BGK2D-Documentation

Giorgio Amati

December 2023

Contents

List of Tables

List of Figures

1   Introduction

Reference document for `BGK2D` open source code.
This code was developed for:

- Exploit Lattice Boltzmann method

- Perform some classical CFD experiment, e.g. Lid driven cavity

- Exploit performance issues

    - Single-core performance
    - Compute node performance
    - Performance portability using GPUs

2 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.

2.1 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`.

5

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

2.2 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. subsectionTEST 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.

2.3 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.

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	$1.7976931348623157E+308$	$2.2204460492503131E-016$

- 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.

2.4 LBM Implementations

There are two different LBM implementations to choose from:

- ORIGINAL: classical implementation with two separated routines (Fig. 1).

  - 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. 2).
  Using pointers populations are swapped in a flip/flop way every time step.
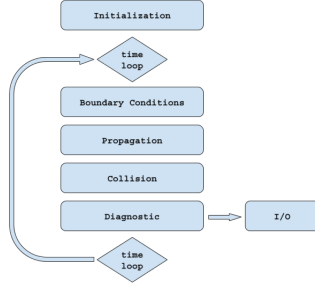 Respect ORIGINAL version the total Bandwidth is halved.

Figure 1: Flowchart for the ORIGINAL implementation
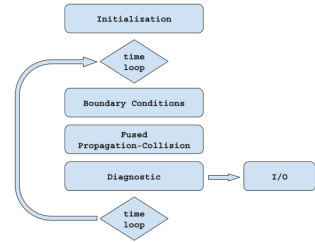
Figure 2: Flowchart for the FUSED implementation

2.5 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 [8].

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

- INTEL Data Center GPU Max 1100 @ 48 GB

---

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

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	$\simeq 1600$	$\simeq 11200$	$\simeq 22400$
INtel MAX 1100	$\simeq 1300$	$\simeq 9100$	$\simeq 18200$

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 [6, 4, 5, 1]

2.6 Licence

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

3 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

- `DOC`: In this directory some documentation

- `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.

4 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 does not need to be recompiled if you need to vary the size of the simulation box.

5 Code 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. All these scripts are intended to be used interactively.

5.1 Poiseuille Flow

The plane Poiseuille flow (POF) is a laminar, stationary flow in a channel between two infinite no-sleep walls. The streamwise velocity follows a parabolic profile

$$u(y) = 4u_0(y - y^2) \tag{1}$$

where u_0 is the centerline velocity, y is the normal to wall direction and $y = 0$ and $y = 1$ are the no-slip wall location. In the code, we impose periodic B.C. along the x direction and no-slip B.C. along the y direction. A volume force along the x direction is imposed. The flow is at rest at $t = 0$. As shown in the left picture of fig. 3, the velocity profile is reached for a single precision simulation. In the right picture, we can see for different Re numbers different timestep to reach the stationary tate (eq. 1).
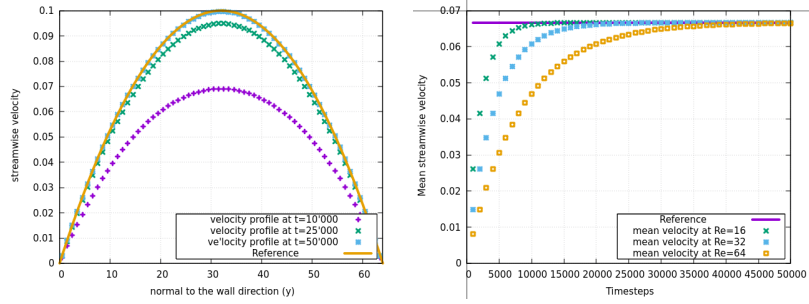
Figure 3: Poiseuille flow. Left: mean streamwise velocity $< u >$ for different Re Numbers. Right: velocity profile $u(y)$ at three different timesteps, at $Re = 64$

5.2 Taylor-Green Vortex

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

$$u(x, y) = A \cos(ax) \sin(by) \tag{2}$$

$$v(x, y) = B \sin(ax) \cos(by) \tag{3}$$

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:

$$u(t, x, y) = (\cos(x) \sin(y)) e^{t/\tau} \tag{4}$$

$$v(t, x, y) = -(\sin(x) \cos(y)) e^{t/\tau} \tag{5}$$

where

$$\frac{1}{\tau} = \frac{2\pi\nu}{(h/2)^2} \tag{6}$$

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.

$$E(t) = E(t = 0) e^{t/\tau} \tag{7}$$

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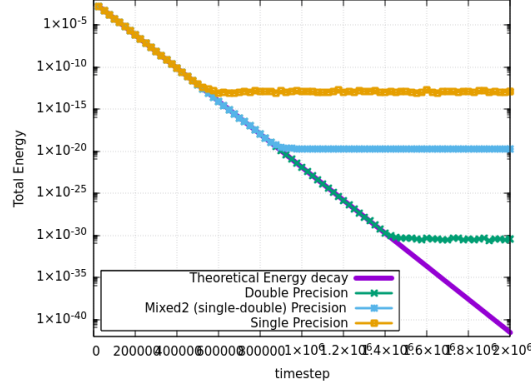
Figure 4: Taylor Green Flow: energy decay for different precision

5.3 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 [2]. 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.
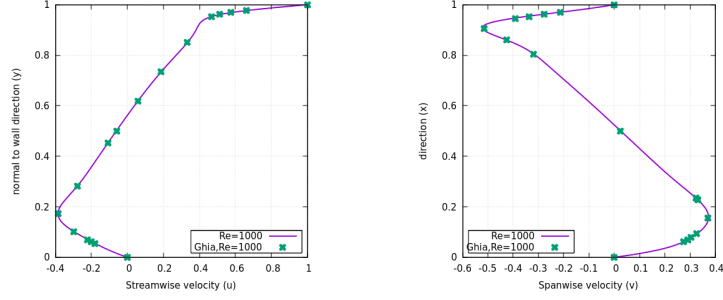
Figure 6: Driven Cavity at $Re = 1000$ compared with Ghia's data for a single precision simulation. Up: streamwise velocity u as a function of y direction. Down: spanwise velocity v as a function of x direction.
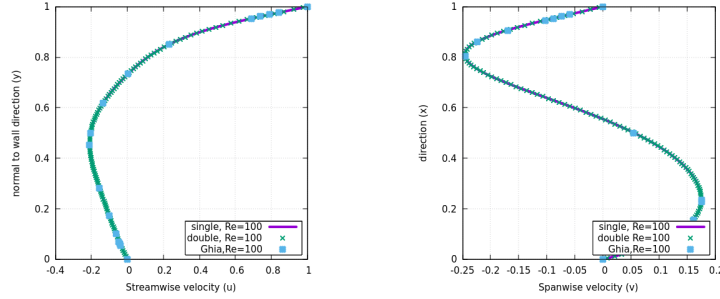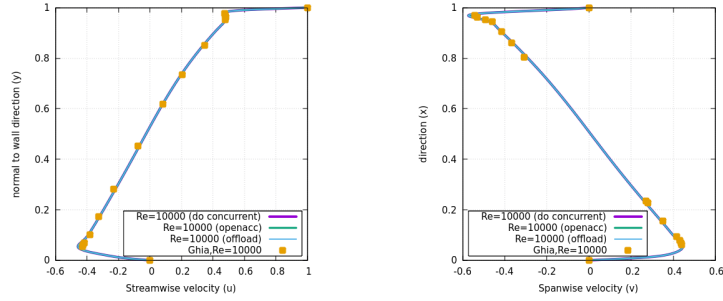

Figure 7: Driven Cavity at $Re = 10000$ compared with Ghia's data using all three available parallel models for GPU. Up: streamwise velocity u as a function of y direction. Down: spanwise velocity v as a function of x direction.

5.4   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` [7] 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.

13
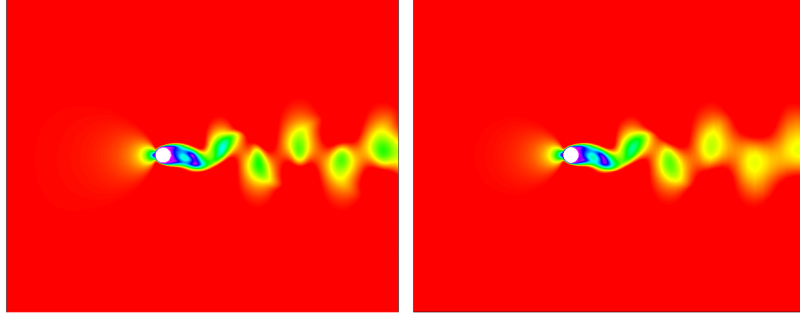
Figure 8: Von Karman Street: velocity field for $Re = 100$ a at $t \simeq 500$, Up: BGK2d, Down: OpenFoam
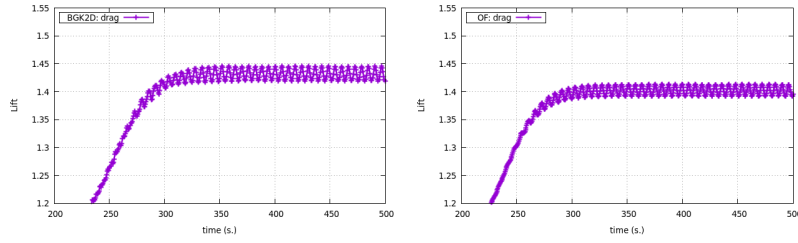
Figure 9: Von Karman Street: Drag history for $R = 100$, Up: BGK2D, Down: OpenFoam

6 Performance Figures

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

- CPU (Serial, single core): `bgk.core.input`
  - Size= 512^2
  - Re=100
  - Single precision
  - Timestep=100'000

- 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

14
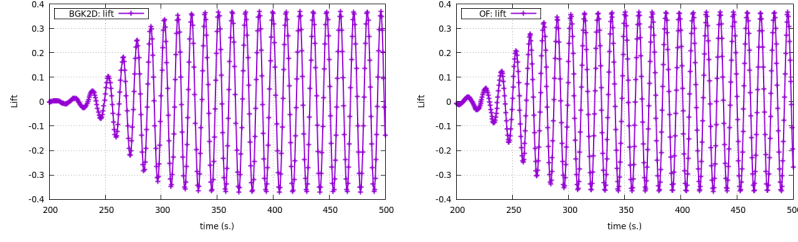
Figure 10: Von Karman Street: Lift history for $R = 100$, Up: BGK2D, Down: OpenFoam

6.1 CPU Serial performance

For the single-core performance the following HW is used:

- **Intel1**: Intel Xeon Platinum 8358 CPU @ 2.60GHz[3]

  - nvfortran rel. 23.5
  - ifx rel. 23.0.0
  - gfortran 11.3.0

- **Intel2**: Intel Xeon Platinum 8260 CPU @ 2.40GHz[4]

  - nvfortran rel.22.3
  - ifx rel. 23.2.0
  - gfortran 10.2.0

- **Intel3**: Intel Xeon Platinum 8480+ CPU @ 2.00GHz[5]

  - ifx rel. 23.2.0
  - gfortran 11.3.1

- **AMD1** AMD EPYC 7742 64-Core Processor[6]

  - nvfortran rel. 22.x
  - gfortran 9.4.0

- **AMD2**: 1x AMD Ryzen 5 5625U [7]

  - nvfortran rel. 22.11
  - gfortran 11.4.0

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

In tab. 5 performance, in terms of MLUPS is reported for the tested HW

---

[3]Cluster leonardo.cineca.it

[4]Cluster g100.cineca.it

[5]inteldev.primo.cineca.it

[6]Cluster dgx.cineca.it

[7]laptop

Table 5: Single core performance

	Compiler	version	mlups	coll (%)	move (%)	b.c (%)
Intel1	nvfortran	original	194	52	47	1
Intel1	nvfortran	fused	423	98	-	1
Intel1	ifx	original	212	56	43	1
Intel1	ifx	fused	335	98	-	2
Intel1	GNU	original	103	79	20	1
Intel1	GNU	fused	85	99	-	1
Intel2	nvfortran	original	151	55	44	1
Intel2	nvfortran	fused	270	98	-	2
Intel2	ifx	original	124	65	34	1
Intel2	ifx	fused	176	99	-	1
Intel2	GNU	original	68	77	22	1
Intel2	GNU	fused	64	99	-	1
Intel3	ifx	original	229	52	47	1
Intel3	ifx	fused	421	99	-	1
Intel3	GNU	original	XX	77	22	1
Intel3	GNU	fused	130	99	-	1
AMD1	nvfortran	original	206	62	36	2
AMD1	nvfortran	fused	314	98	-	2
AMD1	GNU	original	-	-	-	-
AMD1	GNU	fused	75	99	-	1
AMD2	GNU	original	58	82	17	1
AMD2	GNU	fused	121	99	-	1

6.2 CPU multicore Performance

In this section, performance figures and features of different Compute Nodes are presented. `DO CONCURRENT` multicore parallelization is used.

Table 6 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[8]

    – nvfortran rel. 23.5
    – ifx rel. 2023.0.0

- **Intel2**: 2xIntel Xeon Platinum 8260 CPU @ 2.40GHz[9]

    – nvfortran rel.22.3
    – ifx rel. 2023.2.0

- **Intel3**: Intel Xeon Platinum 8480+ CPU @ 2.00GHz[10]

    – ifx rel. 23.2.0
    – gfortran 11.3.1

- **AMD1**: 2x AMD EPYC 7742 64-Core Processor[11]

    – nvfortran rel.22.3

- **AMD2**: 1x AMD Ryzen 5 5625U [12]

    – nvfortran rel. 22.11
    – gfortran 11.4.0

6.3 GPU Performance

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

---

[8]leonardo.cineca.it
[9]g100.cineca.it
[10]inteldev.primo.cineca.it
[11]dgx.cineca.it
[12]laptop

Table 6: Compute Node performance, only fused version and single precision figures are reported

	Compiler	version	mlups	collision (%)	b.c (%)	threads
Intel1	nvfortran	fused	1054	99	1	22
Intel1	ifx	fused	930	99	1	26
Intel2	nvfortran	fused	7362	93	7	48
Intel2	ifx	fused	1216	97	3	48
Intel3	gnu	fused	-	-	-	-
Intel3	ifx	fused	3217	-	-	96
AMD1	nvfortran	fused	2248	79	21	-
AMD1	gnu	fused	-	-	-	-
AMD2	gnu	fused	?	-	-	6
AMD2	nvfortran	fused	898	97	3	12

- nvfortran rel.23.11

- **PTV-1**: Intel MAX 1100 @ 48GB

  - ifx rel.23.2.0

- **MI250**: AMD MI250X, half (Adastra)

  - ftn rel.16.0.0
  - flang rel.15.0.0

Table 7 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).

Table 7: 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%
PTV-1	ifx	do concurrent	¿ 1000	99	1	-%
PTV-1	ifx	offload	4497/8165	99	1	-%
MI250X (half)	ftn	offload	12880	96	4	58%
MI250X (half)	ftn	openACC	11711	97	3	53%
MI250X (half)	flang	offload	4858	96	4	20%

6.4 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 [3]. 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.

6.5   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.

7   Compiler options

7.1   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`

7.2 Compiler options for multicore version

- GNU: `-fopenmp -ftree-parallelize-loops=4`

- INTEL: `-qopenmp`

- NVIDIA:`-stdpar=multicore`

- AMD: `-h thread_do_concurrent`

7.3 Compiler options for GPU version

- GNU

  - Do concurrent: ????
  - Offload:????
  - OpenACC:????

- INTEL

  - Do concurrent: `-fiopenmp -fopenmp -fopenmp-target-do-concurrent -fopenmp-targets=spir64`
  - Offload: `-fiopenmp -fopenmp-targets=spir64_gen -Xopenmp-target-backend "-device *"`
  - OpenACC: not supported

- 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`