Astronomy & Astrophysics manuscript no. output
May 11, 2026
©ESO 2026

CUDA implementation of Mancha3D code

M. Modestov[1,2], I. Alonso Asensio[1,2], and E. Khomenko[1,2]

[1] Instituto de Astrofísica de Canarias, 38205 La Laguna, Tenerife, Spain
[2] Departamento de Astrofísica, Universidad de La Laguna, 38205, La Laguna, Tenerife, Spain

Submitted

ABSTRACT

The Mancha3D code is a versatile tool for numerical simulations of magnetohydrodynamic (MHD) processes in solar and stellar atmospheres. This paper describes the implementation of a CUDA version of the code, which enables it to run efficiently on modern supercomputers with GPUs. We carefully studied the numerical performance, with detailed comparisons to the CPU version, together with strong and weak scaling of the code. Low-level CUDA optimizations allowed us to reach an astonishing 600-fold higher performance with respect to the CPU version. Weak scaling, along with tuned communications between GPUs, enables the yield of more than 85% of the performance for up to 400 GPUs.

Key words. MHD code, GPU, CUDA Fortran, HPC

1. Introduction

The Mancha3D is a well-established code for realistic magnetohydrodynamic (MHD) simulations of the solar and stellar convection, as well as fundamental studies of waves, instabilities, prominence dynamics, and other phenomena in the solar atmosphere (Modestov et al. 2024). The code has been evolving since 2006 and it currently includes nonideal physics (ohmic diffusion and the Hall effect), as well as ambipolar diffusion derived from plasma partial ionization, along with a realistic equation of state, optically thick radiative losses by solving a radiative transfer equation (RTE), and thermal conduction. Splitting variables into equilibrium and perturbed components is one of the code's features and pairing it with a perfectly matched layer (PML) makes it possible to carry out accurate simulations of waves and instabilities.

The code is written in modern Fortran language, fully parallelized with a message passing interface (MPI), using spatial domain decomposition; input and output files are handled via a parallel hierarchical data format (HDF5). The code solves the time-dependent equations of the nonideal MHD on a 3D Cartesian grid and it can solve either full or linearized MHD equations. By default, the governing MHD equations are evolved with a memory-saving variant of the explicit Runge-Kutta scheme of the third order.

The Mancha3D code has a modular structure. It consists of the main time-advancing loop where the solver is employed. The main modules are: artificial diffusion module, including hyperdiffusion and high-frequency filtering; nonideal terms module for computation of the ohmic and Hall terms, as well as the ambipolar terms; an equation of state (EoS) module, which interpolates secondary variables from an externally precomputed table; and thermal conduction and RTE modules. The computation of nonideal terms, under certain conditions of magnetization (see Khomenko et al. 2014), requires modifications to be made to the main Runge-Kutta integration by using a super time stepping (STS) approach (for the ambipolar and ohmic terms) or a Hall diffusion scheme (HDS; for the Hall term), as described in (González-Morales et al. 2018). For high-temperature regions, thermal conduction often implies serious limitations on the time step, which can be overcome by using an additional hyperbolic equation for the heat flux (Navarro et al. 2022). The RTE module requires a specific parallelization since the integration of the RTE is done along several rays using the short characteristic method; it requires iterations at the boundaries of subdomains. Taking into account all the physics and techniques the code uses from 120 to nearly 200 variables per grid cell, depending on included physics, which means about 1.5GB of memory usage per 100^3 grid cells. Despite this complexity and high memory demand, the code demonstrates a good level of performance and excellent scaling characteristics on CPUs (Modestov et al. 2024).

In the next stage of the code evolution, the aim is to follow the modern computational trend and expand the code to perform high-quality simulations using graphics processing units (GPUs). There are several conventional approaches to GPU-target computing, such as language and compiler extensions such as CUDA and OpenCL/FortranCL, directive-based standards such as OpenMP and OpenACC, high-level interfaces such as SYCL, as well as programming models, for instance, KOKKOS (Bispo et al. 2021; Trott et al. 2022).

Wong et al. were the first to use CUDA for ideal MHD simulations. With a simplified finite volume-like solver, they achieved 160 speedup on a single GPU with respect to the CPU core for a 3D problem (128^3 grid cells) and double precision, demonstrating the great potential of CUDA and GPUs in future simulation studies (Wong et al. 2011). In the Asteroth code, a significant amount of effort was put into the effective implementation of high-order finite difference method (Pekkilä et al. 2017). Subsequently, Gyenge et al. revealed one of the problems with multi-GPU calculations that the actual computation would take as little as 1%

Article published by EDP Sciences, to be cited as https://doi.org/10.1051/0004-6361/202558286

of the overall time, while the rest was taken by communication between GPUs (Gyenge et al. 2018). In recent years, this issue has almost been resolved by both technological progress and improvements to compilers.

Nonetheless, CUDA offers two essential disadvantages: it supports only NVIDIA's GPUs and implementing it for an existing code is time-consuming, as it actually requires rewriting most of the code. As an alternative, we can use the pragma-based OpenACC: it enables GPU parallelization with small efforts and can be used on GPUs from different vendors. It might be beneficial for big codes, but as it is a higher level approach, it cannot, in principle, provide superior performance as compared to the low-level CUDA. For example, recent upgrades of the LEONARDO and the MURaM codes utilize OpenACC and demonstrate a very nice weak scaling, although speedup with respect to the CPU version is around $30 - 100$ (Del Zanna et al. 2024; Wright et al. 2021). Finally, several codes used a KOKKOS system (e.g., Athena, Diablo, IDEFIX, (Stone et al. 2024; Delorme et al. 2025; Lesur et al. 2023)) and tested the codes on GPUs and CPUs of different vendors. They all showed a similar level of performance gain to the OpenACC codes.

Taking into account all the advantages and disadvantages of the existing methods, we decided to use CUDA for Mancha3D code. As a low-level language extension, it allows for full control of the memory allocation and usage, as well as an optimal configuration for kernels, which should result in a better performance (Caddy & Schneider 2024). As the original Mancha3D is written in Fortran, in combination with C-based OpenCL, it might show losses in terms of performance. Using the OpenACC approach would pollute the code with numerous directives with limited control and a lack of fine-tuned microparallelization. One of the drawbacks of CUDA, being limited only to NVIDIA's GPUs, can now be overcome with the help of the HIPify tool, which transforms CUDA source code into HIP, enabling it to be run on AMD GPUs (Bispo et al. 2021; Caddy & Schneider 2024). In addition, nowadays most of the supercomputers use NVIDIA's GPUs, so this limitation is of minor importance. At the same time, we would like to keep the CPU code version as unaltered as possible so that it can be adequately run on platforms with CPUs.

2. Several features of CUDA implementation in Mancha3D

One of the challenges faced in this work has been to keep both CPU and GPU versions within the same code. First of all, it is needed for back-compatibility with previous simulations and samples developed for the code validation. It simplifies comparison between versions and ensures that they produce similar results. For this reasoning the GPU version has been implemented as separate modules with CUDA kernels and governing subroutines. It is switched on and off with a single preprocessor directive in a control file.

Regardless of the architecture and programming language, there is always a CPU core that controls the computational load on a GPU. There are three main models of how it can be organized within a compute node: a single CPU core is a host for a single GPU; a single CPU core works with multiple GPUs; and several CPU cores work with a single GPU. We work in the first paradigm when each GPU has its own host CPU core. This choice unifies communications within the compute node and across different nodes. It also allows us to retain the same domain decomposition and parallelization schemes for CPU and GPU versions.

One of the main factors limiting the performance of a GPU code is frequent data transfer between the host CPU core and its GPU. To exclude this problem, all the arrays used in simulations are allocated and stored in the GPU memory. It is usually significantly smaller than the CPU RAM, so it determines the maximal size of the computational domain. Still, the GPU capacity is expected to increase over time as computer hardware advances, and now there are GPUs with 64 GB or larger, which is enough to handle domains as large as 400^3 grid cells. Therefore, after the initialization stage, the data is transferred between the GPU and the associated CPU core only for saving snapshots, which usually happens once in a large number (sometimes hundreds or thousands) of iterations; thus, it does not significantly affect the code performance. A minor difference between the GPU and the CPU versions of the code consists of the fact that in the GPU version, we used a slightly smaller number of arrays compared to the CPU version, as it is computationally cheaper to recompute some of the data than keep it in a separate array. Exchanges between GPUs are greatly facilitated by the CUDA-aware MPI, which does not explicitly involve host CPUs.

Before comparing the performance of the CPU and the GPU versions, we profile the main algorithmic parts of the code. The Mancha3D code solves nonideal MHD equations, written in conservative form:

$$\frac{\partial \mathbf{u}}{\partial t} = -\nabla \mathbf{F}(\mathbf{u}) + \mathbf{S}(\mathbf{u}) + \left(\frac{\partial \mathbf{u}}{\partial t}\right)_{\rm diff}, \tag{1}$$

where the first term on the right-hand side corresponds to the advection flux, the second one stands for possible sources, and the last one is the artificial diffusion term, needed for stability of the simulations. The diffusion term is defined as

$$\left(\frac{\partial u_j}{\partial t}\right)_{\rm diff} = \sum_i \frac{\partial}{\partial x_i}\left[\nu_i(u_j)\frac{\partial u_j}{\partial x_i}\right], \tag{2}$$

where $\nu_i(u)$ is an artificial diffusivity coefficient, consisting of constant, hyper, and shock counterparts. The equations and numerical techniques are thoroughly described in (Modestov et al. 2024), while here we are interested in the amount of computational work needed for different terms. From Eqs. 1 and 2, we see that advection flux terms involve taking the first derivative, and the diffusion flux term requires multiple calculations of the second derivative. The hyper-diffusion part of the diffusivity term also involves derivative-like calculations.

For our analysis, we use a 3D setup of the Kelvin-Helmholtz instability with 720^3 grid cells. To include the impact of MPI exchanges, we run it on eight GPUs with $2 \times 2 \times 2$ division into subdomains, so that each GPU handles a subdomain of 360^3 grid cells. Eight GPUs constitute two full compute nodes on the MareNostrum 5 machine. For comparison with the CPU version, we decided to also use two full nodes with 160 CPU cores in total, for the same domain (each CPU core handles $180 \times 144 \times 90$ subdomain). The actual number of CPU cores is not very important, as

here we are interested in the portion of time spent in different parts of the code and not in the overall timing. Comparing computational subdomains of the same sizes for the CPU and the GPU versions is not practical, as it is typically necessary to submit a job of a fixed size to a compute node, where it runs either on CPUs or on GPUs.

In Fig. 1, we present the time portions spent in the main algorithmic parts of the code for the two versions. As for real times, for the GPU version (runs on 8 GPUs) it takes about 0.5 seconds per iteration, and for the CPU version (runs on 160 CPU cores), it takes about 14 seconds per iteration, resulting in 560 times GPU-CPU speedup. For the profiling, we use the standard Fortran timing function `cpu_time` for the CPU version and `cudaevent` functions for the GPU version to measure exactly what we needed. Moving counterclockwise from the top, the first two segments correspond to the most time-consuming procedures of computing advection and diffusion flux terms. Surprisingly, they take the same portions for both versions. The first noticeable difference appears for computing artificial diffusivity coefficients, $\nu_i(u)$, shown in yellow in the diagram. It involves numerous operations with large 4D arrays, including computing nonlocal terms, such as hyper-diffusion, with more details available in Sect. 3.4 of (Modestov et al. 2024). For the CPU version, which is cache memory-bound, handling lots of big arrays simultaneously results in subpar performance when executed in parallel with many CPU cores. This was done in a manner that would favor the code readability and allow for easier maintainability. In the CUDA version, the similar calculations are naturally split into multiple consecutive kernels, which eliminates cache memory bound issue of the CPU version. In addition, some variables are recomputed on-the-fly instead of keeping them in separate arrays, reducing memory load, resulting in a more efficient GPU code.

The next important segment is related to the MPI exchanges (light green color), which is more than two times larger for the GPU version than for the CPU one. This difference should be attributed to the different sizes of the subdomains for GPU and CPU versions, as the larger amount of exchanged data per core scales with the subdomain size. Still, the time for MPI exchanges is shorter than 10% of the overall time, which clearly demonstrates that the MPI exchanges are not a limiting factor for the code. Another noticeable difference between the two versions is in the computation of the iteration time step based on the CFL condition (shown in dark green in the diagram). This operation requires searching for maximum/minimum in the entire domain, which is a slow process for the GPU codes. Again, the GPU subdomain is a few times larger than the CPU one. This explains the fourfold difference in the profiling results.

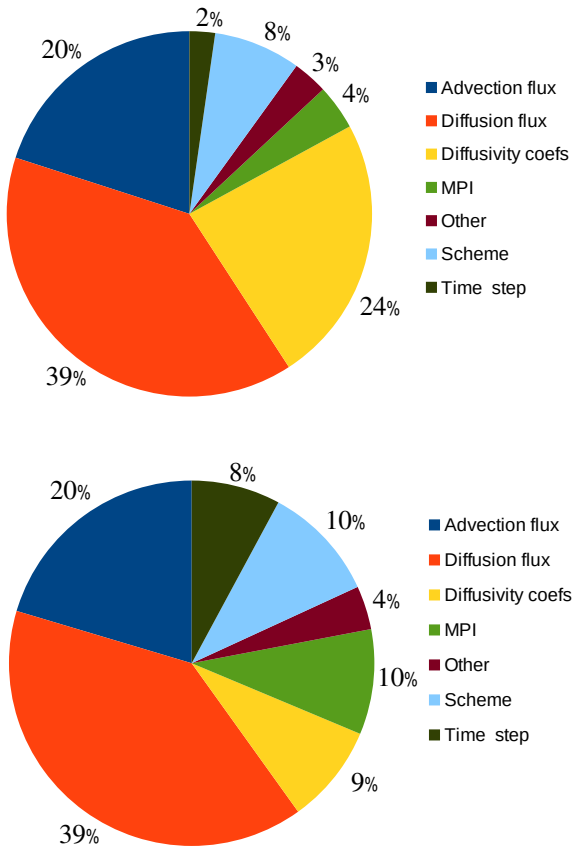

Fig. 1. Profiling for the CPU (top) and GPU (bottom) versions of the code with respect to the main algorithmic parts.

2.1. Hybrid CPU+GPU model

In the section above, we describe our comparison of the efficiency between two versions of the Mancha3D code, which work either on CPU cores or on GPUs. In our study, we also implemented a hybrid version, where a part of the computational domain is treated by GPUs, and another part is treated by CPU cores. This idea comes from the fact that in all supercomputers equipped with GPUs, a compute node consists of several GPUs (usually 2-8) and several dozens of CPU cores (up to 160). For the MareNostrum machine, a single node has 4 GPUs and 80 CPU cores. According to our results (see Section 3 below), a GPU performs about 200-600 times faster than a CPU core; hence the net gain of such a hybrid mode would account for a few percent at most. Such a mixed implementation requires a more complex communication scheme and could still be beneficial for specific problems or other machines with less powerful GPUs and a large number of CPU cores. Nevertheless, the mixed GPU/CPU implementation was beneficial from an educational point of view and in terms of comparison of the two versions. In the following, we focus on the pure GPU setup.

2.2. Optimal kernel configuration

A CUDA kernel is the main programming unit acting as a subroutine or a function, which is executed on a GPU by a large number of threads. Usually, one thread processes one element (i, j, k); for convenience, threads are grouped into 1D, 2D, or 3D blocks depending on the programmer's needs. In all modern GPUs, the number of threads per block is limited to 1024. Blocks are further grouped in 1D, 2D or 3D grids; group sizes are also limited, although the total number can be as large as $2^{63} \approx 9.2 \cdot 10^{18}$, which exceeds any GPU memory by several orders of magnitude. The optimal number of threads per block depends on the specific application and the hardware. A balance must be struck between having enough threads to saturate the hardware and hide latency.

In CUDA, it is a programmer's responsibility to determine the thread blocks and the block grids. In Mancha3D we do this in the following way:

```
block1 = dim3(nb1,nb2,1)
grid1 = dim3(CEILING(mx/nb1),my,CEILING(mz/nb2)).
```

Here, `(mx,my,mz)` are the numbers of grid points in the subdomain received by one core, while `nb1,nb2` are the block sizes. The numbers composing the grid array ensure that threads cover the entire computational domain. In most of the kernels (except for those computing Y derivative and similar), the indexes are built as

```
i = (blockIdx%x-1)blockDim%x + threadIdx%x
j = blockIdx%y
k = (blockIdx%z-1)blockDim%y + threadIdx%y.
```

Here, `blockId`, `blockDim`, `threadIdx` are CUDA reserved variables, related to the kernel configuration. It implies that within a thread block, only two indexes vary. This choice is made to unify the 2D and 3D setups in a simple way.

Most of the kernels used in the code can be divided into two types: local and nonlocal. A local kernel implies computation that requires the elements of arrays only from a single grid point. For example, computing temperature and pressure from internal energy and density is a local kernel. The implementation of local kernels is straightforward and resembles standard Fortran code written by elements. Nonlocal kernels use data from neighboring points; a typical example is computing derivatives. The effective calculation of derivative with CUDA kernels was carefully studied in the Asteroth code, (Pekkilä et al. 2017). Access speed to the neighboring elements depends on direction within the array and whether it corresponds to the contiguous location or not. Starting from their original purpose, GPUs are much more efficient for local computing tasks than for nonlocal ones. For this reason, we have paid special attention to the effective performance of kernels with nonlocal computations.

The execution time of a kernel on a GPU depends on several factors, including the number of threads per block and the block configuration mentioned above. In Table 1 we present timing results for 34 kernel configurations (varying `nb1,nb2` numbers) for five different kernels. The first column (`nb1`) always refers to the contiguous dimension of an array, which, for Fortran, is the first index, X-direction. The second column usually stands for the third index or Z-direction, except for the case of the Y-derivative, where it stands for the second index. Kernels for derivatives in X (`derx`) and Y (`dery`) use a simple algorithm, while for Z-derivative (`derz`), the shared memory algorithm is used, as described in (Bispo et al. 2021). The `small` kernel stands for a local kernel performing one or two operations with a small number of variables. Such kernels are very common and used, for example, for preparing data for derivative operations, or computing logarithmic density from the linear one, and many others. Roughly speaking, the main part of a `small` kernel has 1-5 lines, while for a `big` kernel it can be 20-30 or more lines. Large kernels (`big`) handle heavier calculations with multiple variables and require more memory registers for execution, while the number of registers is limited per thread block. To be able to run such kernels, it is necessary to reduce the number of threads per block and use different kernel configurations as compared to the small kernel one. For this reason, there are no data at the end of the last column of the table. A `big` kernel simply does not run when there are not enough resources available due to too many threads per block reduces the number of registers available to each thread.

Table 1. Timing (ms) for 5 different kernels for different configurations.

nb1	nb2	derx	dery	derz	small	big
1	64	20.0	14.41	17.41	17.97	195.92
2	32	11.3	7.64	10.66	8.65	113.54
4	16	7.37	6.29	7.32	8.05	97.21
8	8	6.2	5.73	6.57	7.72	91.89
16	4	5.71	5.58	6.16	7.32	86.25
32	2	5.64	5.77	5.53	6.5	82.53
64	1	5.63	5.91	5.82	5.94	**80.48**
1	128	21.67	14.37	17.9	18.13	201.81
2	64	12.12	7.56	11.63	8.71	112.12
4	32	7.53	6.18	7.02	8.12	98.86
8	16	6.49	5.61	7.01	7.73	93.87
16	8	5.85	5.52	6.36	7.24	88.33
32	4	5.69	**5.48**	6.08	6.43	84.66
64	2	5.64	5.69	**5.51**	5.88	82.56
128	1	5.61	5.87	5.84	5.69	80.569
1	256	21.84	14.15	18.28	17.59	186.1
2	128	12.4	7.59	12.17	8.67	133.66
4	64	7.79	6.18	7.16	8.53	124.32
8	32	6.46	5.71	6.92	7.91	115.38
16	16	5.94	5.62	6.97	7.21	108.43
32	8	5.76	5.56	6.37	6.4	104.48
64	4	5.67	5.53	6.14	5.83	101.97
128	2	5.6	5.73	5.65	5.67	99.68
256	1	5.59	5.93	6.16	**5.645**	98.2
1	512	21.78	14.16	18.06	16.75	
2	256	12.7	7.43	12.1	9.15	
4	128	7.66	6.33	7.26	8.79	
8	64	6.53	6.01	6.97	8.26	
16	32	6.06	5.89	6.97	7.31	
32	16	5.89	5.8	7.04	6.29	
64	8	5.79	5.74	6.47	5.87	
128	4	5.68	5.73	6.24	5.71	
256	2	5.59	5.95	6.2	5.68	
512	1	**5.55**	6.01	6.33	5.66	

Notes. `nb1` and `nb2` define the kernel configuration; `derx`, `dery`, `derz` are the kernels for computing derivatives in X, Y, and Z, respectively; and `small` and `big` designate kernels with small and large amounts of computations. Four horizontal sections stand for different numbers of threads in a block, namely $64, 128, 256, 512$. The optimal configurations (minimal time) are printed in bold.

We ran multiple tests with different numbers of threads per block and different configurations. For convenience, we marked optimal configurations for every kernel in bold and grouped all the configurations into four horizontal sections based on the number of threads in a block, (`nb1` $\cdot$ `nb2`). Several interesting observations can be made from this table. We see that the execution time of a specific kernel differs only slightly for various configurations; however, the ratio between the slowest and the fastest cases often exceeds 3. It is interesting to note that five different kernels have optimal configurations with four different numbers of threads

per block. All the kernels have their optimal configurations with large `nb1` and small `nb2` numbers, indicating the importance of the contiguousness of the data. Such a situation is naturally expected for local kernels and for X-derivative (see columns `derx`, `small`, and `big` in Table 1). However, for Y- and Z-derivatives (columns `dery`, `derz`), we might have expected better results with large variation along the second index (larger `nb2` values). It is even more expected when the shared memory is used (kernel `derz`), as several elements are being reused. Nevertheless, for nonlocal kernels, the effect of faster access to contiguous elements is stronger and almost overwhelms the effect of reusing elements from noncontiguous directions.

In Table 1, we do not show the results for 32 and 1024 number of threads per block, as for the current setup, they do not provide optimal results. It should be remembered that these optimal configurations depend on a particular machine, on the size of the computational domain, and the resulting performance will vary for different subdivisions. In Mancha3D such optimization tests run automatically during the initialization stage every time the code is launched. In total, we tested 14 different kernels with 40 possible configurations with several block sizes. It takes a few extra seconds, and at the end we have 8 different configurations used in the code for different kernels, which guarantee optimal kernel performance regardless of the machine and the size and shape of the computational domain.

In addition to the previous profiling based on the main modules of the code, in Fig. 2, we provide one more profiling chart, with respect to different types of kernels of the GPU code version. It can be considered as a technical zooming into `Advection flux`, `Diffusion flux`, and partially `Diffusivity coefs` segments depicted in Fig. 1. Here, `Diffusion_local` and `Advection_local` refer to local kernels within corresponding physical modules, while derivative kernels are used in both advection and diffusion modules. Fig. 2 demonstrates that the nonlocal kernels computing derivatives take half of the time in computing diffusion and advective terms. In the CPU version, it is difficult to build a similar profiling as multiple computations are combined due to Fortran's ability to handle array variables. On the contrary, in the GPU version, the same calculations are naturally broken down into several kernels, which makes such profiling straightforward. While, in principle, it might be possible to follow the same strategy for the CPU version, it would require additional arrays; in turn, this would increase the load for memory usage and further slow down the CPU code, leading to an unfair comparison to the GPU version. Still, we measured the portion of time spent on derivatives in the CPU version, and it takes about $7-8\%$ of the overall time, while the rest is taken by local procedures. This is because local calculations on GPUs are considerably faster than those on CPUs, while nonlocal ones are relatively slower. This distinction stems from the different organization of the memory access in the two architectures. In Fig. 2, we also split derivatives by direction to show that with the help of a proper choice for the kernel configuration, the computation of different derivatives takes a similar portion of time. On our first attempts, computations of Y- and Z-derivatives were two to three times slower than the X-derivative due to a wrong assumption on optimal kernel configurations.

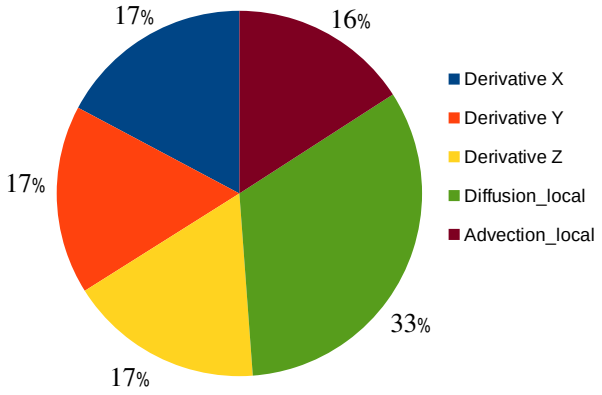

Fig. 2. Profiling of the GPU version for different types of kernels.

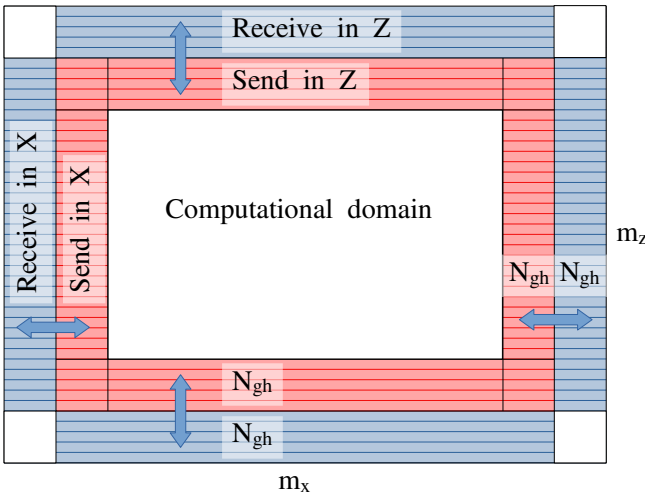

Fig. 3. Schematic of a computational domain with the ghost cells to exchange. Horizontal lines stand for the direction of contiguous elements in Fortran arrays.

2.3. Exchanges between GPUs

In brief, Mancha3D is fully parallelized with the MPI standards for distributed memory machines. The computational domain can be decomposed into subdomains in all three directions. In (Modestov et al. 2024), we explored the parallel efficiency of the CPU version of the code and did not see any noticeable difference in the code performance with respect to decomposition direction. However, while testing various options for the GPU version, we observed a significant difference in exchange times. In particular, when several GPUs are used in X dimension, the communication time is about 50 times longer than in the case when decomposition is applied in Y or Z dimensions.

To get a better insight into the problem, we sketched a 2D subdomain in X-Z coordinates, which can also be viewed as a Y cross-section of a 3D subdomain, see Fig 3. During exchange, the outer light blue parts are received from the neighbors, and the inner pink parts are sent to the neighbors. For simplicity, we considered the case of exchanging a 3D array without corners with five ghost cells. In the CPU version, the data transfer is managed with the help of pre-

defined MPI derived types and sending a 3D data in the X direction to the left neighbor is done with

```
CALL MPI_ISEND(var,1,mpitype_left,rank_left, &
               tag,exchange_comm,mpi_reqs,error),
```

where `var[mx,my,mz]` is a variable array to send and `mpitype_left` is the predefined MPI type, determining what part of the array should be sent to the neighbor.

The CUDA-aware MPI facilitates communication between different GPUs, so that exactly the same procedure can be performed with arrays allocated in the GPU memory. It works fine for exchanges in Y and Z directions, but for the X direction, such an exchange is extremely slow. For exchanges in the Z dimension, a large chunk of the elements are contiguous and there are only five so-called jumps between noncontiguous parts of the array, see 5 long horizontal lines in Fig. 3. On the contrary, when exchanging data in X dimension, only a small number of elements are contiguous, split by a big chunk of noncontiguous elements.

To speed up communications between different GPUs in the X dimension, we used the following three-stage procedure. Prior to the exchange, the necessary elements are first copied into pre-allocated small 3D arrays, separate for each direction. This can be implemented as a separate kernel and it takes a negligible amount of time to fulfill. Consequently, we have a small 3D contiguous array, `var1[Ngh,ny,nz]`, where `Ngh=5` is the number of ghost cells required by the discretization schemes. This array is sent to the corresponding neighbor and as all the data elements are contiguous, the communication time is reduced significantly. In addition, It does not require any MPI-derived types. We can perform this procedure via

```
CALL MPI_ISEND(var1,nvs,MPI_DOUBLE_PRECISION, &
               rank_left,tag,exchange_comm, &
               mpi_reqs,error),
```

where `nvs` is the number of elements in this array. Afterward, the received elements are copied to the main array with another kernel. Despite being a more complicated conceptually, it speeds up communication in X direction to the same level as communication in Y and Z directions. Moreover, a similar procedure can be applied for exchanges in the Y and Z directions and speed them up as well.

In Mancha3D we usually exchange 4D arrays containing all the variables. It increases the role of such a procedure for exchanges in the Y and Z directions. We demonstrate this in Fig. 7 by adding the weak scaling results with nonoptimized exchanges using MPI derived types as in the CPU version. Thus, for efficient GPU code, it is crucially important to have proper control on the GPU memory access, whether that is meant for kernel execution or for exchanges between different GPUs.

3. Results

All the simulations are performed on the GPU partition of the MareNostrum 5 supercomputer. It comprises 1,120 nodes based on Intel Xeon Sapphire Rapids processors and NVIDIA Hopper GPUs. Each node is equipped with the following:

2x Intel Xeon Platinum 8460Y+ 40C 2.3GHz (80 cores per node);
4x NVIDIA Hopper H100 64GB HBM2;
16x DIMM 32GB 4800MHz DDR5 (512GB main memory per node, 6.25GB usable per core);
480GB NVMe local storage;
4x ConnectX-7 NDR200 InfiniBand (800Gb/s bandwidth per node);

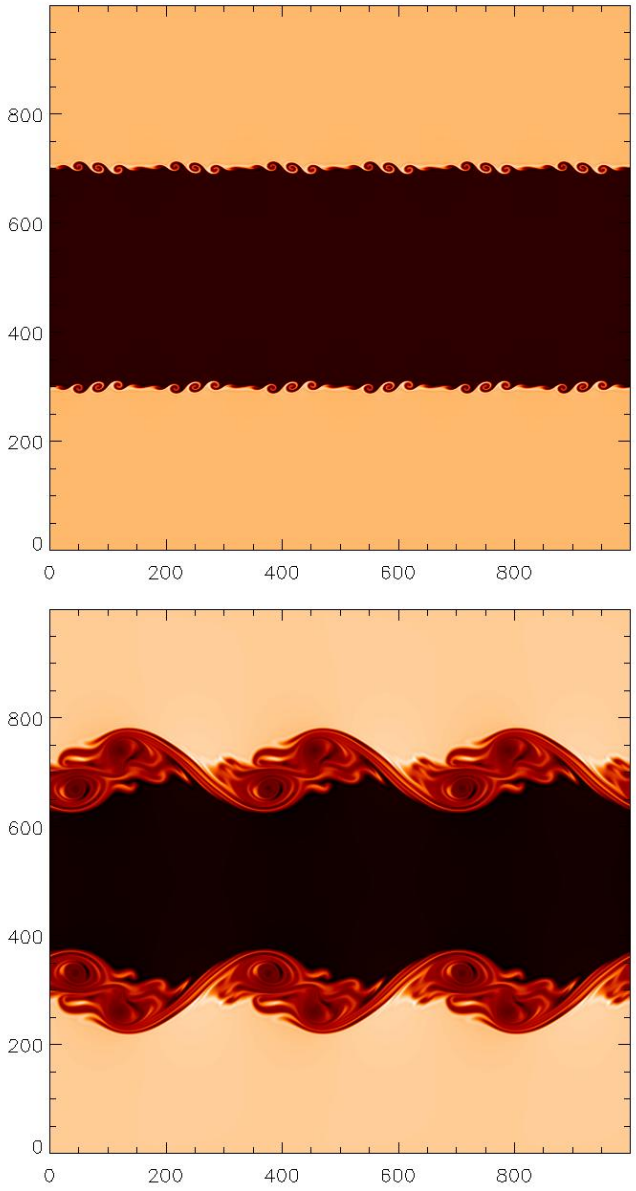

Fig. 4. Temperature after 8000 (top) and 50000 (bottom) iterations. The inner part of the jet corresponds to 10^4K, the outer plasma is at $9 \cdot 10^3$K.

The compiler is `nvfortran 23.11-0` and we use standard optimization flags:
for CPU: `nvfortran -Ofast`;
for GPU: `nvfortran -cuda -Ofast`.

As a test sample, we used a 2D/3D Kelvin-Helmholtz instability setup, with a jet in the middle of the domain. The density of the jet differs by 10% from the ambient background density. Initially, pressure is set uniformly in the entire domain and the temperature is recomputed from the ideal gas equation. To launch the instability, we set small perturbations of vertical velocity at the interface of

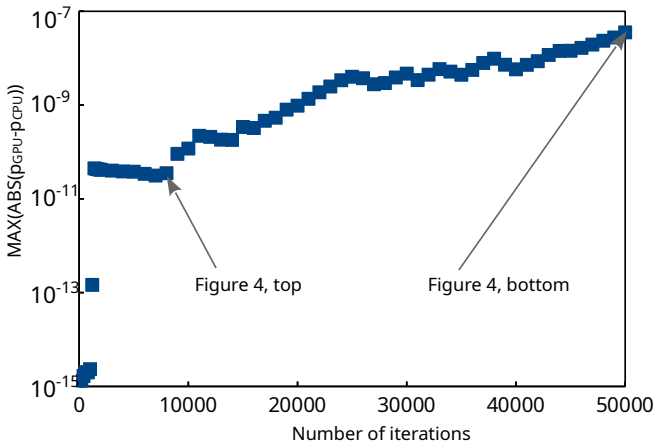

Fig. 5. Maximal absolute difference between pressures computed by the GPU and CPU versions.

the jet. We have periodic boundary conditions set up in X and simple outflow in Z (and Y in the 3D case). This setup is easily scalable in both 2D and 3D.

First, we checked that our GPU version produces similar results to the CPU one. For that purpose, we used a 2D KHI setup with $[1000 \times 1000]$ domain. In Fig. 4, we show the temperature field at the beginning and at highly nonlinear stages of the instability. The difference between the CPU and the GPU versions is indistinguishable by eye, although it is confirmed to be nonzero. We cannot expect bit-to-identity between the two versions; even though they use the same compiler, they are computed on different physical computational units that are organized differently. We demonstrate a maximal absolute difference between pressures computed by the CPU and GPU versions versus the number of iterations (see Fig. 5). The pressure is chosen because the initial one is close to unity ($P_0 \approx 0.83$); relative differences of other variables behave in a similar manner. During the first iterations, the difference is in the numerical accuracy of double precision calculations ($\approx 10^{-15}$). At the later stage, the difference grows slowly; however, even at a highly nonlinear stage shown in the bottom panel of Fig. 4, it remains below 10^{-7}. In Mancha3D we use compound artificial diffusion coefficients including hyper diffusivity and shock diffusivity. Tiny numerical differences in computing these terms result in a jump around 1200 iterations when there are multiple small shocks traveling in the domain. In a test without diffusivities, that difference stays at the level of $3 \cdot 10^{-15}$.

3.1. GPU versus CPU performance

Achieving high performance on a single GPU is the key point before moving on to simulations with multiple GPUs. Due to technological and conceptual differences between GPU and CPU architectures, it is difficult to make a fair comparison. From the technological point of view, the most correct comparison should include the so-called streaming multiprocessor (SM) of NVIDIA GPU (or compute unit of AMD GPU), as it can be considered as an analogue of a CPU core. For example, the NVIDIA H100 used in this work has 132 SMs, and each SM launches multiple threads on 128 CUDA cores. However, the user does not control the number of SMs used by a code, nor is it guaranteed that all the SMs will be used when the code is running on a GPU. Most often, code performances are compared based on a single-CPU-core-single-GPU setup, (Wong et al. 2011; Gyenge et al. 2018; Wright et al. 2021; Caddy & Schneider 2024). This is not really ideal, as a single GPU has hundreds or thousands of arithmetic units while a CPU core has one or very few arithmetic units. On the other hand, it corresponds to the minimal computational unit that can be used to run a code and that does not require user-controllable communications, as when including an MPI. Alternatively, we can compare code performances between devices, such as (Stone et al. 2024), including all the cores of a CPU, which can reach 128 in modern supercomputers. It is fairer in terms of computational resources, but for the CPU case, such a setup involves communication between different cores, which is missing in the GPU case.

In this work, we used the second option and computed the code performance as the number of grid cell updates per core per second via

$$P = \frac{M_x M_y M_z N_{\rm iter}}{N_{\rm core} t}. \quad (3)$$

Here M_x, M_y, M_z are the numbers of grid cells in the X, Y, Z dimensions, respectively. These numbers refer to the computational domain, which is usually splitted between several CPUs or GPUs; they do not include the ghost cells, so that the total number of grid cells used in the code is slightly larger[1]. $N_{\rm iter}$ stands for the total number of time steps; each time step includes computation of diffusivities (which takes a significant amount of time) and three Runge-Kutta substeps. The time, t, is measured as the physical elapsed time (in seconds) minus the time spent on the initialization stage and $N_{\rm cores}$ represents the number of GPUs or CPU cores used in a run. Hence, the metric (3) is very useful for comparing code performance regardless of the actual size of the computational domain, number of cores, and number of iterations. For example, for weak scaling, the number of grid cells per core is fixed, and we use $N_{iter} = 1000$ for all the runs. For strong scaling, the number of grid cells per core varies, and we vary the number of iterations from 1000 to 50000, to keep the elapsed time around 10 minutes. This metric is ideal for comparing two versions of the same code, as the same number of calculations is performed. Comparison with other codes should be done with care, as different codes have different amounts of calculations per grid cell.

In Fig. 6, we present code performance for GPUs and CPUs versus the size of the computational domain, computed following Eq. 3. The maximal size of the domain is limited by the size of the GPU memory, which is 64GB for a single GPU and 256GB for four GPUs for one node. Fig. 6 contains two sets of data, solid lines depict simulations performed on a full node, which is 4 GPUs or 80 CPU cores; the dashed lines stand for a single GPU and single CPU core tests. For a single core, the size of the computational domain varies from $64 \times 64 \times 64$ to $384 \times 400 \times 400$ and for a single node setup, the size varies from $80 \times 120 \times 120$ to $640 \times 620 \times 620$; the GPU memory is 97% full at the largest domain. For the full node case, the domain sizes must be divisible by 4 and 80; therefore, they are not always cubic. Lines with squares and triangles represent code performance for GPU and CPU versions, respectively; they are

[1] For large 2D domains the difference is negligible, $< 0.2\%$, while for a 3D setup the difference is more significant, reaching 10% for small domains

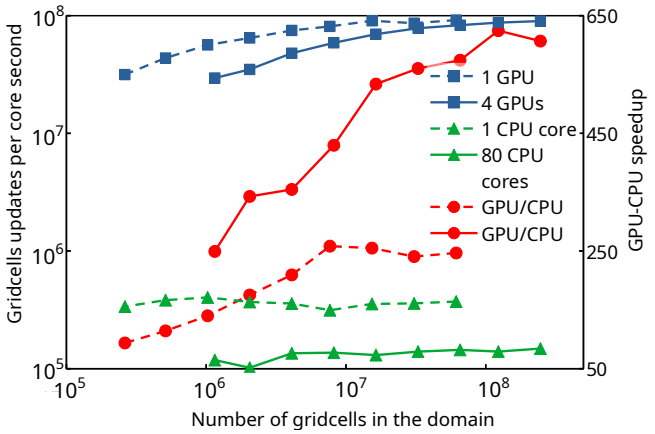

Fig. 6. Code performance for GPU and CPU versus the size of the computational domain; solid lines stand for full node (4 GPUs or 80 CPU cores), dashed lines stand for a single GPU/CPU core. The code performance is plotted by blue lines with squares for GPU and green lines with triangles for CPU; they are measured along the left axis. The red lines with circles show the GPU-CPU speedup; they are measured along the right axis.

measured along the left axis. Red lines with circles show the ratios between the GPU and the CPU performances, which are measured along the right axis.

We start our discussion with the case of one GPU versus one CPU core depicted in dashed lines. Although it is not practical from the point of view of large-scale simulations, it helps in understanding differences between GPU and CPU performance and it is also the fairest and most fundamental case for comparison. The most striking feature is the more than two orders of magnitude difference between the GPU and the CPU performances; however, they behave differently. The CPU performance is nearly constant, varying only within 10% around the level of $3.7 \cdot 10^5$, which is in line with our previous estimates, see (Modestov et al. 2024). The smallest domain is noticeably larger than the cache memory, so that the CPU is memory-bound within the whole range of the domain size. The GPU performance reaches $9.1 \cdot 10^7$ for large domains, and it is almost three times smaller for the smallest domain. This is due to the fact that for a smaller domain, the GPU uses fewer thread blocks and, hence, it doesn't operate on its full capacity. This difference between GPU and CPU hardware features, together with significantly larger bandwidth of the GPU, explains the initial increase and then saturation of the GPU-CPU speedup.

It is interesting to notice that if we compare the GPU performance with respect to SM, assuming that all SMs are used, then we will find that a single SM has similar performance as a CPU core. To obtain this result, the numbers on the right axis of Fig. 6 should be divided by 132. It partially confirms the discussion at the beginning of this section that the GPU-CPU performance comparison could be done based on a CPU core and a GPU SM.

Next, we considered the case of a full single-node performance, depicted in solid lines. This type of setup is more reasonable as compared to the first case, as it involves communications between different cores, which are unavoidable in large-scale simulations. The GPU performance remains at the same level as for a single GPU case; for large domains, the communication overheads are about 4% of the overall time. It also degrades towards smaller domains due to not fully occupied GPU cores and growing overheads due to exchanges. On the contrary, the CPU performance of the full node with 80 cores drops by 2.5 times with respect to a single CPU core case. This drop is due to limited CPU cache memory, which is now shared among all the cores. We have also checked that for the CPU case, the communication overheads are the same 4% of the overall time for large domains. It is mostly determined by the relative size of the exchanged area with respect to the subdomain size, which for a large domain is a small number $< 10\%$, hence the communications between CPU cores are not the limiting factor of performance in a multi-core configuration. Hence, for a multi-core setup, we obtain an unprecedented GPU-CPU speedup, reaching 620 for larger domains.

For 2D setups, the GPU performance is about 20% higher than for 3D and reaches $1.1 \cdot 10^8$. It is mostly due to a smaller amount of calculation per grid cell as all the y-derivative terms are skipped. In addition, the exchange time between different cores reduces drastically.

Finally, to complete the discussion on the code performance, it is of interest to compare the GPU performance of Mancha3D with other codes. For example, (Caddy & Schneider 2024) has a slightly larger limit for the domain size per GPU (450^3) and twice higher performance of $\approx 2 \cdot 10^8$ grid cells updates per core per second. They also used CUDA and applied some optimization to the kernel configuration. Higher performance can be attributed either to a smaller amount of computation per grid cell or to the fact that they run their tests on the Frontier machine, which is currently the most powerful GPU machine. They do not have the CPU version. (Del Zanna et al. 2024) with OpenACC technique, declare similar performance on GPUs, $\approx 2 \cdot 10^8$ grid cells updates per core per second for 512^3 domain. However, they have very moderate GPU-CPU speedup, around 30 due to nonoptimized exchanges between GPUs. A well-known in the Solar society MURaM code, also uses OpenACC to run their code on GPUs and get about 70 speedup with respect to performance on CPUs, (Wright et al. 2021). This moderate speedup could be explained by the fact that some computations, such as radiative transfer, cannot be fully parallelized for GPU kernels. (Stone et al. 2024) with the KOKKOS model tested their code on several GPUs and demonstrated noticeably higher performance, $\approx 6 \cdot 10^8$ grid cells updates per core per second. They also compared GPU and CPU performances with a slightly different metric scaled to a device, not to a core, and achieved 300 times speedup. To follow their strategy, we need to compare the dashed blue line to the solid green line in Fig. 6, resulting in 1500 GPU-CPU speedup for our code. Thus, we conclude that our code demonstrates state-of-the-art performance characteristics with excellent GPU-CPU speedup.

3.2. Weak scaling

Weak scaling refers to performance tests when the domain size increases together with the number of cores, so that each core always treats the same number of grid cells. Partially, we have already considered weak scaling in Fig. 6, which demonstrates almost the same performance for 4 GPUs as for a single GPU. In Fig. 7, we present weak scal-

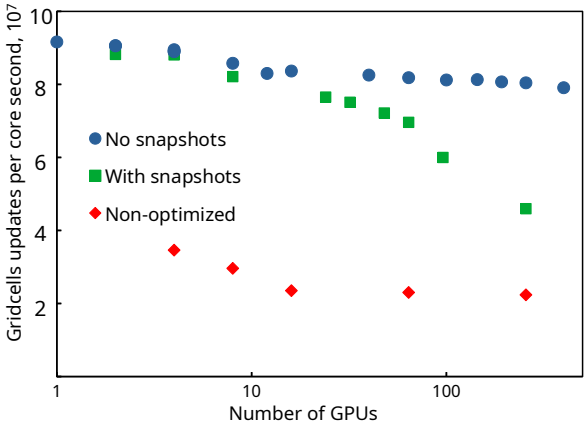

Fig. 7. Weak scaling performance for a 3D setup with (green squares) and without (blue circles) saving snapshots. Red rhombuses stand for nonoptimized data exchanged used in the CPU version. See Section 2.3 for details. The latter data points start with 4 GPUs; the leftmost red rhombus in the figure belongs to the annotation

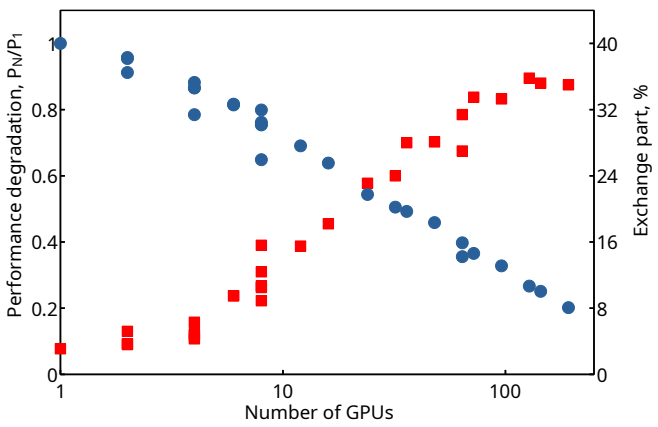

Fig. 8. Strong scaling for a 3D setup together with the role of communications. The blue dots stand for the code performance computed by N GPUs with respect to a single GPU. They are plotted along the left axes. The red squares plotted along the right axes demonstrate the relative role of the exchanges between GPUs.

.

ing results for up to 400 GPUs [2]. A single GPU treats a 360^3 domain, which fills its memory by 75%; the maximal setup treated by 400 GPUs contains almost $2 \cdot 10^{10}$ grid cells.

The blue dots show the main scaling trend for a test without saving snapshots. The code performance drops by 14% for 400 GPUs with respect to a single GPU, which demonstrates very good parallelization efficiency. The efficiency of weak scaling is often affected by saving snapshots, which is an inevitable part of real simulations. In Mancha3D the snapshot is saved into HDF5 format and for the largest setup it has a size of about 1 TB. With green squares, we plot weak scaling when several snapshots are saved during the test. The performance drops by about 50% due to writing a huge amount of data on a hard-drive which is the slowest process in the supercomputer workflow. These results are very similar to those obtained for the CPU version of Mancha3D, see Fig. 12 of (Modestov et al. 2024).

For these two tests, we used the optimized MPI exchanges between GPUs as described in Sect. 2.3. To demonstrate the effect of this optimization, we also performed additional tests with the "standard" way of exchanges, depicted with a red rhombus. The parallel efficiency drops significantly and saturates at the level of 25% of a single GPU. It should be mentioned that for these tests, we used a decomposition only in the Y and Z dimensions, as decomposition in X is about 50 times slower. It clearly demonstrates that using standard CUDA-aware MPI is not enough to achieve an efficient parallelization with GPUs.

3.3. Strong scaling

Strong scaling implies a fixed-size problem run on a single core, compared to running on several cores. It shows parallel efficiency of the code and may also determine optimal subdomain size, if any. From the point of view of the HPC, the strong scaling is not a very relevant test for GPUs, in addition to the growing communication overhead, a GPU performs fewer and fewer computations, which negates its main advantage. Still, for the purposes of completeness, we have performed strong scaling tests and calculated degradation of performance as a ratio of multiple GPUs to a single GPU performance, computed due to expression 3. In Fig. 8, it is plotted with the blue circles, measured along the left vertical axis. We have also measured the relative role of exchanges between GPUs with respect to the overall time, which is plotted as the red squares and measured along the right vertical axis.

The computational domain is $400 \times 384 \times 384$, which takes 93% of the memory of a single GPU. When the number of GPUs exceeds 100, the memory of each GPU is filled by less than 2%. In addition, for such a small subdomain, the number of thread blocks reduces considerably, and the GPU operates on a small portion of its capacity. In this sense, it is different from the CPU performance, where the reduced subdomain size treated by individual CPU cores leads to more cache-efficient computation. For the maximum 192 GPUs used in this test, the GPU performance drops by 5 times, when the subdomain size decreases nearly by 200 times, and the exchange role saturates at the level of 35%.

We also checked the effect of different decompositions; for example, for $N_{GPU} = 8$, we performed five possible tests. The lowest point of 0.6 corresponds to decomposition in X, while the most optimal 0.8 value stands for decomposition in Z directions. It is related to different speeds of memory access for the arrays allocated in the GPU's memory. Although the effect of different decompositions is not very significant, it should be taken into account for large-scale simulations.

4. Conclusions and perspectives

We implemented the CUDA version of theMancha3D code to run it on GPUs. We reached a performance of about 10^8 with is in line with other modern GPU codes, (Del Zanna et al. 2024; Lesur et al. 2023; Stone et al. 2024), demonstrating that with CUDA Fortran it is pos-

[2] 400 GPUs is the maximum number of GPUs that we could use on the MareNostrum supercomputer under the code development access

sible to write very effective MHD codes. We also obtained an unprecedented 600-fold speedup in performance with respect to the CPU code. Such a speedup results in multiple optimizations in kernel configurations and communication procedures.

Another way to compare efficiency of the CPU and the GPU code versions could be the power consumption. However, technologically it is difficult and not always possible. On the MareNostrum5 machine, the measurable power consumption of a compute node does not depend on whether it uses CPU cores or GPUs (each node has 4 GPUs and 80 CPU cores). In this case, the gain in power consumption becomes equivalent to the ratio of nodes needed to run the same job on CPUs and on GPUs. With a 600-fold speedup, a full node with 4 GPUs corresponds to full 30 nodes using CPU cores; hence, the GPU version will consume 30 times less power. Still, this estimate is very rough and may no represent actual numbers.

We have also explored the possibility of a hybrid mode when GPUs and CPUs work together. However, with such efficient computation on GPUs, the hybrid mode can only add a few percent to the code performance, while the communication scheme is much more complicated.

As a next step, we plan to extend the CUDA implementation to the RTE module. It is a necessary step before going to realistic simulations of solar convection. As pointed out in (Wright et al. 2023), the radiative transfer can significantly slow down the GPU-CPU speedup. This is a challenging task as the core of the radiative solver involves an integration process, which basically requires sequential access to the data; in turn, this is in opposition to the internal GPU micro-parallelization. Taking into account the experience gained during this work, we believe that it will be possible to implement the radiative transfer module in CUDA efficiently enough. At that point, we will have access to a flexible numerical tool for solving various MHD problems on CPU and GPU supercomputers.

Acknowledgements. MM and EK thank the support from Agencia Estatal de Investigación del Ministerio de Ciencia e Innovación and the European Regional Development Fund (ERDF "A way of making Europe") through grants PID2021-127487NB-I00 and PID2024-156538NB-I00; and from the OSCARS project No. 01-454 "Federation of Solar Data (FSD)". IAA is supported by the Spanish Ministry of Science, Innovation and Universities (Ministerio de Ciencia, Innovacion y Universidades, MICIU) through the research grant PID2021-122603NB-C22. The simulations have been done in the supercomputer MareNostrum at Barcelona Supercomputing Center - Centro Nacional de Supercomputación (The Spanish National Supercomputing Center); the access has been arranged under the Development project EHPC-DEV-2024D09-033.

References

Bispo, J., Barbosa, J., Silva, P., et al. 2021, in Best Practice Guide: Modern Accelerators (Elsevier)
Caddy, R. V. & Schneider, E. E. 2024, The Astrophysical Journal, 970, 44
Del Zanna, L., Landi, S., Serafini, L., Bugli, M., & Papini, E. 2024, Fluids, 9
Delorme, M., Durocher, A., Sacha Brun, A., et al. 2025, Journal of Physics: Conference Series, 2997, 012014
González-Morales, P. A., Khomenko, E., Downes, T. P., & de Vicente, A. 2018, 615, A67
Gyenge, N., Griffiths, M., & Erdélyi, R. 2018, Advances in Space Research, 61, 683, mHD Wave Phenomena in the Solar Interior and Atmosphere
Khomenko, E., Collados, M., Díaz, A., & Vitas, N. 2014, Phys. Plasmas, 21, 092901
Lesur, G. R. J., Baghdadi, S., Wafflard-Fernandez, G., et al. 2023, A&A, 677, A9
Modestov, M., Khomenko, E., Vitas, N., et al. 2024, Sol. Phys., 299, 23
Navarro, A., Khomenko, E., Modestov, M., & Vitas, N. 2022, 663, A96
Pekkilä, J., Väisälä, M. S., Käpylä, M. J., Käpylä, P. J., & Anjum, O. 2017, Computer Physics Communications, 217, 11
Stone, J., Mullen, P., Fielding, D., et al. 2024, AthenaK: A Performance-Portable Version of the Athena++ AMR Framework
Trott, C. R., Lebrun-Grandié, D., Arndt, D., et al. 2022, IEEE Transactions on Parallel and Distributed Systems, 33, 805
Wong, H.-C., Wong, U.-H., Feng, X., & Tang, Z. 2011, Computer Physics Communications, 182, 2132
Wright, E., Brown, C., Przybylski, D., et al. 2023, in Proceedings of the SC '23 Workshops of the International Conference on High Performance Computing, Network, Storage, and Analysis, SC-W '23 (New York, NY, USA: Association for Computing Machinery), 1929–1938
Wright, E., Przybylski, D., Rempel, M., et al. 2021, in Proceedings of the Platform for Advanced Scientific Computing Conference, PASC '21 (New York, NY, USA: Association for Computing Machinery)