Performance evaluation of a two-dimensional lattice Boltzmann solver using CUDA and PGAS UPC based parallelisation

MÁTÉ SZŐKE, University of Bristol
TAMÁS ISTVÁN JÓZSA, University of Edinburgh
ÁDÁM KOLESZÁR, Soliton Systems Europe
IRENE MOULITSAS, Cranfield University
LÁSZLÓ KÖNÖZSY, Cranfield University

The Unified Parallel C (UPC) language from the Partitioned Global Address Space (PGAS) family unifies the advantages of shared and local memory spaces and offers a relatively straight forward code parallelisation with the Central Processing Unit (CPU). In contrast, the Computer Unified Device Architecture (CUDA) development kit gives a tool to make use of the Graphics Processing Unit (GPU). We provide a detailed comparison between these novel techniques through the parallelisation of a two-dimensional lattice Boltzmann method based fluid flow solver. Our comparison between the CUDA and UPC parallelisation takes into account the required conceptual effort, the performance gain, and the limitations of the approaches from the application oriented developers' point of view. We demonstrated that UPC led to competitive efficiency with the local memory implementation. However, the performance of the shared memory code fell behind our expectations, and we concluded that the investigated UPC compilers could not treat efficiently the shared memory space. The CUDA implementation proved to be more complex compared to the UPC approach mainly because of the complicated memory structure of the graphics card which also makes GPUs suitable for the parallelisation of the lattice Boltzmann method.

CCS Concepts: •**Computing methodologies** → **Parallel computing methodologies; Modelling and simulation;** •**Applied computing** → **Physical sciences and engineering;** •**Mathematics of computing** → *Mathematical software performance;*

Additional Key Words and Phrases: Partitioned Global Address Space, PGAS, Unified Parallel C, UPC, nVidia, Compute Unified Device Architecture, CUDA, Computational Fluid Dynamics, CFD, lattice Boltzmann method, LBM

ACM Reference Format:
M. Szőke, T. I. Józsa, Á. Koleszár, I. Moulitsas and L. Könözsy, 2016. Performance evaluation of a two-dimensional lattice Boltzmann solver using CUDA and UPC based parallelisation. *ACM Trans. Math. Softw.* 99, 99, Article 99 (Month 2016), 22 pages.
DOI: 0000001.0000001

1. INTRODUCTION

In the world of High Performance Computing (HPC), the effective implementation and parallelisation are vital for novel scientific software. Computational Fluid Dynamics (CFD) targets fluid flow modelling, which is a typical application field of HPC. While the Message Passing Interface (MPI) [Message Passing Interface Forum 2012] has become the dominant technique in parallel computing, other approaches, like the Par-

Author's addresses: M. Szőke, (Current address) Faculty of Engineering, University of Bristol, UK; T. I. Józsa, (Current address) School of Engineering, University of Edinburgh, UK; Á. Koleszár, (Current address) Soliton Systems Europe A/S, Taastrup, Denmark; I. Moulitsas and L. Könözsy, School of Aerospace, Transport and Manufacturing, Cranfield University, UK.

DOI: 0000001.0000001

titioned Global Address Space (PGAS) [PGAS 2015] and the General-Purpose Computing on Graphics Processing Units (GPGPU), reared their heads in the last decade.

Co-Array Fortran, Chapel, X-10, Titanium and Unified Parallel C (UPC) [Chauwvin et al. 2007] are members of the PGAS model family. These languages attempt to offer an easier way for parallel programming on multi-core Central Processing Units (CPU) based systems compared to MPI. This involves keeping the code (a) portable: optimisation is made by the compiler in terms of architecture; (b) readable and productive: such languages were shown to be easier to code and easier to read [Cantonnet et al. 2004]; (c) well performing: it was shown that such languages offer the same or even better performance than MPI [Johnson 2005; Mallón et al. 2009].

Performance-centred investigations were carried out to compare commercial UPC compilers (IBM, Cray, HP) with open source UPC compilers (GNU, Michigan, Berkeley). Husbands et al. [2003] reported that Berkeley UPC (BUPC) is competitive with the commercial HP compiler. The former achieved high performance in pointer-to-shared arithmetic because of its own compact pointer representation.

Mallón et al. [2009] compared UPC to MPI. They found that UPC showed poor results in collective performance against MPI [Taboada et al. 2009] due to high start-up communication latencies. In other aspects, their UPC code performed better than MPI.

Zhang et al. [2011] implemented the Barnes-Hut algorithm in UPC. They reported that the problem can be conveniently approached in UPC because the algorithm has dynamically changing communication patterns that can be handled by the implicit communication management of UPC. They reported poor performance because of the lack of efficient data management when their code relied on shared memory. The problem was resolved with the help of additional language extensions and optimisations ensuring that the data is cached accordingly from the global memory prior to it is requested.

Most of the performance evaluations were done via synthetic benchmarks such as FFT calculations, N-Queens problem, NAS benchmarks from NASA [El-Ghazawi et al. 2006] etc., and only a limited number of papers focused on the application of UPC for physical problems. One of these is the work of Markidis and Lapenta [2010], where a particle-in-cell UPC code was implemented to simulate plasma. They experienced performance degradation for a high number of CPUs, the effect was dedicated to a specific part of their solver.

Johnson [2005] used Berkeley UPC compiler for an in-house code on a Cray supercomputer [Cray Inc. 2012] to run CFD simulations. Their code solved the incompressible Navier-Stokes Equations (NSE) using the finite element method. The UPC code showed better performance than the MPI version. The performance difference was bigger for a higher number of CPUs: UPC performed better than MPI, especially above 64 threads. It was shown that MPI required more time to pass small sizes of messages than UPC. Above a certain message size, UPC still performed better but the difference was negligible.

Although HPC centres are dominated mainly by multi-core CPUs, researchers discovered the potential for scientific computing on Graphics Processing Units (GPU) in the early 2000s [McClanahan 2010]. The Compute Unified Device Architecture (CUDA) Software Development Kit (SDK) was released to popularise GPGPU on nVidia graphical cards [Sanders and Kandrot 2010]. The CUDA libraries can be added to several languages, along with Fortran, Java, Matlab and Python, but most of the time it is used with C/C++. The essentially parallel nature of the graphical cards proved to be applicable in several fields, from pure mathematics [Manavski and Valle 2008; Zhang et al. 2010], through image processing [Stone et al. 2008], to physics [Anderson et al. 2008], including fluid flow modelling [Chentanez and Müller 2011; Ren et al. 2014].

The lattice Boltzmann method has become quite popular on parallel architectures because of the local nature of the operations which is discussed in Section 2. The first parallel solvers were relying on CPUs, and were reported in the 1990s [Amati et al. 1997; Kandhai et al. 1998]. Later on, GPU architectures were proven to be a good basis to parallelise the LBM. Two-dimensional CUDA implementation was published by Tölke [2010], where a speedup of $\approx$20 was reported. Several descriptions of three-dimensional solvers can be found, such as Ryoo et al. [2008], where a speedup of 12.5 was measured. Rinaldi et al. [2012] reached a speedup of 131 using advanced strategies with CUDA. The LBM solver was proved to be highly efficient in multi-GPU environment as well [Xian and Takayuki 2011]. As far as the authors know, only Valero-Lara and Jansson [2015] considered using UPC for the LBM.

The advantages of the LBM are its good scalability, explicit time step formulation, applicability for multiphase flows [Succi 2001] and applicability for flows with relatively high Knudsen number [Mohamed 2011]. The latter one is one of the main limitations of the NSE. The LBM has been developed for incompressible subsonic flows, it has second order of accuracy, and relatively high memory requirements due to the discretisation of particle directions.

In this paper, we present and compare the performance gain achieved after the CUDA and UPC parallelisation of an in-house LBM code. The solver handles two-dimensional fluid flow problems using the LBM. The comparison of the two currently applied architectures is not widely discussed from the performance point of view. Since HPC is often used by mathematicians, physicists, and engineers with limited computer science knowledge, we also aim to inspect how user-friendly CUDA and UPC are. We investigated the effect of the following factors:

a. memory structure of UPC: shared and local variables;
b. spatial resolution;
c. hardware;
d. collision models;
e. data representation: single and double precision;
f. required programming effort for the different codes.

The UPC implementation was evaluated on two different clusters, and the results were compared to the CUDA parallelisation on two different architectures. To quantify the performance the speedup was defined as $SU = t_{\mathrm{serial}}/t_{\mathrm{parallel}}$, where t_{serial} is the execution time of the serial code and t_{parallel} is the execution time of the parallel code.

2. THE IMPLEMENTATION OF THE LATTICE BOLTZMANN METHOD

The Boltzmann equation was derived to describe the motion of a large number of particles on a statistical basis. The idea behind the LBM is to discretise the Boltzmann equation so that particle propagation is allowed only in certain discrete directions. The governing equation is written as Eq. (1), where $f = f(\vec{x}, \vec{p}, t)$ is the distribution function, which represents the probability of "finding a molecule around position $\vec{x}$ at time t with momentum $\vec{p}$" [Succi 2001]. $\vec{c}$ is the microscopic particle velocity, and $\Omega = \Omega(f)$ is the collision operator. The Chapman-Enskog procedure forms a clear bridge between the LBM equation (1) and the NSE (in the artificial compressibility form proposed by Chorin [1967]), thus the method can be used to describe incompressible fluid flows [He and Luo 1997].

$$\frac{\partial f}{\partial t} + (\vec{c} \cdot \nabla) f = \Omega \tag{1}$$

We applied the D2Q9 model, which means that the resolved flow field was two-dimensional, and particle propagation was allowed in nine discrete directions (Fig. 1). The time marching was divided into four steps from the implementation point of view:

I. Collision. The collision modelling treats the right hand side of Eq. (1). Two collision models were analysed:

a. In the BGKW model, the collision is described by Eq. (2) which was derived by Bhatnagar et al. [1954] and by Welander [1954]. Here τ is the relaxation factor, calculated from the lattice viscosity of the fluid, and f^{eq} is the so called local equilibrium distribution function described by the Maxwell-Boltzmann distribution [Maxwell 1860]. It is important to note that the collision term can be computed independently for every direction after the discretisation.

$$\Omega = \frac{1}{\tau}(f^{\mathrm{eq}} - f), \tag{2}$$

b. The Multi Relaxation Time (MRT) model was presented and reviewed by d'Humières [1992] and d'Humières et al. [2002]. In this case, instead of using a single constant (τ^{-1}) to describe the collision, the model applies a matrix which depends on the resolved directions (D2Q9). In our case, this matrix has a dimension of 9×9. This approach yields a matrix multiplication for each lattice. Despite the fact that this model is computationally more expensive than the BGKW, it is widely applied since the flow field can be more accurately resolved.

II. Streaming. The *streaming* process occurs when the directional distribution functions "travel" to the neighbouring cells: second term on the left hand side of Eq. (1). This process is presented in Fig. 1(b).

III. Boundary treatment. These steps are followed by the handling of the boundaries. In the current paper, the boundary description suggested by Zou and He [1997] was used for the moving wall, and the so called bounce-back boundary condition [Succi 2001] was used to handle the no-slip condition at the stationary walls.

IV. Update macroscopic. Once the distribution functions were known at the end of the time step, the macroscopic variables (density, x- and y-directional velocity components) had to be computed from the distribution functions to recover the flow field.

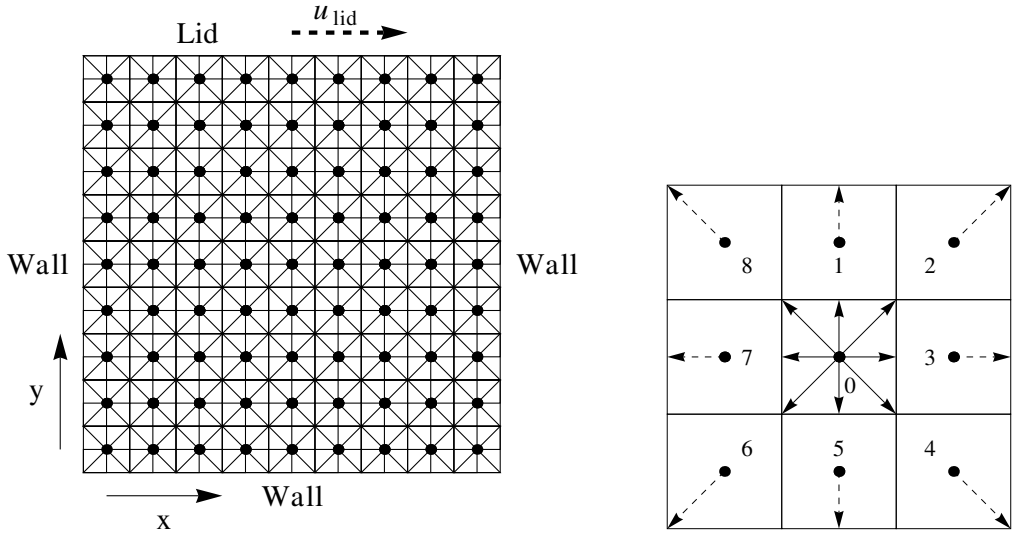
The sum of the listed items is referred from now on as the *main loop*. The grid generations and the initialisation of the microscopic and macroscopic variables took place before the *main loop*.

In terms of the computational grid, the method is based on a uniform Cartesian lattice, which is represented in Fig. 1(a), where the nine directions of the *streaming* are displayed as well. In order to examine the effect of the spatial resolution on the performance gain, four different grids were investigated. In the following, the lattices are referred to based on the names given in Table I.

Table I: Applied mesh sizes

Name	Size $n_x \times n_y$	Lattices
Coarse	128×128	16,384
Medium	256×256	66,536
Fine	512×512	262,144
Ultra Fine	1024×1024	1,038,576

The simulated fluid flow was the well known lid-driven cavity, which is a common validation case for CFD solvers [Ghia et al. 1982]. The aspect ratio of the domain was



(a) Lattice and the applied boundary conditions

(b) *Streaming* and the nine discrete directions ($\vec{c}_i$)

Fig. 1: Structure of the computational domain

unity. The lid on the top moved with a defined positive x-directional velocity, while all the other boundaries were stationary walls (see Fig. 1(a)). From these conditions, the Reynolds number in the domain was defined based on the lattice quantities as $Re = n_x u_{\text{lid}}/\nu_l$, where $n_x = n$ is the number of lattices along the x-direction, u_{lid} is the lid velocity, and ν_l is the lattice viscosity, which was set to be 0.1. The Reynolds number of the simulations was 1000. At the end of the computations, the coordinates and the macroscopic variables were saved. A qualitative validation of the flow field can be found in Appendix 2. For a more comprehensive validation and verification, we refer to the work of Józsa et al. [2016].

3. PARALLELISATION

The serial code was written in C, and the code was built up as it was presented in Section 2. The serial implementation was based on one-dimensional Array of Structures (AoS), similarly to the UPC version. The `Cells` structure included the macroscopic variables (velocity components as `U` and `V`, density as `Rho`, etc.) as scalars, while the distribution function `F` was stored as a nine-dimensional array within the `Cells` structure. Thus `(Cells+i)->F[k]` referred to the $\texttt{i}^{\text{th}}$ lattice in the domain and the corresponding distribution function in the $\texttt{k}^{\text{th}}$ discrete direction. The parallelisation process can be followed in Appendix 1. The serial simulations were performed on Archer, the United Kingdom National Supercomputing Service.

First, we parallelised the solver with UPC and ran the codes on Archer and Astral. The latter is the HPC cluster of Cranfield University. The hyper-threading technology [Intel 2015] was switched off on both architectures. The properties of the two clusters are listed in Table II. A higher performance can be expected on Archer since it holds several optimisation properties such as hardware supported shared memory addressing. Note that the interconnection between the nodes are different in the two investigated clusters. On Archer the commercial Cray C compiler was available, while the open source BUPC compiler was installed on Astral.

Table II: Properties of the used CPU clusters

Properties	Astral (Intel)	Archer (Cray)
Compiler	BUPC	Cray C
Intel processor number	E5-2660	E5-2697
Processor clock rate [GHz]	2.2	2.7
Processors per node	2	2
Threads per processor	8	12
Threads per node (used)	16 (16)	24 (16)
Memory per node [GB]	64	64
Cache per processor [MB]	20	30
Interconnect type	Infiniband	Cray Aries
Recommended customer price	$1445	$2890

Second, the CUDA parallelisation was tested. The performance gain was evaluated on two different nVidia GPUs. The relevant properties of the graphical cards are listed in Table III. While the GeForce cards are cheaper devices, as they are primarily designed for computer games, the Tesla cards are more expensive, directly designed for scientific computing. The code development was carried out on a desktop using the GTX 550Ti card, while the Tesla GPU of the University of Edinburgh‘s Indy cluster was used for further investigations.

Table III: Properties of the used GPUs

Properties	GeForce GTX 550Ti	Tesla K20
CUDA release	v7.5	v6.0
Number of CUDA cores	192	2496
Global memory type	GDDR5	GDDR5
Global memory bandwidth [GB/s]	98.5	208
Global memory size [GB]	1	5
Single precision performance [GFLOPs]	691	3524
Double precision performance [GFLOPs]	unknown	1175
Recommended customer price	$230	$2300

3.1. Unified Parallel C Approach

UPC is an extension of the standard C language but it requires its own compiler. The novelty of UPC relies on its memory structure, where the user has the opportunity to lay down variables in the shared address space and in the local memory at the same time. A schematic draw of the local and the shared memory spaces is showed in Figure 2. Implementing UPC based codes therefore offers the opportunity to combine the advantages of the conventional parallelisation methods, such as OpenMP and MPI, whose restrict the programmer to rely purely on either shared or local memory only. The UPC compiler is designed to manage and handle the data layout between the threads and nodes. This keeps the architecture based problems hidden from the user, i.e. optimisation is expected to be performed by the compiler.

The syntax of the local memory declarations is the same as in the standard C language. Data exchange between the threads is performed via the `upc_memput`, `upc_memget` and `upc_memcpy` functions. The first function copies local data to the shared space, and the second copies data from the shared to the local memory. The last function performs data copy from shared to shared address space. Note that the first and second functions are similar to the `MPI_Send` and `MPI_Recv` functions.

The shared memory based data needs to be declared according to the UPC standards. In this case, the programmer must declare a compile time constant block size, which defines how many elements of a vector belongs to each thread. The data is laid down

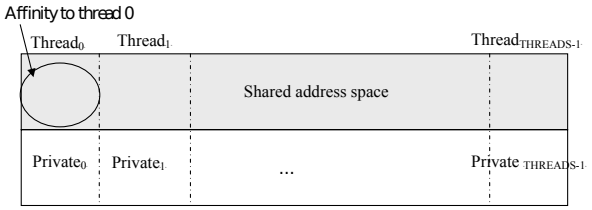


Fig. 2: Memory model of the local and shared memory spaces

between the threads with a round-robin fashion using the corresponding block size. We can see that the compile time constant restriction is the biggest drawback of the investigated language. If the programmer wishes to lay down the whole mesh, for example here in the shared memory, then the mesh size needs to be known in advance of the compilation. In other words, different executable files are needed for different mesh sizes.

To perform shared memory based operations, UPC offers the usage of `upc_forall`, which is an extension of the standard C `for` loop. Each shared variable within a vector has an affinity term that describes which thread the given element belongs to. Based on this information, the `upc_forall` distributes the computational load between the threads. UPC also offers the usage of barriers, locks, shared pointers, collectives etc. For further description we refer to Chauwvin et al. [2007].

In our case, two UPC codes were implemented: (a) one with shared memory implementation, and (b) another code relying on the local memory. The *streaming* step of these codes are given as an example in Appendix 1. In the former code, the data is laid down in the shared memory, and in the latter one, the data is stored in the local memory. The shared memory based code exploits the novelty of UPC, i.e. this code relies on the `upc_forall` function and shared pointer declarations. This leads to a more easily readable code. The second approach, which exploits data locality and follows the logic of MPI implementations, offers better speed and lower latency time. As a disadvantage, the `upc_memput`, `upc_memget` memory operations were required; therefore this code is more complex and consists of more lines. The computational load was distributed equally between the threads in both implementations, since the mesh size was divisible by the number of threads during the simulations.

During the compilation of the UPC codes, similarly to the serial code, the performance affecting flags were avoided. The compile command of the UPC codes, for example using four threads, was `upcc -T=4 *.c -lm -o LBMSolver`. The execution simplified to `upcrun LBMSolver` on both HPC systems.

3.2. nVidia CUDA

The CUDA SDK contains several additional functions to make the programming of nVidia GPUs possible. The programming of the graphical card, from now referred to as ‘device’, requires some basic knowledge of its structure. The working units are the so called threads, which form separate blocks. Theoretically, every thread can work in parallel. Originally the threads could perform operations only on the data which was stored in the device memory. This meant that the programmer had to work out the data transfer between the host and device memory. The most recent solution of nVidia to bridge this problem is the unified memory (available since v6.0), which enables automatic data migration between the host and the device. Nevertheless, the host memory bandwidth is evidently the bottle neck of the available performance gain. It is a rule of thumb that the data transfer between the host and the device should be minimised for high efficiency.

After the data has been copied to the device, CUDA offers an opportunity to manage the multilayer memory structure of the GPUs. In the current study, only the global and the constant memories were used. When data is copied to the device, it is stored originally in the global memory (Table III). Every thread has access to this data, however this memory has the smallest bandwidth on the device. To quicken the speed of the calculations, the parameters can be stored in the constant memory which has a higher bandwidth but a smaller size.

It is worth mentioning that the shared memory, as it has an important role when the communication between the blocks is high. For instance, in the case of the so called prefix sum when every element of a vector are summed. Originally, the threads within different blocks can communicate only through the relatively slow global memory. However, the threads have access to the shared memory only within the blocks, the appropriate management of this memory layer often leads to significant performance gain because of its high bandwidth and low latency. For further description on the memory structure of the GPU and on CUDA programming we refer to [Nickolls et al. 2008] and [Sanders and Kandrot 2010].

The CUDA parallelisation included the following main steps (presented through the *streaming* in Appendix 1):

CUDA 1. Several smaller structures were used instead of the `Cells` structure, for example `Cells_var_9d_d` to store the nine-dimensional distribution functions (Appendix 1, CUDA parallelisation step 1). The inefficient `for` loops were avoided, so that the *streaming* was performed theoretically in parallel for every node in every discrete direction.

CUDA 2. The so called structures of array (SoA) approach was implemented, which proved to be more efficient compared to the AoS approach [Ryoo et al. 2008]. This means that every variable (e. g. density and velocity components) was stored in separate one-dimensional arrays. This SoA version of the code proved to be $\approx$5 times faster than the AoS version. The maximum speedup with this implementation, measured with the ultra fine lattice on the Tesla K20 card using SP, was $\approx$20. This is similar to the speedup achieved by Tölke [2010] on a 2D lattice, but seems infinitesimal compared to the 131 speedup reported by Rinaldi et al. [2012] using a 3D domain.

CUDA 3. The SoA approach was kept but the threads were assigned to the lattices and the discrete directions were taken step by step as it is shown in Appendix 1. We note that the operations related to the 0^{th} discrete direction were ignored during the *streaming* in the last version of the CUDA code because they do not have any physical role. This simplification did not influence the performance significantly.

After the *boundary treatment* was identified as the bottleneck of the computations (see Chapter 4, Fig. 6), the global search, which was performed based on a boolean mask at every time step to find the boundary lattices, was replaced. In the last version the boundary lattices were selected during the initialisation so that the *boundary treatment* kernel function "knew" the location of the boundaries in advance.

In every case, one-dimensional grids and blocks were used; furthermore 256 threads were initialised within every block (block size). The number of the blocks (grid size) varied automatically as a function of the mesh size. This set-up proved to be the most efficient computationally, although it resulted in a strong limitation in terms of the maximum mesh size. The theoretical maximum thread number of the devices is 65535×1024 (maximum grid size times maximum block size). In the first two implementations the threads were assigned to every distribution function in every lattice which led to a maximum cell number $65535 \times 256/9 \approx 1864207$. This limitation was

overcome in the third CUDA parallelisation step by assigning the threads to the cells. This way the maximum number of lattices was nine times higher.

The last version of the CUDA code was compiled with the `nvcc -arch=sm_20 -rdc=true` command. Here the first flag defines the virtual architecture of the device, while the second one allows the user to compile the files separately and link them at the end. (The first and the second version did not need the `-rdc=true` flag, since the kernel functions were in one file.) The authors note that compiling the code with a more recent virtual architecture for the K20 GPU, for instance `-arch=sm_35`, would probably result in an enhanced parallel performance. The `-arch=sm_20` flag was used because this was the most recent virtual architecture supported by both of the tested GPUs. The detailed analysis of the code performance can be found in Chapter 4.

4. RESULTS AND DISCUSSION

In this section, we will discuss the achieved speedup of the *main loop*. Firstly, the serial results will be analysed. Secondly, the UPC approaches will be evaluated. Finally, the nVidia CUDA parallelisation will be examined and compared to the UPC approaches. In addition we will evaluate the applied parallel approaches from the code developer's point of view.

4.1. Serial simulations

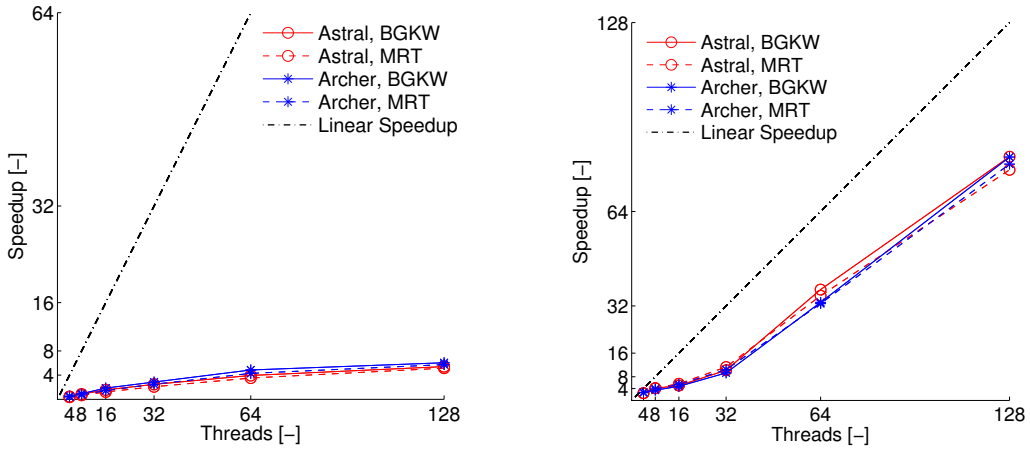
The measured iteration times of the serial simulations are listed in Table IV. As expected, the simulation time increased with the grid size. A factor of approximately four can be identified between the computational cost of the coarse-medium and fine-ultra fine grids, which is logical since the grid size increased exactly by the same factor. This observation is not valid between the medium and fine meshes. While the variables for the coarse and the medium mesh can be fitted into the cache, the memory requirement of the fine and the ultra fine meshes exceeds the cache size. As we expected, the MRT model was computationally more expensive than the BGKW. Compared to the BGKW model, the MRT execution time was typically 10–50% higher, and as the number of cells increased the MRT model became relatively cheaper.

Table IV: Wall time per iteration, serial simulations [ms]

		Grid			
Collision model	Precision	Coarse	Medium	Fine	Ultra fine
BGKW	Single	1.204	4.974	45.01	177.6
BGKW	Double	1.264	9.664	59.77	240.5
MRT	Single	1.696	7.092	51.56	204.1
MRT	Double	1.956	11.946	65.75	263.1

4.2. Performance analysis of the UPC codes

The shared and local approaches were compared in terms of the achieved speedup in Fig. 3. The dash-dotted line shows the theoretical limit, the linear speedup ($SU = N_{\mathrm{threads}}$) in all figures presented in the current subchapter. Fig. 3(a) presents the shared memory based speedup achieved on Astral and Archer using the fine mesh and double precision. We can see that the shared memory based code led to poor parallel performance. By using any number of threads, the gained performance was far below the linear speedup. Furthermore, for lower number of threads (4 and 8), the parallel code was slower than the serial. By crossing nodes, the speedup continued to increase indicating appropriate communication between the nodes.



(a) Shared memory approach (b) Local memory approach

Fig. 3: speedup of the *main loop* as a function of the parallelisation approach, fine mesh, double precision arithmetic

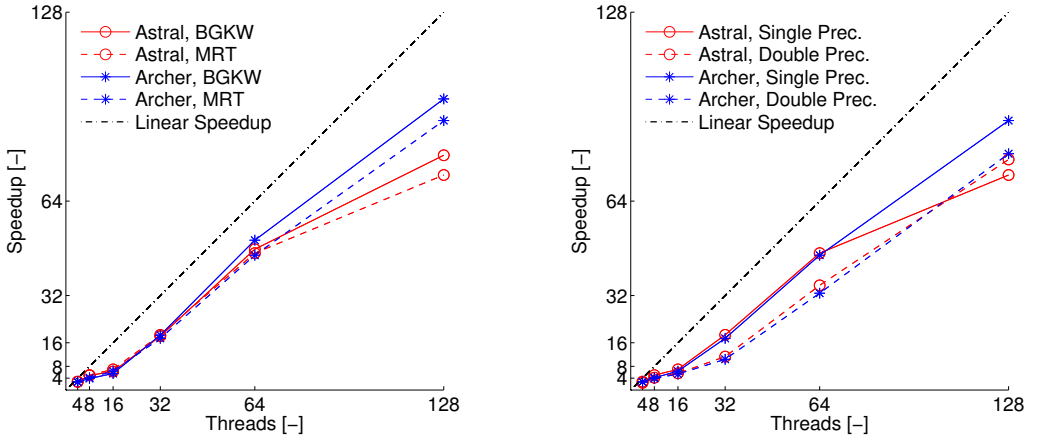
Fig. 3(a) indicates that none of the simulations exploit the maximum performance of the supercomputers. Despite the hardware support of the shared memory operations on Archer, the simulations did not show better speedup results compared to Astral. We hypothesised that the compilers could not handle the shared pointers and manage the data between the shared and local memory spaces effectively. As a first step, performance analysis was conducted on Archer using CrayPat [Cray 2015], which showed that the data was managed and "fed" to the CPUs properly.

As a next step, we tried to find other reasons for the problem. We reckon that the poor performance might have been caused by one of the following factors, and so we took measures to overcome them:

a. usage of shared pointers. All of them were tested with static variables;
b. inappropriate time measurement. We tested different approaches such as the `clock()`, and the `MPI_Wtime()` commands;
c. inappropriate usage of the `upc_forall` command. Different methods were examined to distribute the computational load, for instance working threads were defined based on affinity of shared variables (`&Cells[i]`) or modular division of integers (`i % THREADS`);
d. usage of `Cells` structure. The structure was eliminated (see the code samples in Appendix 1);
e. lack of optimisation flags. All of the available optimisation flags (-O1, -O2, -O3) were tested.

None of these modifications resulted in better speedup in the case of shared variables, i.e. the experienced performance of the shared code was the same for each listed factors. Therefore, we concluded that the compiler cannot handle the data properly without additional modifications, and for this particular application the compiler is still not mature enough. As it was presented by Zhang et al. [2011], the problem could be resolved by outer libraries and user implemented machine level data management. Adding these low level modifications to the shared code would eliminate the main advantage of UPC, namely the quick and user-friendly parallelisation environment. To

overcome this problem the data was rather transferred to the local memory and another, MPI-like code was developed.



(a) Local memory based speedup, single precision arithmetic

(b) Comparison of double and single precision speedup, MRT collision model

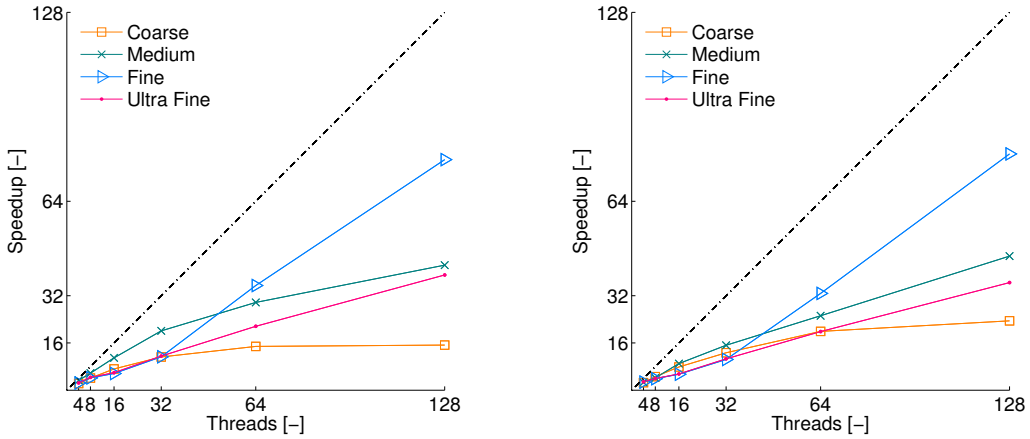
Fig. 4: *Main loop* speedup results on fine mesh

Fig. 3(b) shows the local memory based speedup using the fine mesh and double precision. This approach gave significantly better results. Here we can see that crossing a node did not introduce significant latency either, the compilers were capable of managing the halo swap between the nodes.

Fig. 4(a) shows the speedup as a function of the collision model. The two models had similar parallel efficiency. We can see that the BGKW collision model (solid lines) enabled slightly better speedup results than the MRT collision model. Fig. 4(b) gives us a basis for an explicit comparison between the performance of the single and double precision executions. This graph shows the speedup achieved on the fine mesh with the MRT collision model. We can see that the single precision results (represented by the dashed lines) are better above 16 threads than the double precision ones (continuous lines). Between 16 and 32 threads the first node was crossed on both architectures. The difference between the single and double precision curves above 16 threads are originated from the communication costs. The double precision approach requested more data handling resulting in lower speedup.

We plotted the effect of mesh size on the speedup in Fig. 5(a) and 5(b) for Astral and Archer, respectively. If we consider more than 32 threads, then we may conclude that better speedup were achieved with increasing mesh size. Unexpectedly, this finding was not valid for the ultra fine mesh, where performance degradation was experienced on both architectures. To find the “leakage” in the performance we measured the time spent with the data transfer on the fine and the ultra fine meshes using 128 threads. With this set up, the halo included twice as much data on the ultra fine mesh than on the fine mesh, so that the data transfer should take roughly twice as long as well. In contrast, the data transfer took six times longer on the ultra fine mesh compared to the fine. The performance degradation experienced on the ultra fine mesh was caused by the increased communication costs, which seems to be a relatively strong limitation. We note that the ultra fine mesh consists of approximately one million cells, so

the computational load of the processors is still reasonably low when 128 threads are allocated.



(a) Astral based results (b) Archer based results

Fig. 5: The effect of mesh size on the *main loop* speedup, double precision number representation

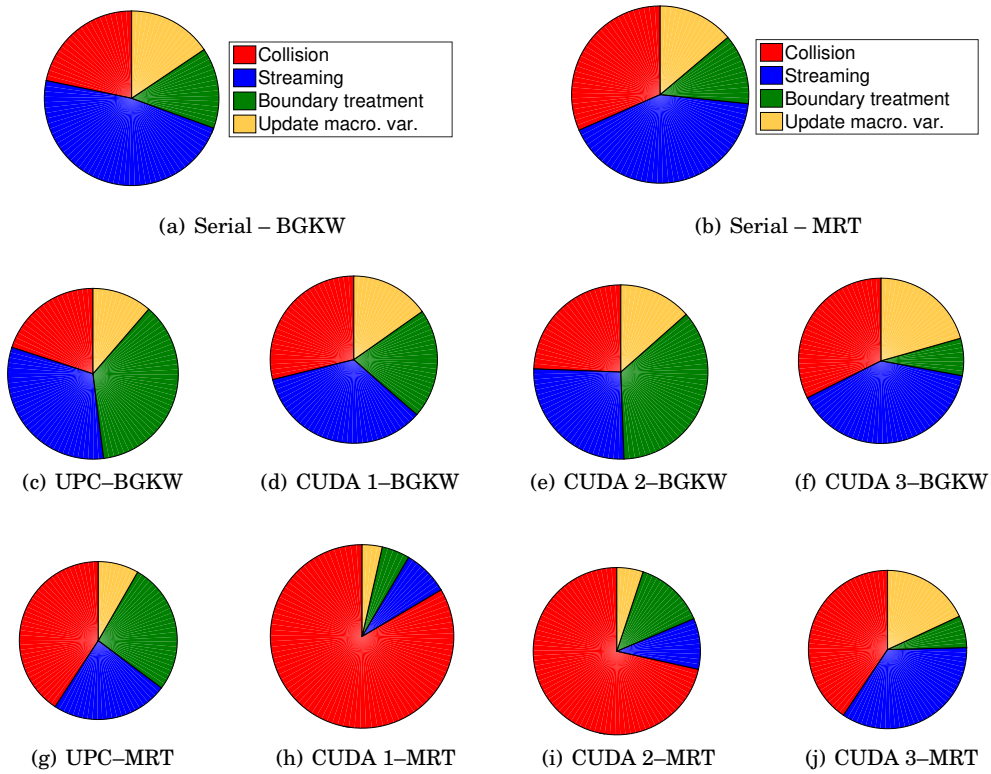
4.3. Performance comparison of the UPC and CUDA codes

In this subsection only the local memory based UPC implementation run on 64 threads will be analysed and compared to the serial and CUDA simulations. Fig. 6 shows profiling results of the different codes. Based on Figs. 6(a) and 6(b) one can identify the *collision* and the *streaming* as the most expensive operations. We can conclude from these two figures that the MRT model is slightly more expensive than the BGKW model. Although the *boundary treatment* is relevant only for the nodes on the perimeter of the domain, it still needs approximately as much time as the *update macroscopic* operation, because the boundary nodes are treated based on a global search.

Compared to the serial results, the UPC implementation drew attention to the the inefficient *boundary treatment* which became one of the most expensive operation in the UPC approach (Fig. 6(c) and 6(g)). Furthermore, the parallelisation highlighted the increased cost of the *collision* in the MRT model, which was more expensive compared to the other steps (Fig. 6(g)).

The MRT collision model was found to be computationally more expensive on the GPU as well. We can see how the *collision* step became less expensive during the development (Figs. 6(h) 6(i) and 6(j)), although it still took approximately 40% of the *main loop*. The MRT model includes summations along the discrete directions. In the first and the second CUDA development steps these summations required operations between the blocks, while in the final version the summation happened within the blocks, so that it could be done more efficiently. While the first and the second CUDA development steps were more favourable for the *streaming*, the third step was specifically developed to decrease the cost of the *collision*.

The scalability of the codes as a function of the hardware is displayed in the bar charts in Fig. 7. As we can see in Fig. 7(a) and 7(b), the speedup of the BGKW and the MRT models were bounded around 50 on Archer, and around 80 on the Tesla K20



(a) Serial – BGKW (b) Serial – MRT

(c) UPC–BGKW (d) CUDA 1–BGKW (e) CUDA 2–BGKW (f) CUDA 3–BGKW

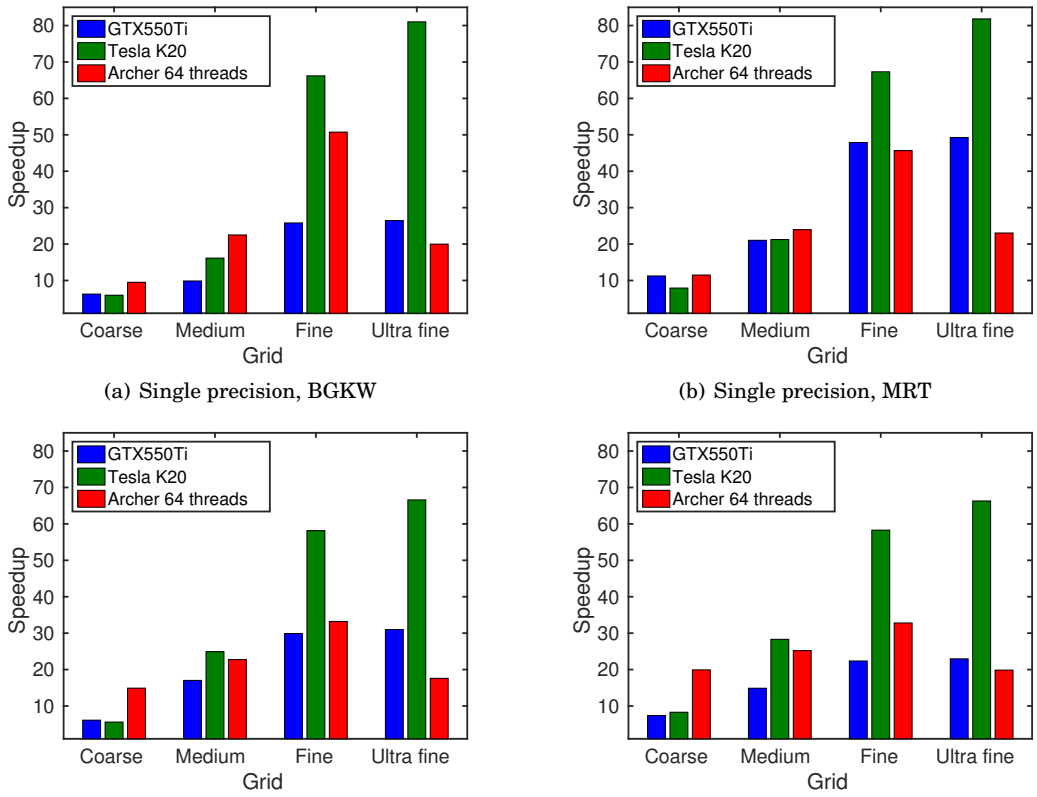
(g) UPC–MRT (h) CUDA 1–MRT (i) CUDA 2–MRT (j) CUDA 3–MRT

Fig. 6: Profiling results based on single precision simulations with fine mesh on the GTX 550Ti graphical card, and on Archer using 64 threads

card. In case of the K20 card, the performance gap between double and single precision execution is clearly visible: while a maximum speedup of around 80 was measured with single precision arithmetic (Fig. 7(a)), a speedup around 65 could be achieved with double precision arithmetic in the case of the BGKW model (Fig. 7(c)). A similar trend can be seen in the case of the MRT model as well, with a slightly wider gap between the single precision and double precision arithmetic (Figs. 7(b) and 7(d)). Considering that the double precision processing power of the K20 unit is approximately a third of its single precision processing power (Table III), it might seem surprising that the performance gap is only around 20%. If we also consider that two of the main steps in the LBM, (*streaming* and *boundary treatment*) are essentially data copying, then the relatively small gap makes more sense: the high memory bandwidth of the GPU compensated for the lack of computing power.

Interestingly, the GTX550Ti device showed better parallel performance with the MRT model compared to the BGKW model (Fig. 7(a) and 7(b)). While this card gave higher speedup with DP in the case of the BGKW model, using DP led to a drastic performance drop with the MRT model (compare Fig. 7(a) with 7(c) and Fig. 7(b) with 7(d)).

Based on the charts, the K20 card performed in almost every case better than the GTX550Ti. In fact, the K20 card was slightly slower than the GTX550Ti only when the grid size was small. In our case, the medium grid proved to be big enough to utilise



(a) Single precision, BGKW

(b) Single precision, MRT

(c) Double precision, BGKW

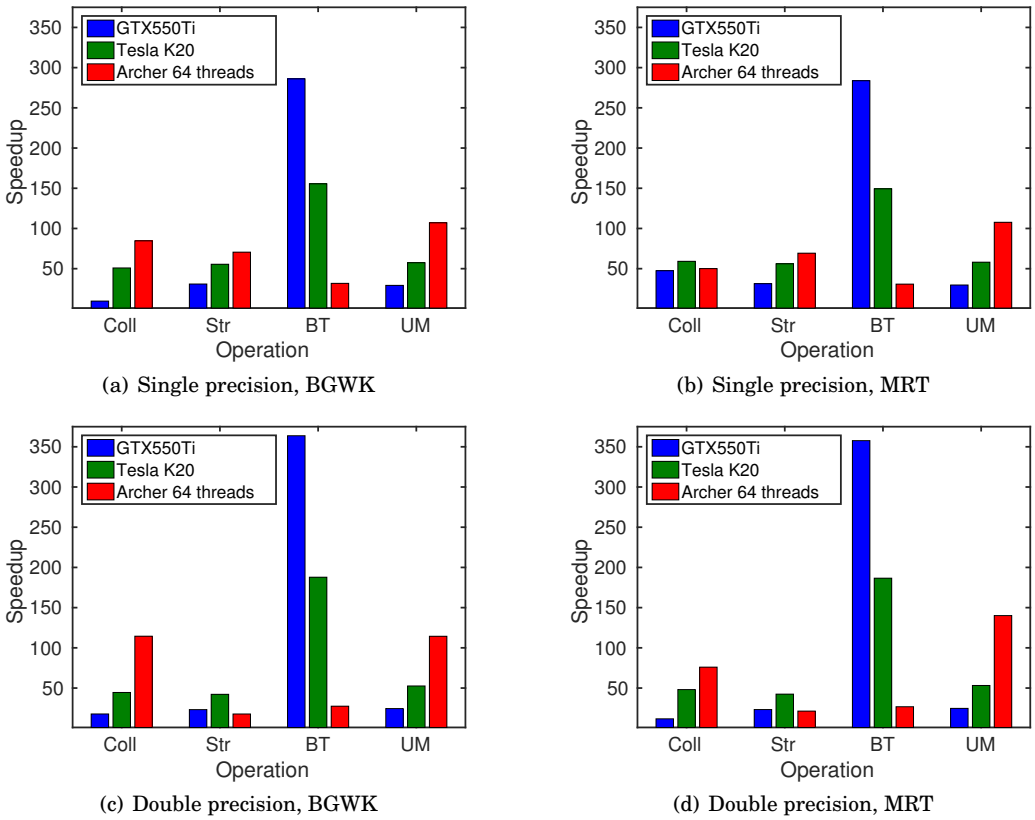
(d) Double precision, MRT

Fig. 7: Speedup of the *main loop* as a function of the grid spacing and the hardware

the better potentials of the K20 GPU. As the grid size increased we could measure an increasing speedup in the case of the K20 device, while the GTX550Ti card reached its limits at the fine mesh. These results mirror the GPUs' evolution, and correlate well with the hardware parameters (e. g. CUDA cores) given in Table III.

Ideally, in the case of the CPU parallelisation, when the number of threads is kept constant and the grid size changes, a nearly constant speedup can be expected. After looking at Fig. 7, it becomes clear how far away this application is from an ideal situation: increasing the number of lattices up to a certain point (fine mesh), resulted in an increasing speedup. This happened because as the problem size increased, the time spent with the halo swap decreased relative to the time spent with the computation of the different operations. The speedup on Archer with 64 threads was close to the ideal when single precision arithmetic was used on the fine mesh (Figs. 7(a) and 7(b)). However, using double precision arithmetic means an increased computational load for each threads, it also means increased communication between the threads. Probably this is the reason why the parallel performance of the double precision execution was lower on Archer compared to the single precision simulations when the fine mesh was investigated. (Figs. 7(c) and 7(d)).

In order to gain a better understanding of the results, deeper analyses of the code is required. The speedup of the main parts of the code are shown in Fig. 8. It is important to recognize that, theoretically, only the speedup of the *collision* step should



(a) Single precision, BGWK

(b) Single precision, MRT

(c) Double precision, BGWK

(d) Double precision, MRT

Fig. 8: speedup of the different operations on the fine grid as a function of the hardware. (Coll–*Collision*; Str–*Streaming*; BT–*Boundary treatment*; UM–*Update macroscopic*)

change when we consider different models. Indeed, the other operations show only a small deviation (compare Figs. 8(a), 8(c) with 8(b), 8(d)). After a first look, we can see that the *boundary treatment*, which was identified as the bottle neck of the performance (Fig. 6), was significantly improved in the final step of the CUDA code. This high speedup was measured, because the global search of the boundaries was replaced (see Chapter 3.2). Furthermore, it is also visible that the other parts of the code had a relatively uniform speedup in the case of the CUDA implementation: for the K20 card around 50 with single precision and 40 with double precision. When compared with the K20, the GTX550Ti device showed better speedup of the *boundary treatment* but a worse speedup of the other operations. The unexpected behaviour of the GTX550Ti card when compared to K20 can be caused by its structure which was designed for gaming, or the different CUDA release (see Table III).

The speedup of the operations measured on Archer shows less deviation. However, it was found that the measured speedup exceeded the theoretical limit (64) several times, especially for the *collision* and the *update macroscopic* parts, while other times the parallel code underperformed. This behaviour seems to be logical if we take into account that the *collision* and the *update macroscopic* operations do not require any communication between the processes. Furthermore, the data of the partitioned mesh

fit the cache of the nodes, while the same data exceeded the cache size of a single node in the case of the serial execution.

4.4. UPC and CUDA beyond performance

Because of the complex memory structure of the GPUs, the CUDA parallelisation was more cumbersome and time consuming. We can see that certain parts of the code (e. g. *boundary treatment*) needed significant reformulation in order to reach higher performance. Furthermore, it is important to note that for a CPU related parallelisation it is reasonably well understood how the efficiency can be improve, but for a complex application, like the current study, the CUDA related optimisation requires more conceptual effort and the identification of performance loss is more complex. All in all, the final version of the CUDA code required roughly twice as much working hours than the UPC parallelisation. The final code was achieved via three steps, and there are still several opportunities to further increase the performance, for instance, using the shared memory of the cards or multiple graphical cards.

Table V: Number of lines in the different codes

Code name	Number of lines in implementation: *Main loop*	Overall
Serial	1148	2215
UPC: Archer – Shared	1294	2413
UPC: Archer – Local	1359	2566
UPC: Astral – Shared	1294	2413
UPC: Astral – Local	1337	2629
CUDA: Step 1	1596	2455
CUDA: Step 2	1748	2547
CUDA: Step 3	1676	4684

Although the straightforward shared memory approach of UPC proved to be inefficient, the classical MPI-like local parallelisation technique gave acceptable results. Thanks to its simple syntax, it needed less programming effort compared to CUDA. The corresponding number of lines for the different codes are listed in Table V. We can see that the most efficient CUDA implementation took $\approx$40% more lines, while the longest UPC implementation needed only $\approx$15% more lines compared to the serial code. It is arguable whether counting the number of lines is representative of the programming effort but in this case it is well correlated to the required work. The efficient parallelisation with CUDA required roughly twice as much office hours than the two UPC implementations. Without a doubt, we can conclude that the highest amount of conceptual effort was required by the CUDA approach followed by the local UPC code and the shared UPC code.

It is important to see what kind of compromises application oriented developers need to make. The situation can be described with the help of the triangle shown in Fig. 9. The edges of the triangle contain the good properties of a high performance programming approach: low conceptual effort, high performance and low hardware costs. For the lattice Boltzmann method, the first possible scenario includes a low cost hardware (Tesla K20 of $2300) and a reasonably high performance but we have to pay the price of conceptual effort because of the GPU's programming environment. Another extreme scenario is when a higher budget is available (Intel E5 processors, each for $2900), and we can work with a more flexible programming environment, and probably end up with an efficient code in a shorter period of time. This situation makes the local memory based UPC programming environment a suitable candidate. Additionally to these two cases, when the available budget is limited, one can consider running single

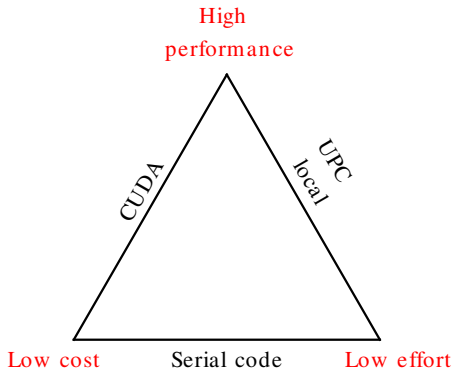


Fig. 9: Compromise triangle of high performance scientific programmers

core simulations on a cheap hardware. This would clearly require longer computations because of the low performance. The choice between the three scenarios is still usually made by the time frame, the available hardware, and skill set. However, the shared memory approach of UPC aims to provide another low effort, high performance scenario, our investigations highlighted that the compilers need further development to achieve this goal.

5. CONCLUSIONS

We parallelised an in-house, two-dimensional lattice Boltzmann solver using CPU and GPU parallelisation approaches. We presented the UPC implementation of the lattice Boltzmann method and compared the parallel efficiency of our CUDA and UPC codes using the two-dimensional physical problem of the lid-driven cavity. The UPC codes were tested with two different compilers on two different clusters, while the CUDA codes were run on two different GPUs. A detailed performance analysis of the different implementations was performed to provide an insight into the parallel capabilities of UPC and CUDA when it comes to the lattice Boltzmann method.

The parallelisation of the *collision* proved to be crucial since this is the part in the algorithm where the majority of the computation happens. Based on our experience the efficiency of this part determines the globally experienced efficiency, and it typically means favourable implementation for the *update macroscopic* operation as well. We would like to draw attention to the *boundary treatment* as well, since it can easily become the bottle neck of the parallel code, although its execution time is essentially limited by the memory bandwidth similarly to the *streaming*.

The UPC code using the shared memory approach showed surprisingly low performance compared to the serial code. We found that the investigated compilers could not automatically manage the data transfer between the threads efficiently. The further development of the compilers may solve this issue and make the UPC approach more user-friendly and attractive for future scientific programmers. Until then, we can enjoy the simple syntax of UPC for local memory based implementation, which was proven to be more efficient and suitable for the parallelisation of the lattice Boltzmann method.

The CUDA development was presented through three different steps which highlighted that the used data structures (namely the AoS and the SoA approaches), and the data distribution strategies have a significant effect on the parallel performance.

We can confirm that the nVidia graphics cards, especially the ones designed for scientific computing, are highly suitable for the parallelisation of the lattice Boltzmann method. Based on our measurements a single GPU might compete with 3-4 supercomputer nodes (around 80 threads) or more, for a significantly lower price. To reach this high performance developers need more specific skills and programming effort when it is compared to the local UPC implementation.

Code availability

The developed codes are available as open source and can be downloaded from GitHub at https://github.com/mate-szoke/ParallelLbmCranfield. The codes are available under the MIT license. The folders also include all mesh and setup files used to perform the documented simulations.

ACKNOWLEDGMENTS

This work used the ARCHER UK National Supercomputing Service (http://www.archer.ac.uk). The authors would like to thank the Edinburgh Parallel Computing Centre for providing access to the INDY cluster. We are also grateful for the ASTRAL support given by the Cranfield University IT team. Further thanks go to Tom-Robin Teschner and Anton Shterenlikht for useful discussions, Kirsty Jean Grant for the proofreading, and Gennaro Abbruzzese for the original C++ version of the code and the mesh generator which was used for our simulations.

REFERENCES

G. Amati, S. Succi, and R. Piva. 1997. Massively parallel lattice-Boltzmann simulation of turbulent channel flow. *International Journal of Modern Physics C* 8, 4 (1997), 869–877.

J. A. Anderson, C. D. Lorenz, and A. Travesset. 2008. General purpose molecular dynamics simulations fully implemented on graphics processing units. *Journal of Computational Physics* 227, 10 (2008), 5342–5359.

P. L. Bhatnagar, E. P. Gross, and M. Krook. 1954. A model for collision processes in gases I: Small amplitude processes in charged and neutral one-component systems. *Physical Review* 94, 3 (1954), 511.

F. Cantonnet, Y. Yao, M. Zahran, and T. El-Ghazawi. 2004. Productivity analysis of the UPC language. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium*. IEEE, 254.

S. Chauwvin, P. Saha, F. Cantonnet, S. Annareddy, and T. El-Ghazawi. 2007. *UPC Manual*. The George Washington University, Washington, DC. Version 1.2.

N. Chentanez and M. Müller. 2011. Real-time Eulerian water simulation using a restricted tall cell grid. 30, 4 (2011), 82.

A. J. Chorin. 1967. A numerical method for solving incompressible viscous flow problems. *Journal of Computational Physics* 2, 1 (1967), 12–26.

Inc. Cray. 2015. *Performance Measurement and Analysis Tools* (s-2376-63 ed.). Cray.

Cray Inc. 2012. Cray standard C and C++ reference manual. (2012).

D. d'Humières. 1992. Generalized lattice-Boltzmann equations. In *Rarefied gas dynamics: Theory and simulations* (ed. B. D. Shizgal & D. P. Weaver) (1992).

D. d'Humières, I. Ginzburg, M. Krafczyk, P. Lallemand, and L. S. Luo. 2002. Multiple–relaxation–time lattice Boltzmann models in three dimensions. *Philosophical Transactions of the Royal Society of London. Series A: Mathematical, Physical and Engineering Sciences* 360, 1792 (2002), 437–451.

T. A. El-Ghazawi, F. Cantonnet, Y. Yao, S. Annareddy, and A. S. Mohamed. 2006. Benchmarking parallel compilers: A UPC case study. *Future Generation Computer Systems* 22, 7 (2006), 764 – 775.

U. Ghia, K. N. Ghia, and C. T. Shin. 1982. High-Re solutions for incompressible flow using the Navier-Stokes equations and a multigrid method. *Journal of Computational Physics* 48, 3 (1982), 387–411.

X. He and L.-S. Luo. 1997. Lattice Boltzmann model for the incompressible Navier-Stokes equation. *Journal of Statistical Physics* 88, 3-4 (1997), 927–944.

P. Husbands, C. Iancu, and K. Yelick. 2003. A performance analysis of the Berkeley UPC compiler. In *Proceedings of the 17th Annual International Conference on Supercomputing*. ACM, 63–73.

Intel. 2015. Automated Relational Knowledgebase (ARK). (2015). http://ark.intel.com/ Accessed 15/02/2015.

A. Johnson. 2005. Unified Parallel C within computational fluid dynamics applications on the Cray X1. In *Proceedings of the Cray User's Group Conference. Albuquerque*. 1–9.

I. T. Józsa, M. Szőke, T.-R. Teschner, L. Könözsy, and I. Moulitsas. 2016. Validation and Verification of a 2D lattice Boltzmann solver for incompressible fluid flow. *ECCOMAS Congress 2016 - Proceedings of the 7th European Congress on Computational Methods in Applied Sciences and Engineering* 1 (2016), 1046–1060.

D. Kandhai, A. Koponen, A.G. Hoekstra, M. Kataja, J. Timonen, and P.M.A. Sloot. 1998. Lattice-Boltzmann hydrodynamics on parallel systems. *Computer Physics Communications* 111, 1–3 (1998), 14 – 26.

D. A. Mallón, A. Gómez, J. C. Mouriño, G. L. Taboada, C. Teijeiro, J. Touriño, B. B. Fraguela, R. Doallo, and B. Wibecan. 2009. UPC performance evaluation on a multicore system. In *Proceedings of the 3rd Conference on Partitioned Global Address Space Programing Models*. ACM, 9.

S. A. Manavski and G. Valle. 2008. CUDA compatible GPU cards as efficient hardware accelerators for Smith-Waterman sequence alignment. *BMC Bioinformatics* 9, Suppl 2 (2008), S10.

S. Markidis and G. Lapenta. 2010. Development and performance analysis of a UPC Particle-in-Cell code. In *Proceedings of the 4th Conference on Partitioned Global Address Space Programming Model*. ACM, 10.

J. C. Maxwell. 1860. Illustrations of the dynamical theory of gases. *Philosophical Magazine Series 4* 20, 130 (1860), 21–37.

C. McClanahan. 2010. History and evolution of GPU architecture. In *A Paper Survey*.

Message Passing Interface Forum. 2012. MPI: A Message-Passing Interface Standard. (September 2012).

A. A. Mohamed. 2011. *Lattice Boltzmann method: Fundamentals and engineering applications with computer codes*. Springer, London.

J. Nickolls, I. Buck, M. Garland, and K. Skadron. 2008. Scalable parallel programming with CUDA. *Queue* 6, 2 (2008), 40–53.

PGAS. 2015. Partitioned Global Adress Space Consortium. (2015). http://www.pgas.org/ Accessed 15/02/2015.

B. Ren, C. Li, X. Yan, M. C. Lin, J. Bonet, and S.-M. Hu. 2014. Multiple-fluid SPH simulation using a mixture model. *ACM Transactions on Graphics* 33, 5 (2014), 171.

P. R. Rinaldi, E. A. Dari, M. J. Vénere, and A. Clausse. 2012. A lattice-Boltzmann solver for 3D fluid simulation on GPU. *Simulation Modelling Practice and Theory* 25 (2012), 163–171.

S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W.-M. W. Hwu. 2008. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*. ACM, 73–82.

J. Sanders and E. Kandrot. 2010. *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley Professional.

S. S. Stone, J. P. Haldar, S. C. Tsao, W.-M. Hwu, B. P. Sutton, Z.-P. Liang, and others. 2008. Accelerating advanced MRI reconstructions on GPUs. *Journal of Parallel and Distributed Computing* 68, 10 (2008), 1307–1318.

S. Succi. 2001. *The lattice Boltzmann equation for fluid dynamics and beyond*. Oxford.

G. L. Taboada, C. Teijeiro, J. Tourio, B. B. Fraguela, R. Doallo, J. C. Mourino, and D. A. Mallon. 2009. Performance evaluation of unified parallel C collective communications. In *11th IEEE International Conference on High Performance Computing and Communications*. IEEE, 69–78.

J. Tölke. 2010. Implementation of a lattice Boltzmann kernel using the Compute Unified Device Architecture developed by nVIDIA. *Computing and Visualization in Science* 13, 1 (2010), 29–39.

P. Valero-Lara and J. Jansson. 2015. LBM-HPC-An Open-source tool for fluid simulations. Case study: Unified Parallel C (UPC-PGAS). In *IEEE International Conference on Cluster Computing*. IEEE, 318–321.

P. Welander. 1954. On the temperature jump in a rarefied gas. *Arkiv Fysik* 7 (1954).

W. Xian and A. Takayuki. 2011. Multi-GPU performance of incompressible flow computation by lattice Boltzmann method on GPU cluster. *Parallel Computing* 37, 9 (2011), 521–535.

J. Zhang, B. Behzad, and M. Snir. 2011. Optimizing the Barnes-Hut Algorithm in UPC. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 75:1–75:11.

Y. Zhang, Jo. Cohen, and J. D. Owens. 2010. Fast tridiagonal solvers on the GPU. *ACM Sigplan Notices* 45, 5 (2010), 127–136.

Q. Zou and X. He. 1997. On pressure and velocity boundary conditions for the lattice Boltzmann BGK model. *Physics of Fluids* 9, 6 (1997), 1591–1598.

Appendix 1

The following code sections cover the *streaming*:

— Serial code

```
for (i=0; i<(*m)*(*n); i++){ //sweep through the domain
  if ( (Cells+i)->Fluid == 1 ){ //if the lattice is in the fluid domain
    for(k=0; k<9; k++){ //sweep along the nine discrete directions
      //if streaming is allowed in the current direction
      if ( ((Cells+i)->StreamLattice[k]) == 1){
        //the current distr. fct. travels to the corresponding neighbour
        (Cells+i)->F[k] = ( Cells+i+c[k] )-> METAF[k];
}  }  }  }
```

— UPC parallelisation relying on local pointer variables

```
for(i = 0;  i<(n_sub_x * n_sub_y);  i++){ //sweep in the whole sub-domain
  if ((Cells+i)->Fluid == 1) { //if the lattice is in the fluid domain
    for(k=0; k<9; k++) { //sweep along the nine discrete directions
    //if streaming is allowed in the current direction
    if ( ((Cells+i)->StreamLattice[k])) == 1 ) {
        //the current distr. fct. travels to the corresponding neighbour
    (Cells+i)->F[k] = ( Cells+i+c[k] )-> METAF[k];
}  }  }  }
```

— UPC parallelisation relying on shared static variables

```
upc_forall(i=0; i<((*nx)*(*ny)); i++; &Fluid[i]){ //sweep in the whole
    domain
  if (Fluid[i] == 1) { //if the lattice is in the fluid domain
    for(k=0; k<9; k++) { //sweep along the nine discrete directions
      //if streaming is allowed in the current direction
      if ( StreamLattice[i][k] == 1 ) {
        //the current distr. fct. travels to the corresponding neighbour
        F[i][k] = METAF[i+c[k]][k];
}  }  }  }
```

— CUDA parallelisation step 1, *streaming* kernel function

```
int bidx = blockIdx.x; //index of the current block
int tidx = threadIdx.x; //index of the current thread within the block
//global index of the threads up to 9*nx*ny
int ind = tidx + bidx*blockDim.x;
//index for the lattices up to nx*ny
int ind_l = ind - ( (*nx_d)*(*ny_d) * (int)(ind/ ( (*nx_d)*(*ny_d) ) ) );
//index for the nine layers from 0 to 8
int ind_c = (int)(ind/ ( (*nx_d)*(*ny_d) ) );
//limit computations to the physical domain
if ( ind<( 9*(*nx_d)*(*ny_d) ) ){
  //limit computations to the fluid domain
  if (Cells_const_d[ind_l].Fluid == 1){
      if ( (Cells_const_9d_d[ind].StreamLattice) == 1 ){
        //the current distr. fct. travels to the corresponding neighbour
        //the travelling occurs theoretically in the same time
        Cells_var_9d_d[ind].F = Cells_var_9d_d[ind+c_d[ind_c]].METAF;
} } }
```

— CUDA parallelisation step 2, *streaming* kernel function

```
int bidx = blockIdx.x; //index of the current block
int tidx = threadIdx.x; //index of the current thread within the block
//global index of the threads up to 9*nx*ny
int ind = tidx + bidx*blockDim.x;
//index for the lattices up to nx*ny
int ind_l = ind - ( (*nx_d)*(*ny_d) * (int)(ind/ ( (*nx_d)*(*ny_d) ) ) );
```

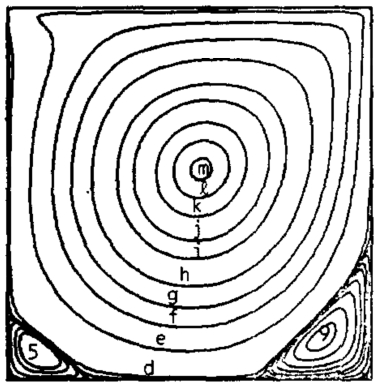
```
//index for the nine layers from 0 to 8
int ind_c = (int)(ind/ ( (*nx_d)*(*ny_d) ) );
//limit computations to the physical domain
if ( ind<( 9*(*nx_d)*(*ny_d) ) ){
  //limit computations to the fluid domain
  if (Fluid_d[ind_l] == 1){
      if ( (StreamLattice_d[ind]) == 1 ){
        //the current distr. fct. travels to the corresponding neighbour
        //the travelling occurs theoretically in the same time
        F_d[ind] = METAF_d[ind+c_d[ind_c]];
} } }
```

— CUDA parallelisation step 3, *streaming* kernel function

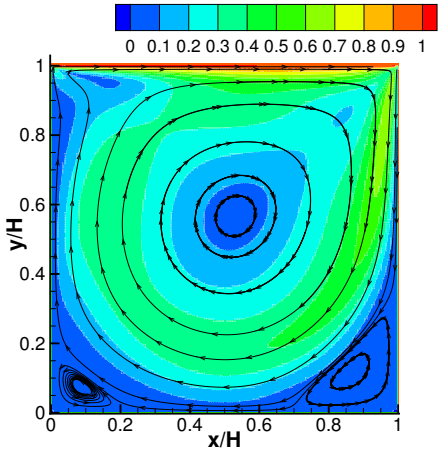
```
//global index of the threads up to nx*ny
int ind = blockIdx.x * blockDim.x + threadIdx.x;
//number of cells in a layer = nx*ny
int ms = width_d*height_d;
FLOAT_TYPE *f, *mf;
int n = height_d;
//limit computations to the physical fluid domain
if (ind < ms && fluid_d[ind] == 1){
  f_d[ind] = fColl_d[ind];
  f = f_d + ms;
  mf = fColl_d + ms;
  //the current distr. fct. travels to the corresponding neighbour
  //stream_d[ind] does not allow streaming from the boundaries
  f[ind]      = (stream_d[ind]      == 1) ? mf[ind-1]        : mf[ind];
  f[ind+ms]   = (stream_d[ind+ms]   == 1) ? mf[ind+ms-n]     : mf[ind+ms];
  f[ind+2*ms] = (stream_d[ind+2*ms] == 1) ? mf[ind+2*ms+1]   : mf[ind+2*ms];
  f[ind+3*ms] = (stream_d[ind+3*ms] == 1) ? mf[ind+3*ms+n]   : mf[ind+3*ms];
  f[ind+4*ms] = (stream_d[ind+4*ms] == 1) ? mf[ind+4*ms-n-1] : mf[ind+4*ms];
  f[ind+5*ms] = (stream_d[ind+5*ms] == 1) ? mf[ind+5*ms-n+1] : mf[ind+5*ms];
  f[ind+6*ms] = (stream_d[ind+6*ms] == 1) ? mf[ind+6*ms+n+1] : mf[ind+6*ms];
  f[ind+7*ms] = (stream_d[ind+7*ms] == 1 && ind < ms-n+1) ? mf[ind+7*ms+n-1]
     : mf[ind+7*ms];
}
```

Appendix 2

Qualitative validation of the computed velocity field at $Re = 1000$.



(a) Streamlines, [Ghia et al. 1982]



(b) Streamlines and the velocity contour, serial MRT simulation on medium mesh, results coloured by u/u_{lid}