An Evaluation of GPU Performance Portability Using BGK2D, a Lattice Boltzmann Code

Giorgio Amati
g.amati@cineca.it
CINECA
Roma, Italy

Filippo Spiga
fspiga@nvidia.com
NVIDIA
Cambridge, England

Georgios Markomanolis
georgios.markomanolis@amd.com
AMD
Grenoble, France

ABSTRACT

Performance portability is a key issue in today's HPC arena. In this article, we will present an opensource *agnostic* 2D CFD code, based on the Lattice Boltzmann Method (LBM), explicitly developed to explore different directive-based parallel models. It can be used as a tool to find a performance baseline for these approaches and to understand the level of performance that can be achieved.

CCS CONCEPTS

• **Computing methodologies** → **Concurrent computing methodologies**; **Concurrent programming languages**; • **Software and its engineering** → **Parallel programming languages**.

KEYWORDS

Directive-based parallel mode, GPU, OpenACC, OpenMP offload, Lattice Boltzmann Method

ACM Reference Format:
Giorgio Amati, Filippo Spiga, and Georgios Markomanolis. 2023. An Evaluation of GPU Performance Portability Using BGK2D, a Lattice Boltzmann Code. In *Proceedings of In Platform for Advanced Scientific Computing (PASC'24) (PASC24).* ACM, New York, NY, USA, 10 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Performance Portability is a key aspect of today's High-Performance Computing (HPC): Exascale-class supercomputers are few, heavily based on accelerators like General Purpose GPU (GPGPU) [13], and ask for different parallel paradigms, depending on the different accelerators used [11]. For many code developers could be almost impossible to develop and maintain different versions, one for each different accelerator.
Moreover, Performance Portability for the Urgent Computing framework is mandatory, like the Center of Excellence (CoE) Cheese2P [3, 6] where the need to perform in a fixed time window a workflow to compute a map risk related to a possible Tsunami, or an Earthquake, ask to be *quite* efficient on all possible available accelerators at the moment.

PASC24, June 03–05, 2024, Zurich, CH

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

In this article we will focus on directive-based programming models that, from the programming point of view which, *should be* less *intrusive* because it *should not* request heavy code refactoring and *should* allow to run *smoothly* on serial, multicore, and on GPUs.
Actually, the three main vendors for GPGPU present different solutions for performance that are not portable. Focusing on `Fortran` language, only `OpenMP offload` is fully supported by the three vendors, but, to our knowledge, the effective sustained performance is not clear, in particular for an even simple, real-world code [9]. Other parallel models, like `OpenaACC` or intrinsic standard Fortran for multi-threading like `do concurrent` are not supported by all three vendors. The AMD solution, `HIPfortran`, is an open-source solution that could be applied also to other GPU like NVIDIA or Intel [24].
But, to our knowledge, there Is no clear idea of the possible performance that the different approaches could achieve.
This article will present an open-source Code, based on Lattice Boltzmann Method (LBM), tailored to benchmark performance portability using `fortran90`. We'll try to give an answer to a simple but important question: which performance a directive-based parallel model could achieve on today's GPGPUs with *relatively simple* coding efforts? We are aware that total performances depend on many different aspects, like compiler maturity, but to define the problem, the first step is to understand how far we are from the theoretical limit, given by the Roofline Model [23]. In this code there are three different ways to exploit GPGPU:

- `OpenMP offload`
- `OpenACC`
- `do concurrent`

Is possible to switch between different options through pre-processing directives. For every loop ported on GPGPU we have the following structure:

```
#ifdef OFFLOAD
!$OMP target teams distribute parallel do simd
        do j=1,m
#elif OPENACC
!$acc parallel loop independent
        do j=1,m
#else
        do concurrent (j=1:m)
#endif
....do something.....
```

The `do concurrent` structure is used by default. If `-DOPENACC` preprocessing flag is used at compile-time, the `OpenACC` parallel model is activated. While using `-DOFFLOAD` the `OpenMP offload` parallel model is used.
We are aware that other approaches, like `cudaFortran` for NVIDIA

or `hipfortran` for AMD, could be performing more efficiently, but the idea behind this work is to understand the baseline level of performance that could be achieved using a directive-based approach to reduce *deep* code modification/refactoring.
`BGK2D` is a 2D CFD code for single-phase incompressible flows. It is used because, with respect to many other classical Computational Fluid Dynamics (CFD) schemes, is a highly parallel numerical scheme: only a few loops, less than 10, must be *modified* to run on GPGPU efficiently.
The code is not meant to be the fastest but is *quite* efficient to get a reasonable baseline of performance portability for this class of codes. The full 3D code, based on `MPI+OpenACC`, was developed using also Leonardo Supercomputer [21].
The article is organized as follows: in Sec. 2 the LBM is briefly introduced, together with its main performance issues. In Sec. 3 the main aspect of `BGK2D` code is presented with its main features. In Sec. 4 code validation for different classical CFD test cases is shown. In Sec. 5 performance figures for GPGPUs are shown. Comments and conclusions are presented in Sec. 6. In the Appendix, more details on the repository structure and compiler's options are shown.

2 LATTICE BOLTZMANN METHOD

The code presented here is a "downsize" of a 3D incompressible flows [1, 4, 20]. The Lattice Boltzmann Method (LBM) was developed in the late 1980s as lattice gas cellular automata evolution. It presented a huge number of applications across a broad spectrum of complex flow problems, from fully developed turbulence to micro and nanofluidics [5, 14], up to quark-gluon plasmas [18, 19].
The main idea is to solve a minimal Boltzmann kinetic equation for a set of discrete distribution functions, usually defined as *populations*, $f_i(\vec{x};t)$, expressing the probability of finding a particle at position $\vec{x}$ and at time t, with a discrete velocity $\vec{v} = \vec{c}_i$. The set of discrete velocities must be chosen in such a way as to secure enough symmetry to comply with mass-momentum-energy conservation laws of macroscopic hydrodynamics, as well as with rotational symmetry. Fig. 1 shows the lattices used for 2D LB simulations, with a set of nine discrete velocities (a.k.a. D2Q9). We do not solve directly Navier-Stokes equations so each for gridpoint 9 populations with a different $\vec{c}$ velocity direction, including the one with $\vec{c} = (0,0)$ should to be stored, while is not necessary to store derived quantities like velocity.
In its compact form, the main LB equation reads as follows:

$$f(\vec{x}+\vec{c}_i, t+1) - f_i(\vec{x};t) = -\omega(f_i(\vec{x};t) - f_i^{eq}(\vec{x};t)) + S_i, \quad i = 0, b \quad (1)$$

where $\vec{x}$ and $\vec{c}_i$ are position and velocity vectors in ordinary space, f_i^{eq} is the equilibrium distribution function; time advance is in lock-step mode and the lattice time step is made unit, so that $\vec{c}_i$ is the length of the link connecting a generic lattice site node $\vec{x}$ to its b neighbours, $\vec{x}_i = \vec{x} + \vec{c}_i$.
The local equilibria are provided by a lattice truncation, to the second order in the Mach number $M = u/c_s$, of the Maxwell-Boltzmann distribution, where c_s is the lattice sound speed, namely

$$f_i^{eq}(\vec{x};t) = w_i \rho (1 + u_i + q_i) \quad (2)$$

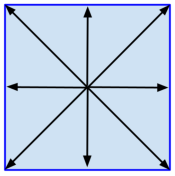

Figure 1: D2Q9 lattice, composed of a set of 9 discrete velocities, including the one with both components equal to 0.

where w_i is a set of weights normalized to the unit and,

$$u_i = 3\frac{\vec{u}\cdot\vec{c}_i}{c_s} \qquad q_i = 9\frac{(\vec{u}\cdot\vec{c}_i)^2}{2c_s^2} - 3\frac{u^2}{2c_s^2} \quad (3)$$

In eq. 3 the left term is linear in velocity, the right one quadratic. Finally, S_i is a source term for the fluid interaction with external (or internal) sources.
The eq. represents the following situation: the collision step (right-hand side) where the populations are locally *relaxed* to the equilibrium and the streaming step (left-hand side) where the populations were scattered away to the corresponding neighbour in $\vec{x} + \vec{c}_i$ at time $t+1$. The above scheme can be shown to reproduce the Navier-Stokes equations for an isothermal quasi-incompressible fluid of density and velocity

$$\rho = \sum_i f_i \qquad \vec{u} = (\sum_i f_i \vec{c}_i)/\rho \quad (4)$$

providing the lattice with suitable symmetries, and the local equilibria. The relaxation parameter ω dictates the viscosity of the lattice fluid according to the relation

$$\nu = c_s^2(\omega^{-1} - 1/2) \quad (5)$$

Full details can be found in the vast literature on the subject [12].

2.1 Performance Issues

The appealing feature of the LBM is the simple "*move and collide*" scheme. From the computational point of view, neglecting boundary conditions, all flops are performed in the collision term (right hand side of eq. 1). Then you have to perform a *rigid* data streaming of the i-th population from $\vec{x}$ to $\vec{x} + \vec{c}_i$. This approach, we define as `ORIGINAL` is shown in Fig. 2 where two different populations have to be used in order to have an efficient streaming step implementation.
To reduce the total bandwidth request, streaming and collision steps can be fused into one single routine, with a scattering read from a pre-collision population f^{pre} and write back the result on a post-collision populations f^{post} and swapping $f^{post} \rightarrow f^{pre}$ at time $t+1$. This approach, we call `FUSED`, is shown in Fig. 3, allows using pointers, to reduce by a factor 2 the memory bandwidth requirements. To achieve *good* performance a code refactoring was done, in detail:

- Only the quantities to be integrated according to the eq. 2 are stored in arrays. All derived quantities like velocity $\vec{v}$ are computed *on the fly* via eq. 4 without using arrays.
- All computations are performed using local variables or populations f_i.

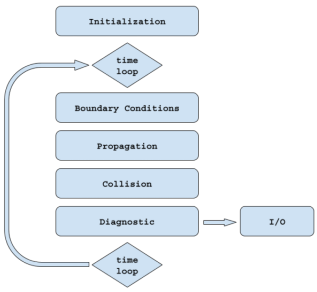

Figure 2: Flowchart for the ORIGINAL implementation

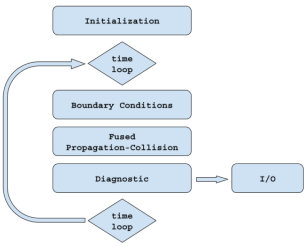

Figure 3: Flowchart for the FUSED implementation

- No subroutines or functions are used to compute f^{eq} or derived quantities like $\vec{v}$. The idea is to exploit *Common Subexpression Elimination & Constant folding*: even if these operations could be done by a compiler is really useful to explicitly make by hand, because compilers are by definition conservative and could not perform these optimizations if not forced. As an example for computing velocity (eq. 4) you have to expand vector $\vec{c}_i$: the only values it assumes are 0 or ±1. This means that there's no need to perform any multiplications as the compact form could wrongly suggest. The same holds for computing f^{eq}(eq. 2).
- The memory layout used was Structure of Array (SoA): nine different arrays are defined, one for each f_i. For the collision step, we have nine different data streams, but each with unitary stride. For the streaming step we have, for each f_i, a memory access with unitary stride. More complex Data Layouts can be found in [10].

With this code refactoring at the end, the `ORIGINAL` version, has only two 2D loops (in `streaming` and `collision` step) and four 1D loops for boundary conditions to handle.
For the `FUSED` version we have only one 2D loop (`fused_collision`) to handle, plus 4 1D loops for boundary conditions.

3 CODE STRUCTURE

The code, written in `Fortran90` with dynamic allocation, is developed to be controlled at compile-time using pre-processing directives, i.e., with `-D<DIRECTIVE>` flags to pass to the compiler via the `make` Linux utility.
The only software requirements are a standard `FORTRAN90` compiler, a standard `C` compiler, and `make` utility. Output can be visualized using `gnuplot` for the 0-D and 1-D diagnostic and a visualization tool like `paraview` for `visit` for the 2D `*.vtk` files.
Different options and test cases can be chosen at compile time. There are four different options to choose from and the syntax is:

```
make target <COMPILER>=1
            <PRECISION>=1
            <IMPLEMENTATION>=1
            <TEST_CASE>=1
```

For example, to compile the double precision simulation of Lid-Driven Cavity, using the `FUSED` implementation and OpenACC parallel model with NVIDIA compiler, the command line is:

```
make openacc NVIDIA=1
             DOUBLE=1
             FUSED=1
             LDC=1
```

This means that 800 different combinations could be activated, but some are not supported by some compilers (see tab. 1 and tab. 2). In the following sections a more detailed description of the different available options and those supported by compilers are presented.

3.1 Makefile Targets

There are five different targets to choose from:

- `serial`: With this target a serial code for the CPU is generated.
  The exe file will be `bgk2d.serial.x`. This is the default target.
- `manycore`: With this target a multi-threaded code for CPU is generated, laying on Fortran's `do concurrent` language standard parallelism.
  The exe file will be `bgk2d.multicore.x`
- `doconcurrent`: With this target a GPU code, based on Fortran 2008 `do cuncurrent` will be generated, if supported by the used compiler.
  The exe file will be `bgk2d.doconcurrent.x`.
- `openacc`: with this target a GPU code, based on `openacc` parallel model, will be generated if supported by the used compiler.
  The exe file will be renamed `bgk2d.openacc.x`.
- `offload`: With this target a GPU code will be generated, based on `OpenMP offload` parallel model if supported by the compiler.
  The generate exe file will be renamed as `bgk2d.offload.x`.

3.2 Compilers options

There are five different tested compilers to choose from. For all different compilers standard equivalent optimization flags are used. More details will be found in the Appendix

- `GNU`: With this option GNU `gfortran` compiler is be used. This is the default compiler.
- `NVIDIA`: With this option NVIDIA `nvfortran` compiler will be used.
- `INTEL`: With this option Intel `ifx` compiler will be used.
- `CRAY`: With this option CRAY `ftn` compiler is be used.
- `AMD`: With this option AMD `flang` compiler will be used.

In Table. 1, the compiler support for the different targets is presented.

Table 1: Compiler's support respect different targets

	nvfortran	ftn	flang	ifort	gnu
serial	yes	yes	yes	yes	yes
manycore	yes	no	yes	yes	yes
doconcurrent	yes	no	no	yes	no
openacc	yes	yes	no	no	yes
offload	yes	yes	yes	yes	yes

3.3 TEST Case Options

There are four different predefined test cases to choose from.

- `POF`: Poiseuille Flow, flow in a channel with volume force. The initial condition is a rest flow.
- `TGV`: Taylor-Green Vortex, decaying flow in a squared periodic box. The initial condition is a developed Taylor-Green vortex
- `LDC`: Lid Driven Cavity. forced flow in a squared block with no-slip walls. The initial condition is rest flow. This is the default test case.
- `VKS`: Von Karman Street. Flow around a Cylinder with inflow/outflow and periodic boundary conditions up and down. The initial condition is rest flow.

3.4 Precision options

The stored quantities, i.e. $f_i(\vec{x})$, can have a different kind with respect to the one used for computations. In the code, all the floating point operations are performed using scalar variables that can present a different precision with respect to the stored one. There are four different precision options to choose from.

- `MIXED1`: With this option all stored quantities are in half-precision (i.e., 2 bytes per word). All computations are done using single precision (i.e., 4 bytes).
- `SINGLE`: With this option all stored quantities are in single precision and all the floating point operations are done using single precision (i.e., 4 bytes). This is the default precision.
- `MIXED2`: With this option all stored quantities are in single precision (i.e., 4 bytes per word). All computations are done using double precision (i.e., 8 bytes)
- `DOUBLE`: With this option all the stored quantities are in single precision and all the computations are done using double precision (i.e. 8, bytes)

In Table 2 the precision support for the different compilers is presented, while in Table 3 the range for the different precision is presented.

3.5 LBM Implementations

There are two different LBM implementations to choose from:

- `ORIGINAL`: classical implementation with two separated routines (Fig. 2).
  - `movef`: In this subroutine populations $f_i(\vec{x})$ are streamed according to their velocity set $\vec{c}_i$. Two arrays are used to avoid data races.
  - `col`: In this subroutine, there is the local update of the population $f_i(\vec{x})$. Two arrays are used to avoid data races.
- `FUSED`: in this implementation only a single subroutine is used (`col_MC`) with a scattered read from the old population and a write of the updated populations on a different field (fig. 3).
  Using pointers populations are swapped in a flip/flop way every time step. Respect `ORIGINAL` version the total Bandwidth is halved.

3.6 Theoretical Performance

The roofline model can be used to estimate, according to its Arithmetic Intensity[1], we can extract the performance limit of the code [23].
The code presents an Arithmetic Intensity of 1.4 in single precision and 0.7 in double precision for the `FUSED` implementation. This code, like almost all CFD code, is Bandwidth Limited using any today's devices, both CPU and GPU.
For LBM, a good performance metric is Mega Lattice Updated per second (MLUPs), i.e., how many millions of gridpoints will be updated in 1 second. We will use this Figure of Merit, knowing that the code performs about 100 flops per single gridpoint.
According to the theoretical BW for a single GPU or GCD we can extract the theoretical peak for a simulation using GPU. These are the GPUs tested so far.

- Nvidia V100 @ 32GB
- Nvidia A100 @ 64GB
- Nvidia H100 @ 96GB (SXM module)
- AMD Instinct™ MI250X @ 128GB

In Table 4 the performance limit for the used GPU is shown. For a double-precision simulation, the limit is half of the single-precision one. It should be noted that for all the experiments we use 1 GCD of the MI250X as the code has not integrated yet MPI, we expect that it would double the performance when we can use both GCDs. More detail about the GPU specs can be found in [2, 15–17]

3.7 Licence

This code is released under the MIT license. It can be downloaded at `https://github.com/gamati01/BGK2D_GPU`.

4 CODE VALIDATION

To validate the code four different test cases are available in the `TEST` directory. This article will present figures for the Taylor-Green vortex, lid-driven cavity, and Von Karman streets.

[1] Arithmetic Intensity is the ratio between floating point operations to be performed and byte moved back and forth to complete these operations.

Table 2: Compiler's support respect different targets

	nvfortran	ftn	flang	ifort	gnu
HALF	yes	no	no	no	no
SINGLE	yes	yes	yes	yes	yes
DOUBLE	yes	yes	yes	yes	yes

Table 3: Precision's range: maximum value and epsilon for different precision are presented using NVIDIA compiler.

	huge	epsilon
HALF	$6.5504E+4$	$9.7656E-4$
SINGLE	$3.4028235E+38$	$1.1920929E-07$
DOUBLE	$1.7976931348623157E+308$	$2.2204460492503131E-016$

Table 4: Roofline Performance limit, in MLUPs, for single precision simulation. For MI250X we refer to a single Graphics Complex Die (GCD)

	Bandwidth (GB/s)	ORIGINAL	FUSED
Nvidia V100	$\simeq$ 900	$\simeq$ 6300	$\simeq$ 12600
Nvidia A100	$\simeq$ 1600	$\simeq$ 11200	$\simeq$ 22400
Nvidia H100	$\simeq$ 4000	$\simeq$ 28000	$\simeq$ 56000
AMD MI250X	$\simeq$ 1600	$\simeq$ 11200	$\simeq$ 22400

4.1 Taylor-Green Vortex

Taylor-Green Vortex (TGV) is a decaying vortex flow. Its initial condition consists of a set of counter-rotating vertices,

$$u(x,y) = A\cos(ax)\sin(by) \tag{6}$$

$$v(x,y) = B\sin(ax)\cos(by) \tag{7}$$

where u is the velocity component in the x direction, v is the velocity component in the y direction.
For an incompressible flow exists an analytic solution, for $A = B = a = b = 1$ it holds:

$$u(t,x,y) = (\cos(x)\sin(y))e^{t/\tau} \tag{8}$$

$$v(t,x,y) = -(\sin(x)\cos(y))e^{t/\tau} \tag{9}$$

where

$$\frac{1}{\tau} = \frac{2\pi\nu}{(h/2)^2)} \tag{10}$$

is the relaxation time and it depends from the viscosity ν and the box size h. As a consequence the total energy decays with an exponential law.

$$E(t) = E(t=0)e^{t/\tau} \tag{11}$$

In fig. 4 the energy decay, using different precision is shown. A 1024^2 box was used with $\nu = 0.3$. It is evident the range of different precision. Below $\simeq \epsilon^2$ floating point precision is no more correct (tab. 3)[2].

[2]We have reached the zero machine for that precision.

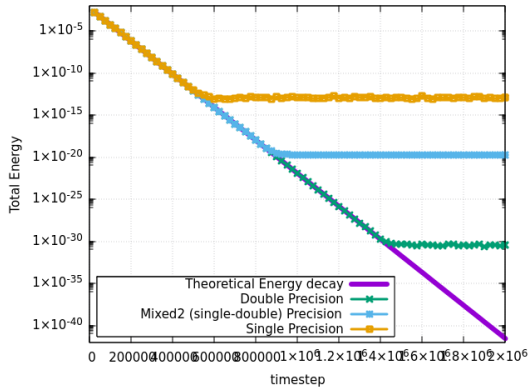

Figure 4: Taylor Green Flow: energy decay for different precision

4.2 Lid Driven Cavity

The Lid Driven Cavity (LDC) is a classical CFD benchmark. It consists of a closed box with no-slip walls and the flow is driven by a moving wall with a tangential fixed velocity u_0. Here we will present the results of three different Re numbers, compared with the results from Ghia [7]. Fig. 5 compares for a simulation at $Re = 100$ and a 128^2 grid is used. Fig. 6 compares for a simulation at $Re = 1000$ and a 256^2 grid is used. Fig. 7 compares for a simulation at $Re = 10000$ and a 512^2 grid is used while running on GPU.
All Re numbers yielded good results.

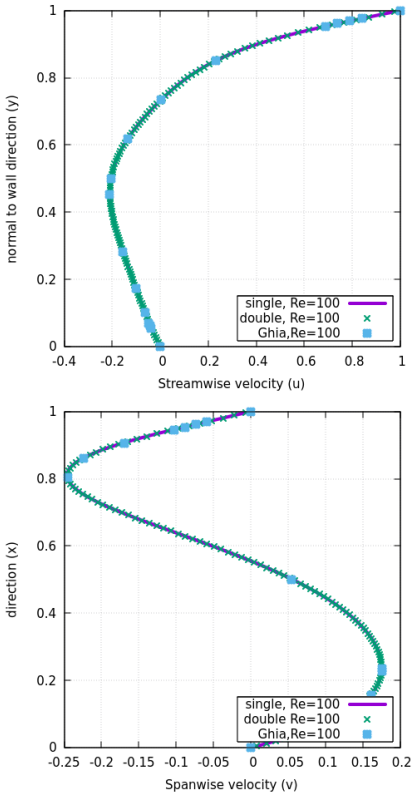

Figure 5: Driven Cavity at $Re = 100$ compared with Ghia's data for single and double precision. Up: streamwise velocity u as a function of y direction. Down: spanwise velocity v as a function of x direction.

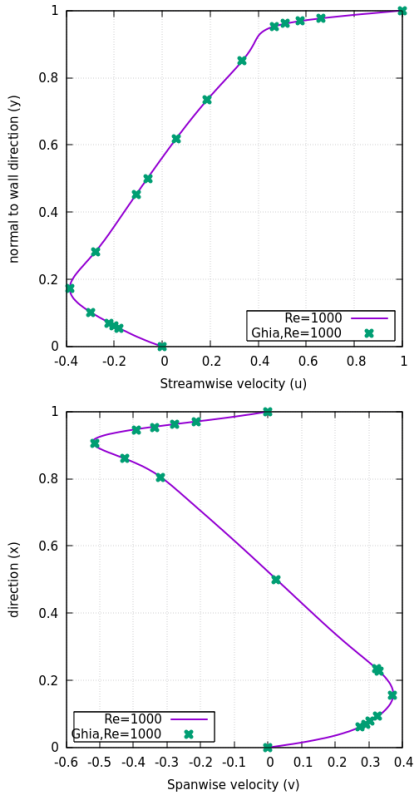

Figure 6: Driven Cavity at $Re = 1000$ compared with Ghia's data for a single precision simulation. Up: streamwise velocity u as a function of y direction. Down: spanwise velocity v as a function of x direction.

4.3 Von Karman Street

Von Karman Street is a pattern of swirling vortices caused by vortex shedding. It occurs, for a flow around a cylinder, at $Re = 100$. To validate the code, a comparison with similar simulations done using `OpenFOAM` [22] is presented. In Fig. 8 a velocity magnitude snapshot of the flow with the two methods is presented, Fig. 9 shows the drag coefficient history and Fig. 10 shows the lift coefficient history for the two different numerical methods.

The lift and the drag history are qualitatively *the same* for BGK and OF. The differences can be explained in terms of different resolutions: for BGK method the obstacle is *staircase-like* one, while for OpenFoam is more smooth.

5 PERFORMANCE FIGURES

These are the input files used to get the performance figures for the different HW at compute node and GPU level:

- CPU (Multicore, single compute node): `bgk.node.input`
  - Size= 1024^2
  - Re=1000
  - Single precision
  - Timestep=500000
- GPU (single GPU or GCD): `bgk.gpu.input`
  - Size= 4096^2
  - Re=10000
  - Single precision
  - Timestep=100'000

5.1 CPU Multithreaded Performance

In this section, performance figures and features of different Compute Nodes are presented. `DO CONCURRENT` multicore parallelization is used.
Table 5 shows the performance in terms of MLUPs as reported for the tested CPU. Only fused version figures and single precision are reported.

- **Intel1**: 1x Intel Xeon Platinum 8358 CPU @ 2.60GHz (leonardo.cineca.it)
  - nvfortran rel. 23.5
  - ifx rel. 2023.0.0
- **AMD**: 2x AMD EPYC 7742 64-Core Processor (dgx.cineca.it)
  - nvfortran rel.22.3

5.2 GPU Performance

In this section, performance figures and features of different GPUs are presented.

Table 5: Compute Node performance, only fused version and single precision figures are reported

	Compiler	version	mlups	collision (%)	b.c (%)
intel1	nvfortran	fused	964	98	2
intel1	ifx	fused	452	98	2
AMD	nvfortran	fused	2248	79	21

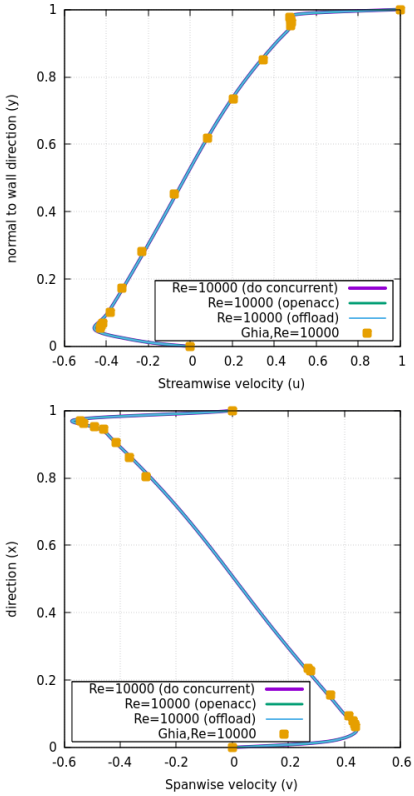

Figure 7: Driven Cavity at $Re = 10000$ compared with Ghia's data using all three available parallel model for GPU. Up: streamwise velocity u as a function of y direction. Down: spanwise velocity v as a function of x direction.

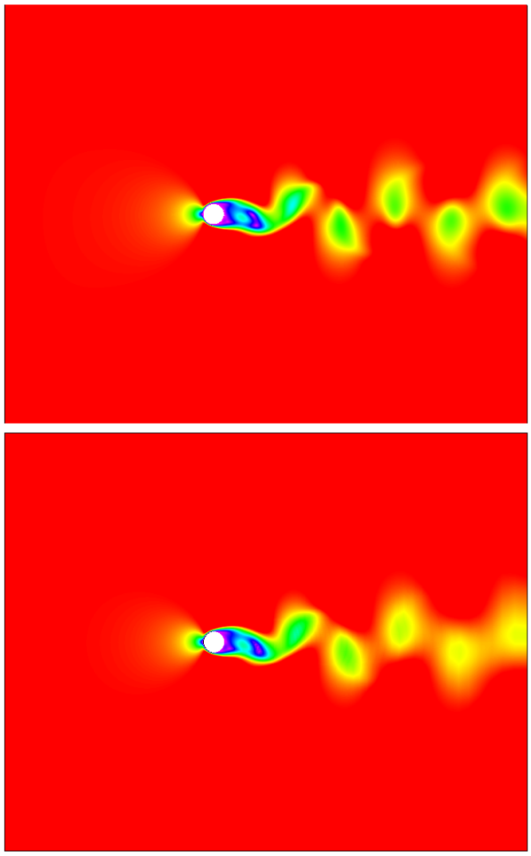

Figure 8: Von Karman Street: velocity field for $Re = 100$ a at $t \simeq 500$, Up: BGK2d, Down: OpenFoam

These are the GPUs and the compiler used. Figures using GNU compiler are not reported because they presented very low performance.

- **V100**: NVIDIA V100 @ 32GB (g100)
  - nvfortran rel.22.3
- **A100**: NVIDIA A100 @ 64GB (Leonardo)
  - nvfortran rel.23.5
- **A100-1**: NVIDIA A100 @ 40GB (DGX)
  - nvfortran rel.22.3
- **H100**: NVIDIA H100 @ 96GB (SXM module)
  - nvfortran rel.23.11
- **MI250**: AMD MI250X, half (Adastra)
  - ftn rel.16.0.0
  - flang rel.15.0.0

Table 6 shows the performance, in terms of MLUPS, as reported for the tested GPU. Only FUSED version and SINGLE precision is reported. It is worth noting that, for GPU, the sustained performance is between 20% and 75% of the theoretical peak (Table 4).

6 COMMENTS & CONCLUSIONS

In this article, we show that even with simple directive-based parallel models, quite interesting performance figures are obtained. According to tab. 6 we get, as a baseline for BGK2D code, from $\simeq$ 20% up to 70% of the theoretical limit according to the Roofline Model (tab. 4). In this sense, we can say that a *relative* performance portability can be achieved.
Also, It is worth noting that the performance reached using 1 GPU is between 5 and 16 times faster with respect to 1 GPU (tab. 5). The

Table 6: GPU performance: all figures refer to single precision and fused version

	Compiler	paradigm	MLUPs	collision (%)	boundary (%)	% Peak
V100	nvfortran	offload	9099	97	3	72%
V100	nvfortran	openacc	7779	98	2	62%
V100	nvfortran	do concurrent	7721	98	2	61%
A100	nvfortran	offload	16634	96	4	74%
A100	nvfortran	openacc	14901	97	3	67%
A100	nvfortran	do concurrent	14491	97	3	65%
A100-1	nvfortran	offload	15467	93	6	69%
A100-1	nvfortran	openacc	16008	95	5	71%
A100-1	nvfortran	do concurrent	15993	95	5	71%
H100	nvfortran	offload	26219	89	7	47%
H100	nvfortran	openacc	30619	91	8	55%
H100	nvfortran	do concurrent	31350	92	7	56%
MI250X (half)	ftn	offload	12880	96	4	58%
MI250X (half)	ftn	openACC	11711	97	3	53%
MI250X (half)	flang	offload	4858	96	4	20%

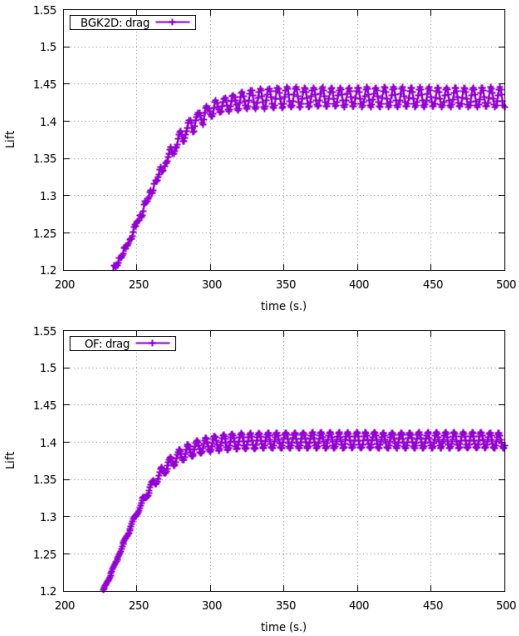

Figure 9: Von Karman Street: Drag history for $R = 100$, Up: BGK2D, Down: OpenFoam

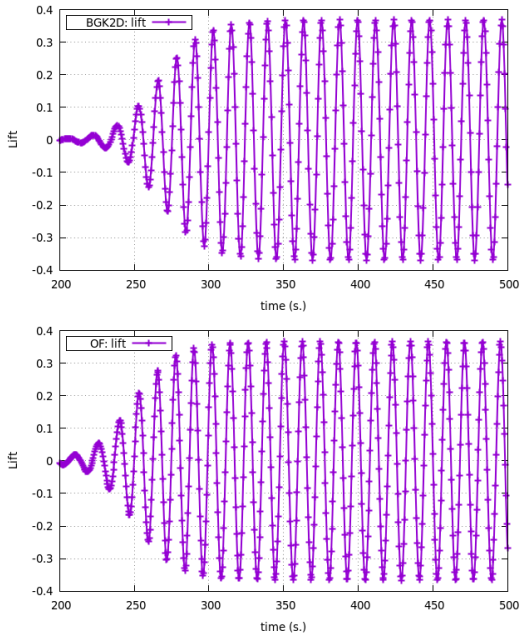

Figure 10: Von Karman Street: Lift history for $R = 100$, Up: BGK2D, Down: OpenFoam

factor $\simeq 3$ in performance can be explained by the different levels of maturity of the compilers used in this work and in a *implicit* bias towards NVIDIA GPU, where the original 3D code was developed and optimized.
A fine-tuning phase, oriented on a particular GPU, and consequently yelding a certain loss of portability, is requested to further speed up performance.
We'd like also to underline that the main idea was not to make a ranking between GPUs but a way to set a reasonable performance portability baseline for an LBM-based code written in `Fortran`. The Next step will be to port, with the same framework, a more complex, at least in terms of data management, 3D LBM-based code.

A REPOSITORY STRUCTURE

The code is developed to be self-consistent. No external libraries are needed to compile/run the code. The directory structure is the following

- `RUN`: In this directory the executable file will be created (i.e., `bgk2d.*.x`) after the compilation step.
- `SRC`: In this directory all the source files, with the `Makefile`, are present.
- `UTIL`: In this directory some utility scripts and some input files are present.
- `TEST`: In this directory the four test cases, with some scripts, used for the code validation are present, each in a different directory.
- `BENCH`: In this directory the three benchmark scripts (for single core, single compute node, and GPU) are present, each in a different directory.
- `CI`: In this directory some script to perform a *fast* check for compilation and execution of all possible configurations.

B HOW TO COMPILE

These are the steps to compile the code:

(1) Go in the `SRC` directory.
(2) Compile the desired configuration: e.g.,
`make serial DOUBLE=1 FUSED=1 LDC=1 GNU=1`
for a lid-driven cavity test in double precision, with the fused implementation and GNU compiler.
(3) If the compilation is successful, the exe file, i.e., `bgk2d.serial.x`, will be copied in the `../RUN/` directory.
(4) Go in the `RUN` directory.
(5) Copy from the `UTIL` directory a `bgk.input` file.
For the single core run: `../UTIL/bgk.core.input bgk.input`.
(6) Run the code: `./bgk2d.serial.x`.

The code uses dynamic allocation, so, once fixed, the test case has no need to recompile if you need to vary the size of the simulation box.

B.1 Validation

In the `TEST` directory you will find four different directories, each for a different test case. In each directory there are some scripts `run.*.x` that will compile, copy the `*.exe`, and run in a dedicated directory. Scripts `runMe.*.x` will perform different tests.

B.2 Further Pre-processing Flags

Other directives available are:

- `DEBUG_*`: Different debugging level. With `DEBUG_1` activated there is debug info at subroutine level, except for those subroutines in the time loop. With `DEBUG_2` activated, there is debug info also at loop level. With `DEBUG_3` activated, there is info for each single iteration for collision subroutines.
- `NO_SHIFT`: With this option activated there Is no *shift* of equilibrium distribution [8]. It reduces the precision range of the simulation.
- `TRICK_1`: With this option a loop merging is activated in the boundary conditions for Lid-Driven Cavity test is done to increase performance, sensible for size < 2048.
- `TRICK_2`: With this option we explicitly set `thread_limit` and `num_teams`. It improves the performance using `flang` compiler.
- `NO_BINARY`: With this pre-processing flag activated the vtk visualization files will be ASCII one. It could be useful for debugging purposes.
- `KERNELS`: With this option activated, for the `OpenACC` paradigm, `kernel` clause will be used instead of the `parallel` clause.

B.3 Input/Output

The code will produce the following file, together with some standard output:

- Input/Log files (ASCII)
  - `bgk.input`: input file. It contains all the information needed by the run, like timestep to perform, diagnostic and so on.
  - `bgk.log`: log file of the simulation.
  - `bgk.time.log`: log file with history of the time spent in the different sections (collision, boundary condition, etc.).
  - `bgk.perf`: file with some performance figure (Mlups) and time per section (collision, diagnostic, boundary condition, streaming etc.).
- 0D diagnostic files (ASCII)
  - `diagno.dat`: mean value for velocity and pressure for all the computational box.
  - `probe.dat`: instantaneous velocity and pressure figures for a defined gridpoint.
- 1D diagnostic files (ASCII)
  - `prof_i.dat`: instantaneous values of velocity and pressure along x direction.
  - `prof_j.dat`: instantaneous values of velocity and pressure along y direction.
  - `u_med.dat`: mean value of of velocity and pressure along y direction.
- 2D diagnostic files (Binary/ASCII)
  - `tec_*.vtk`: visualization file, using vtk standard.
  - `save.bin`: save file.
  - `restore.bin`: restart file.

C COMPILER OPTIONS

C.1 Compiler options for serial version

- `GNU`: `-Ofast`
- `INTEL`: `-O3 -xCORE-AVX512 -mtune=skylake-avx512 -assume contiguous_pointer`
- `NVIDIA`: `-O2 -Mnodepchk -Mcontiguous`
- `AMD`: `-O3`
- `CRAY`: `-O3`

C.2 Compiler options for multicore version

- `GNU`: `-fopenmp -ftree-parallelize-loops=4`
- `INTEL`: `-qopenmp`
- `NVIDIA`:`-stdpar=multicore`
- `AMD`: `-h thread_do_concurrent`

C.3 Compiler options for GPU version

- NVIDIA
  - Do concurrent:-stdpar=gpu
  - Offload:-mp=gpu
  - OpenACC: O2 -acc -ta=tesla:cc80/cc70,managed
- AMD
  - Do concurrent: not supported
  - Offload: -fopenmp --offload-arch=gfx90a
  - OpenACC: not supported
- CRAY
  - Do concurrent: not supported
  - Offload: -h omp
  - OpenACC: -h acc

D ACKNOWLEDGMENTS

This work has been made possible thanks to the input and feedback from many people including Prof. Giacomo Falcucci (Univ. Tor Vergata, Rome), Dr. Vesselin Krastev (Univ. Tor Vergata, Rome) and the CINES Adastra system Support Staff.

REFERENCES

[1] Giorgio Amati, Sauro Succi, Pierluigi Fanelli, Vesselin K. Krastev, and Giacomo Falcucci. 2021. Projecting LBM performance on Exascale class Architectures: A tentative outlook. *Journal of Computational Science* 55 (2021), 101447. https://doi.org/10.1016/j.jocs.2021.101447

[2] AMD. 2022. AMD CDNA 2 Architecture datasheet. https://www.amd.com/content/dam/amd/en/documents/instinct-business-docs/white-papers/amd-cdna2-white-paper.pdf.

[3] CHEESE2P. 2023. Centre of Excellence for Exascale in Solid Earth. https://cheese2.eu/.

[4] Giacomo Falcucci, Giorgio Amati, Pierluigi Fanelli, and et al. 2021. Extreme flow simulations reveal skeletal adaptations of deep-sea sponges. *Nature* 595 (2021), 537–541. https://doi.org/10.1038/s41586-021-03658-1

[5] Giacomo Falcucci, Giorgio Amati, Vesselin K Krastev, Andrea Montessori, Grigoriy S Yablonsky, and Sauro Succi. 2017. Heterogeneous catalysis in pulsed-flow reactors with nanoporous gold hollow spheres. *Chemical Engineering Science* 166 (2017), 274–282.

[6] Arnau Folch, Claudia Abril, Michael Afanasiev, and Giorgio Amati et al. 2023. The EU Center of Excellence for Exascale in Solid Earth (ChEESE): Implementation, results, and roadmap for the second phase. *Future Generation Computer Systems* 146 (2023), 47–61. https://doi.org/10.1016/j.future.2023.04.006

[7] 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. *J. Comput. Phys.* 48, 3 (1982), 387–411. https://doi.org/10.1016/0021-9991(82)90058-4

[8] Farrel Gray and Edo Boek. 2016. Enhancing computational precision for lattice Boltzmann schemes in porous media flows. *Computation* 4, 1 (2016), 11. https://doi.org/10.3390/computation4010011

[9] Jeff R Hammond, Tom Deakin, James Cownie, and Simon McIntosh-Smith. 2022. Benchmarking Fortran DO CONCURRENT on CPUs and GPUs Using Babel-Stream. In *2022 IEEE/ACM International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*. IEEE, IEEE, ., 82–99.

[10] Gregory Herschlag, Seyong Lee, Jeffrey S Vetter, and Amanda Randles. 2018. GPU data access on complex geometries for D3Q19 lattice Boltzmann method. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, IEEE, ., 825–834.

[11] Andreas Herten. 2023. GPU Vendor/Programming model compatibility Matrix. https://github.com/AndiH/gpu-lang-compat.

[12] Timm Krueger, Halim Kusumaatmaja, Alexandr Kuzmin, Orest Shardt, Goncalo Silva, and Erlend Magnus Viggen. 2016. *The Lattice Boltzmann Method: Principles and Practice*. Springer, .

[13] Top500 Supercomputing list. 2023. Top500 list, November 2023. https://www.top500.org/lists/top500/2023/11/.

[14] Andrea Montessori, Pietro Prestininzi, Michele La Rocca, and Sauro Succi. 2015. Lattice Boltzmann approach for complex nonequilibrium flows. *Physical Review E* 92, 4 (2015), 043308.

[15] NVIDIA. 2020. NVIDIA V100 Tensor Core GPU datasheet. https://images.nvidia.com/content/technologies/volta/pdf/volta-v100-datasheet-update-us-1165301-r5.pdf.

[16] NVIDIA. 2021. Ampere A100 Tensor Core GPU datasheet. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/a100/pdf/nvidia-a100-datasheet-us-nvidia-1758950-r4-web.pdf.

[17] NVIDIA. 2023. NVIDIA H100 Tensor Core GPU datasheet. https://resources.nvidia.com/en-us-tensor-core/gtc22-whitepaper-hopper.

[18] Sauro Succi. 2008. Lattice Boltzmann across scales: from turbulence to DNA translocation. *The European Physical Journal B* 64, 5 (2008), 471–479.

[19] Sauro Succi. 2015. Lattice boltzmann 2038. *Europhysics Letters* 109, 5 (2015), 50001.

[20] Sauro Succi, Giorgio Amati, Massimo Bernaschi, Giacomo Falcucci, Marco Lauricella, and Andrea Montessori. 2019. Towards Exascale Lattice Boltzmann computing. *Computers & Fluids* 181 (2019), 107–115. https://doi.org/10.1016/j.compfluid.2019.01.005

[21] Matteo Turisini, Giorgio Amati, and Mirko Cestari. 2023. LEONARDO: A Pan-European Pre-Exascale Supercomputer for HPC and AI Applications. arXiv:2307.16885 [cs.DC]

[22] Henry G. Weller, Gavin Tabor, Hrvoje Jasak, and Christer Fureby. 1998. A Tensorial Approach to Computational Continuum Mechanics using Object-Oriented Techniques. *Computers in Physics* 12, 6 (1998), 620–631. https://doi.org/10.1063/1.168744

[23] Samuel Williams, Andrew Waterman, and David Patterson. 2009. Roofline: An Insightful Visual Performance Model for Multicore Architectures. *Commun. ACM* 52, 4 (apr 2009), 65–76. https://doi.org/10.1145/1498765.1498785

[24] Jisheng Zhao. 2023. How to use CHIP-SPV on JLSE. https://github.com/jz10/hip-training.

Received 1 December 2023