexing values. Ideally, only one set of propagation indices is needed for all but the center distribution value in each lattice site. For simulations where 32-bit indices are sufficient, the solver would need $(Q-1)*32$ bits of space for the indexing values per lattice site. When using porous media, or another media with a large amount of solid lattice sites in the simulation geometry, the extra memory space required is quickly offset by the saving coming from not storing solid lattice sites.

The performance impact of using indirect addressing will depend on the algorithm used. Ideally, the distribution values will only be read and written once per iteration. In the case where double-precision values are used for the distribution values, $2*64*Q$ bits ideally need to be accessed for the distribution values while only $(Q-1)*32$ bits are needed for the indexing. The result is that the theoretical performance impact of using indirect addressing should only be around 24% lower than in the case we run a fluid only simulation.

For porous media, using direct addressing will have a significant performance impact, in addition to the wasted memory space from storing all the solid sites. Since some of the cache lines fetched will include data that will not be used by the simulation, it will waste bandwidth that could be used for actual simulation data. Figure 3.3 shows result from testing on an Nvidia Tesla M2050, placing solid sites at random locations to achieve a certain percentage of solid sites. The performance is measured in millions of fluid lattice sites updated per second, MFLUPS. We only count the fluid sites since no computation happens at the solid sites, and in fact since we are using indirect addressing, we do not even allocate memory for them. At roughly 10% of the lattice volume filled with solid sites, direct and indirect addressing performed the same [64]. Increasing the percentage of solid

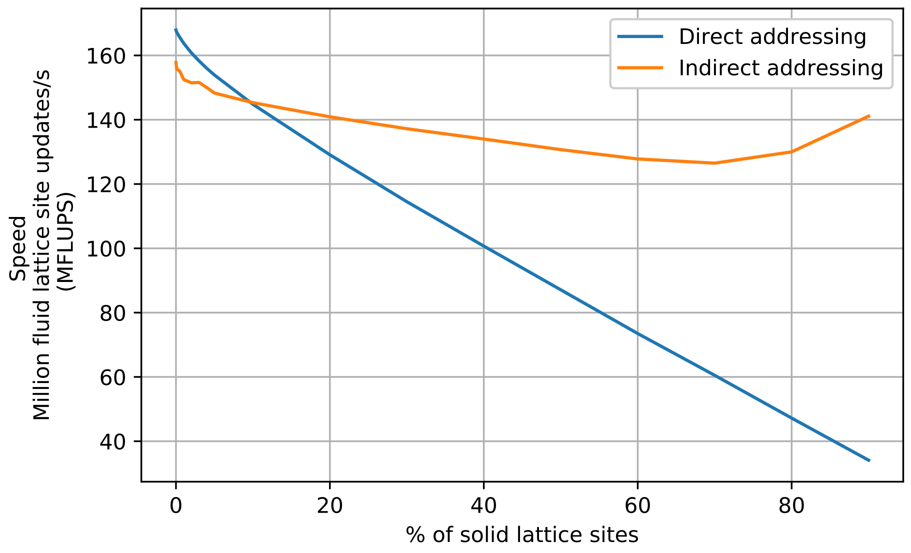

Figure 3.3: The performance at different percentages of solid sites, compared to the performance of the direct and indirect addressing schemes on a Tesla C2050 GPU. Solid sites are placed at random into the simulation domain.

sites in the lattice will have the performance of the direct addressing scheme falling further in a linear relation to the number of solid sites. The indirect variant will see a declining performance while going towards 70% solid sites, and a performance increase when going to a more solid simulation geometry.

3.2.2 Two-step algorithm

The two-step algorithm is one of the well-known implementation algorithms for LBM [65]. The algorithm works by iterating over the lattice two times. The first iteration is to apply the relaxation to each lattice site and the second is to perform the propagation of values from site to site. The relaxation can be trivially parallelized, but the propagation needs to be done in a carefully selected order, in order not to introduce race conditions into the computation [66].

The benefits are that this algorithm only requires one location in memory for the distribution values associated with a lattice site and one set of indices for the propagation. This method is memory bandwidth intensive, as it requires the program to iterate over all lattice sites twice, while the propagation step also requires the indices for indirect propagation.

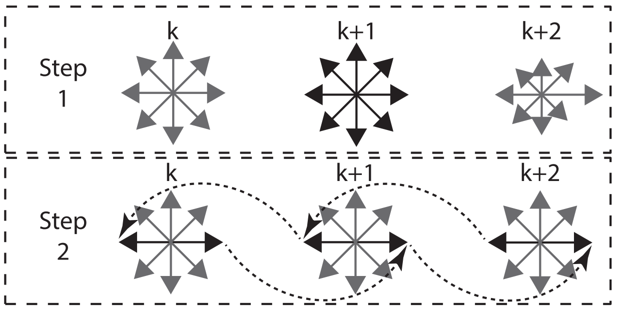

Figure 3.4: Two step algorithm illustrated using a D2Q9 example. The site update happens in two discrete steps, one applying the relaxation the other the propagation.

3.2.3 Two-lattice algorithm

The second well-known algorithm for implementing the LBM is the two-lattice algorithm [65]. With this algorithm, the propagation step can be parallelized by having each lattice site being associated with two memory locations. In this case, it is unnecessary to iterate over the lattice site twice. The program simply iterates over the lattice sites once, applying both the relaxation and the propagation at the same time in a fused operation. For the propagation on even time steps, data is read from lattice A and written to lattice B after the relaxation and propagation, and then for odd time steps the procedure is reversed, reading from lattice B and writing to lattice A, as illustrated in figure 3.5. The propagation can be done in one of two ways. If the propagation is done before the relaxation, it is a so-called pull scheme, and if the relaxation is done after the propagation, it is a push scheme. Since there are no dependencies between lattice sites, this algorithm can be trivially parallelized with the post relaxed distribution values being written to a completely different location than they are read from.

Iterating over the lattice sites once significantly reduces the required memory bandwidth. The downside of this approach is that it requires two memory locations for the distribution values within a lattice site, which doubles the memory space required for the distribution values.

3.2.4 Swap algorithm

Another way to handle the inherent data dependencies in the LBM is the swap algorithm presented by Mattila et al. [67]. The swap algorithm has the benefit of only needing one storage location for the distribution values. At the same time, it is possible to fuse the relaxation and propagation such that the lattice can be iterated over just once each time step.

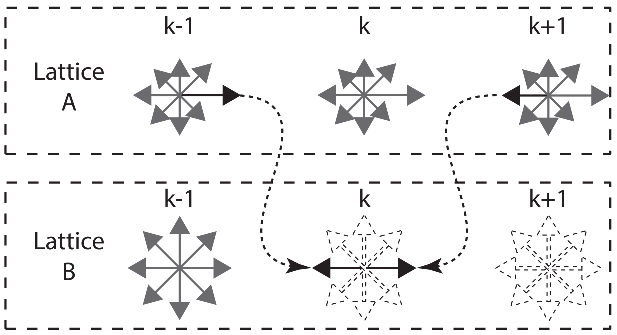

Figure 3.5: Two lattice algorithm illustrated using a D2Q9 example. At each site, the values are read from one lattice and written to another. Once all sites have been updated, the lattices are swapped for the next time step.

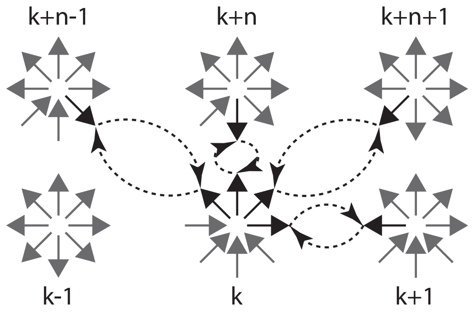

Figure 3.6: The swap algorithm illustrated using a D2Q9 example. The values propagating to sites not visited for this time steps are swapped with the values from the non-visited lattice sites.

The novelty of the swap algorithm is that the lattice sites need to be iterated over in a very specific order, starting from one corner updating one site at a time, and moving to the next. At each site, depending on if the solver is using either the pull or the push scheme, the propagation is first applied to all distribution values that will propagate to a site the iteration has not yet visited. These propagations are done by swapping the values with the opposing distribution value for that site, as illustrated in figure 3.6. Once the values have been swapped, the relaxation is applied to all values in the current site, alternatively, if the push scheme is used, the order of relaxation and propagation is reversed.

Since the algorithm requires the sites to be iterated over in a very specific order, it makes parallelizing the swap algorithm hard. The only way is to divide the problem into as many parts as cores in the system, and run the parts of the simulation as a single thread on the cores, i.e. pure MPI parallelism or similar task-based parallelism. This also limits the vectorization

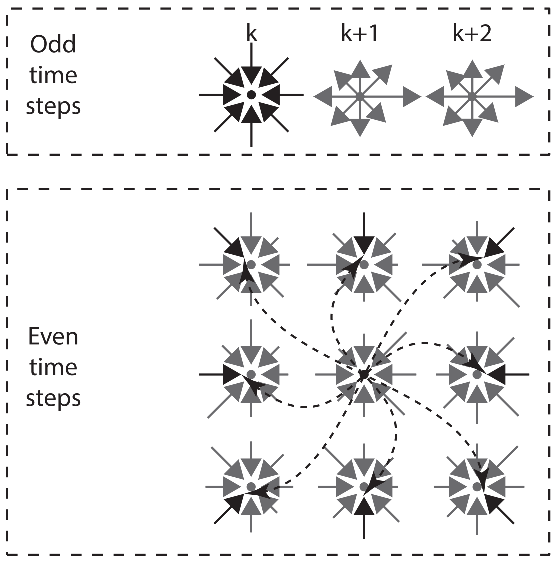

Figure 3.7: A-A algorithm illustrated using a D2Q9 example. At odd time steps no propagation is applied, and at even time steps propagation is applied both when reading and writing the lattice data.

that can be done to vectorizing a single lattice site instead of updating multiple sites at once. With these restrictions, it is hard to achieve the massive levels of parallelism needed for efficient utilization of GPUs.

3.2.5 AA algorithm

The AA pattern introduced by Bailey et al. [68] allows the lattice sites to be updated in any order and for this reason the solver can be easily parallelized. This comes with the requirement to have just one storage location for the distribution values associated with each lattice site. Finally, it allows for a fused relaxation and propagation implementation, and therefore the values for each lattice site need to be accessed only once every time step.

The basic idea behind the AA pattern is to handle the propagation differently between odd and even time steps. At odd time steps only the collision is applied. As illustrated by figure 3.7, the distribution values are written back to the opposing locations in the local site. At even time steps, the propagation is applied twice. First, the values needed for the current lattice site update are pulled from the neighboring sites, then the relaxation is applied, and afterwards the newly updated values are pushed out to the neighboring lattice sites. Due to how the odd and even time steps interact, the values that are pulled in at the start represent the distribution values for the opposite directions from which they are read.

Since the site update for both the odd and even time steps only use distinct memory locations, there are no race conditions introduced in the site update code. This means that the updates can be executed in any order. The solver also only needs one location for the distribution values. Another benefit is that the same memory locations are used for both read and write, which can, in some architectures, improve the performance. This performance increase comes from the fact that the values are written back into memory segments that may already be in the cache.

The indexing values used for the propagation at the even time steps are the same for both the pull propagation and the push propagation. Since both propagations use the same values, only one set of propagation indices are needed. Additionally, since the propagation indices are only needed at even time steps, it saves some memory traffic as the indexing variables are read only at the even time steps.

3.2.6 Algorithm summary and applicability

The algorithms presented thus far all have varying degrees of suitability for parallel computation. All of them can be parallelized if each core in the system would be handled as a separate task using the pure MPI based paradigm. But such approaches are difficult to scale and not applicable to running on computational accelerators. Finally, it also limits us to vectorizing the site update for a single site rather than vectorizing over multiple sites. The algorithms presented also have different memory space requirements and different memory bandwidth requirements. The bandwidth being one of the major factors in the performance of the LB model, the amount of memory bandwidth needed is a good indicator of the performance of the different algorithms. This is discussed further in chapter 3.3.

Table 3.1 shows a summary of the memory bandwidth requirements as well as the memory space requirements for the presented algorithms. The table also includes an estimate of the number of bytes needed for an ideal site update with the D3Q19 discrete velocity set. This estimate is made assuming double-precision, 8-byte values are used as distribution values and 4-byte integers as indexing values. These memory bandwidth numbers do not account for any cache reuse that might reduce the actual data moved, nor does it account for some architectures having to read in a cache line before it can be modified. The table also does not account for extra data that might get accessed when applying the boundary conditions. Regarding the memory space requirements for the solvers there are only two options. The algorithm relying on two lattices will require twice the amount of space to store the distribution values and the others will only require one location for the distribution values. The space needed for the indexing data is the same for all algorithms.

Table 3.1: Memory space usage and memory bandwidth requirements for the different algorithms, assuming no cache effects. It is assumed that integers are used as index values.

Algorithm	Memory space (Bytes)	Memory bandwidth per site update (Bytes)	Bytes accessed per D3Q19 site update
Two-lattice	2*Q*sizeof(f_i)+ (Q-1)*sizeof(int)	2*Q*sizeof(f_i)+ (Q-1)*sizeof(int)	376
Two-step	Q*sizeof(f_i)+ (Q-1)*sizeof(int)	2*2*Q*sizeof(f_i)+ (Q-1)*sizeof(int)	680
AA	Q*sizeof(f_i)+ (Q-1)*sizeof(int)	2*Q*sizeof(f_i)+ ((Q-1)/2)*sizeof(int)	340
Swap	Q*sizeof(f_i) + (Q-1)*sizeof(int)	sizeof(f_i)*((3*Q)-1)+ sizeof(int)*(Q-1)/2	476

The two-step algorithm iterates twice over the lattice data and will require more memory bandwidth, which will have a significant impact on the performance. This makes it less attractive for a high-performance solver. The swap algorithm, although it has low space requirements and does not require the lattice sites to be iterated over twice, still requires the solver to read some values twice. It is also significantly harder to parallelize without resorting to a task based parallelism model. This difficulty is due to its reliance on iterating over the sites in a specific order. Algorithms that can only be effectively parallelized using task based parallelism are less attractive for massively parallel systems like GPUs and vectorization.

The two-lattice algorithm is an attractive algorithm, since it does not come with a high memory bandwidth requirement and is easy to make massively parallel, but it has the downside of a larger memory space requirement. Finally, the AA algorithm appears to be the best of both worlds. It is easy to parallelize, does not require two storage locations for the distribution values and does not require us to iterate twice over the lattice data. Additionally, since the solver only needs the index values every second time step, the solver has the possibility to perform better than the two-lattice algorithm, and as such it appears to be the optimal one of the presented algorithms.

3.3 Simple estimate of the performance potential of the LBM

The performance of the LBM depends on the amount of memory bandwidth required by the solver. This can be determined by a simple analysis of the algorithm used. The easiest way to highlight this is to look at the arithmetic

Table 3.2: Arithmetic intensity for some of the architectures discussed in this work.

Architecture	Theoretical peak GFLOPS (64-bit)	Bandwidth (GB/s) stream triad	Arithmetic intensity (flops/bytes)	
Tesla K20x	1312	182	7.2	[71]
Tesla P100	4670	550	8.5	[4]
Xeon Phi 7210	2253	436.6	5.2	[4]
Xeon E5-2698v3	1178	116	10.2	[71]

intensity, flops per bytes of data accessed [69] by the solver. The arithmetic intensity of the solver can then be compared to the flops per bytes of memory bandwidth provided by the architecture the solver is running on to obtain an estimate of the maximum performance achievable.

The BGK relaxation operator for the D3Q19 velocity set involves approximately 160 floating-point operations per site update [70]. The TRT operator for the same velocity set is more complex and involves around 213 to 257 floating-point operations per site update, depending on implementation [5]. Using the AA algorithm, which has the lowest bandwidth requirements, and the upper limit of the reported floating-point operations for the TRT relaxation function, the arithmetic intensity is only 0.76 floating-point operations per byte of data accessed.

Table 3.2 shows the theoretical peak floating-point performance and the stream triad benchmark bandwidth [72] of the architectures utilized within this work. While achieving the theoretical peek floating-point performance might seem far-fetched, programs that rely heavily on linear algebra can often achieve close to the theoretical peak performance of the systems. The most efficient systems on the TOP500 list are achieving over 90% of their theoretical peak performance [13].

The stream benchmark is often used to measure the realistic memory bandwidth a system can deliver. The idea behind the stream triad benchmark [72] is to read data from two different arrays, A and B, multiply the value from B with a scalar K, add that result to the value from A, and store the result into array C. The sizes of the arrays are chosen in such a way that the data cannot fit into the cache structure of the processor. The operation performed by the benchmark includes only two arithmetic operations, and it reads two values and stores one. The scalar K should be accessible from the cache and does not contribute to the bandwidth used. The stream benchmark is a good indication of the type of bandwidth achievable for a bandwidth bound code, if the solver can efficiently use all the data moved in from main memory.

Table 3.2 shows that, for all the presented processors and accelerators, there is significantly more floating-point performance than memory bandwidth available. Using the TRT relaxation operator as a comparison, it is clear that the available flops to bytes on the presented architectures are over an order of magnitude more than required. To achieve the theoretical peak arithmetic performance of the systems, the solver would have to do 40 to 100 floating-point operations with each double-precision value accessed.

3.4 Arranging the lattice data

Different processor architectures have different requirements for how data is accessed. Most modern processors have some form of cache structure that acts as a buffer between the memory and the processor. The idea behind these caches is to have a smaller high speed lower latency memory space closer to the cores performing the computation. Modern CPUs can have up to three or four levels of caches, some which are shared across the entire processor, and others which are distinct to each core. Generally, caches closer to the cores offer a higher performance at the cost of size, with the smallest caches being the closes ones to the individual cores.

Data is transferred to and between the caches in fixed size segments, referred to as a cache line. The access is done in segments, and even if a single value is accessed, an entire cache line is always brought into the cache structure. For Intel x86 architectures 64-byte cache lines are used, so even if only a single byte is read from memory, there will still be a full 64-byte cache line moved into the CPU. With the caches being significantly smaller than main memory, all data cannot be kept in them indefinitely, so as newer data is moved in, the older data eventually will be evicted from the cache.

On Nvidia GPUs, the L1 cache line length is 128-bytes, but on newer GPUs the loads are only cached in the L2 cache, which has a granularity of only 32-bytes [52]. The GPUs are designed to use the caches more as a coalescing buffer. This allows multiple threads from the same warp to access their part of the cache line, allowing for spatial reuse of cache data. Since the caches are smaller and the GPUs run significantly more active threads than a regular CPU temporal data reuse is discouraged.

When programming for architectures with caches, the data accessed by the program should be arranged in such way that the program is able to use all the data that is transferred into the processor. Any data reuse from the caches should be done before the data is evicted from the cache. The way the data is arranged has a significant impact on how the processor can use the data brought in. The differences in the way the data accesses work results in some architectures benefiting from data arranged in one way, while other architectures benefit from other ways.

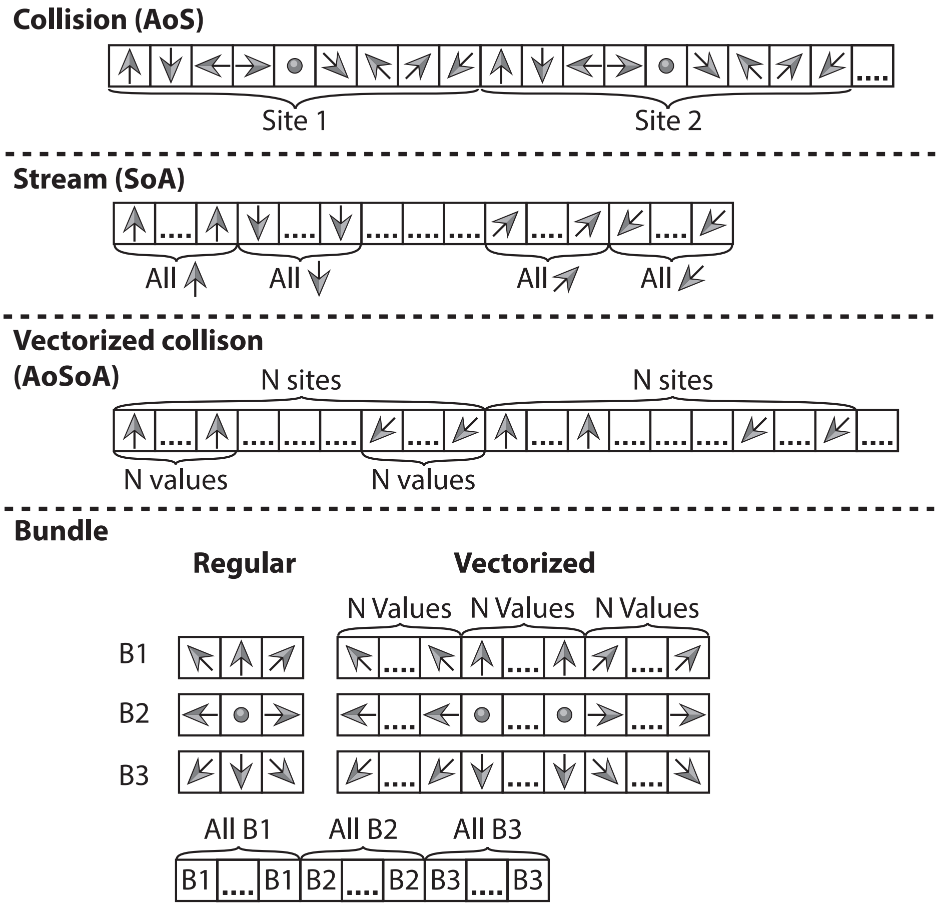

Figure 3.8: Illustration of the data layouts used. Topmost the collision optimized, next the stream optimized. Third is the vectorized collision optimized and at the bottom the bundle layout and how it can be vectorized.

3.4.1 Data layouts

The two elementary data layouts used for the LB method are the so-called stream optimized and the collision optimized layouts [73]. These layouts represent textbook examples of a structure of arrays (SoA) and an array of structures data layout (AoS), respectively. Both layouts come with their separate strengths and weaknesses, which can vary depending on the hardware architecture used.

The collision layout, illustrated first in figure 3.8, has the benefit of placing all values associated with a given lattice site next to each other in memory. This makes manipulation of one lattice site easier. This layout also guarantees that the values for one single lattice site will fit into as few cache lines as possible. This in turn raises the chance that, when processing one lattice site, the next distribution value needed is delivered from a lower latency cache. The downside is that if a thread wants to access the value for a specific distribution value for multiple lattice sites, these accesses get strided in the memory. That is, the distribution values associated with a lattice site are separated in memory by a fixed distance, or stride. These strided accesses are detrimental to architectures such as GPUs, where multiple threads should be fed from the same cache lines, since cache lines accessed will include a lot of data that the threads are not necessarily able to use. These strided accesses are also detrimental to vectorized LB implementations, which cannot access contiguous vectors from the memory and need to assemble the data from strided memory locations.

The stream optimized layout places values representing the same distribution value for different lattice sites next to each other in the memory, as shown in figure 3.8. All values representing a certain distribution value for different lattice sites are placed in contiguous memory locations. If a single thread wants to access the values for one lattice site, the thread now needs to do strided accesses. However, when accessing data for multiple lattice sites, such as for a vectorized solver or GPUs, these accesses can be done from contiguous memory space, which is the far more efficient access pattern for these types of architectures.

There is no real limitation to how the lattice data can be arranged. If the programmer is willing to keep track of this, one could propose more complex data layouts as well. One such more complex data layout is the bundle data layout [66], also illustrated in figure 3.8. This data layout groups together certain distribution values into bundles, for instance for a D2Q9 discrete velocity distribution the bundles could be constructed as illustrated in figure 3.8. The idea behind the bundle is to increase the locality of the data and thus reduce the number of cache misses encountered by the processor running the simulation.

Data layouts such as the collision optimized and bundle layouts have the drawback that they will not perform well on GPUs or vectorized solvers, since they do not conform to their data access requirements. These layouts can however be modified by arranging the data to conform to the access requirements for these architectures. For the array of structures type collision optimized data layout, instead of constructing the inner most structure from data associated with a single lattice site, the structure should be constructed from multiple lattice sites. The number of lattice sites placed into each structure should be the number of values that should be accessed at the same time by a given architecture, or a multiple of the number of lattice sites required. Then, instead of storing single values in the structure, arrays of the size of the required number of lattice sites that are packed into the structure are stored. These arrays should contain the same distribution value from each lattice site as in the stream optimized data layout. This would effectively be an array of structures of arrays (AoSoA) data layout, or a vectorized collision data layout. Figure 3.8 illustrates how to construct this data layout. A similar approach to the vectorized collision layout can be used for the bundle layout. Again, instead of storing single scalar values, the bundles should be constructed of arrays with a size that is a multiple of the required number of sites that need to be accessed at the same time.

A collision optimized layout that is optimized for coalesced access would potentially be the optimal solution. This offers the locality benefits of having all data associated with a lattice site close to each other in memory, as well as being suited for coalesced accesses are required by the vectorized and GPU based solvers.

Modern CPUs are very versatile and can handle a wide variety of different access patterns. They can also reduce the impact of suboptimal access patters though prefetching and multi-level cache structures with large caches. As such, a scalar solver performs similarly with many data layouts in the case that any unused data brought in for the update of one lattice site is reused by the update of another. The large caches help, since they allow unused data to stay in the cache structure longer, increasing the chance some other lattice site update needs this data.

The GPUs benefit from data accesses where the running threads access data from the same coalesced data segment. To get the optimal memory bandwidth out of the system, the threads executing at the same time should be accessing adjacent data. For this, the memory layout should be either the stream optimized one or the vectorized version of the collision or bundle layouts.

With vectorized solvers running on regular CPUs, the data layouts become more important. This is due to the vectorized solvers updating multiple lattice sites at the same time. As such, the basic collision optimized layout is no longer feasible, as it would require the values for the vectors to be assembled from scattered memory locations. This leaves the stream and vectorized layouts as the only valid options for a vectorized lattice Boltzmann solver.

The manycore Xeon Phi processors rely heavily on vectorization for their performance. The data layout used for these should be like the vectorized CPU solver and the GPU solvers, with different vector lengths. Layouts such as stream optimized and the vectorized collision and vectorized bundle layouts are also suitable, with values from at least 8 lattice sites placed in the same structure.

Chapter 4

Simplified GPU programming with OpenACC

With GPUs offering more memory bandwidth than CPU-based systems, the GPUs can offer better performance for LB solvers. The drawback with GPUs is that they require the code to be expressed using a programming language that can offload the computation to the GPU. Converting the code to use these accelerator-specific languages requires parts of the code to be rewritten. The OpenACC programming standard offers accelerator programming using compiler directives to describe the parts of the code that are to be parallelized. OpenACC directives can offload the parallel sections to be executed on the GPU.

In article [1] we examine how OpenACC can be used to offload the computation of the LB method to GPUs. It also examines how the performance of the OpenACC version compared to the native CUDA version implemented using the same techniques as the OpenACC version. This chapter presents how OpenACC directives can be applied to an LB solver to offload the computation to a GPU. This chapter also includes a high-level description of how LB solvers should be implemented in order to efficiently use the resources of a GPU.

The LB solver used for these tests is implemented using hybrid parallelism, i.e. the solver uses MPI to communicate between compute nodes or compute units and some other form of parallelism within the compute nodes or within each accelerator. The LB method has already proven that, if correctly optimized, it will perform very well on GPUs [59]. This makes it a suitable candidate for evaluating the applicability and performance of OpenACC as a simpler method for creating GPU programs.

On a basic level, the LB code can be structured as a collection of loops, iterating over all or part of the lattice sites and applying the propagation and relaxation to the lattice sites. If there are any special boundary conditions being enforced, those are usually done as separate loops. When distributing the solver over multiple compute nodes, the simulation domain is divided into subdomains. To keep the communication of a distributed solver to a minimum, the subdomains should be kept as contiguous blocks. Then the blocks are distributed over the parallel processes running the simulation. Any value propagated out of the local domain on one compute node needs to be moved to the neighboring compute nodes. This usually involves a separate loop, iterating over the lattice site where the propagation moves the distribution values outside the local domain. These loops gather the values into a contiguous memory space. The communication is then done from these contiguous buffers. Once the communication is completed, the new values are scattered back into the local domain.

4.1 Lattice Boltzmann GPU implementation

With the GPU being a separate device, a key factor in performance is to keep data on the device for as long as possible. If the lattice data is initialized on the host, it needs to be moved to the device before the simulation starts. The data should then stay on the device for the entire simulation and not move back and forth between host and device. If the data is shuffled back and forth between host and device, any performance benefit gained from offloading the computation will often be negated by the transfers between host and device. The bandwidth of the bus connecting GPU to the CPU is more than an order of magnitude slower than the memory speed of the device, which will limit the simulation speed.

The lattice site update needs to be described in terms of computational kernels. The kernel describes what a thread running on the GPU will execute. The kernel should apply both the collision and the propagation operation to a single lattice site in a fused operation. These operations should be done at the same time, to reduce the required memory bandwidth per site update. For optimal GPU performance, the code and the data need to be structured in a way that is beneficial to the execution model of the GPU. The algorithm used needs to be one that does not introduce any data dependency between lattice sites, in this case the two-lattice algorithm is used. The lattice data also needs to be laid out by using a scheme that allows the GPU to do coalesced accesses of data. Chapter 3.4 presents data layouts that are optimal for the GPU, the most basic of these would be the stream optimized layout, which is used in this case. Finally, the actual code executed by the kernel needs to be such that it fits the SIMT execution

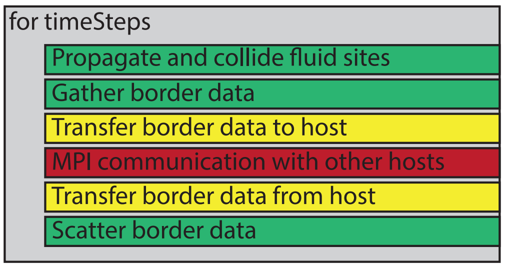
for timeSteps
Propagate and collide fluid sites
Gather border data
Transfer border data to host
MPI communication with other hosts
Transfer border data from host
Scatter border data

Figure 4.1: The basic blocks of the LB solver before we added OpenACC directives.

model of the GPU. In practice, multiple threads will be executing the same kernel at the same time with different indices, and these threads should be executing the same instruction at the same time.

When distributing this solver, the values that are propagated out of the local domain need to be moved to the neighboring compute nodes. These values need to be transferred from the compute device to the network adapter. For optimal performance, these transfers should be done in larger blocks, to reduce overhead associated with the transfers and to improve the network bandwidth of the transfers. To achieve this, the values that will be communicated are placed into a contiguous memory space. Once the communication is done, the values are spread out from these continuous memory locations back to the lattice. Moving the data to and from contiguous memory space is done by a set of small kernels that simply move the data associated with some distribution values.

4.2 Lattice Boltzmann OpenACC implementation

The OpenACC code used for these tests is based on a serial CPU version implemented using the same techniques, algorithm and data layout, as those we used for the CUDA code. This was done to keep the OpenACC and the CUDA versions as similar as possible, for a more direct comparison. The serial code works in the same way as the CUDA code with the relaxation being applied as a fused operation with one function iterating over all the lattice sites. The communication is handled the same way, with values packed into contiguous buffers before being communicated. As the GPU is a discrete device with its own memory space that is separate from the host CPU, data that needs to be communicated must be moved to and from the device. Figure 4.1 shows the basic program flow together with the necessary additional data transfers between the host and the device.

The idea behind OpenACC is to be able to express what part of the code will be offloaded to the accelerator using simple compiler directives. As with

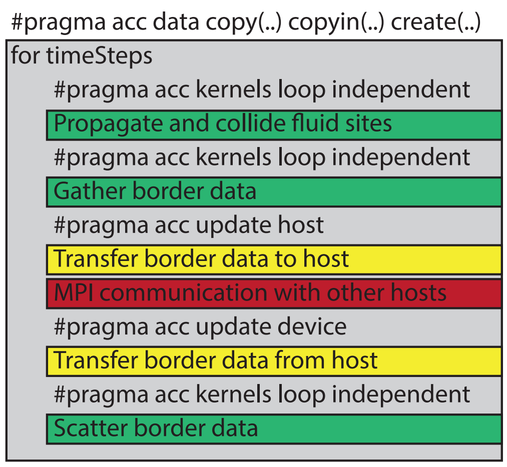
#pragma acc data copy(..) copyin(..) create(..)
for timeSteps
#pragma acc kernels loop independent
Propagate and collide fluid sites
#pragma acc kernels loop independent
Gather border data
#pragma acc update host
Transfer border data to host
MPI communication with other hosts
#pragma acc update device
Transfer border data from host
#pragma acc kernels loop independent
Scatter border data

Figure 4.2: OpenACC directives added to the LB solver. These directives control the data movement and what parts are offloaded to the accelerator.

OpenMP, the code the compiler should work with needs to be placed into a structured block, and this block needs to be annotated with a directive to signal to the compiler what it should do with the code. The directives include hints to as how the compiler should parallelize the code and what parts should be executed on the device.

Data transfers between host and device are also handled using compiler directives. In C code, the data directives are either based on structured blocks or can be added to other directives dictating how the code will be offloaded to the GPU. In the case that they are applied to offload directives, the data transfers will be handled once the execution of the parallel segment starts. When applied to a structured block data transfers will happen as the execution enters and exits the block. The use of data blocks allows the data to be kept on the GPU for longer than just a single parallel loop.

Figure 4.2 shows the practical application of the OpenACC directives to the LB program flow. To minimize the data movement between the host and the device, the entire time stepping part of the code is placed in a `#pragma acc data` block. In the beginning of the block the indexing data is copied in to the device using the `copyin` directive, while the fluid data is moved using the `copy` directive. These directives ensure that the data will be copied to the device, once the execution enters the block, and the fluid data will be copied back to the host once the execution exits the block. Lastly, the additional buffers needed for the communication and the extra lattice data needed by the execution are created using the `create` directives. The create directive allocates space on the device upon entering the block and then deallocates the data at the end of the block.

The main loops of an LB solver are the ones iterating over all lattice sites and updating them. This loop can be offloaded to the GPU using the `#pragma acc kernels loop independent` directive. The `kernels` keyword signifies that the following section should be executed on the accelerator as a sequence of kernel operations. The `loop` construct is added to describe the type of accelerator parallelism to use when executing the iterations of the loop. Finally, `independent` is used to override the compiler dependency analysis of loop dependencies, signaling that the data accesses in the loop are independent. The `independent` directive is needed in this case, since the loop contains indirect data accesses. These indirect accesses restrict the compilers ability to parallelize the code, since the compiler can no longer guarantee they will not cause any data race conditions.

The loop responsible for the gather and scatter of the data needed for the communication is offloaded to the GPU in the same way as the main loop. Again, since the gather and scatter operations involve some type of indirect data access, the `independent` directive is needed for them to be executed in parallel on the device.

The transfers of the communication data to and from the device occur in the middle of the data segment used to keep all the simulation data on the device. These communication buffers cannot easily be transferred to and from the device using the standard data movement functionality. OpenACC does provides a way to update either host or device data from within a data segment in the form of the `#pragma acc update` directive. The `update` directive allows the programmer to specify an array included in a data segment that is to be copied either to the device or host in the middle of a data segment. This allows the communication data to be moved to the host to hand it off to MPI for the transfer between the different ranks, and then back to the device, once the communication is done.

4.3 Performance

The compiler should only convert the sections marked with OpenACC directives into code that can be executed on an accelerator, and handle the movement of data between host and device. The compiler does not modify the code to be more suitable for running on the accelerator, that job still falls on the programmer. The main factor that determines the usability of OpenACC depends on any overhead associated with the way the code is converted, or on any overhead associated with the resulting data movement between host and device. Ideally, with no overhead from the generation of the kernels and the data movement, the OpenACC version should achieve similar performance to a CUDA version of the code implemented in the same way.

We set up a simple test to determine the performance of our OpenACC version of the code and compared it with our CUDA version. Both solvers were implemented using the same algorithm, in this case the two lattice-algorithm. Both solvers used the same D3Q19 discrete velocity set, as well as the same stream optimized data layout. Indirect addressing was used in both cases, making the read and write operations from device memory less trivial. Both solvers functioned in the same way with regards to the communication, with the data moving through the host memory to the other compute nodes using MPI.

The performance test was carried out on a cluster consisting of 8 nodes, each of which contains two Nvidia Tesla C2050 GPUs and two 6-core Intel Xeon processors. For the compilation, the PGI 12.10 compiler was used to compile the OpenACC code, and for the CUDA code version 5.0 of the CUDA compiler was used. The test case used was a simple 3D geometry consisting of the simulation of the fluid flow between two planes, with the edges of the domain being implemented as periodic in the other directions. To test the performance, both solvers were run with varying sized domains from 2^3 lattice sites to 175^3 lattice sites. The only solid lattice sites in the simulation was those at the edge of the domain in the y-direction, representing the two planes.

The sizes of the thread blocks that are being run on the GPU can have an impact on the performance of the code [74]. The OpenACC compiler will automatically choose the size of the thread blocks for any code it compiles, unless a specific size is set using the OpenACC directives. For the main kernels in the LBM code, the OpenACC compiler set the thread block size to 256 threads. The compiler also used the same thread block size for all the other kernels in the program. When compiling both the CUDA based code, as well as the OpenACC one, both ended up using 42 registers for the main LBM relaxation and propagation function.

Figure 4.3 shows the performance measured on a single GPU, comparing the performance of both the CUDA and OpenACC implementations. Both codes are running at the same thread block size, 256 threads. Using this thread block size, the performance difference between OpenACC and CUDA is just 0.5%, with the CUDA version being the faster one. For the CUDA version, using a thread block size of 512 threads yields a better performance as shown in figure 4.3. However, forcing the OpenACC version to the same thread block size, there was no observable improvement in the performance of the OpenACC code. Comparing the OpenACC version with the CUDA version, running with a thread block size of 512 the CUDA version performs 1.05 times better than the OpenACC one.

The distribution of the solver was done using MPI, each GPU was handled as a separate MPI rank. The communication from the GPU passes through the host memory to the network. Figure 4.4 shows how the dis-

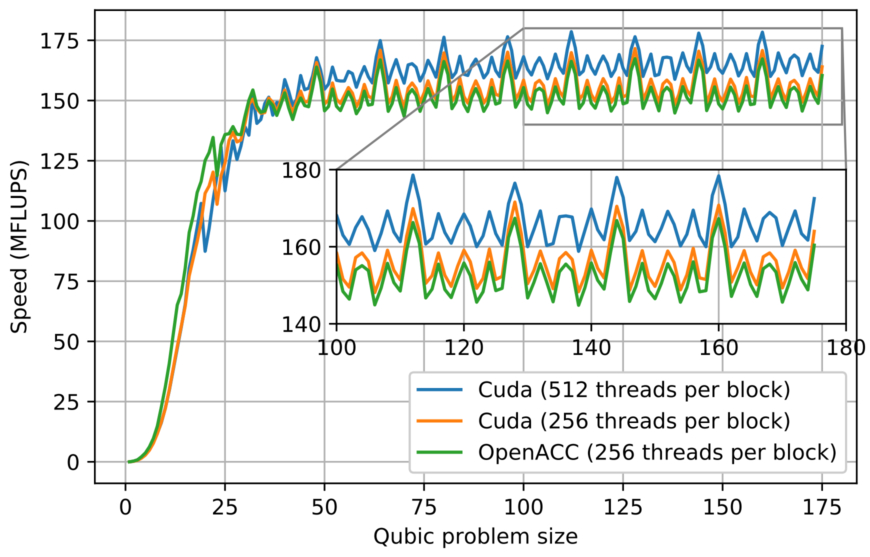

Figure 4.3: Performance of the OpenACC accelerated LB solver running on one Tesla C2050 GPU.

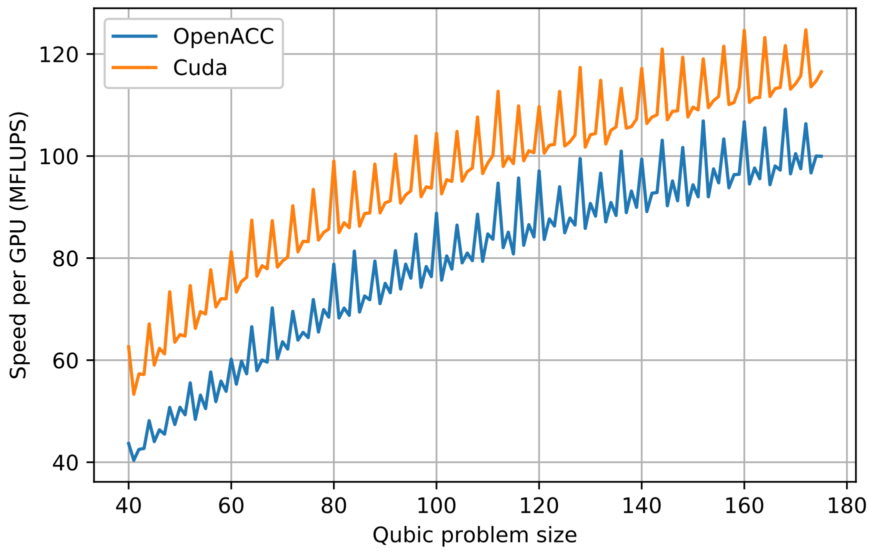

Figure 4.4: Multi GPU performance of the OpenACC accelerated LB solver running on 8 Tesla C2050 GPUs.

tributed versions of the solver perform. In the distributed case, the CUDA version gives, on average, a performance 1.22 times better than the OpenACC version. The reason for the larger performance discrepancy was examined using the Nvidia profiler [75]. The profiler shows that the main reason for the performance difference is down to a small delay between each kernel launch in the OpenACC code that is not present in the CUDA code. Additionally, the way the OpenACC version does the data transfers between the host and the device achieves a transfer speed that is about 1GB/s slower than what was achieved with CUDA. Since, for this comparison, all data transfers were executed synchronously with no overlapping, this speed difference will be visible immediately when comparing the performance of the solvers.

4.4 Conclusion

Even though the OpenACC code performs slightly slower than the CUDA version, OpenACC should not be discarded as an alternative way to program GPUs. When the experiments were run, the OpenACC version performed marginally worse than the CUDA versions on single GPUs, but with a larger difference on distributed solvers. However, the simplicity of just adding compiler directives to existing code is a worthwhile tradeoff. Especially if the goal is to quickly covert an existing code base to use GPUs, it is far easier to get the code to use accelerators using OpenACC than with CUDA. Sections of the code that are performance critical can be further converted to CUDA code for optimization. The comparison made in this chapter shows that, while OpenACC is a worthwhile alternative, it is by no means a replacement for CUDA code. It is still possible to achieve better performance with the CUDA code. Certain functions of the GPUs cannot be used in OpenACC, one prominent example being the usage of the shared memory on the GPU, which means there is still a need for CUDA to program GPUs.

Chapter 5

Large-scale parallel computing using the lattice Boltzmann method

This chapter covers additional GPU implementation techniques from article [2] and [3]. We examine the influence of different data layouts on the performance of GPU solvers, as well as the impact the choice of algorithm has. This chapter also discusses the details of large scale LB simulations from the same articles, and delves into how the I/O was designed. We also discuss the effects of load balancing, and how the LB solver can both strongly and weakly scale on, what was at the time, one of the most powerful supercomputers in the world.

Through the INCITE project [76] we gained access to the Titan supercomputer [77, 78] at the Oak Ridge national laboratory. At the time Titan was the second fastest supercomputer in the world. Titan is a Cray XK7 machine [79], consisting of 18688 compute nodes. It derives most of its computational power from Nvidia Tesla K20X GPUs. Each node has one GPU which has 6GB of memory. The nodes use 16 core AMD Opteron CPUs, with a single CPU powering each node with 32 GB of memory. The interconnect is Cray's proprietary Gemini interconnect [80], which is accessed through the HyperTransport bus of the CPU. Normally, this bus is used for communication between CPUs in a node which require a high bandwidth and low latency connection. The theoretical peak performance of the machine is rated as 27.1 PFLOPS, and in the industry standard LINPACK [14] test Titan can achieve 17.59 PFLOPS. This performance makes it the first supercomputer in the world to break the 10 PFLOPS barrier, placing it at the top of the TOP500 list in November of 2012 [81].

Centers hosting the top tier machines want the people using these machines to be able to use the system efficiently and be able to use a significant

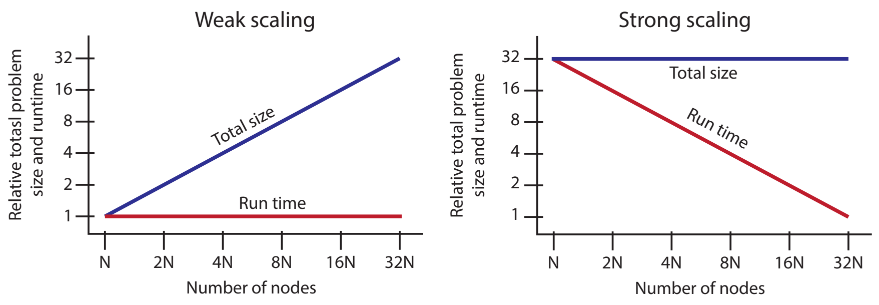

Figure 5.1: Strong (right) and weak (left) scaling illustrated. The graphs show how the ideal runtime and the total load changes in the two scaling scenarios.

part of the machines. Researchers applying for resources on Titan are encouraged to show how their codes perform on the machine or machines of similar scale before they apply for substantial amounts of resources [82]. These benchmarks should also reflect the real work planned to be carried out with the resources. Large machines such as Titan are intended for simulations that require the resources these machines offer. Smaller simulations can be done on simpler and more easily accessible machines. Supercomputing centers often provide other resources to users wanting to run smaller parallel and serial jobs.

With the requirements of being able to efficiently use the large machines, scalability becomes a very important aspect of any software that will be run on these large-scale machines. The programs should show consistent performance as the number of compute nodes the program is distributed over is increased, i.e. show good scaling. In the HPC world, scaling is often divided into two categories, strong and weak scaling.

When scaling a simulation using weak scaling, the problem size is scaled up at the same rate as the number of computational nodes is increased. Thus, the computational load per compute node stays constant. The goal with weak scaling is usually to be able to fit a larger problem into the machine, and these additional nodes are needed due to the memory requirement of the simulation in question. Good weak scaling implies that the runtime of the simulation stays the same as the problem size and node count is increased. Strong scaling, on the other hand, works with a fixed problem size. The goal with strong scaling when adding more nodes is then to reduce the total run time for the simulation. Running on twice as many nodes, ideally the runtime should halve, as each node then gets half the computational load. Figure 5.1 illustrates how the computational load per node changes as the number of compute units used increases.

Amdahl's law [83] should always be taken into consideration when discussing parallel scalability. Amdahl's law is expressed as:

$$S_{latency}(s) = \frac{1}{(1-p) + (p/s)} \tag{5.1}$$

where $S_{latency}$ is the total speedup achievable for the program if the parallel portion p can be sped up by a factor of s. In simplified terms, it governs the theoretical speedup that can be achieved by parallelizing a given workload. In the case that only 50% of the total simulation can be parallelized, the maximum speedup achievable is just 2 times better performance. If 75% can be parallelized, the maximum speedup is 4 times faster. Even in the case that as much as 90% of the computation can be parallelized, the maximum speedup is still only 10 times faster, regardless of the number of processing elements being used for the computation.

Weak scaling is considered to be easier to achieve than strong scaling, since the load on each node will stay the same, and Amdahl's law generally does not become the limiting factor. Collective operations and networks where the bandwidth decreases as more nodes are communicating can still make the program experience sub-optimal weak scaling. Strong scaling, on the other hand, tends to be harder to achieve. With strong scaling the data assigned to a specific node decreases as the number of nodes is scaled up. This gives the local node less data to work with, which equates to less work to be executed in parallel, and any serial sections will affect the total runtime more. The smaller per node computational load also tends to make overhead, caused by communication or threading, become more apparent. In the LB case and other stencil-based solvers, the communication volume does not decrease in the same ratio as the computational load. Any imbalances and bottlenecks in the interconnect will become more pronounced when strongly scaling the simulation, as the communication becomes a large part of the simulation.

5.1 Input sample

For all large-scale simulations, a porous media sample generated by Hilfer et al. [84] was used. This sample is freely available online and represents the microstructure of Fontainebleau sandstone, a subsample of which is shown in figure 5.2. The sample is available in different resolutions, ranging from 458 nm per voxel to 117 µm per voxel, halving the size of the voxels at each step. The porosity of the sample is approximately 13%. While the samples consist of large number of voxels, the fluid will only be simulated for the 13% that is not solid.

These samples are named based on the size of the voxels in them. The naming is based on how many times larger the voxels are compare to the

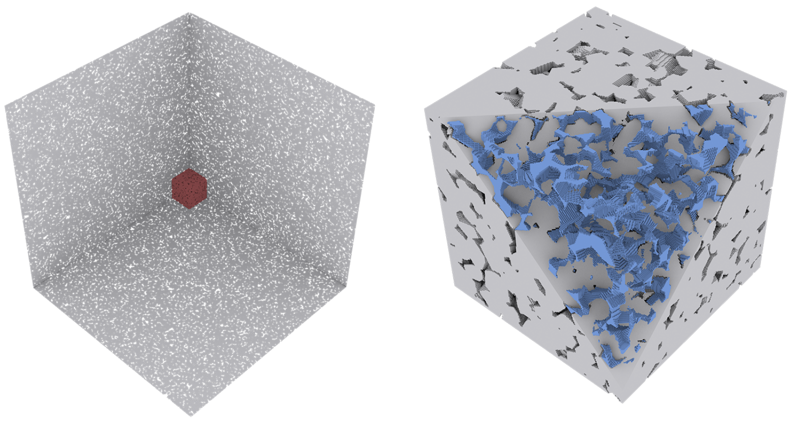

Figure 5.2: An illustration of the sandstone input sample used. Left is a representation of the full input sample representing N^3 voxels. Right is the red cube from the left enlarged, the red cube consists of $(N/8)^3$ voxels. The blue part is the pores within the structure, and with the sandstone removed, it is the fluid flow within these pores that is simulated.

915.5 nm sample. The lowest resolution sample is denoted as A128 and the highest as A0.5. For the simulation runs on Titan, all samples from the A16 sample to half of the A1 sample were used, with voxel sizes from 14.648 µm to 0.9155 µm.

5.2 GPU data layout and algorithm performance

On the GPU, basic implementation decisions can heavily influence the performance of the LB solver. Choosing the right algorithm and, more importantly, data layout is the key to a well performing GPU accelerated LB solver. The OpenACC comparison work was carried out using the two-lattice algorithm, since it is the simplest algorithm that fits the execution model of the GPU. However, it is not the best algorithm to use when running an LB solver on the GPU. On Titan using 1024 compute nodes, the Two-lattice and AA algorithm were compared using the porous media sample. The two-lattice algorithm achieved a per node performance of 286.4 MFLUPS, while the AA algorithm could run at 384 MFLUPS, a 1.34 times better performance.

The data layout is also a major factor in the performance. Since the GPU performs best when doing coalesced memory accesses, the data layout needs to be such that non-propagated data accesses can be done coalesced. The stream optimized data layout would allow for this, and is a viable

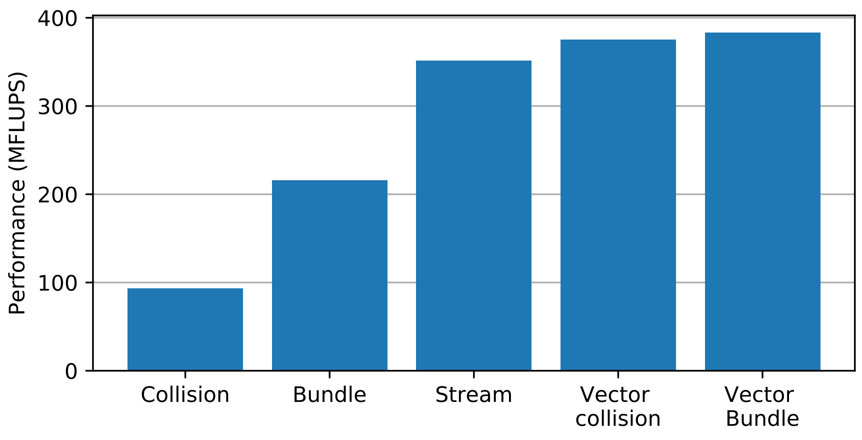

Figure 5.3: Data layout performance for 1024 nodes on Titan. The measurements were carried out using a porous media sample using the AA algorithm.

candidate for running on GPUs. However, testing showed that there are better alternatives. Figure 5.3 shows the measured performance for different data layouts. These tests are from 1024 nodes on Titan using a porous media sample running the AA algorithm.

The collision optimized layout has the lowest performance at 93 MFLUPS, the regular bundle layout achieves 216 MFLUPS, and the stream optimized layout achieves 351 MFLUPS. Both the collision and bundle layouts can be made to perform better by vectorizing them; instead of placing one value into the structure for each site, values from 16 sites are placed into the same structure. This layout allows non-propagated accesses to be done perfectly coalesced. These vectorized collision and bundle layouts can run at 375 MFLUPS and 383 MFLUPS respectively, 1.067 and 1.09 times better performance than the basic stream optimized layout.

5.3 I/O

File I/O is easy to perform on a single system, such as a desktop workstation. It can often be done from a single process using the standard I/O libraries and still be able to achieve satisfactory performance. On larger systems, having a single process being responsible for all the file accesses for the simulation is impractical. The I/O bandwidth becomes limited by the network bandwidth of the single node doing the I/O, even if the file system could deliver more than the network bandwidth of a single node. As the amount of data needed for the simulations and the number of nodes being used grows, using one process for file I/O is simply not feasible after a point. The solution is to use some form of I/O functionality that can distribute the file I/O over multiple processes and compute nodes and thus distribute the I/O load.

On the top clusters and supercomputers in the world, the file system used is always a parallel file system, the most popular being the Lustre file system [85], as it is on the Titan supercomputer. The Lustre file system [86] builds on having one or more metadata servers and one or more object storage servers. These systems usually have orders of magnitude more object storage servers than metadata servers. The metadata servers are responsible for storing the metadata for the file system, such as the filenames, directories, file permissions, etc.

The object storage servers (OSS) are responsible for storing the actual data associated with the files. The actual data is stored on object storage targets (OST) within an OSS, with each server capable of hosting multiple OSTs. The OSTs are then generally backed by several disks in a RAID type configuration that provide data redundancy and availability.

A file stored in the Lustre system is stored in one or more OSTs. When using multiple OSTs the file is striped across the different OSTs using some chosen block size. The more OSTs the file is distributed across, the higher is the theoretical bandwidth available when performing operations involving that file [87]. This means that the parallel performance of a Lustre system is directly related to the number of OSTs available.

Choosing a default value for how many OSTs a file is distributed over is a complicated task, as some cases benefit from files being distributed over many OSTs, while others do not. Usually, a low default stripe count is set between 2 and 8. Even on a system as big as Titan, the default stripe count is only 4. The main reason for a low default stripe count is that in the case that users are using small files, having them striped over many OSTs will be detrimental to performance.

One way to implement parallel I/O is to have one file per process running on the system, the idea being that each process can independently access the file associated with that process. In the case that the files are placed on different OSTs in a Lustre system, the simulation can benefit from the speed offered by accessing multiple OSTs simultaneously. The downside of this scheme is that to change the load balance or the number of nodes participating in each simulation, the files need to be regenerated and rearranged. Many operations with different files also put a higher load on the metadata server. There is also a challenge managing the considerable number of files on larger systems, since the user needs to ensure that the files are distributed over multiple OSTs.

A more convenient solution is to use a single file that is accessed by all the processes in the simulation. A parallel I/O library is then used to enable more than one process to participate in the I/O operation in parallel. One such parallel file I/O library is MPI-I/O [88], which has been part of the

MPI standard since MPI version 2.0 and provides the user with the needed functionality for parallel file access.

MPI-I/O provides parallel I/O functionality along the same lines as the regular send and receive operations of MPI. The library can perform collective I/O, asynchronous I/O, read and write using MPIs data types. When performing parallel I/O, the library has the possibility to merge scattered accesses from multiple ranks into larger requests. The data accesses can be reorganized and distributed over multiple nodes doing the file access.

Using the data type functionality of MPI, MPI I/O provides a convenient way for each process to access a specific subdomain from the input file. MPI datatypes can be used to describe the access pattern a process will have to a file. The view of a process into that file can then be limited, so that the process can see the data it will be accessing, as if it were contiguous. Any collective operations to this file will then be handled by the underlying library and file system. These collective operations can stitch together the non-continuous accesses from separate processes from the same file into larger contiguous transfers.

In the simulations carried out on Titan, the input data is a collection of voxels representing a 3D cuboid. From this cuboid, individual MPI ranks will access a smaller cuboid space, and this type of access fits well with the parallel I/O capabilities of the MPI I/O library. The total amount of data and the number of nodes participating varies between the different samples, the smallest sample used is 1 GB and the largest 4 TB. From the 4 TB file half was used, or 2 TB of data.

The porous media sample was used to test the performance of the default settings of the Lustre file system on Titan. On 1024 compute nodes an aggregate performance of 274 MB/s was measured. At that speed our largest samples would take hours to access. With the computational performance of the GPUs, even when filling up the memory of the GPUs, running the simulations for the largest input samples would take less than an hour. The slow I/O would result in the simulation wasting resources for the many hours it takes to read the data into the machine.

During the first years of its use, the Titan system, the maximum stripe count for a file was 160. This was due to software limitations of the system at that time. With the stripe count set to 160, a performance of 6146 MB/s was measured using 8192 nodes and an input sample of 512 GB in size. At this speed, even the largest of the samples could be read in a reasonable time. After an update to the Lustre system, the full potential of the file system was unlocked and the maximum stripe count for a file grew to 1008. The increased stripe count allowed the solver to read the 512 GB sample at a speed of 177 GB/s using 8192 nodes and at a speed of 314 GB/s using 16384 nodes. For the largest runs on 16384 nodes, this meant the 2 TB sample could be read in just 6.5 seconds, allowing most of the allocated time to be spent running the numerical simulations on the GPUs.

5.4 Load balance

For simulations distributed over many compute nodes, the way that the computational load is distributed over the nodes becomes a key factor in the performance and scalability of the program. Improperly distributed, the simulation ends up with some nodes having more load assigned to them. The load imbalance can cause the program to end up with parts of the simulation running only on a subset of the compute nodes and the others being idle, since those nodes have already completed their work for the current iteration. In the case with one or a few nodes doing more computation, it will effectively slow down the computation on the other nodes, since they will be waiting on the computation on the overloaded nodes.

Having a few nodes with a lower load than the rest is less harmful to the performance, since then the only nodes waiting will be the few nodes that finished their computation early. One thing to note is that modern processors work with dynamically scaling clock frequencies. Thus, there is no guarantee that the computation will progress at the same speed on all nodes. This makes achieving the perfect load balance near impossible. The efforts presented here are only trying to distribute the load evenly among the nodes and not accounting for any performance difference between the nodes.

Figure 5.2 shows a cross section of the input sample. On a large scale, the material appears very homogenous, with the fluid sites in the simulation evenly distributed across the input sample. On a smaller scale with the high-resolution sample distributed over many compute nodes, the load variance between the load assigned to each domain starts to vary significantly.

Figure 5.4 shows how the sizes of the local domains differ from the average size when dividing the A4 sample into different numbers of subdomains, ranging from 1024 to 16384 subdomains. The orange colored line represents the distribution with no load balancing applied, and the domain is divided into equally sized cuboid domains. As the number of subdomains is increased, the difference between the most and the least loaded subdomain grows. At 1024, the most loaded domain has 1.07 times more fluid sites than the average load. Scaling up to 16384 subdomains, the difference is larger, with the largest domain having 1.65 times larger load than the average. The figure shows that the distribution is such that a few domains have significantly larger load assigned to them than the rest of the domains.

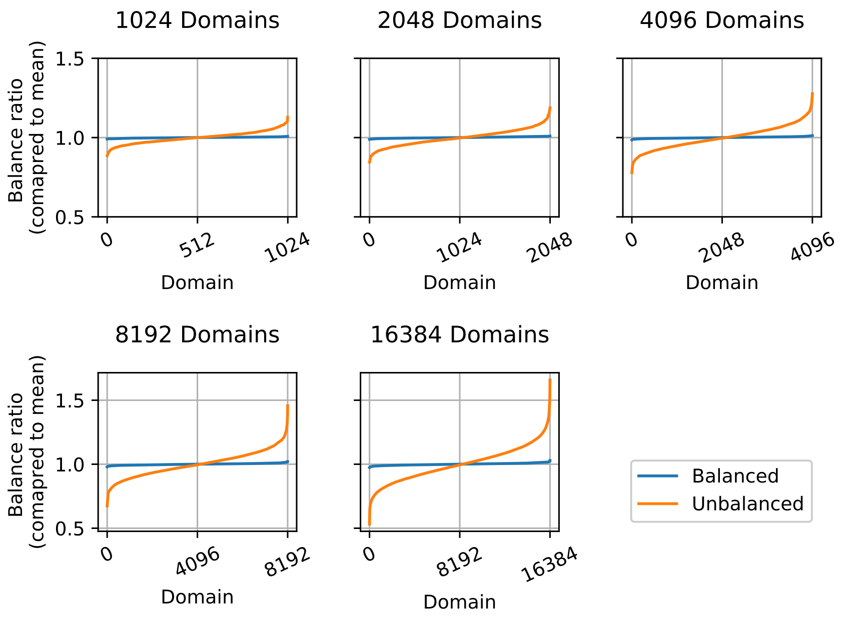

Figure 5.4: Illustration of the effect our load balancing has on the distribution of the fluid sites of the A4 sample, starting from 1024 subdomains and showing the distribution up to 16384 subdomains.

The imbalance in the load will have a significant impact on performance, especially for the simulations using more than half of the supercomputer. To alleviate the imbalance, some form of load balancing scheme is needed. With the input sample being homogeneous, i.e. there are not wildly different geometries in certain parts of it, a basic load balancing scheme can be used and still give satisfactory results.

To improve the load balance, a basic recursive bisection load balancing scheme was implemented. This scheme works by taking the entire simulation domain and dividing it into two cuboid domains, divided in such a way that they contain the same number of fluid sites. Each of these two cuboid domains is then divided into two new domains in the same way as the original split, each of which is further divided into two domains, until the number of needed domains have been generated. The downside of this scheme is that it is limited to a power of two number of domains, and the domains need to be cuboid shaped. The granularity with which the split ratio can be adjusted is limited to one slice of voxels sites, which can result in some minor imbalance when dividing a domain, preventing the load from being balanced perfectly.

With this scheme, the load balance situation of the simulation can be improved. The blue colored line in figure 5.4 shows the distribution of

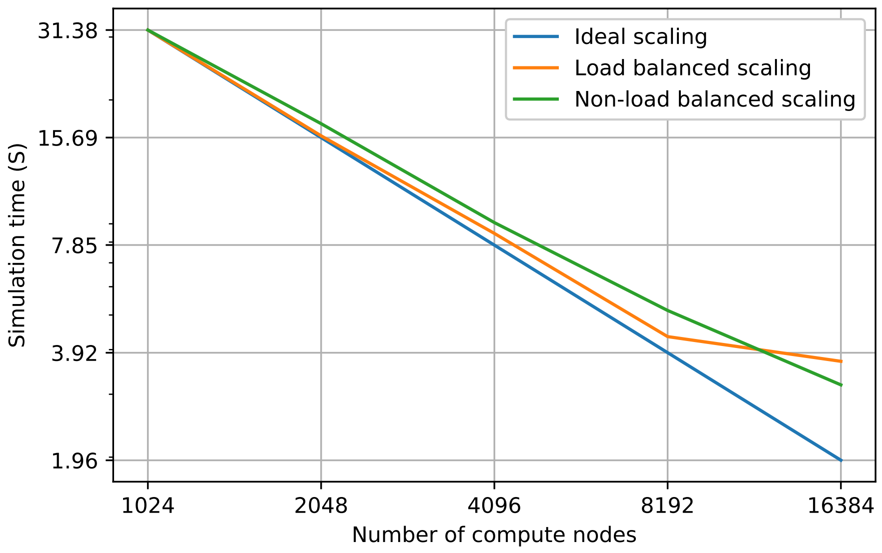

Figure 5.5: Comparison of the performance of the load balanced solver versus the solver with no load balancing. The simulations were carried out using a two-lattice solver with the A4 input sample.

the load for different numbers of subdomains. Even though a simple load balancing algorithm was used, it does improve the balance between the different domains significantly. With the load balancing applied for 16384 subdomains, the maximum number of fluid sites in a subdomain is just 1.028 times larger than the average. Figure 5.5 shows the impact the better load balance has on the performance of the solver. This test was carried out using the A4 resolution of the input sample, starting from 1024 nodes and scaling up to 16384 compute nodes. In this case the two-lattice algorithm was used.

Distributing the solver over 8192 nodes or less, there is a clear performance advantage in performing the load balancing. At 8192 nodes running the solver with load balancing is 1.18 times faster than without load balancing. Assuming perfect scalability, this speedup would be the equivalent of having slightly less than 1500 additional nodes running the unbalanced version of the simulation. Scaling up to 16384 nodes, there is a sharp decline in the performance of the balanced version. This performance hit can be attributed to the non-balanced version having a simpler communication pattern, as all nodes communicate with 18 neighboring nodes. The balanced version can communicate with more nodes and in a less structured way. As a by-product of how the load balance was implemented, the domains are also distributed in a more random fashion across the machine.

5.5 Scalability

The Titan supercomputer at the Oak Ridge national laboratory was used to evaluate how well the LB solver is able to scale to large supercomputer systems. As presented previously, it is a GPU accelerated supercomputer consisting of 18688 compute nodes, each with one K20x GPU and one AMD Opteron 6274 CPU. The interconnect used in Titan is Cray's proprietary Gemini interconnect [80], an evolution of Cray's earlier Sea Star interconnect. The interconnect is laid out as a three-dimensional torus with a pair of compute nodes sharing the same network interface, referred to as a router in the torus. Each router is connected to its nearest neighbors in three dimensions. The routers at the end are then connected to the node at the opposite end of the row, forming a torus in all directions. Each router has a total of 10 torus links that are used to connect to the neighbor routers. In the case of Titan, these links provide 4 links in the X and Z directions, a pair in both the positive and negative X and Z directions, and 2 links in the Y direction, with only one link going to each neighbor in the Y direction. Due to the imbalance in the number of links used, the nodes are physically laid out in such a way as to favor the X and Z dimension and to deemphasize communication in the Y direction. There are 25 pairs of nodes in the X direction and 24 pairs in the Z direction, but only 16 pairs in the Y direction. Bland et al. [89] measured the bidirectional MPI bandwidth per node in the different directions in the torus. In the X direction they achieved 10.6 GB/s, 10.5 GB/s in the Z direction and just 5.40 GB/s in the Y direction.

This type of network topology offers some benefits. The topology avoids the need for network switches and keeps the cable runs short. The network also allows the systems to be easily scaled up by simply adding cabinets of compute nodes. This topology does have some significant drawbacks, with each pair only being connected to its nearest neighbors. Messages going to a compute node further away must pass through multiple other routers on its path to the target compute node. Since the network has no switches, every network router must, in addition to the network traffic caused by the pair of node from the current router, also handle other communication moving through the network. Additionally, as the machines increase in size, the maximum number of hops between two points in the network will increase. At the same time, each router must handle traffic from the additional nodes, further impacting the performance of the network.

For the scaling test, a porous media sample was used. The resolution of the samples used varies, starting from 1024^3 lattice sites to 16384^3. Due to memory restrictions, only half of the 16384^3 sample was used, for all other samples the entire sample was used. For the scaling runs, the AA algorithm was used, in addition to asynchronous communication, the implementation of which is presented in section 6. The simulation was carried out using

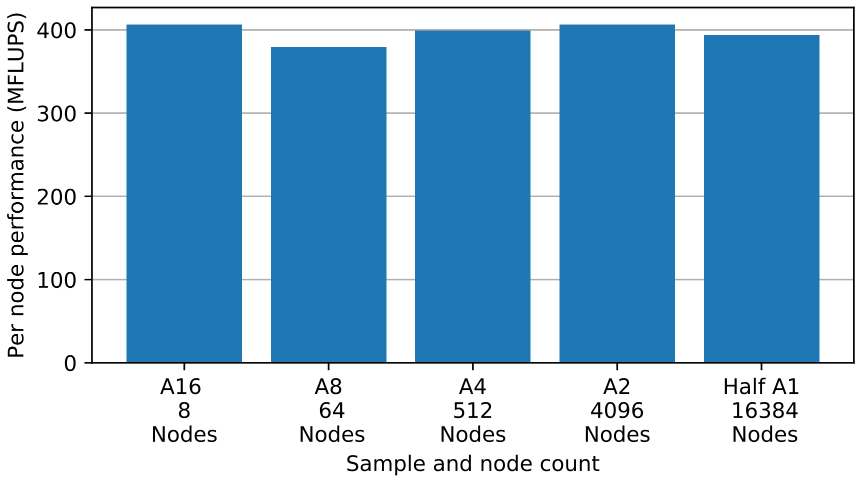

Figure 5.6: Weak scaling results for the porous media samples, starting from 8 nodes up to 16384, running the AA algorithm on Titan.

the vectorized bundle data layout, with values from 16 lattice sites packed into each structure. The load balance used is the simple recursive bisection described in section 5.4.

5.5.1 Weak scaling

The weak scaling results are shown in figure 5.6, with the different resolutions of the input sample being used for different compute node counts, starting at 8 compute nodes with the A16 sample at 1024^3 voxels, with around 144 million fluid sites, scaling up to 4096 compute nodes with the A2 sample with 8192^3 voxels. The largest runs were carried out using 16384 compute nodes, and used half the A1 sample, consisting of $16384 * 16384 * 8192$ lattice sites of which 295 billion are fluid sites. All of these runs took between 43 and 46 milliseconds per iteration.

The initial run with 8 nodes is allocated within a contiguous segment in the machine and benefits from a lower latency between the nodes. Increasing to 64 nodes, there is a small drop in the performance, since the nodes are no longer guaranteed to be located close to each other. As the simulation is scaled to more nodes, there is a minor increase in the performance, going from 64 to 512 and again going to 4096 nodes. This is due to the higher resolution having larger cavities in the sample. Due to the larger cavities, there is less divergence in the site update code, since the ratio of sites that the boundary condition is applied to is lower. Overall, the weak scaling performance is excellent.

5.5.2 Strong scaling

The strong scaling results are more interesting. As the simulation load per node drops, the effect of the network starts to become more apparent. Each strong scaling run starts with around $1.7 * 10^7$ fluid sites per GPU. Testing with the same porous media sample, a single GPU can maintain the same performance using $1.7 * 10^7$ fluid sites as it can with $2.7 * 10^5$ fluid sites. In fact, at $3.4 * 10^4$ fluid sites, a single GPU still maintains 84% of peak performance.

Starting with the smallest sample, A16, which has a resolution of 1024^3, this sample needs at least 8 compute nodes to fit into GPU memory, and the maximum it was scaled out to was 4096 compute nodes. The A16 sample follows the ideal scalability well up to 256 compute nodes, with each GPU assigned $5.5 * 10^5$ lattice sites. When going to 512 compute nodes, where each node is assigned $2.7 * 10^5$ lattice sites, the performance drops off. The performance only improves 1.7 times when doubling the compute nodes from 128 to 256, even though a single GPU is still able to perform well at that load. Further doubling the compute nodes to 512 brings the per GPU load to $1.36*10^5$ lattice sites. This increase in nodes yields a 1.19 times performance improvement, even though the single GPU tests show better performance at that load.

The A8 sample, starting from 64 nodes, starts to deviate from the ideal scalability when going past 1024 compute nodes. It gradually diverges from the ideal scaling up to 4096 compute nodes, and when scaling further to 8192 compute nodes, there is a steep divergence from the ideal scaling. The run on 8192 compute nodes is a fraction of a percentage slower than the one at 4096 compute nodes. For the A8 sample, the run on 4096 compute nodes equals $2.7 * 10^5$ sites per GPU, a size where a single GPU is still able to maintain the same performance as fully loaded. The run at 8192 compute nodes equal $1.36 * 10^5$ fluid sites per GPU, a size where a single GPU is still able to achieve over 84% of the peak performance.

The A4 run also experiences a significant divergence from the ideal scaling, when moving past 4096 compute nodes. Finally, the A2 sample fares better than the others, but is also not able to keep close to the ideal scaling when going past 4096 compute nodes. At 16384 compute nodes, the A2 sample is far off the ideal speed, even though each GPU is assigned $4.4 * 10^6$ lattice sites at that node count, achieving only a 1.24 times speedup from 8192 to 16384 compute nodes.

Overall, the simulation of the different resolution samples can be strongly scaled, at least to some degree. The smaller samples that fit on a smaller number of nodes are far better at scaling than the larger samples. With the

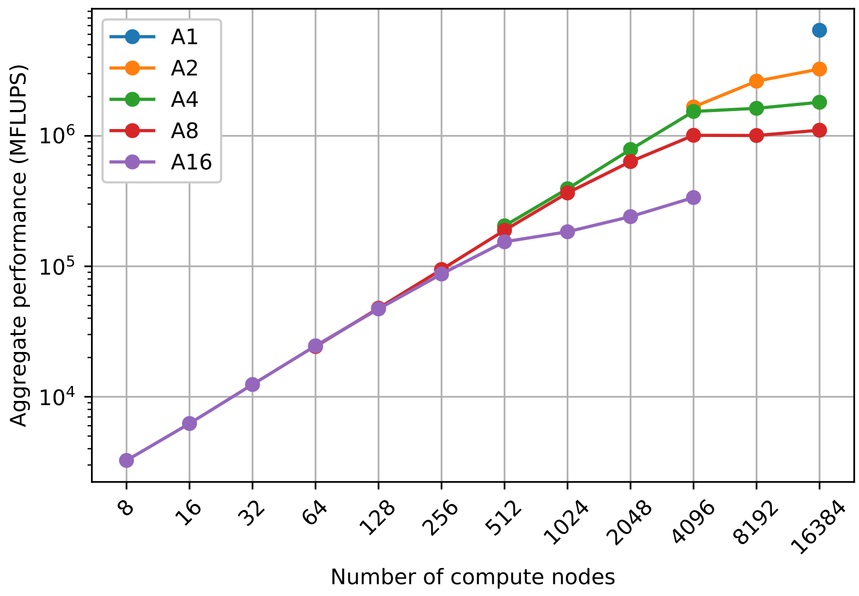

Figure 5.7: Strong scaling for the AA solver using the porous media samples running on Titan.

larger samples, there are significant difficulties scaling up from 4096 compute nodes to 8192 nodes, with multiple samples encountering difficulties at the same number of compute nodes.

5.5.3 Scaling difficulties

When testing the weak scaling, there were no difficulties scaling up. The performance for all different resolutions at their respective node counts are virtually the same. However, when testing strong scaling, the simulation encountered some difficulties scaling to and past 8192 compute nodes.

The load balancing scheme is shown to evenly distribute the computational load across the compute nodes. Section 5.5.2 showed that testing on a single GPU indicates that it still can perform, with the amount of load assigned to the individual GPUs for these runs. Ruling out the individual GPU performance and the load balance, the remaining factor that can affect the scalability of the solver is the network of the machine.

The first potential problem with the network comes from how the nodes in Titan are structured. The network interface sits on the Hypertransport bus, a bus that is normally used to communicate between CPUs in the same node. While this bus provides substantial amounts of bandwidth to the network at a low latency, the downside is that all communication from the GPU must move through the host CPU memory to get to the bus. This additional hop adds latency to each MPI transfer.

The topology of the network is likely a major limiting factor in the strong scaling tests. In theory, the type of interconnect used in Titan is well suited for this type of solver. The LB solver only uses nearest neighbor communication and the interconnect network in Titan is built around each node, being connected to its nearest neighbors. Which means, perfectly distributing and arranging each process of the solver, would result in each node only communicating with its neighbors. However, in practice, Titan is shared among different users, all running their own simulations at the same time. This means that when a single user is allocated a group of nodes, there are no guarantees where they will be located. The different amount of bandwidth that is available in different directions, combined with no exact control over what nodes are allocated to the user, and how the tasks are placed will also affect the scalability of the solver. With no control over the task placement, there is no way of avoiding the slower communication paths.

The difference observed when testing the performance between the load balanced and the non-load balanced solver is that the non-load balanced version has a constant 18 neighbors it communicates with. However, with the introduction of the load balancing scheme, the number of neighbors each compute node needs to communicate with grows. For the balanced version, the worst-case scenario observed was one processes having to communicate with 24 other processes. Another drawback is that, as a by-product of the implementation of the load balancing scheme, the order of the nodes is further shuffled compared to the non-load balanced solver. This can lead to nodes working on adjacent parts of the geometry being physically separated within the machine. The load balanced versions had each node moving data from the nodes to the network at a rate of 2.9 TB/s, and with the way the asynchronous communication works, they are not sending and receiving constantly.

5.6 System utilization

To determine how well the system was utilized by the simulation, the Nvidia profiler was used to determine the floating-point operations performed and the number of bytes accessed per each lattice site update. According to the profiler, each lattice site update consists of 279 floating-point operations and on average 545 bytes of memory transferred. This is the total number of bytes transferred by the GPU, and includes all data transferred between the GPU and the on-board memory, including data that is not used by the simulation.

The largest runs performed were done using 16384 GPUs. This run used half the A1 sample and ran at $6.45 * 10^{12}$ lattice updates per second. Based on the measured floating-point operations per lattice site update, this results in a total floating-point performance of approximately 1.8 PFLOP for the largest simulation run. This is roughly 10% of the measured Linpack performance of the Titan supercomputer, which was measured at 17.6 PFLOP [81].

The total theoretical memory bandwidth for one GPU in Titan is 250 GB/s [90]. Using this bandwidth gives us a theoretical peak memory bandwidth for 16384 GPUs of 3.9 PB/s. The profiler measurements of 545 bytes of data needed per site update, combined with speed of an average GPU from the run, gives us a per GPU bandwidth of 209 GB/s, or 3.3 PB/s for all the GPUs in the simulation. This is roughly 84 % of the total available bandwidth on the GPUs. While the solver was not able to achieve more than 10% of the numerical performance of the machine, it managed to utilize a significant portion of the memory bandwidth of the GPUs in the machine.

5.7 Scalability conclusion

By having access to the Titan supercomputer, we could examine how the LB solver scales to large scale systems. For large systems, how the parallel I/O system is used can significantly reduce the time taken to perform setup of the solver. Using Titan, we showed that the LB solver can scale up to petaflop scale machines when using weak scaling. The largest weak scaling simulation carried out was on 16384 compute nodes and achieved a sustained floating-point performance of 1.8 PFLOP. The simulation at that scale could, however, utilize 84% of the available memory bandwidth of the GPUs in the system. For strong scaling, the network appeared to be the biggest bottleneck, preventing some samples from scaling well to 8192 compute nodes.

Chapter 6

Asynchronous communication

One key aspect in both article [2] and [3] was the usage of asynchronous communication between the GPUs in the system. This chapter examines how the asynchronous communication is implemented and what impact it has on the performance of the solver. It also covers the what steps need to be taken when implementing it to maintain data access patters that are still favorable for the GPU.

Asynchronous communication is a well-known concept in the HPC field. The idea is that the communication between different processes is done while the processors running the tasks execute other work related to the simulation. Ideally, the communication should finish before the data communicated is needed by the receiver. If the communication finishes before the computation, the communication is effectively done for free, since the simulation will never be waiting for the communication, and will be able to work on the computation the entire time.

On pure CPU systems, asynchronous communication requires some intervention from the CPU [91]. Some systems have special hardware in the network adapter allowing it to process part of the communication protocol stack independently of the CPU [92]. Without special hardware, the CPU will need to take part in the communication, while at the same time, it should perform the computation for the simulation. This will either result in a lower performance for the computation or no progress for the asynchronous communication. On a GPU accelerated system, on the other hand, getting asynchronous communication to work is far easier, since the GPU is a separate part of the system that performs the computation without any real involvement from the host CPU, apart from launching the kernels.

This allows the CPU in the system to perform any operation related to the communication for the program, without impacting the performance of the computation.

In the case of the LB solver, the main computation task is to apply the relaxation and propagation to each lattice site as a fused operation. The distribution values that are propagated outside the local domain for a given node is the data that needs to be communicated. This data needs to be transferred to the neighboring compute nodes before the start of the next time step. Before the communication starts, the updates of the sites whose values propagate outside the local domain need to be completed. To enable the LB solver to communicate asynchronously, the lattice sites need to be updated in two parts. The first part will update the lattice sites at the edges of the local domain, the ones that will be communicated. Next, the second part will update the sites at the interior of the local domain. While the interior sites are updated, the data that needs to be communicated can be transferred to the correct neighboring compute nodes.

On the GPU, to run things concurrently, for example multiple kernels at the same time, or data transfers at the same time as kernels, the operations need to be assigned into streams [16]. A stream is a queue of operations the GPU will perform, like kernels launches, data transfers, etc. The operations within a stream are run in the order issued and are not overlapped with each other. Operations from different streams, however, can be executed simultaneously.

For the GPU implementation, the LB solver uses two streams. Into the first stream, the kernel that handles the computation of the sites at the edge of the local domain is issued. Into the first stream, the transfers of the communications data between the host to device are also issued. Additionally, to optimally utilize the bus between the device and the host, data transfers over it should be done in bigger blocks. For this purpose, the program runs two additional helper kernels that gather the data to be communicated into contiguous memory space, before it is transferred to the host. The other helper kernel does the opposite, it scatters the data, once the communication is done. These gather and scatter kernels are also issued into the first stream. Into the second stream, only the kernel that is responsible for updating the lattice sites at the interior of the local domain is issued, these are all the sites that do not participate in the communication.

The order in which the different kernels, communication, and data transfers are started is the key to having the simulation to run correctly and allowing the communication to be performed asynchronously. Each time step starts by updating the lattice sites that contain data that will be communicated. Then the data is gathered into the communications buffers. Once this kernel is done, the kernel that performs the computation for the interior lattice sites is started. At the same time as the interior computation

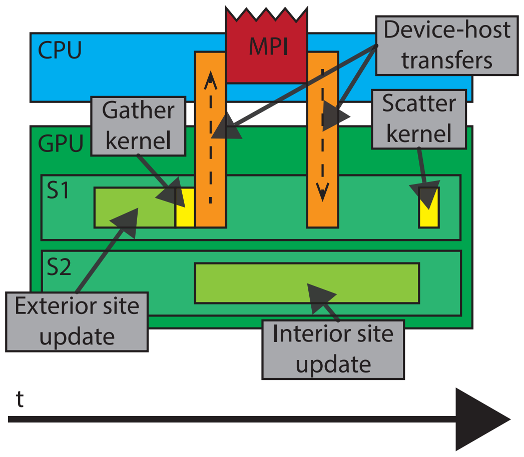

Figure 6.1: The timeline for an asynchronous GPU solver, showing the order the kernels and data transfers are issued.

is started, the transfer of data from the device to the host is also started. This way the transfer between device and host will be overlapped with computation. Once the data is on the host, the communication with relevant compute nodes is done, and once the incoming communication data has been received, the data transfer back to the GPU is started. If the update of the interior lattice sites has not finished by the time the data has been transferred back to the GPU, the communication has been achieved completely asynchronously. Lastly, the data received needs to be scattered back into the correct places in the lattice data for the current compute node. The entire timeline for how the time step is executed is shown in figure 6.1.

The performance impact of running the communication asynchronously depends heavily on the type of simulation geometry used. Factors such as the amount of data that needs to be communicated, the speed of the network used and the amount of computation done by each node affect the potential performance benefit from using asynchronous communication. Programs where either the communication or the computation takes significantly longer than the other will show poor speedup from asynchronous communication, whereas problems where they take roughly the same time asynchronous communication can result in almost two times better performance.

6.1 Dividing the computational domain

To facilitate the overlapping of communication with computation, the computational domain needs to be divided into two separate parts. One part consists of the sites at the edge of the local domain, and another part is the rest of the lattice sites. The naïve approach to dividing the domain is to first update the lattice sites at the very edge of the local domain and then, while the communication happens, update the rest. This approach however will cause memory accesses for all the lattice site updates to become misaligned, and this misalignment will cause memory bandwidth to be wasted.

The misalignment is caused by how most CPUs and GPUs access memory. Instead of a single value multiple values, one cache line, is fetched at once since it is likely that the program will need at least some of the adjacent values as well. With the appropriate data layouts for the GPU, these adjacent values will be the same distribution values from other lattice sites. In this naïve implementation, the kernel updating the lattice sites at the edge of the local domain will not use all the data brought in to the processor, because it only updates some lattice site from the segment and thus only uses some of the values in the cache line. This misalignment also carries over to the interior lattice sites. Cache lines brought in, which contain data from edge sites, will not be fully utilized when updating the interior sites, since the edge sites are already updated, and that data will be brought in unnecessarily. On the GPU, where multiple threads in a warp are fed from the same cache line, this will result in an additional segment being fetched to supply all threads with data. This misalignment will cascade down to any subsequent warps as well, since the current warp consumes data in the segments for the next warp. The effect for the interior sites is, however, often masked by the fact that the additional segment brought in can be used by other warps that can now find that data in the cache structure.

Correcting for this misalignment is easily done by changing how the domain is divided. Instead of processing a single lattice site from the cache line fetched, all the lattice sites that have values that fall within that cache segment are processed. Since that data is already brought into the processor, it should be utilized at that time as well. The only downside is that the wider segments around the edge sites reduce the amount of computation that can be used to hide the communication. Using a whole cache line aligned segment around the edge sites gave 1.16 times better performance for the solver. In this case, the test was carried out on a Kepler basted GPU and using segments of 16 lattice sites.

6.2 Removal of halo sites

To make the implementation of the lattice Boltzmann method and other lattice based solvers easier, an extra layer of lattice sites is often added around each local domain on each compute node. Such a layer, referred to as a ghost or halo layer, simplifies the propagation of values moving outside of the local domain. The idea behind having a layer of ghost sites is that it makes the propagation step easier, since the same propagation operation can be applied to all lattice sites in the same way, regardless of where in the local domain the site is located. The values from these ghost sites are then collected and placed into contiguous memory locations, before the communication starts.

When adding an extra layer around the computational domain, that layer is not used for the collision operation. The effect of this unused data is the same as allocating memory space for the solid lattice sites. Data from these ghost sites are not used by the simulation. Since a full cache line will be fetched for each access, this unused data will be moved around and will consume memory bandwidth and affect the performance of the code.

Indirect addressing enables the solver to arrange the lattice sites in any order, which allows the solver to be implemented without utilizing ghost sites. If, instead of propagating values to the ghost sites, the bounce back boundary condition is applied to the components going out of the local domain, there is no need for the ghost sites. Since all the indices are precomputed, applying the bounce back boundary condition will not change how the propagation is applied, since the propagation still reads the target index from the indexing array. The bounced back location is not needed by any other distribution values, as that is where the values after the communication should be written. Since the locations are not needed, the location can be borrowed for the values that would propagate out of the local domain.

This approach offers some benefits. Since no data is allocated for the ghost sites, this reduces the memory usage to some degree. The major benefit is that it allows the solver to more efficiently utilize the memory bandwidth and improve the alignment of data in the memory. The downside is that the bounce back boundary condition now needs to be enforced at the edges of the local domain, even if the value propagates further in the global domain. Getting rid of the halo sites for a GPU based two-lattice solver using the porous media sample yielded a speedup of 1.18.

Combined with the cache line aligned way of dividing the computational domain, all non-propagated accesses of lattice data can be done as perfectly aligned accesses. Furthermore, if the data layout fulfills the data access requirements of the GPUs and vectorized solvers, this allows us to access contiguous vectors with no need to assemble them.

6.3 Performance of asynchronous communication

The performance impact of asynchronous communication depends heavily on the input samples used, what system the code is run on, what network and network topology is used and a myriad of other factors. For the benchmark, a realistic simulation run was chosen, in no way picked to overemphasize the impact of our implementation. The simulation geometry was the same porous media sample as the one used for the GPU scaling tests, and the system used was the Titan supercomputer, using the same GPU based solver as the one used for the scalability tests. For the asynchronous communication test on Titan, we used a large case with the A4 resolution sample running on 1024 nodes. The solver was implemented using the two-lattice algorithm.

For the synchronous case, the solver was implemented so that it completed all the computation before the communication started, and only when the communication was done did it move to the next time step. While the asynchronous alternative divided the lattice sites into edge and interior lattice sites, it first performed the site updates for the edge lattice sites and then, while it started the communication, also started the computation of the interior lattice sites. Both cases were implemented using no halo sites. The observed speedup of going from synchronous to asynchronous was 1.13. The performance per GPU increased from 255 MFLUPS for the synchronous case to 289 MFLUPS for the asynchronous case.

6.4 Asynchronous communication conclusion

With the GPU being a separate part of the system, it leaves the CPU free for other tasks. One task it can work on is the communication. With the CPU doing the communication at the same time as the GPU is busy working on the computation, the communication can be performed without affecting the runtime of the solver. For this, the computation needs to be divided into two parts, such that the warps of the GPU are able access non-propagated data perfectly aligned. This ensures that the solver effectively utilizes the data accessed by the processor. With bandwidth bound solvers, the usage of halo data around the local domain will introduce unused data into the simulation data, which when accessed will waste memory bandwidth.

Chapter 7

Xeon Phi and vectorized lattice Boltzmann

Article [4] focuses on our efforts with a vectorized LB solver and how to enable the solver to efficiently use the Knights landing generation (KNL) of manycore Xeon Phi processors from Intel. While the focus of the optimization was the KNL architecture, some of the improvements are directly portable to systems based on regular Xeon CPUs as well. The optimization work that was applicable was also ported to a Haswell-based CPU only system.

The KNL system used was a Ninja developer platform [93], which has one Xeon Phi 7210 processor [94] running at 1.3 GHz and 96 GB of DDR4 memory running at 2133 MHz. The KNL processor has an additional 16 GB of fast MCDRAM memory sitting on the same package as the processor. For comparison, a dual socket Haswell-based system was used. The Haswell system included two 2690 v3 CPUs [95] running at 2.6 GHz and 64 GB of DDR4 memory running at 2133 MHz. On the Xeon Phi system, all initial runs were carried out with the MCDRAM in cache mode. This mode allows the MCDRAM to act as a large last level cache to the processor. The core configuration set in quadrant mode which exposes all cores and memory controllers as a single shared memory domain.

This chapter covers the optimization work presented in the article and discusses the performance results achieved. The key optimization techniques focus on how the memory bandwidth can be efficient utilized by choosing the appropriate data layouts and the addition of prefetching instructions into the code. In this chapter, we also present a method for applying SIMD vectorization to the LB solver to enable it to efficiently use the vector instructions available on modern CPUs.

While the original publication uses both a porous media case and a Poiseuille flow case, however, for the sake of brevity this chapter will only cover the porous media case since the Poiseuille flow case is an overly ideal simulation. The porous media case is from the same Fontainebleau sandstone sample as the one used for the large-scale GPU simulations on the Titan supercomputer, presented in chapter 5. For the simulations carried out in this work, a 512^3 voxel sub sample from the A4 sample was used.

7.1 Vectorized lattice Boltzmann

There are two obvious ways of applying vectorization to a lattice Boltzmann implementation. The vectorization can either be done to the update of a single lattice site or to the update of multiple lattice sites at the same time. While the collision operation does include some symmetry, for instance, between values in opposing directions, and it is feasible to pack these into vectors, they will not optimally fill out these vectors. Part of the collision operation also requires summing together all values for a site, and summing the components of a SIMD vector is not the intended use of vectors and requires multiple operations.

A better solution is to vectorize over multiple lattice site updates, filling the vectors with the same distribution values, just from different lattice sites. For the collision operation, vectorizing over multiple sites will apply the same operation to data from multiple lattice sites. This is easily implemented by using vector data types instead of scalar types for the collision function. With vector types that have the common arithmetic operations defined for the vector data types, the collision operator can be written in the same way the scalar version is written. Implementing these operations for the vector data types is easily done with overloaded operators that will call the intrinsic functions for those operations. Listing 7.1 shows how the BGK collision operator is implemented using 512-bit vectors. It also shows how the addition operator is implemented for 512-bit vector types.

Listing 7.1: Pseudo code showing how the vectorized collision operation is implemented using AVX-512 vectors.

```
func Update(__m512d f[Q] ,double g, double tau, double dt)
    __m512d den = 0
    __m512d mom[3] = 0
    for each i do
        den += f[i] end
        mom[0] += C[i][0]*f[i]
        mom[1] += C[i][1]*f[i]
        mom[2] += C[i][2]*f[i]
    end
    __m512d vel[3] = mom/den + dt*g/2

    __m512d feq[Q]
    Eq2(den,vel,feq)
    // Relaxation (BGK) and
    // external forcing (linear)
    double r1 = dt/tau, r2 = 1-r1/2
    for each i do
        __m512d fneq = f[i]-feq[i]
        __m512d cdotg = DotProd8(C[i],g)
        __m512d facc = dt*W[i]*den*cdotg/CT2
        f[i] -= r1*fneq // BGK
        f[i] += r2*facc // forcing
    end
end

func Eq2(__m512d den, __m512d vel[3], __m512d feq[Q])
    __m512d udotu = DotProd8(vel,vel)
    for each i do
        __m512d cdotu = DotProd8(C[i],vel)
        feq[i] = 1-udotu/2CT2
        feq[i] += cdotu/CT2
        feq[i] += cdotu*cdotu/2CT4
        feq[i] *= W[i]*den
    end
end

operator +(__m512d lhs, __m512d rhs)
    return _mm512_add_pd (lhs, rhs)
end
```

Listing 7.2: Pseudo code showing how the propagation step is vectorized using AVX-512 vectors.

```
// iterate and process fluid sites in blocks of 8 consecutive
    sites

func Evolve(double tstep, double g, double tau, double dt)
    // read,update,and
    // write fi values
    for n=0 until n < fsN
        using n+=8 do
        __m512d f[Q] = Gather8(n)
        Update(f,g,tau,dt)
        Store8(f,n)
    end
end

func Gaterh8(int n)
    for each i do
        f[i] = _mm512_i32gather_pd(propIndex[i],buffIn, 8)
    end

func Store8(__m512d f, int n)
    for each i do
        _mm512_store_pd(buffOut, f[i])
    end
end
```

In a vectorized implementation, data should ideally be accessed as contiguous vectors from the main memory of the system. That way the data can be loaded as a complete vector and not assembled into one from scattered values. Ideally, for the best performance these loads should also be done from addresses aligned on vector length boundaries. This access requirement cannot be fulfilled for the propagation step of the LB method, as the propagation will inevitably move the location of the lattice data. At the very least, the addresses written to will no longer be aligned properly. In a more realistic simulation that includes solid lattice sites, the components in one vector will need to be accessed from scattered memory locations. This is because the boundary condition enforced when propagating into a solid site usually causes a divergent memory access from the rest of the vector. Non-propagated accesses, however, can be done as contiguous vectors aligned on vector length boundaries, if the data layout chosen allows for this.

Assembling and disassembling vectors was made easier in the AVX2 and AVX-512 instruction sets, as these introduced gather and scatter operations. The AVX2 instruction set supports only gather operations, while the AVX-512 supports both gather and scatter operations. Gather and scatter instructions take as arguments a pointer from where to start the access and an offset from the pointer for each component in the vector. Using these gather and scatter instructions allow the processor to do indirect access for each component in a vector with a single instruction. Vectorizing the propagation step of the LB solver using these function is a far easier task than without them, since the programmer no longer needs to manually pack and unpack data for the vectors. With these functions, any propagated accesses can be done with a single call to an intrinsic function, since the propagated access of lattice data are effectively indirect memory accesses.

Listing 7.2 shows a pseudo code example for how the propagation can be vectorized using 512-bit vectors. The propagation is done using a pull scheme two lattice algorithm, with the propagation happening when the values are read. The propagation is done using the `_mm512_i32gather_pd` intrinsic. When written back into memory, a normal store function is used that will store the values of the vector into contiguous memory, in the order they are in the vector.

7.2 Algorithm performance

The AA and the two-lattice algorithm were both tested on a regular Haswell generation based Xeon system and the Knights landing Xeon Phi system. Both systems used the stream optimized data layout and ran with the maximum number of threads available. These tests showed that, on both systems, the AA algorithm offers superior performance, compared to the two-lattice approach. On the Haswell system, the AA algorithm can achieve 246.2 MFLUPS with the porous media case, while the two-lattice implementation is able to reach only 123 MFLUPS. On the KNL system, the AA algorithm performs 6 times better than the two-lattice algorithm. In this case, the AA algorithm achieved 178.2 MFLUPS, while the two-lattice algorithm achieved only 30 MFLUPS. On neither system is there any difference between implementing the two-lattice algorithm using either a push or a pull scheme.

7.3 Thread count and vector length

The Haswell system supports up to 256-bit vectors through the use of the AVX2 instruction set which allows a core to process 4 double-precision values with one instruction. The KNL architecture is the first system to use the 512-bit version of the AVX instruction set, called AVX-512, allowing it to process vectors which fit 8 double-precision values.

Figure 7.1 shows the measured performance for different vector lengths used on the two tested systems. On the Haswell system, the performance increase from using scalar values to using 256-bit AVX2 vectors was 1.055 for the porous media case. For the KNL system, switching from scalar to AVX2 for the LB solver yielded a speedup of 1.77. Further expanding the

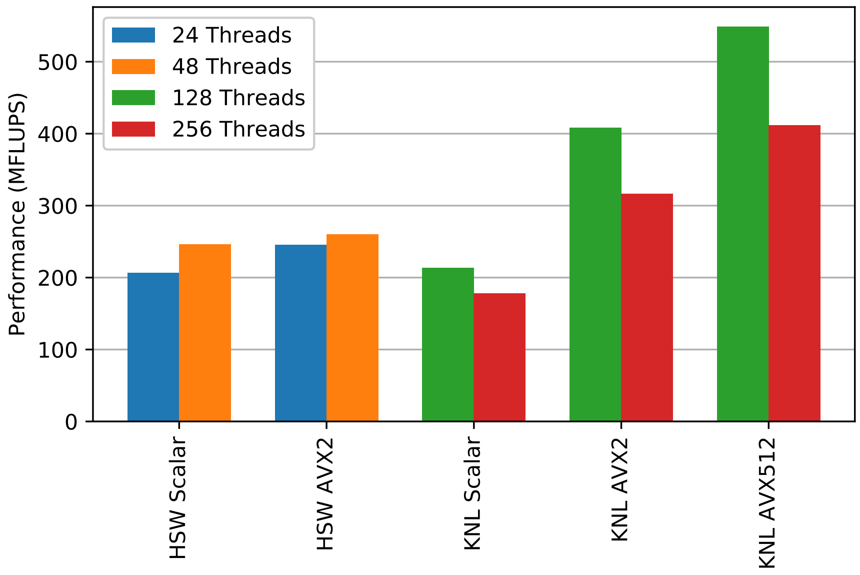

Figure 7.1: Performance of different vector lengths and thread counts on a dual socket Haswell system and a Knights landing Xeon Phi.

vectors to AVX-512 vectors shows a performance increase of 1.3, compared to AVX2 vectors. The total speedup from scalar to fully AVX-512 vectorized code on the KNL system is 2.31.

The KNL processor supports up to 4-way simultaneous multi-threading (SMT), allowing the 64 core KNL processor to run up to 256 threads. The regular Xeon CPUs support only 2-way SMT, resulting in a 12 core CPU being able to run 24 treads simultaneously. SMT improves the efficiency of superscalar CPUs by allowing multiple threads to use the resources of the core at the same time. Increasing the number of threads each core is running does have the side effect of increasing the contention for the resources that exist within a core.

On the Haswell system, running the simulation with SMT using 2 threads per core instead of just one yields a 1.05 times better performance for the porous media case. On the KNL system, running too many threads has a detrimental impact on the performance. The solver performs 1.33 times better running with only 128 threads, compared to 256 threads. More active threads on the KNL system leads to less resources, cache space and prefetching units available per thread. Since two cores already share some of those resources [37], each thread ends up with far fewer resources than on the Haswell comparison system.

7.4 Prefetching

The latency of the computers main memory is high, in the order of 100 times higher compared to the latency of the internal caches of the processor [24]. Most modern x86 based CPUs will try and predict what data the program will need next, and pre-emptively start moving the data the processor predicts will be needed into the CPU before it is accessed by the program. This prefetching of data is often implemented in hardware, where the processor has a number of prefetching units per core that attempt to identify the access patterns used, and predict what data will be accessed next. This way, when the program eventually needs the data, it will already reside in the much lower latency caches of the processor, thus the full latency of the main memory can be avoided. While prefetching cannot improve the memory bandwidth utilization of a program past what the theoretical bandwidth of the slowest memory used is, it is key to hiding the latency of that memory.

In case the hardware is not able to predict the data access patterns, the programmer can still try to help the hardware by using software prefetching instructions inserted into the program. These instructions can be used to manually start the data movement before the data is needed. This is especially useful for indirect memory accesses, where the programmer might know what data will be accessed next, while the processor might not be able to deduce this from the previous access patterns.

In the LB solver, there is a loop that iterates over the lattice sites being updated. Prefetching instructions are added as an additional step each iteration of the loop. This step issues prefetching instructions for the values needed for a future iteration of the loop. Testing how many iterations ahead the prefetching should be issued showed what prefetching distance offered the best performance.

In the LB case, running on regular Xeon CPUs of the Haswell generation, prefetch instructions were only added to the odd time steps. Since the odd time steps apply the propagation which involves indirect memory accesses, it is harder for the processor to predict these. There was also no observable performance increase when adding the instructions to the even time steps. When running with the porous media sample, adding prefetching for the propagated data accesses provided a speedup of 1.05.

In the KNL processor, the need for manual prefetching is far greater. On this architecture, the prefetching hardware has been slightly extended, compared to what is available on the Haswell system. However, in the KNL processor, the hardware capable of prefetching data from the main memory is shared between two cores, resulting in each core having less prefetching resources than the HSW core. Combined with the fact that the KNL cores can run four threads instead of two, this further diminishes the prefetching resources available for a single thread, if all threads are used.

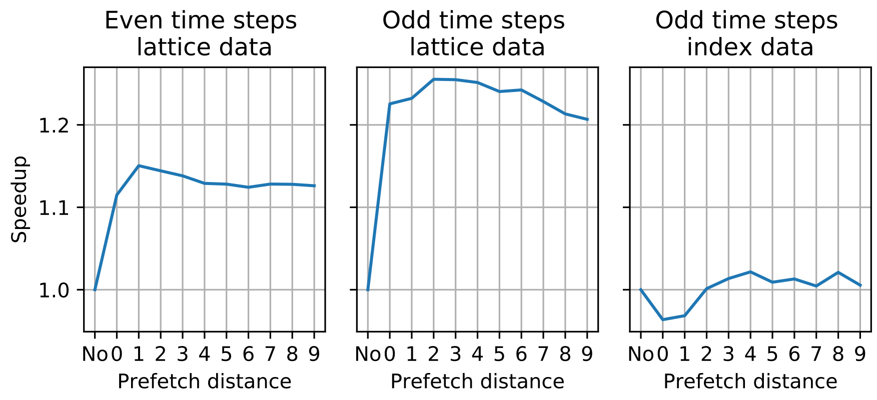

Figure 7.2: A comparison of the performance of the LB solver running on a Xeon Phi with different prefetching distances.

On the KNL system, both the non-propagated and the propagated accesses of lattice data benefitted from the addition of prefetching instructions. In addition, there is also an observable increase in the performance when adding prefetching instructions for the index variables used for the propagation. Figure 7.2 examines how far ahead data should be prefetched. It is split into odd and even time steps, and shows the performance when prefetching the lattice data N iterations ahead of the current one. The figure also includes the performance for the odd time step when adding prefetching instructions for the index data used for the propagation. For the even time step accesses of the distribution values, the optimal distance is to issue the prefetching instruction 1 iteration ahead, for the lattice data for the odd time steps the ideal distance is 2 iterations. Finally, the indexing data for the odd steps gives the best performance if prefetched 4 iterations ahead. The total speedup when adding prefetching to the LB solver was 1.333. The figure also shows how the performance decreases if prefetches are done too far ahead.

7.5 Data layout

As with the GPU, the data layout affects the performance the solver can achieve on CPUs and Xeon Phis. With vectorization being key on both architectures, the data layouts that are not suited for vectorized accesses are discarded. Performance comparison was carried out using the stream optimized and the vectorized version of the collision optimized data layout, the AoSoA layout, with different numbers of lattice sites packed into the structure. The data layout performance was measured before and after the prefetching instructions were added.

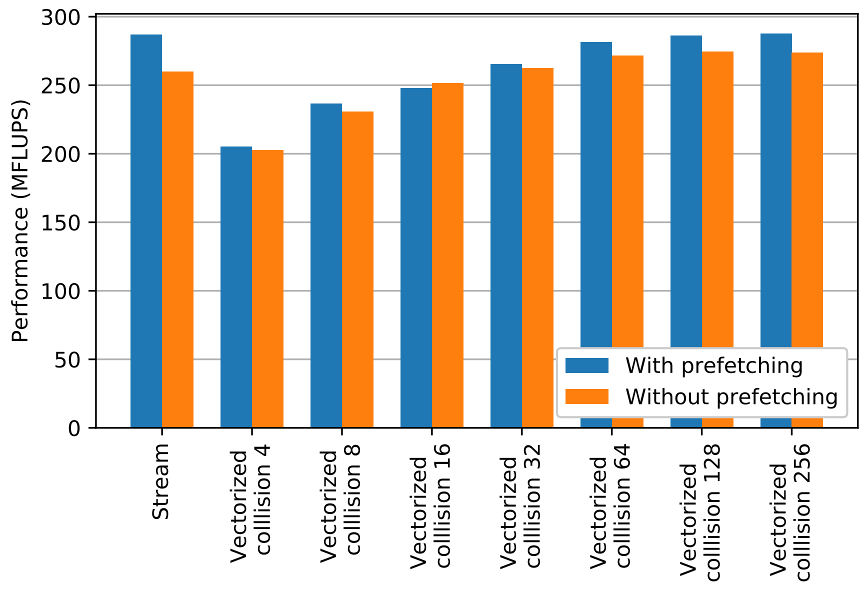

Figure 7.3: The effect different data layouts have on the performance of the solver running on a dual socket Haswell system.

Figure 7.3 shows the performance for the different data layouts, with and without prefetching on the Haswell system. With no prefetching, the peak performance is achieved using the AoSoA layout, with 256 sites in each structure. Adding prefetching does not change what data layout is the most optimal, but the stream optimized layout sees a larger performance increase, to about the same performance as the AoSoA layout.

On the KNL system, shown in figure 7.4, the addition of prefetching changes which data layout is optimal. With no prefetching added to the code, the optimal data layout was the AoSoA, with 64 sites packed into each structure. With the prefetching added, the optimal data layout was still the AoSoA layout, however now with only 8 sites packed into each structure, as tightly packed as possible, with AVX-512 vectors containing 8 values. With this data layout, the performance achievable was 841.8 MFLUPS, 1.52 times better for the same layout without prefetching.

7.6 MCDRAM and NUMA modes

The high speed MCDRAM on the KNL system can be configured in three modes, the simplest being caching mode, which uses the MCDRAM as a large last level cache and hides it from the user. It is, however, possible to configure it as a separate memory space that the programmer can use,

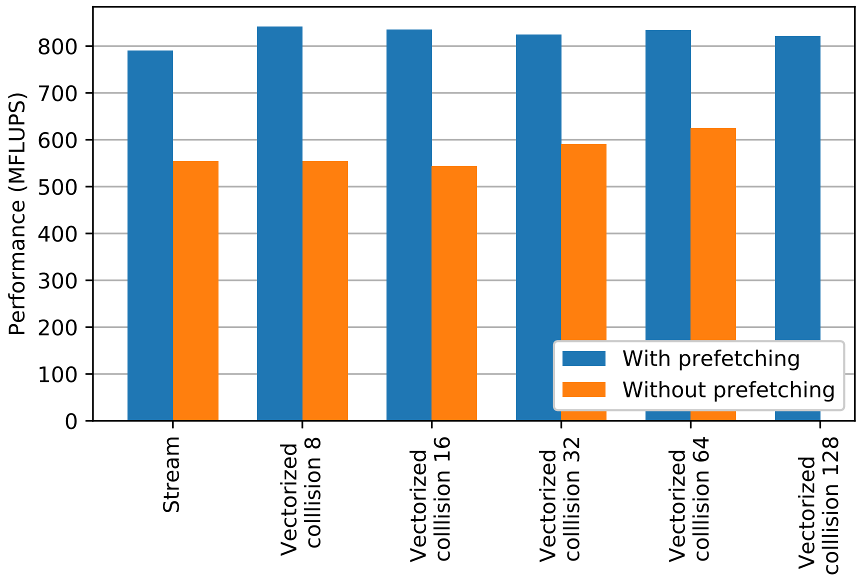

Figure 7.4: The performance for different data layouts for the solver running on a Knight landing Xeon Phi system.

referred to as flat mode. It can also be configured as a hybrid cache and flat memory, where some of it acts as a cache and some as a separate memory space. Utilizing the memory in flat mode requires the use of specialized allocation functions that allocate space from the MCDRAM. These are used to store data that need high speed access. In flat mode, there is no risk of data used by the simulation being evicted from the cache, and it can provide some additional bandwidth [37].

The mesh network connecting the cores to each other within the KNL processor can be configured in varying cluster modes. The default for most systems is quadrant mode, which presents the processor as one unified memory space and allows the processor to be used as a regular multicore CPU. Of the other modes offered, the most interesting one for running an LB solvers is the sub NUMA cluster (SNC) mode, particularly the SNC-4 mode, which divides the processor into 4 parts. In this mode, the MCDRAM is divided so that each cluster has access to a quarter of the memory through two memory controllers. The benefit of SNC-4 is that, by dividing the processor into smaller parts, the memory used by the cores within a cluster is allocated from the nearest MCDRAM location. Accessing memory from the nearest memory space lowers the latency for the memory accesses [37].

Switching the memory into flat mode and allocating the lattice data and propagation indices explicitly in the faster memory yielded a speedup of 1.083 for the porous media case. Switching the core to core interconnect from the default quadrant mode to SNC-4 when the MCDRAM is in flat mode gives another 1.02 times speedup.

7.7 Streaming stores

Streaming or non-temporal stores refers to store operations that can bypass the cache structure of the processor. Normally, on an X86 architecture, when a value from memory is updated, if the cache line of that value is not in the cache structure, the cache line needs to be read into the cache structure for the value to be modified. In the case where all the values in the cache line are updated, it is possible to circumvent the need to bring the cache line in by using streaming stores. These will overwrite the entire cache line in the memory. Overwriting the entire cache line could possible halve the memory bandwidth needed to store that data by eliminating the read for ownership operation [96]. In the case that a streaming store is issued for a value already in the cache structure, on most X86 architecture that also marks the cache line for eviction. Evicting the line could free up space in the cache and save data that is still needed from being evicted from the cache.

Since the vector length is the same as the cache line length in the KNL system, the use of streaming stores is straightforward. When storing a vector back into memory, the programmer can, using intrinsic functions, issue a streaming store for that operation, if the data is not needed again immediately. With the AA algorithm, there is only one point where streaming stores are applicable, and that is when storing non-propagated lattice data. Since the propagated writes are going to scattered location, and in some cases updating only one value from a cache line in the process, it is not advisable to issue streaming stores for such accesses. When issuing streaming stores for these runs, the risk of evicting data that will still be used is high. In fact, no streaming scattered store functions exist.

From a performance perspective, the use of streaming stores has only a marginal effect on the performance. In flat mode, the speedup observed for the porous media case was 1.023, in cache mode the performance increase was 1.032. However, since in the AA algorithm any locations that data are stored to are already in the cache structure, the only benefit is that the programmer can influence what data is evicted from the cache.

7.8 Conclusion

In the regular Haswell system, vectorizing the LB solver resulted in minor performance gain of around 1.05. Further choosing the optimal data layout yielded a speedup of 1.06, and adding prefetching to the accesses of the lattice data for the odd time steps gave another 1.05 times better performance. Overall, the speedup achieved from vectorizing and tuning the LB solver on the Haswell system was 1.17, or 41.5 MFLUPS, resulting in a final speed of 287.7 MFLUPS.

The KNL system saw a larger performance increase when vectorizing the solver. Comparing scalar to full AVX-512 vectors yields 2.31 times better performance. Reducing the number of running threads from 256 to 128 gives another 1.35 times speedup. Changing the data layout from the stream optimized layout to the AoSoA yields another 1.14 times better performance. With the more limited prefetching of a KNL core, the addition of manual prefetching to the code gave 1.3 times higher performance. Tweaking the parameters of the data layout with prefetching gave a minor 1.025 times increase in the performance. Switching the MCDRAM to flat mode increased the performance 1.083 times, and the final tuning with SNC mode and streaming stores gave another 1.033 and 1.02 times better performance, respectively.

Starting from the baseline AA algorithm with the optimization, the total speedup was 5.4 or 783.4 MFLUPS, with the maximum performance measured for the porous media case being 961.6 MFLUPS. Most of the large speedups come from the way memory is used by the program. The data layout used and the addition of prefetching instructions into the code being key to how efficiently the memory bandwidth can be utilized.

It should be noted that while the optimization techniques used are likely to at least to some degree be applicable to future architectures, the specific values used are likely to change. How far ahead data needs to be prefetched and how many lattice sites are packed into each structure will probably change. We already see that the Haswell and Knights Landing systems achieve the best performance with different number of lattice sites used in each structure in the AoSoA data layout. However, techniques such as vectorization and the addition of prefetching work on both architectures.

With the AVX-512 vectorized solver, each site update requires 340.5 bytes of data to be moved. The memory needed comes from reading and writing 19 double-precision values. The odd steps also need 18 indexing values, and each vectorized site needs one integer for loop iteration to lattice site mapping. That results in at least 327 GB/s of bandwidth to sustain that simulation speed.

An evolution of the same GPU solver used on Titan can on a P100 GPU achieve 1181.5 MFLUPS. When comparing with the GPUs, one needs to consider that the GPUs deliver a higher memory bandwidth. While the Xeon Phi 7210 processor can deliver 436.5 GB/s of bandwidth in the stream triad benchmark, the P100 GPU delivers 550 GB/s in the same benchmark, 113.4 GB/s more than the KNL processor. Using the naïve estimate of 327 GB/s of bandwidth required for the Xeon Phi to run at 961.6 MFLUPS results in the code being able to achieve 75 % of the stream bandwidth. This is a significant improvement from what others reported being achievable on the KNC [97]. Doing the same estimate for the GPU, using 344 bytes per site update, gives 406.4 GB/s for 1181.5MFLUPS or 74 % of the stream bandwidth. Both the GPU and the KNL are similarly efficient at running the porous media case. Taking into account that the Xeon Phi can run much the same code as a regular Xeon CPU and the KNL processor is self-hosted and does not need an additional system to run, we conclude that the KNL processor is an attractive choice for LB simulations.

Chapter 8

The lattice Boltzmann method, a petaflop and beyond

The large-scale simulation runs demonstrated that our LB solver can run on one of the top supercomputers in the world and scale well past one petaflop. Even though it was only able to achieve about 10% of the measured peak arithmetic performance of the machine, we could utilize significant parts of the available memory bandwidth of the machine. With memory bandwidth being the limiting factor performance-wise, the KNL generation of Xeon Phi is a contender for running accelerated LB solvers. This is due to its faster MCDRAM on package memory and an improved architecture from the KNC generation. The newer Pascal generation GPUs also offer a significant performance increase compared with the older Kepler generation, due to their use of HBM2, which operates at up to 3 times higher bandwidth than the GDDR5 memory used on older cards [29].

While these accelerators with their faster memory offer superior performance, this performance comes at the cost of the size of the simulation domain that can fit into a single accelerator. Both the Pascal P100 GPUs and the KNL generation of Xeon Phis have 16GB of fast memory. For the GPU, that is all the memory it has, and in case more data is needed, that must be exchanged over the PCI-e bus, which at 16 GB/s is slow compared to the memory bandwidth of these accelerators. For the few types of systems that support it, currently only PowerPC systems, the NVLink interface can be used to communicate to the host CPU. The first generation of NVLink runs at 20 GB/s per lane, with each Pascal GPU having 4 lanes. Most systems, however, use these lanes for communication between GPUs and not to communicate with the CPU. Both the PCI-e and NVLink busses are still significantly slower than the GPUs own memory, and moving simulation data to and from the GPU during the simulation would severely impact the performance of the solver. The Xeon Phi can also support up to 384 GB of DDR4 memory, in addition to its MCDRAM. If the simulation data does not fit into the MCDRAM and spills over the DDR memory, that will effectively reduce the performance of the solver to run at DDR4 memory speeds. While this is a better situation than the GPU case, it raises the question why to use the accelerator at all for simulations of large systems. A system based on commodity CPUs could be used instead, and those systems are widely more accessible.

Regular CPU based systems have the benefit of supporting vastly more memory per node than the accelerators. The latest generation Intel CPUs support up to 1.5 TB of memory per socket, and AMD systems support up to 2 TB per socket. This would allow the same simulation that required large amounts of memory to be run on far fewer compute nodes using CPUs instead of accelerators. One example comes from our work with Mattila et al. [5], which includes our performance results from the large scale runs on Titan, but also those carried out on a CPU only system. In this case the system used was the ARCHER computer at EPCC. The system consists of 3008 nodes, with two 12 core CPUs and 64 GB of memory per node. On that system, the A2 sample could fit into 320 compute nodes and the entire A1 sample could be simulated using 2880 compute nodes. On ARCHER, half of the A1 sample would fit into 1440 compute nodes, which is less than a tenth of the number of nodes used on Titan. Naturally, the CPU only system ran at a lower performance than the accelerated system, achieving only a little over 150 MFLUPS per node for the A2 sample and around 130 MFLUPS for the A1 sample. With similar CPUs as the ARCHER system, our vectorized CPU solver can, however, perform better than what was reported by Mattila et al.

For simulations of large fluid systems, it is clear that on CPU systems it is far easier for large simulations to fit into the systems. While one can build an accelerated system with the same amount of fast memory, the sheer number of nodes and hardware cost of such systems would outweigh any performance benefits. The downside with the CPU only systems is the speed the simulations will run at. The good news is that the bandwidth of the CPU systems is increasing. Intel's Skylake CPUs come with 6 channels of memory, offering a theoretical performance of 250 GB/s for a dual socket system. AMD‘s Epyc CPUs offer 340 GB/s for a dual socket system, which is more than the theoretical performance of the K20X GPUs in Titan. The bandwidth improvement in both CPU architectures should translate into an improved LB performance on those systems.

On the accelerated side, the memory performance has also continued to increase. Nvidia‘s Volta GPU architecture offers up to 900 GB/s of theoretical bandwidth with the V100 GPUs. This architecture also comes with an improved memory controller, and is claimed to offer 850 GB/s of bandwidth in the stream triad benchmark [98]. Other accelerators, such as the SX-10+ vector processor from NEC which uses 6 HBM stacks [18], compared to the 4 used on Nvidia‘s GPUs, and offer a theoretical memory bandwidth of 1.2 TB/s.

For outright performance and time-critical simulations, the better choice is currently to use accelerated systems. Considering how the evolution of the performance of both accelerators and CPU systems have progressed, the performance advantage of the accelerators is unlikely to change in the near future. However, for large fluid systems, obtaining access to large enough accelerated machines that will fit these simulations will be out of reach for most scientists. In those cases, the additional bandwidth of the newer CPUs will result in per node performance comparable to the performance we achieved using the accelerated Titan supercomputer.

Chapter 9

Conclusion

The work in this thesis has focused on the lattice Boltzmann method and how it behaves on current accelerated supercomputer systems. We have shown the benefit of using computational accelerators and that they offer great performance for running LB solvers. While the solver is not relying on the computational performance of GPUs and manycore accelerators, they do provide significantly higher memory bandwidths than regular CPU systems. With the performance of the LB method being highly reliant on the memory bandwidth of a system, accelerators provide an excellent platform for high speed LB solvers.

We have shown that it is possible to create programs for GPUs without resorting to low level languages, using OpenACC, while reaching similar performance. Thus, the usage of accelerators should not be dismissed because they require specialized programming languages. For the Xeon Phi manycore accelerators, we showed that, while the Xeon Phi processor can run standard C++ code, it is important to fine tune the solver for that architecture to get the best performance. Unsurprisingly, being memory bound, most of the key optimizations applied revolved around how the system used the smaller on package high bandwidth memory.

Large scale accelerated systems are becoming more and more prevalent, and we showed that it is possible to scale a GPU-accelerated LB solver to well over one petaflop of computational performance using these systems. While this was only 10% of the Linpack performance of the system, we utilized 84% of the available memory bandwidth. Therefore, bandwidth bound simulations, such as LB simulations, would benefit performance-wise if new computing systems are built with sufficient memory bandwidth and not focusing solely on the arithmetic performance.

Bibliography

[1] F. Robertsén, K. Mattila, and J. Westerholm. Lattice Boltzmann method on GPUs: a comparison between OpenACC and CUDA. Submitted 2016.

[2] 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 *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 604–609. IEEE, mar 2015.

[3] F. Robertsén, J. Westerholm, and K. Mattila. Designing a graphics processing unit accelerated petaflop capable lattice Boltzmann solver: Read aligned data layouts and asynchronous communication. *The International Journal of High Performance Computing Applications*, 31(3):246–255, aug 2016.

[4] F. Robertsén, K. Mattila, and J. Westerholm. High-performance SIMD implementation of the lattice-Boltzmann method on the Xeon Phi processor. *Concurrency and Computation: Practice and Experience*, 2018. Minor revision Jan 2018.

[5] K. Mattila, T. Puurtinen, J. Hyväluoma, R. Surmas, M. Myllys, T. Turpeinen, F. Robertsén, J. Westerholm, and J. Timonen. A prospect for computing in porous materials research: Very large fluid flow simulations. *Journal of Computational Science*, 12:62 – 76, 2016.

[6] J. Anderson. *Computational Fluid Dynamics*. McGraw-Hill Education - Europe, 1995.

[7] Fédération Internationale de l'Automobile. *2017 F1 Sporting Regulations*, 2017.

[8] S. Anthony. Formula 1: A technical deep dive into building the world's fastest cars, 2017. Available at: `https://arstechnica.com/cars/2017/04/formula-1-technology/` Retrived: 2017-12-07.

[9] N. Hemsoth. A look inside China's chart-topping new supercomputer, 2016. Available at: `https://www.nextplatform.com/2016/06/20/look-inside-chinas-chart-topping-new-supercomputer/` Retrived: 2017-12-07.

[10] T. Damkroger. Unleashing high-performance computing today and tomorrow. Available at: `https://itpeernetwork.intel.com/unleashing-high-performance-computing/` Retrived: 2017-12-07.

[11] M.D. Mazzeo and P.V. Coveney. HemeLB: A high performance parallel lattice-Boltzmann code for large scale fluid flow in complex geometries. *Computer Physics Communications*, 178(12):894–914, jun 2008.

[12] What is high performance computing? Available at: `https://insidehpc.com/hpc-basic-training/what-is-hpc/` Retrived: 2017-11-16.

[13] TOP500.org. Top500 description. Available at: `https://www.top500.org/project/top500\_description/` Retrived: 2018-17-03.

[14] J. Dongarra, P. Luszczek, and A. Petitet. The LINPACK benchmark: past, present and future. *Concurrency and Computation: Practice and Experience*, 15(9):803–820, 2003.

[15] J. McCalpin. Memory bandwidth and system balance in HPC systems. In *SC16*, 2016.

[16] R. Farber. *CUDA Application Design and Development.* Elsevier Inc., 2011. ISBN: 978-0-12-388426-8.

[17] J. Jeffers J. and Reinders. *Intel Xeon Phi Processor High Performance Programming.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2013.

[18] T. Morgan. A deep dive into NEC's Aurora vector engine, 2017. Available at: `https://www.nextplatform.com/2017/11/22/deep-dive-necs-aurora-vector-engine/` Retrived: 2017-12-11.

[19] S. Behling, R. Bell, P. Farrell, H. Holthoff, F. O'Connell, and W. Weir. *The POWER4 Processor Introduction and Tuning Guide.* 2001.

[20] J. De Gelas and I. Cutress. Sizing up servers: Intel's Skylake-SP Xeon versus AMD's EPYC 7000 - the server CPU battle of the decade?, 2017. Available at: `https://www.anandtech.com/show/11544/intel-skylake-ep-vs-amd-epyc-7000-cpu-battle-of-the-decade` Retrived: 2017-12-11.

[21] P. Kennedy. AMD EPYC 7000 series architecture overview for Non-CE or EE Majors, 2017.

[22] J. De Gelas. Intel Xeon E5 version 3: Up to 18 Haswell EP cores, 2014. Available at: `https://www.anandtech.com/show/8423/intel-xeon-e5-version-3-up-to-18-haswell-ep-cores-` Retrived: 2017-12-11.

[23] J. De Gelas. Intel's Xeon E5-2600 V2: 12-core Ivy Bridge EP for Servers, 2013. Available at: `https://www.anandtech.com/show/7285/intel-xeon-e5-2600-v2-12-core-ivy-bridge-ep` Retrived: 2017-12-11.

[24] U. Drepper. What every programmer should know about memory, 2007.

[25] D. Tullsen, S. Eggers, and H. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings 22nd Annual International Symposium on Computer Architecture*, pages 392–403, June 1995.

[26] M. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, C-21(9):948–960, Sept 1972.

[27] Intel. Intel advanced vector extensions programming reference. Technical report, June 2011.

[28] Intel. Intel architecture instruction set extensions programming reference. Technical report, August 2015.

[29] NVIDIA Corporation. NVIDIA Tesla P100.

[30] NVIDIA Corporation. *NVIDIA Tesla P100 GPU accelerator*, 2016.

[31] PCI-SIG. *PCI Express® Base Specification Revision 2.1.* PCI-SIG, 2009.

[32] S. Gupta. IBM & NVIDIA present the NVLink server you've been waiting for, 2016.

[33] G. Chrysos. Intel Xeon Phi coprocessor (codename Knights Corner). In *2012 IEEE Hot Chips 24 Symposium (HCS)*, pages 1–31, Aug 2012.

[34] A. Sodani. Knights landing (KNL): 2nd generation Intel Xeon Phi processor. In *2015 IEEE Hot Chips 27 Symposium (HCS)*, pages 1–24, Aug 2015.

[35] G. Crimi, F. Mantovani, M. Pivanti, S.F. Schifano, and R. Tripiccione. Early experience on porting and running a lattice Boltzmann code on the Xeon-phi co-processor. *Procedia Computer Science*, 18(Supplement

C):551 – 560, 2013. 2013 International Conference on Computational Science.

[36] E. Calore, A. Gabbana, S.F. Schifano, and R. Tripiccione. Optimization of lattice Boltzmann simulations on heterogeneous computers. *The International Journal of High Performance Computing Applications*, page 1094342017703771.

[37] J. Jeffers, J. Reinders, and A. Sodani. *Intel Xeon Phi Processor High Performance Programming: Knights Landing Edition Edition.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2016.

[38] J. Kim, W. J. Dally, S. Scott, and D. Abts. Technology-driven, highly-scalable Dragonfly topology. In *2008 International Symposium on Computer Architecture*, pages 77–88, June 2008.

[39] InfiniBand Trade Association. *InfiniBand Architecture Specification Volume 2*, November 2012.

[40] M. Birrittella, M. Debbage, R. Huggahalli, J. Kunz, T. Lovett, T. Rimmer, K. Underwood, and T. Zak. Intel® Omni-path architecture technology overview. 2015.

[41] B. Alverson, E. Froese, L. Kaplan, and D. Roweth. *Cray® XC^{TM} Series Network.* Cray Inc., 2012.

[42] Y. Ajima, T. Inoue, S. Hiramoto, Y. Takagi, and T. Shimizu. The Tofu interconnect. *IEEE Micro*, 32(1):21–31, Jan 2012.

[43] NVIDIA Corporation. Summit and Sierra supercomputers: An inside look at the U.S. department of energy's new pre-exascale systems. 2014.

[44] J. Dongarra. Report on the sunway TaihuLight system. Tech report, University of Tennessee, 2016.

[45] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard Version 3.0*, 2012.

[46] R. Rabenseifner, G. Hager, and G. Jost. Hybrid MPI/OpenMP parallel programming on clusters of multi-core SMP nodes. In *2009 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing*, pages 427–436, Feb 2009.

[47] S. Páll, M. J. Abraham, C. Kutzner, B. Hess, and E. Lindahl. *Tackling Exascale Software Challenges in Molecular Dynamics Simulations with GROMACS*, pages 3–27. Springer International Publishing, Cham, 2015.

[48] B. Chapman. *Using OpenMP*. MIT University Press Group Ltd, 2007.

[49] OpenMP Architecture Review Board. *OpenMP Application Program Interface*, July 2015.

[50] OpenMP. OpenMP compilers & tools. Available at: `http://www.openmp.org/resources/openmp-compilers/` Retrived: 2017-12-02.

[51] OpenMP Architecture Review Board. *OpenMP Application Program Interface*, July 2013.

[52] *Professional CUDA C Programming*. Wrox Press Ltd., Birmingham, UK, UK, 1st edition, 2014.

[53] OpenACC.org. *The OpenACC^{TM} Application Programming Interface*, November 2015.

[54] S. Succi, R. Benzi, and F. Higuera. The lattice Boltzmann equation: A new tool for computational fluid-dynamics. *Physica D: Nonlinear Phenomena*, 47(1):219 – 230, 1991.

[55] D. Groen, J. Hetherington, H. B. Carver, R. W. Nash, M. O. Bernabeu, and P. V. Coveney. Analysing and modelling the performance of the HemeLB lattice-Boltzmann simulation environment. *Journal of Computational Science*, 4(5):412–422, sep 2013.

[56] A. Ribeiro, D. Casalino, and E. Fares. *Lattice-Boltzmann Simulations of an Oscillating NACA0012 Airfoil in Dynamic Stall*, pages 179–192. Springer International Publishing, Cham, 2016.

[57] 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.

[58] 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, 2013.

[59] J. Tolke and M. Krafczyk. Teraflop computing on a desktop PC with GPUs for 3D CFD. *Int. J. Comput. Fluid Dyn.*, 22(7):443–456, August 2008.

[60] Y. Qian, D. d'Humières, and P. Lallemand. Lattice BGK models for Navier-Stokes equation. *Europhysics Letters*, 17(6):479–484, 1992.

[61] I. Ginzburg and D. d'Humières. Multireflection boundary conditions for lattice Boltzmann models. *Physical Review E*, 68(6):066614, 2003.

[62] P. Lavallée, J. P. Boon, and A. Noullez. Boundaries in lattice gas flows. *Physica D: Nonlinear Phenomena*, 47(1):233 – 240, 1991.

[63] P. Balaji, D. Buntinas, D. Goodell, W. Gropp, S. Kumar, E. Lusk, R. Thakur, and J. Träff. *MPI on a Million Processors*, pages 20–30. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[64] F. Robertsén. Implementing a high performance lattice-Boltzmann solver on multi GPU systems. Master's thesis, 2013.

[65] 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.

[66] 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.

[67] K. Mattila, J. Hyväluoma, T. Rossi, M. Aspnäs, and J. Westerholm. An efficient swap algorithm for the lattice Boltzmann method. *Computer Physics Communications*, 176(3):200–210, 2007.

[68] P. Bailey, J. Myre, S. Walsh, D. Lilja, and M. Saar. Accelerating lattice Boltzmann fluid flow simulations using graphics processors. *2013 42nd International Conference on Parallel Processing*, 0:550–557, 2009.

[69] S. Williams, A. Waterman, and D. Patterson. Roofline: An insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):65–76, April 2009.

[70] J. Qi, K. Jain, H. Klimach, and S. Roller. Performance evaluation of the LBM solver musubi on various HPC architectures. In *Advances in Parallel Computing*, volume 27, pages 807–816. 04 2016.

[71] T. Deakin, J. Price, M. Martineau, and S. McIntosh-Smith. *GPU-STREAM v2.0: Benchmarking the Achievable Memory Bandwidth of Many-Core Processors Across Diverse Parallel Programming Models*, pages 489–507. Springer International Publishing, Cham, 2016.

[72] J. McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pages 19–25, December 1995.

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

[74] Y. Torres, A. Gonzalez-Escribano, and R. Llanos. Understanding the impact of CUDA tuning techniques for Fermi. In *2011 International Conference on High Performance Computing & Simulation*. IEEE, jul 2011.

[75] NVIDIA. *Profiler User's Guide*. NVIDIA, April 2017.

[76] J. White. INCITE overview, 2012. Available at: `http://www.doeleadershipcomputing.org/wp-content/uploads/2011/01/2013\_INCITE\_Overview\_10\_31\_2012.pdf` Retrived: 2017-12-07.

[77] L. Anand. Inside the titan supercomputer: 299k AMD x86 cores and 18.6k NVIDIA GPUs, 2012. Available at: `https://www.anandtech.com/show/6421/inside-the-titan-supercomputer-299k-amd-x86-cores-and-186k-nvidia-gpu-cores` Retrived: 2017-23-09.

[78] Introducing Titan — the world's #1 open science supercomputer. Available at: `https://www.olcf.ornl.gov/titan/` Retrived: 2014-04-08.

[79] Cray inc. *Cray XK7*. Cray Inc., 2011.

[80] Cray inc. *The Gemini Network*. Cray Inc., 2010.

[81] TOP500 News Team. Oak ridge claims no. 1 position on latest TOP500 list with Titan, 2012.

[82] J. Osborn, M. Norman, and J. White. NCITE proposal writing webinar, 2013.

[83] G. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM.

[84] R. Hilfer and T. Zauner. High-precision synthetic computed tomography of reconstructed porous media. *Physical Review E*, 84(6):062301, 2011.

[85] T. Morgan. Intel shuts down Lustre file system business, 2017. Available at: `https://www.nextplatform.com/2017/04/20/intel-shuts-lustre-file-system-business/` Retrived: 2017-12-07.

[86] Oracle. *Lustre File System^{TM} Operations Manual for Lustre*, January 2011.

[87] Lustre.org. *Introduction to Lustre Architecture*, 2017.

[88] P. Corbett, D. Feitelson, S. Fineberg, Y. Hsu, B. Nitzberg, J. Prost, M. Snirt, B. Traversat, and P. Wong. Overview of the MPI-IO parallel I/O interface. In *Input/Output in Parallel and Distributed Computer Systems*, pages 127–146. Springer, 1996.

[89] A. Bland, J. Wells, O. Messer, O. Hernandez, and J. Rogers. Titan: Early experience with the Cray XK6 at Oak Ridge national laboratory. *Cray User Group*, 2012.

[90] NVIDIA. *NVIDIA® TESLA® GPU ACCELERATORS*. NVIDIA, October 2013.

[91] D. Buettner, J. T. Acquaviva, and J. Weidendorfer. Real asynchronous MPI communication in hybrid codes through OpenMP communication tasks. In *2013 International Conference on Parallel and Distributed Systems*, pages 208–215, Dec 2013.

[92] H. Pritchard, D. Roweth, D. Henseler, and P. Cassella. Leveraging the Cray Linux environment core specialization feature to realize MPI asynchronous progress on Cray XE systems. In *Proceedings of the Cray User Group Conference*, 2012.

[93] Intel. Developer access program (DAP) for Intel Xeon Phi processor. Available at: `http://dap.xeonphi.com/` Retrived: 2017-23-09.

[94] Intel. Intel® Xeon Phi$^{\text{TM}}$ processor 7210. Available at: `https://ark.intel.com/products/94033/Intel-Xeon-Phi-Processor-7210-16GB-1\_30-GHz-64-core` Retrived: 2017-12-07.

[95] Intel. Intel® Xeon® Processor E5-2690 v3. Available at: `https://ark.intel.com/products/81713/Intel-Xeon-Processor-E5-2690-v3-30M-Cache-2\_60-GHz` Retrived: 2017-12-07.

[96] M. Papamarcos and J. Patel. A low-overhead coherence solution for multiprocessors with private cache memories. *SIGARCH Comput. Archit. News*, 12(3):348–354, jan 1984.

[97] W. McIntosh-Smith and D. Curran. Evaluation of a performance portable lattice Boltzmann code using OpenCL. In *Proceedings of the International Workshop on OpenCL 2013 & 2014*, IWOCL '14, pages 2:1–2:12, New York, NY, USA, 2014. ACM.

[98] NVIDIA Corporation. *NVIDIA TESLA V100 GPU ARCHITECTURE*, 2017.

Paper I

Lattice Boltzmann method on GPUs: a comparison between OpenACC and CUDA

F. Robertsén and K. Mattila and J. Westerholm

Paper II

Lattice Boltzmann Simulations at Petascale on Multi-GPU Systems with Asynchronous Data Transfer and Strictly Enforced Memory Read Alignment

F. Robertsén and J. Westerholm and K. Mattila (2015). In *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 604-609.

Paper III

Designing a graphics processing unit accelerated petaflop capable lattice Boltzmann solver: Read aligned data layouts and asynchronous communication

F. Robertsén and J. Westerholm and K. Mattila (2016). In , pages 246–255.

Paper VI

High-performance SIMD implementation of the lattice-Boltzmann method on the Xeon Phi processor

F. Robertsén and K. Mattila and J. Westerholm (2018). *Concurrency and Computation: Practice and Experience*

128. **Tero Jokela**, Design and Analysis of Forward Error Control Coding and Signaling for Guaranteeing QoS in Wireless Broadcast Systems
129. **Ville Lukkarila**, On Undecidable Dynamical Properties of Reversible One-Dimensional Cellular Automata
130. **Qaisar Ahmad Malik**, Combining Model-Based Testing and Stepwise Formal Development
131. **Mikko-Jussi Laakso**, Promoting Programming Learning: Engagement, Automatic Assessment with Immediate Feedback in Visualizations
132. **Riikka Vuokko**, A Practice Perspective on Organizational Implementation of Information Technology
133. **Jeanette Heidenberg**, Towards Increased Productivity and Quality in Software Development Using Agile, Lean and Collaborative Approaches
134. **Yong Liu**, Solving the Puzzle of Mobile Learning Adoption
135. **Stina Ojala**, Towards an Integrative Information Society: Studies on Individuality in Speech and Sign
136. **Matteo Brunelli**, Some Advances in Mathematical Models for Preference Relations
137. **Ville Junnila**, On Identifying and Locating-Dominating Codes
138. **Andrzej Mizera**, Methods for Construction and Analysis of Computational Models in Systems Biology. Applications to the Modelling of the Heat Shock Response and the Self-Assembly of Intermediate Filaments.
139. **Csaba Ráduly-Baka**, Algorithmic Solutions for Combinatorial Problems in Resource Management of Manufacturing Environments
140. **Jari Kyngäs**, Solving Challenging Real-World Scheduling Problems
141. **Arho Suominen**, Notes on Emerging Technologies
142. **József Mezei**, A Quantitative View on Fuzzy Numbers
143. **Marta Olszewska**, On the Impact of Rigorous Approaches on the Quality of Development
144. **Antti Airola**, Kernel-Based Ranking: Methods for Learning and Performace Estimation
145. **Aleksi Saarela**, Word Equations and Related Topics: Independence, Decidability and Characterizations
146. **Lasse Bergroth**, Kahden merkkijonon pisimmän yhteisen alijonon ongelma ja sen ratkaiseminen
147. **Thomas Canhao Xu**, Hardware/Software Co-Design for Multicore Architectures
148. **Tuomas Mäkilä**, Software Development Process Modeling – Developers Perspective to Contemporary Modeling Techniques
149. **Shahrokh Nikou**, Opening the Black-Box of IT Artifacts: Looking into Mobile Service Characteristics and Individual Perception
150. **Alessandro Buoni**, Fraud Detection in the Banking Sector: A Multi-Agent Approach
151. **Mats Neovius**, Trustworthy Context Dependency in Ubiquitous Systems
152. **Fredrik Degerlund**, Scheduling of Guarded Command Based Models
153. **Amir-Mohammad Rahmani-Sane**, Exploration and Design of Power-Efficient Networked Many-Core Systems
154. **Ville Rantala**, On Dynamic Monitoring Methods for Networks-on-Chip
155. **Mikko Pelto**, On Identifying and Locating-Dominating Codes in the Infinite King Grid
156. **Anton Tarasyuk**, Formal Development and Quantitative Verification of Dependable Systems
157. **Muhammad Mohsin Saleemi**, Towards Combining Interactive Mobile TV and Smart Spaces: Architectures, Tools and Application Development
158. **Tommi J. M. Lehtinen**, Numbers and Languages
159. **Peter Sarlin**, Mapping Financial Stability
160. **Alexander Wei Yin**, On Energy Efficient Computing Platforms
161. **Mikołaj Olszewski**, Scaling Up Stepwise Feature Introduction to Construction of Large Software Systems
162. **Maryam Kamali**, Reusable Formal Architectures for Networked Systems
163. **Zhiyuan Yao**, Visual Customer Segmentation and Behavior Analysis – A SOM-Based Approach
164. **Timo Jolivet**, Combinatorics of Pisot Substitutions
165. **Rajeev Kumar Kanth**, Analysis and Life Cycle Assessment of Printed Antennas for Sustainable Wireless Systems
166. **Khalid Latif**, Design Space Exploration for MPSoC Architectures

ISBN 978-952-12-3692-1
ISSN 1239-1883

Fredrik Robertsén

The Lattice Boltzmann Method, a Petaflop and Beyond

Fredrik Robertsén

The Lattice Boltzmann Method, a Petaflop and Beyond