Master in High Performance Computing

Porting of DSMC to multi-GPUs using OpenACC

Supervisor(s):
Gianluca Di Staso Supervisor,
Ivan Girotto Supervisor,
Sebastiano Fabio Schifano Supervisor

Candidate:
Marco Celoria

8th edition
2021–2022

Master in
High Performance Computing

Abstract

The Direct Simulation Monte Carlo has become the method of choice for studying gas flows characterized by variable rarefaction and non-equilibrium effects, rising interest in industry for simulating flows in micro-, and nano-electromechanical systems.
However, rarefied gas dynamics represents an open research challenge from the computer science perspective, due to its computational expense compared to continuum computational fluid dynamics methods.

Fortunately, over the last decade, high-performance computing has seen an exponential growth of performance. Actually, with the breakthrough of General-Purpose GPU computing, heterogeneous systems have become widely used for scientific computing, especially in large-scale clusters and supercomputers.
Nonetheless, developing efficient, maintainable and portable applications for hybrid systems is, in general, a non-trivial task.
Among the possible approaches, directive-based programming models, such as OpenACC, are considered the most promising for porting scientific codes to hybrid CPU/GPU systems, both for their simplicity and portability.

This work is an attempt to port a simplified version of the *fm_dsmc* code developed at FLOW Matters Consultancy B.V., a start-up company supporting this project, on a multi-GPU distributed hybrid system, such as Marconi100 hosted at CINECA, using OpenACC.
Finally, we perform a detailed performance analysis of our DSMC application on Volta (NVIDIA V100 GPU) architecture based computing platform as well as a comparison with previous results obtained with x64_86 (Intel Xeon CPU) and ppc64le (IBM Power9 CPU) architectures.

Contents

Chapter 1

Introduction

Over the last decade, heterogeneous systems, where general-purpose Central Processing Units (CPUs) are supported by hardware accelerators, have become widely used for scientific computing, especially in large-scale clusters and supercomputers.
There are two main reasons for this trend.

The first key factor, in the success of accelerators, is power efficiency, as performance per watt has been playing a major role in the recent evolution of supercomputing technologies. Remarkably, the Green500 (November, 2022) [1] shows that heterogeneous systems are consistently the most energy-efficient ones.

The second key factor is the need to optimize larger and larger systems for certain specific workloads, given the fact that scientific and industrial applications are combining different technologies into complex work-flows, ranging from scientific simulations to data analytic and machine learning [2].
Actually, at the time of writing, 7 out of 10 most powerful HPC systems, according to Top500 list (November, 2022) [3], employ Graphics Processing Units (GPUs) accelerators, highly parallel multi-core systems allowing efficient manipulation of large blocks of data.
In fact, GPUs were originally designed for acceleration of real time graphics in applications such as video games, but it was then realized that their highly parallel computing power could also be used for scientific computations or machine learning. In general, the use of GPUs in non-graphical applications is known as General-Purpose Graphics Processing Unit (GPGPU) programming.

However, developing efficient, maintainable and portable applications for such hybrid systems is in general a non-trivial task and several approaches are possible.
The first possibility is using a dedicated parallel computing platform such as Computed Unified Device Architecture (CUDA), developed by NVIDIA since 2006 [4, 5]. Working directly at the programming language level with an interface based on C/C++ results in maximum flexibility, although there are some drawbacks, the most relevant ones being portability and complexity.

As far as portability is concerned, CUDA-enabled GPUs are only available from NVIDIA. As an open alternative to CUDA, the Open Computing Language (OpenCL) has been developed since 2008 by the non-profit technology consortium Khronos Group [6].

However, from the complexity perspective, both CUDA and OpenCL are low-level Application Programming Interfaces (APIs) and developing or porting programs to GPUs

within these frameworks may be difficult.
For these reasons, recent interest has been focused on directive-based approaches such as OpenMP [7] and OpenACC [8, 9], where, unlike low-level programming models, the complexity is handled mainly by the compiler, allowing GPGPU programming to non-CUDA/OpenCL experts. Such directives represent a useful tool to accelerate intensive applications, significantly reducing the work required to port an existing code to any accelerator platform, although generally limiting the performance of the GPU compared to CUDA (see for instance [10]).

In this work, we focus on OpenACC, a programming standard for parallel computing designed to simplify parallel programming of heterogeneous CPU/GPU systems. Developed by Cray, CAPS, NVIDIA and PGI in 2012, it is currently maintained by the non-profit OpenACC corporation.
Similarly to OpenMP, the programmer can annotate C, C++ and Fortran source code to identify the areas that should be accelerated using compiler directives. As a result, GPGPU programming becomes, not only straightforward, but also portable across parallel and multi-core processors.
In conclusion, OpenACC can be seen as a promising tool for developing or porting scientific codes to hybrid CPU/GPU systems.
In this work, we present the OpenACC implementation of a Direct Simulation Monte Carlo (DSMC) algorithm, a numerical method proposed by Bird [11, 12, 13] for modeling rarefied gas flows. Actually, the coarse-grained distribution of work across processors is effectively handled via MPI, whereas the fine-grain parallelism is accelerated using OpenACC by carefully designing the data structures and the algorithms to fully exploit the multi-GPUs capabilities.

This thesis is outlined as following.
In Chapter 2, we introduce the standard approaches to rarefied gas modeling both from a theoretical as well as numerical perspective, describing the DSMC model.
In Chapter 3, we describe the GPGPU architecture and its massively data parallel capabilities, including a review of two programming models, namely CUDA and OpenACC.
Then, in Chapter 4, we present the details of our DSMC implementation, focusing on the algorithm we have designed and the main OpenACC directives we have used.
Finally, the results that we achieved are shown in Chapter 5, while the conclusions can be found in Chapter 6.
The work presented in this document was performed as a training thesis of the Master in High Performance Computing - MHPC, and supported by FLOW Matters Consultancy B.V. fm, a start-up company originated from the Eindhoven University of Technology.

Chapter 2

Direct Simulation Monte Carlo method for modeling rarefied gas flows

Fluid dynamics, describing the flow of fluids, liquids and gases, typically involves the calculation of various macroscopic properties of the fluid, such as flow velocity, pressure, density, and temperature, as functions of space and time.
By combing the conservation laws (conservation of mass, linear momentum and energy) with a continuum assumption (valid on length scales much greater than the inter-atomic distances) and the thermodynamic equation of state (relating pressure, temperature and density), these slowly-varying macroscopic properties are well-defined at infinitesimally small points in space and vary continuously from one point to another.
Under the further assumption that the flow velocity is small compared to the speed of light and after specifying a constitutive law for the stress tensor, the dynamics of the fluid is described by a non-linear set of differential equations, called Navier-Stokes equations.
Such equations do not have a general closed-form solution, and are mainly solved using computational fluid dynamics.

However, the situation changes completely for rarefied gas flows, where the mean free path of a molecule is of the same order (or greater) than a representative physical length scale. In this case, the continuum assumption of fluid dynamics might not be valid, and the Navier-Stokes equations can be inaccurate.
To be more precise, we can introduce the Knudsen number (Kn), a dimensionless number defined as the ratio of the molecular mean free path length λ to a representative physical length scale L

$$\mathrm{Kn} = \frac{\lambda}{L}, \tag{2.1}$$

where L is defined taking into account the overall dimensions of the flow, e.g. the height of a channel, or according to the macroscopic flow gradients

$$L = \frac{Q}{|dQ/d\ell|} , \tag{2.2}$$

where Q might be chosen as the gas density, velocity, temperature, or other hydrodynamic quantities and ℓ is the smallest hydrodynamic length scale.
According to the Knudsen number, we can classify the gas flows into the continuous or

hydrodynamic (Kn < 0.01), slip ($0.01 <$ Kn < 0.1), transition ($0.1 <$ Kn < 10) and free molecular (Kn > 10) regimes, see Figure 2.1.

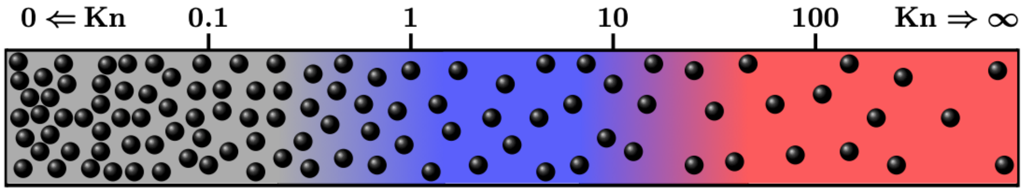

Figure 2.1: Visual representation of flows at different levels of rarefaction, with the associated Knudsen number [14].

The Navier-Stokes equations are valid for Kn < 0.1 that is the hydrodynamic as well as the slip regime (using specific slip boundary conditions).
However, for Kn > 0.1 the continuum assumption does not hold and we need to rely on kinetic equations (such as the Boltzmann, Enskog or the collisionless Boltzmann equation for free molecular flow). For a comprehensive treatment of the subject see [13, 15, 16].
Such discrete particles methods are typically studied numerically within Molecular Dynamics (MD) simulations by solving Newton's equations of motion for a system of interacting particles, where forces between the particles and potential energies are often calculated using interatomic potentials or molecular mechanics force fields.

Note that quantum mechanical effects can be ignored if the dynamics of electrons are so fast that they can be considered to react instantaneously to the motion of their nuclei (Born-Oppenheimer approximation) and if de Broglie wavelength $\lambda = h/p$ (where $h = 6.62607015 \times 10^{-34}$ m^2 Kg s^{-1} is the Planck constant and p is the particle momentum) associated to nuclei, which are much heavier than electrons, is substantially smaller than the intermolecular distances a. Actually, we can estimate the de Broglie thermal wavelength with

$$E_{kin} = \frac{p^2}{2m} = \frac{h^2}{2m\lambda^2} \simeq \frac{3}{2} k_B T \qquad \Longrightarrow \qquad \lambda \simeq \frac{h}{\sqrt{3mk_BT}} \,, \tag{2.3}$$

where $k_B = 1.380649 \times 10^{-23}$ m^2 Kg s^{-2} K^{-1} is the Boltzmann constant.
As an example, for Argon gas ($m = 6.63 \times 10^{-26}$ Kg) at $T = 300$ K, where the typical inter-molecular distance is $a = 4$ Å, we get $\lambda \simeq 0.1$ Å $\ll 4$ Å.

Under these hypothesis, we can model the gas as a collection of N (of the order of the Avogadro number, $N_A = 6.02214076 \times 10^{23}$) point-like objects interacting via an intermolecular potential in a box of volume V at temperature T and satisfying the Newton equations of motion.

However, we are not typically interested in microscopic information such as the final position or the velocity of each particle, but rather we are interested in macroscopic observables obtained by statistical averages over microscopic degrees of freedom. For this reason, such microscopic approach is not only computationally impracticable but also useless, since what we are really interested in is the statistical behavior of a thermodynamic system generally out-of-equilibrium, thus described by the Boltzmann equation.

2.1 From the Boltzmann equation to hydrodynamics

In this section we follow [17, 18, 19].
The Boltzmann equation can be derived by considering the probability distribution function (pdf) f_1 to find a molecule of mass m at position $\mathbf{r}$ with momentum $\mathbf{p} = m\mathbf{v}$ at time t.
More formally, consider a set of coordinates in a 6-dimensional phase space $(\mathbf{r}, \mathbf{p}) = (x, y, z, p_x, p_y, p_z) \in V \subset \mathbb{R}^3 \times \mathbb{R}^3$, parametrised by time $t \in [0, t_{max}]$.
Then, the one-particle pdf f_1 at time t is the non negative function such that

$$dN = f_1(\mathbf{r}, \mathbf{p}, t)\, d^3\mathbf{r}\, d^3\mathbf{p} = f_1(\mathbf{r}, \mathbf{p}, t)\, dx\, dy\, dz\, dp_x\, dp_y\, dp_z \tag{2.4}$$

is the number of molecules which all have positions lying within a volume element $d^3\mathbf{r}$ about $\mathbf{r}$ and momenta lying within a momentum space element $d^3\mathbf{p}$ about $\mathbf{p}$, at time t.
Therefore, the fluid number density is defined as

$$n(\mathbf{r}, t) = \int_{\mathbb{R}^3} d^3\mathbf{p}\ f_1(\mathbf{r}, \mathbf{p}, t)\ . \tag{2.5}$$

The average velocity of the particles is

$$\mathbf{u}(\mathbf{r}, t) = \int_{\mathbb{R}^3} d^3\mathbf{p}\ \frac{\mathbf{p}}{m} f_1(\mathbf{r}, \mathbf{p}, t) = \langle \mathbf{v} \rangle\ . \tag{2.6}$$

Integrating over a region of position and momentum space $V \subset \mathbb{R}^3 \times \mathbb{R}^3$ gives the total number of particles which have positions and momenta in that region

$$N = \int_V d^3\mathbf{r}\ d^3\mathbf{p}\ f_1(\mathbf{r}, \mathbf{p}, t) \qquad \forall t\ . \tag{2.7}$$

Notice that, from the Liouville's theorem, the streaming motion of the particles along the trajectories connected with the external force field $\mathbf{F}(\mathbf{r}, t)$, not due to other particles, satisfies

$$\frac{df_1}{dt} = \frac{\partial f_1}{\partial t} + \frac{\partial f_1}{\partial r_i}\frac{dr_i}{dt} + \frac{\partial f_1}{\partial p_i}\frac{dp_i}{dt} = 0\ , \tag{2.8}$$

where we are using the convention that we sum over the repeated index.
However, taking also into account the forces acting between particles in collisions, we finally get the Boltzmann equation

$$\frac{df_1}{dt} = \left(\frac{\partial f_1}{\partial t}\right)_{coll}\ . \tag{2.9}$$

Let us focus on the collision term.
Consider a particle sitting at $(\mathbf{r}, \mathbf{p})$ in phase space and colliding with a particle at $(\mathbf{r}, \mathbf{p}_2)$.
After the collision, the resulting particles emerge with momenta $\mathbf{p}_1'$ and $\mathbf{p}_2'$.
The details of the collision are captured by the scattering function ω containing the information about the dynamics of the process. Notice that any scattering which is invariant under time reversal, parity and translational invariance must obey

$$\omega(\mathbf{p}, \mathbf{p}_2 | \mathbf{p}_1', \mathbf{p}_2') = \omega(\mathbf{p}, \mathbf{p}_2 | \mathbf{p}_1', \mathbf{p}_2')\ . \tag{2.10}$$

Now, the typical timescales involved in the problem are the time between collisions, τ, known as the scattering time or relaxation time, and the time it takes for the process of collision between particles to occur, τ_{coll}, known as collision time. For dilute gas, we have

$$\tau \ll \tau_{coll} \ . \tag{2.11}$$

In this regime, we can assume the molecular chaos hypothesis (Stosszahlansatz), that is the velocities of colliding particles are uncorrelated as well as independent of position, and multi-body collisions can be neglected.
Then, the collision term can be written as a momentum-space integral over the product of one-particle distribution functions

$$\begin{aligned}\left(\frac{\partial f_1}{\partial t}\right)_{coll} &= \int d^3\mathbf{p}_2\ d^3\mathbf{p}_1'\ d^3\mathbf{p}_2'\ \omega(\mathbf{p}_1',\mathbf{p}_2'|\mathbf{p},\mathbf{p}_2)\left[f_1(\mathbf{r},\mathbf{p}_1')f_1(\mathbf{r},\mathbf{p}_2') - f_1(\mathbf{r},\mathbf{p})f_1(\mathbf{r},\mathbf{p}_2)\right] \\ &= \int d\Omega\ d^3\mathbf{p}_2\ gI(g,\Omega)[f_1(\mathbf{r},\mathbf{p}_1')f_1(\mathbf{r},\mathbf{p}_2') - f_1(\mathbf{r},\mathbf{p})f_1(\mathbf{r},\mathbf{p}_2)]\ ,\end{aligned} \tag{2.12}$$

where $\mathbf{p}$ and $\mathbf{p}_2$ are the momenta of any two particles before a collision, $\mathbf{p}_1'$ and $\mathbf{p}_2'$ are the momenta after the collision,

$$g = |\mathbf{p}_2 - \mathbf{p}| = |\mathbf{p}_2' - \mathbf{p}_1'| \tag{2.13}$$

is the magnitude of the relative momenta, and $I(g,\Omega)$ is the differential cross section.
Note that the Variable Hard Sphere (VHS) model is characterized by [20]

$$I(g,\Omega) = C_\alpha g^{\alpha-1} \tag{2.14}$$

where C_α is a positive constant. The Hard Sphere model corresponds to the case $\alpha = 1$.

2.1.1 Conserved quantities and hydrodynamics

Recall that hydrodynamics describes the dynamics of systems that are in local equilibrium, with density $\rho(\mathbf{r},t)$, temperature $T(\mathbf{r},t)$ and velocity $\mathbf{u}(\mathbf{r},t)$ that vary slowly in space and in time.
Actually, for macroscopic physical processes involving space-time scales much larger than relaxation time τ (and relaxation length ℓ), the only relevant long-wavelength variables are those associated with conserved quantities, as non-conserved quantities will have typically relaxed back to equilibrium. More precisely, conserved quantities are slowly varying variables in "local equilibrium" that provide the background for fast relaxing non-conserved quantities [21]. Remarkably, perturbations in conserved quantities can relax back to equilibrium only by transports, see Figure 2.2.

Let's see how hydrodynamics equations can be derived from the Boltzmann equation.
Consider a general function $A(\mathbf{r},\mathbf{p})$ over the single particle phase space.
Integrating over the momenta, we define the average

$$\langle A(\mathbf{r},t)\rangle = \frac{1}{n(\mathbf{r},t)}\int_{\mathbb{R}^3} d^3\mathbf{p}\ A(\mathbf{r},\mathbf{p})f_1(\mathbf{r},\mathbf{p},t)\ . \tag{2.15}$$

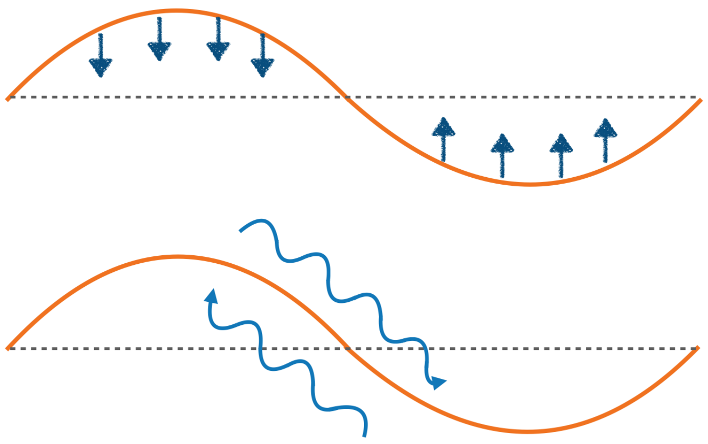

Figure 2.2: Relaxation of different types of excitations [21]. The grey dashed lines denote the global equilibrium values and the orange lines denote values of perturbed quantities. In the upper panel, perturbations in non-conserved quantities can relax back to equilibrium values locally in a time of order of the relaxation time τ. In the lower panel, conserved quantities can only relax through transports, i.e. excesses have to be transported to regions with deficits to achieve equilibrium.

Notice that

$$n(\mathbf{r},t)\left\langle A(\mathbf{r},t)\right\rangle = \frac{1}{n(\mathbf{r},t)}\int d^3\mathbf{p}\; n(\mathbf{r},t)A(\mathbf{r},t)f_1(\mathbf{r},\mathbf{p},t) = \left\langle n(\mathbf{r},t)A(\mathbf{r},t)\right\rangle \ . \tag{2.16}$$

Now, any local equilibrium distribution function, obeying the detailed balance condition

$$f_1(\mathbf{r},\mathbf{p}_1')f_1(\mathbf{r},\mathbf{p}_2') = f_1(\mathbf{r},\mathbf{p})f_1(\mathbf{r},\mathbf{p}_2) \qquad \Longrightarrow \qquad \left(\frac{\partial f_1}{\partial t}\right)_{coll} = 0 \tag{2.17}$$

has the form of a Maxwell-Boltzmann equilibrium distribution function

$$f_1^{\text{local}}(\mathbf{r},\mathbf{v},t) = \frac{n(\mathbf{r},t)}{(2\pi m k_B T(\mathbf{r},t))^{3/2}}\exp\left(-\frac{m(\mathbf{v}-\mathbf{u}(\mathbf{r},t))^2}{2k_BT(\mathbf{r},t)}\right) \tag{2.18}$$

where the number density $n(\mathbf{r},t)$ (defined in Equation 2.5), drift velocity $\mathbf{u}(\mathbf{r},t)$ (defined in Equation 2.6) and the temperature $T(\mathbf{r},t)$ are slowly varying over space and time. In particular, the temperature $T(\mathbf{r},t)$ of a non-equilibrium gas, in general, is defined by

$$T(\mathbf{r},t) = \frac{m}{3k_B}\left\langle [\mathbf{v}-\mathbf{u}(\mathbf{r},t)]^2\right\rangle \ . \tag{2.19}$$

From the Boltzmann equation, we get

$$\int_{\mathbb{R}^3} d^3\mathbf{p}\; A(\mathbf{r},\mathbf{p})\left(\frac{\partial}{\partial t}+\frac{\mathbf{p}}{m}\cdot\frac{\partial}{\partial \mathbf{r}}+\mathbf{F}\cdot\frac{\partial}{\partial \mathbf{p}}\right)f_1(\mathbf{r},\mathbf{p},t) = \int_{\mathbb{R}^3} d^3\mathbf{p}\; A(\mathbf{r},\mathbf{p})\left(\frac{\partial f_1}{\partial t}\right)_{coll} \ . \tag{2.20}$$

The function $A(\mathbf{r},\mathbf{p})$ is a collisional invariant if

$$\int_{\mathbb{R}^3} d^3\mathbf{p}\; A(\mathbf{r},\mathbf{p})\left(\frac{\partial f_1}{\partial t}\right)_{coll} = 0 \ . \tag{2.21}$$

For collisional invariants, integrating by parts, throwing away boundary terms[1], and using the fact that $A(\mathbf{r}, \mathbf{p})$ has no explicit time dependence, we get

$$\frac{\partial}{\partial t}\int_{\mathbb{R}^3} d^3\mathbf{p}\ A\ f_1\ + \frac{\partial}{\partial \mathbf{r}}\cdot \int_{\mathbb{R}^3} d^3\mathbf{p}\ \frac{\mathbf{p}}{m}\ A\ f_1 - \int_{\mathbb{R}^3} d^3\mathbf{p}\ \frac{\mathbf{p}}{m}\cdot\frac{\partial A}{\partial \mathbf{r}}\ f_1 - \int_{\mathbb{R}^3} d^3\mathbf{p}\ \mathbf{F}\cdot\frac{\partial A}{\partial \mathbf{p}}\ f_1 = 0 \tag{2.22}$$

and so

$$\frac{\partial}{\partial t}\langle nA\rangle + \frac{\partial}{\partial \mathbf{r}}\cdot\langle nA\mathbf{v}\rangle - n\left\langle \mathbf{v}\cdot\frac{\partial A}{\partial \mathbf{r}}\right\rangle - n\left\langle \mathbf{F}\cdot\frac{\partial A}{\partial \mathbf{p}}\right\rangle = 0\ . \tag{2.23}$$

Density

Now, consider the mass conservation. Substituting $A = m$ in Equation 2.23, we get

$$\frac{\partial \rho}{\partial t} + \frac{\partial}{\partial \mathbf{r}}\cdot(\rho\ \mathbf{u}) = 0\ , \tag{2.24}$$

where we have defined $\rho(\mathbf{r}, t) = m\ n(\mathbf{r}, t)$.

Momentum

Similarly, for momentum conservation, setting $A = m\mathbf{v}$ in Equation 2.23, it is possible to get the following expression

$$\frac{\partial}{\partial t}(\rho u_i) + \frac{\partial}{\partial r_j}\langle \rho v_j v_i\rangle - \langle nF_i\rangle = 0\ . \tag{2.25}$$

Note that

$$\begin{aligned}\rho\langle v_j v_i\rangle &= \rho\langle (v_j - u_j)(v_i - u_i)\rangle + \rho(u_i u_j + u_j u_i - u_i u_j)\\ &= \rho\langle (v_j - u_j)(v_i - u_i)\rangle + \rho\ u_i u_j = P_{ji} + \rho\ u_i u_j\ ,\end{aligned} \tag{2.26}$$

where we have defined the pressure tensor as $P_{ij} = \rho\langle (v_i - u_i)(v_j - u_j)\rangle$.
Finally, using Equation 2.24 and $\langle F_i(\mathbf{r})\rangle = F_i(\mathbf{r})$ (since, by assumption, the force can depend on position but not on momentum), we get

$$\rho\left(\frac{\partial}{\partial t} + u_j\frac{\partial}{\partial r_j}\right)u_i = \rho\ D_t u_i = \frac{\rho}{m}F_i - \frac{\partial P_{ij}}{\partial r_j}\ , \tag{2.27}$$

where we have introduced the material derivative

$$D_t \equiv \frac{\partial}{\partial t} + \mathbf{u}\cdot\frac{\partial}{\partial \mathbf{r}} = \frac{\partial}{\partial t} + u_j\frac{\partial}{\partial r_j} \tag{2.28}$$

capturing the rate of change of a quantity as seen by an observer swept along the streamline of the fluid.

[1]Dropping boundary terms is justified because f_1 is normalized and so $f_1 \to 0$ in asymptotic parts of phase space

Kinetic Energy

The last collisional invariant is the kinetic energy of the particles. However, to simplify calculations, we consider the relative kinetic energy.
Actually, substituting $A = \frac{m(\mathbf{v}-\mathbf{u})^2}{2}$ into Equation 2.23, we get

$$\frac{1}{2}\frac{\partial}{\partial t}\left\langle \rho(\mathbf{v}-\mathbf{u})^2 \right\rangle + \frac{1}{2}\frac{\partial}{\partial r_i}\left\langle \rho v_i(\mathbf{v}-\mathbf{u})^2 \right\rangle - \rho\left\langle v_i \frac{\partial u_j}{\partial r_i}(\mathbf{v}-\mathbf{u})^2 \right\rangle = 0 \ , \tag{2.29}$$

since $\langle \mathbf{v}-\mathbf{u}\rangle = \mathbf{u}-\mathbf{u} = 0$ and $\langle (\mathbf{v}-\mathbf{u})\cdot \partial\mathbf{u}/\partial t\rangle = \langle \mathbf{v}-\mathbf{u}\rangle \cdot \partial \mathbf{u}/\partial t = 0$.
Now, consider

$$\begin{aligned} \frac{m\rho}{2}\left\langle v_i(\mathbf{v}-\mathbf{u})^2\right\rangle &= \frac{m\rho}{2}\left\langle (v_i-u_i)(\mathbf{v}-\mathbf{u})^2\right\rangle + \frac{1}{2}m\rho u_i\left\langle (\mathbf{v}-\mathbf{u})^2\right\rangle \\ &= q_i + \frac{3}{2}\rho u_i k_B T \ , \end{aligned} \tag{2.30}$$

where we have used the out of equilibrium expression for the temperature (Equation 2.19) and defined the heat flux as

$$q_i = \frac{m\rho}{2}\left\langle (v_i-u_i)(\mathbf{v}-\mathbf{u})^2\right\rangle \ . \tag{2.31}$$

Then, Equation 2.29 becomes the conservation of energy

$$\frac{3}{2}\frac{\partial}{\partial t}(\rho k_B T) + \frac{\partial}{\partial r_i}\left(q_i + \frac{3}{2}\rho u_i k_B T\right) + mP_{ij}\frac{\partial u_j}{\partial r_i} = 0 \ . \tag{2.32}$$

Finally, using Equation 2.24, we get

$$\rho k_B D_t T + \frac{2}{3}\frac{\partial q_i}{\partial r_i} + \frac{2}{3}mU_{ij}P_{ij} = 0 \ , \tag{2.33}$$

where the symmetric tensor $U_{ij} = \frac{1}{2}\left(\frac{\partial u_i}{\partial r_j} + \frac{\partial u_j}{\partial r_i}\right)$ is known as the rate of strain.
As a concluding remark, notice that the set of Equations 2.24, 2.27, 2.33, are not closed since the equation for ρ depends on $\mathbf{u}$; the equation for $\mathbf{u}$ depends on P_{ij} and the equation for T depends on $\mathbf{q}$. The problem is that, to determine any of these, we need to solve the Boltzmann equation and compute the distribution f_1. How can we do this?

2.1.2 Splitting approach

In the framework of DSMC, to solve the full Boltzmann equation we use an operator splitting algorithm scheme. Actually, the solution after one time step Δt can be obtained by the sequence of two steps: a free flow step (i.e. streaming of the particles) in which we solve the collisionless Boltzmann equation over a time interval Δt

$$\begin{cases} \dfrac{\partial h_1(\mathbf{r},\mathbf{v},t)}{\partial t} + \mathbf{v}\cdot\dfrac{\partial h_1(\mathbf{r},\mathbf{v},t)}{\partial \mathbf{r}} = 0 \\ h_1(\mathbf{r},\mathbf{v},0) = f_1^0(\mathbf{r},\mathbf{v}) \ , \end{cases} \tag{2.34}$$

and the collision step where we integrate the space homogeneous equation

$$\begin{cases} \dfrac{\partial h_2(\mathbf{r},\mathbf{v},t)}{\partial t} = Q \\ h_2(\mathbf{r},\mathbf{v},0) = h_1(\mathbf{r},\mathbf{v},\Delta t) \ , \end{cases} \tag{2.35}$$

where Q is the collision operator.
The final solution $h_2(\mathbf{r},\mathbf{v},\Delta t)$ after one time interval Δt represents an approximate form of the solution of the Boltzmann equation, $f_1(\mathbf{r},\mathbf{v},\Delta t)$.

2.2 Basics of the DSMC method

In this section, we present the basis of the DSMC method, following [14, 13]. Specifically, we focus on the classical method originally developed by Bird [13] known as No Time Counter (NTC) algorithm, providing a good compromise between accuracy and efficiency. The typical DSMC simulation flowchart is depicted in Figure 2.3.
However, within the DSMC framework, several schemes have been proposed, such as the time counter [22], the majorant collision frequency [23] or the Bernoulli trials scheme [24] that differ in the determination of the number of potential collision pairs during a time step.

2.2.1 Initialization of the simulation

During the *Initialization* phase, the grid composed of cells is created.
Notice that, in general, we can consider variable-sized grid cells, in order to group similar numbers of particles even when the spatial density of particles varies dramatically.
In our implementation, for simplicity, we consider a Cartesian grid and the very same cells are used both for sampling the microscopic properties as well as processing collisions. However, other approaches are possible, see for instance [14].
One of the rules of thumb typically employed in DSMC simulations prescribes that the cell size, Δx, should be comparable to the microscopic molecular mean free path λ, and specifically $\Delta x \leq \lambda/3$.

As an example, consider a gas composed of Argon atoms (with diameter $d = 3.66\times10^{-10}$ m) that occupies a square box of size $L = 1$ mm at a temperature $T = 273$ K and pressure $p = 10^3$ Pa. Using $nk_BT = p$, we get $n = 2.6531 \times 10^{23}$ m^{-3}. The mean free path, calculated according to the Hard Sphere model, is roughly $\lambda = \frac{1}{\sqrt{2}\pi d^2 n} \simeq 6.3$ μm giving $\Delta x = 2.1$ μm. This means a total of 1.1×10^8 cells to fully discretize the domain.

Once we have created the geometry, we need to initialize the particles.
Note that each particle in the DSMC framework represents, in reality, a very large number of real molecules that are moving along the same direction with the same velocity, thus corresponding to the definition of the single particle distribution function $f_1(\mathbf{r},\mathbf{v},t)$.
In order to reduce the risk of repeated collisions between the same particles, we need about 20 DSMC particles per cell.

Having in mind the example above, we have $\Delta x^3 \simeq (2.1\ \mu\text{m})^3 \simeq 9.3\times10^{-18}$ m$^3 \simeq 10^{-17}$ m^3, so each cell would contain $N_{real} = \frac{p\Delta x^3}{k_BT} \simeq 2.5 \times 10^6$ real particles. On the other hand,

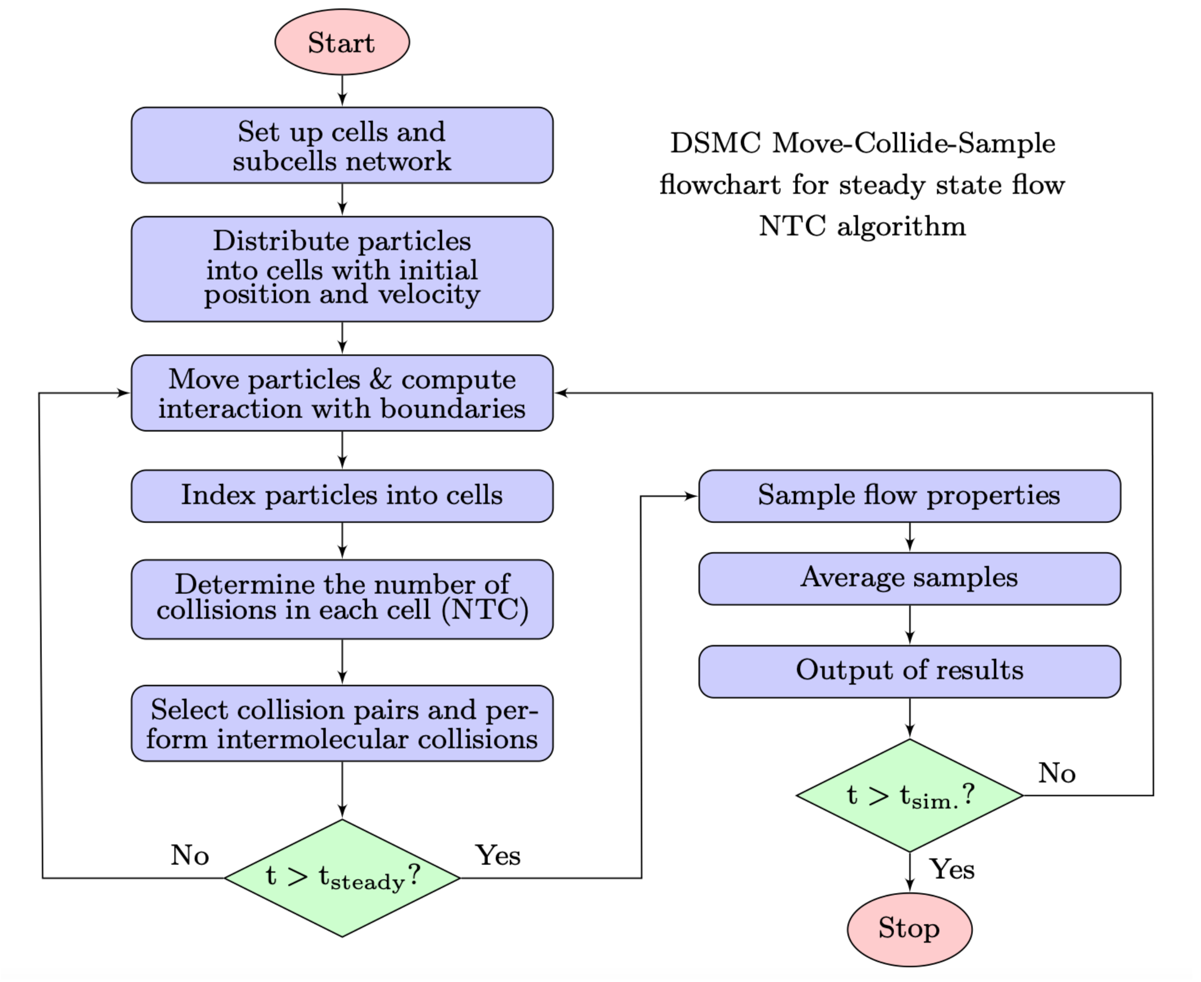
Start
Set up cells and subcells network
DSMC Move-Collide-Sample flowchart for steady state flow NTC algorithm
Distribute particles into cells with initial position and velocity
Move particles & compute interaction with boundaries
Index particles into cells
Sample flow properties
Determine the number of collisions in each cell (NTC)
Average samples
Output of results
Select collision pairs and perform intermolecular collisions
$t > t_{sim.}$?
No
Yes
No
$t > t_{steady}$?
Yes
Stop

Figure 2.3: Typical DSMC simulation flowchart for the NTC algorithm and for steady state flow. The represented order of the operations, Move-Collide-Sample (MCS), is the classical one, but other options exist [14].

considering 20 DSMC particles per cell, we have that each DSMC particle represents about $N_{real}/20 \simeq 1.25 \times 10^5$ real Argon atoms.

So far, we have described how to create the grid and populate it with particles, but we still need to initialize the particles' positions and velocities. Actually, one possibility is to assign to each particle a random initial position within a cell and sample the three velocity components from a probability distribution function taking into account the local flow properties.

The most common pdf used for this initialization is the Maxwell Boltzmann distribution. However, this is not the only option, since what we really need is that the Equation 2.19 is satisfied.

2.2.2 Streaming step

After the *Initialization*, we enter inside the time loop where the sequence of streaming and collisions are performed.
Let's focus on the *Streaming* step during which the particles' positions $\mathbf{r}_i$ are updated according to their velocity $\mathbf{v}_i$ and the fixed time step Δt.
In order to choose Δt properly, a few considerations are in order. First, the most probable velocity is given by

$$v_{mp} = \sqrt{\frac{3k_BT}{m}} \ . \tag{2.36}$$

Second, the residence time for a particle in a cell is

$$t_{res} = \frac{\Delta x}{v_{mp} + U_f} \ , \tag{2.37}$$

where U_f is the estimated flow velocity. Then, in order to minimize inaccuracies in the collisional transport, the rule of thumb is to impose

$$\Delta t \leq \frac{t_{res}}{4} \ . \tag{2.38}$$

In this way, particles will likely stay within the same cell or move as far as the nearest neighbor cells during one time step, thus reducing possible errors related to an excessive collisional transport.
Then, in absence of external forces, we can integrate the equation of motion via the Euler method:

$$\mathbf{r}_i(t + \Delta t) = \mathbf{r}_i(t) + \mathbf{v}_i(t)\Delta t \ . \tag{2.39}$$

Moreover, during the streaming step, we also need to take into account the interactions with boundaries. Specifically, there are three main possibilities:

- the periodic boundary conditions
- the specular wall model
- the diffusive wall model

In this work, we consider for simplicity periodic boundary conditions, often chosen for approximating large (infinite) systems. When one molecule diffuses across the boundary of the simulation box it reappears on the opposite side.

For completeness, let us just briefly describe the other two possibilities.
The specular wall model prescribes that, once a particle hits a wall, the velocity component normal to the wall is reverted while the other two components are unchanged. In this case, the underlying hypothesis is that the surface of the wall is perfectly smooth.
On the contrary, assuming a microscopically rough surface, we have the diffusive wall model, which prescribes that the post-interaction velocity components are sampled from a biased Maxwellian distribution in the frame of reference of the wall.
In any case, after the solid boundary interactions, the particles complete their advection for the residual time.
See, for example, Figure 2.4 for a schematic representation of the *Streaming* step.

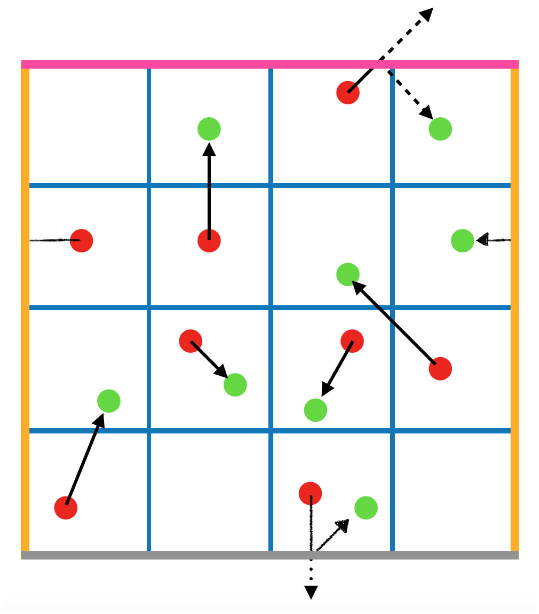

Figure 2.4: Particles configuration before (red dots) and after the streaming step (green dots). Particles are free to stream along any direction with any speed. Periodic boundary conditions are imposed on orange walls. Specular boundary conditions are imposed on the pink wall. Diffusive boundary conditions are imposed on the grey wall.

2.2.3 Indexing and Communication steps

After the *Streaming* step is completed, we need to sort and index particles by grid cell. The reason is twofold: on one hand the inter-molecular collisions are performed only between particles in the same cell. On the other hand, macroscopic fields are computed by evaluating the properties of particles residing in the same cell, for example by averaging the velocities of the particles within a cell.
The main idea is to have a map between each cell and the particles residing in that cell. However, because particles continuously move to new cells, we need to constantly update such mapping.

Notice that, for DSMC codes in HPC, the grid cells are typically distributed among many processors. Moreover, each processor also owns ghost cells, which are copies of grid cells owned by other processors that overlap its extended bounding box.
Therefore, after the *Indexing* step, we have a *Communication* step, where particles which ended their advection in a ghost cell owned by another processor are sent to that processor.

2.2.4 Collision step

Once particles are sorted and indexed in cells, stochastic binary collisions are performed within particles residing in the same cells.
Actually, consider the collision between two particles labeled by 1 and 2, with mass m_1 and m_2 and initial velocities $\mathbf{v}_1$ and $\mathbf{v}_2$, respectively.
Then, the goal of the *Collision* step is to determine the post-collision velocities $\mathbf{v}_1^*$ and $\mathbf{v}_2^*$

satisfying conservation of momentum

$$m_1\mathbf{v}_1 + m_2\mathbf{v}_2 = m_1\mathbf{v}_1^* + m_2\mathbf{v}_2^* = (m_1+m_2)\left[\frac{m_1\mathbf{v}_1^* + m_2\mathbf{v}_2^*}{m_1+m_2}\right] = (m_1+m_2)\mathbf{v}_{cm}^* \ , \tag{2.40}$$

together with the conservation of energy

$$m_1|\mathbf{v}_1|^2 + m_2|\mathbf{v}_2|^2 = m_1|\mathbf{v}_1^*|^2 + m_2|\mathbf{v}_2^*|^2 \ . \tag{2.41}$$

Moreover, the magnitude of the relative velocity should remain unchanged

$$|\mathbf{v}_r| = |\mathbf{v}_1 - \mathbf{v}_2| = |\mathbf{v}_r^*| = |\mathbf{v}_1^* - \mathbf{v}_2^*| \ . \tag{2.42}$$

In order to determine the post-collision relative velocity $\mathbf{v}_r^*$, let's focus on the collision solid angle. Several approaches are available in the literature, the simplest one being the so-called Hard Sphere (HS) model for which fully elastic interactions between two particles are assumed to occur. Actually, for the HS model, the scattering is isotropic in the center of mass frame of reference, that is

$$\mathbf{v}_r^* = |\mathbf{v}_r|\left[(\sin\chi\cos\phi)\hat{\mathbf{x}} + (\sin\chi\sin\phi)\hat{\mathbf{y}} + (\cos\chi)\hat{\mathbf{z}}\right] \ , \tag{2.43}$$

where the elevation angle χ is sampled from a distribution of the form:

$$P_\chi(\chi)d\chi = \frac{1}{2}\sin\chi d\chi \tag{2.44}$$

while the azimuthal angle ϕ is sampled from the uniform distribution between 0 and 2π. After the solid angle has been determined, the post-collision velocities are given by

$$\mathbf{v}_1^* = \mathbf{v}_{cm} + \left(\frac{m_2}{m_1+m_2}\right)\mathbf{v}_r^* \qquad \mathbf{v}_2^* = \mathbf{v}_{cm} - \left(\frac{m_1}{m_1+m_2}\right)\mathbf{v}_r^* \ . \tag{2.45}$$

See, for example, Figure 2.5 for a sketch representing collisions between particles.

Notice that, at macroscopic scales, the HS model implies that the dynamic viscosity scales with the temperature as $\mu \sim T^\omega$ with $\omega = 1/2$. However, experimentally, we measure $\omega \simeq 3/4$. The reason can be traced back to the fact that, within the HS model, the collision cross-section is $\sigma_T = \pi d^2$, where the molecular diameter $d = d_{ref}$ is independent of the relative velocity.
In order to recover the experimental dependency, more refined collision models have been proposed, such as the Variable Hard Sphere (VHS) and Variable Soft Sphere (VSS).
For example, within the VHS model, the diameter of the molecules d, and therefore the collision cross-section $\sigma_T = \pi d^2$ is expressed as an inverse power law function of the relative velocity, as

$$d = d_{ref}\left[\frac{(2k_BT_{ref}/(m_r\mathbf{v}_r\cdot\mathbf{v}_r))^{\omega-1/2}}{\Gamma(5/2-\omega)}\right]^{1/2} \ , \tag{2.46}$$

where

$$m_r = \frac{m_1m_2}{m_1+m_2} \ . \tag{2.47}$$

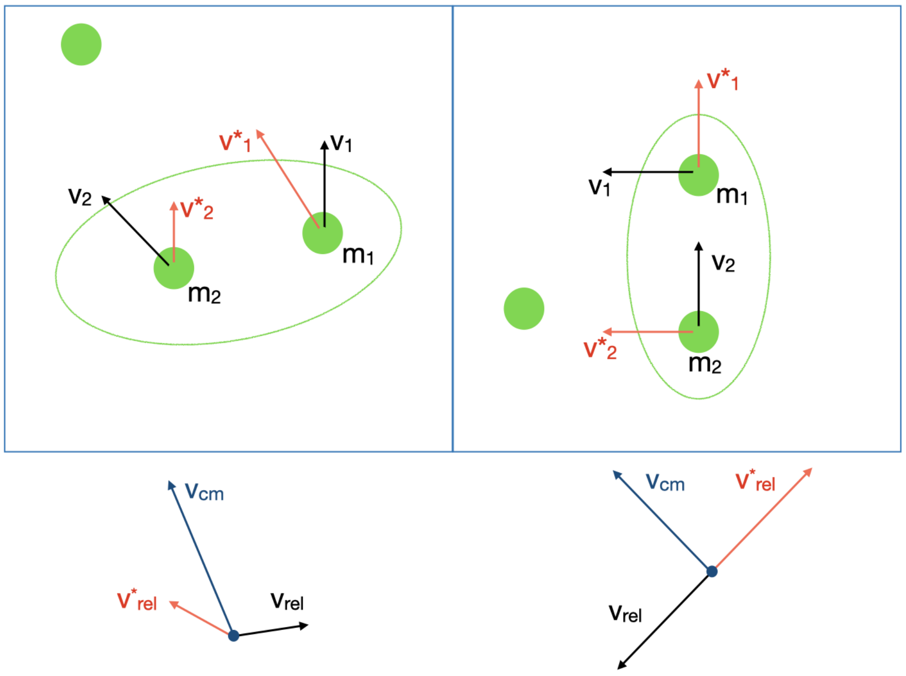

Figure 2.5: Sketch representing collisions between particles of equal mass in two neighboring cells. In the figure, the pre-collision velocities are labeled with $\mathbf{v}_1$, $\mathbf{v}_2$, post-collision velocities with $\mathbf{v}_1^*$, $\mathbf{v}_2^*$. The velocity of the center of mass as well as the pre- and post-collision relative velocities are also indicated below and labeled with $\mathbf{v}_{cm}$, $\mathbf{v}_r$ and $\mathbf{v}_r^*$, respectively. Note that in DSMC, it is not required that the two particles touch in order to have a collisional event.

As a result, on macroscopic scales, we recover the following relation for the dynamic viscosity:

$$\mu = \mu_{ref} \left(\frac{T}{T_{ref}} \right)^{\omega} \, . \tag{2.48}$$

Finally, we need to determine the total number of collisions to be performed in each cell. Again, several approaches are possible. As mentioned before, in this work we have considered the NTC algorithm, setting the number of attempted collisions M_{cand} between two randomly selected partners as

$$M_{cand} = \frac{1}{2} \frac{N(N-1)\Delta t}{\Delta V} (\sigma_T |\mathbf{v}_r|)_{max} \, , \tag{2.49}$$

where N is the instantaneous number of particles in the cell and ΔV is the cell volume.
For each cell, we keep track of the maximum value of the product $(\sigma_T |\mathbf{v}_r|)_{max}$ and continuously update such quantity.
Then, an attempted collision is accepted with probability:

$$P(\mathbf{v}_1, \mathbf{v}_2) = \frac{\sigma_T |\mathbf{v}_r|}{(\sigma_T |\mathbf{v}_r|)_{max}} \tag{2.50}$$

expressing the fact that pairs with larger relative velocity are more probable to collide.

2.2.5 Sampling and averaging flow properties

For the last step of the main time loop, recall that we are interested in hydrodynamics variables, so, we finally need to measure the flow macroscopic properties by statistical sampling.
Actually, consider a generic microscopic property γ. For volume-averaged measurements, given the number of acquired samples S and the number of particles $N(t)$ residing in the cell with volume ΔV, the average moments can be evaluated as

$$\langle \gamma(\mathbf{v}) \rangle = \frac{1}{S} \sum_{s=1}^{S} \frac{1}{\Delta V} \sum_{i=1}^{N(s,t)} \gamma(\mathbf{v}_i(t)) \ . \tag{2.51}$$

Using this relation, we can finally compute

$$\begin{aligned}
\rho &= \langle m \rangle \\
\mathbf{u} &= \langle \mathbf{v} \rangle \\
T &= \frac{m}{3k_B} \left\langle (\mathbf{v}-\mathbf{u})^2 \right\rangle = \frac{m}{3k_B} \langle \mathbf{v}\cdot\mathbf{v} + \mathbf{u}\cdot\mathbf{u} - 2\mathbf{v}\cdot\mathbf{u} \rangle = \frac{m}{3k_B} \left(\langle \mathbf{v}\cdot\mathbf{v} \rangle - \mathbf{u}\cdot\mathbf{u} \right) \\
P_{ij} &= \rho \left\langle (v_i - u_i)(v_j - u_j) \right\rangle \\
q_i &= \frac{m\rho}{2} \left\langle (v_i - u_i)(\mathbf{v}-\mathbf{u})^2 \right\rangle \ .
\end{aligned} \tag{2.52}$$

Note that two successive samples should not be temporally correlated, and so the sampling operation shall not be performed at each time step, but rather with a prescribed frequency (typically one sampling operation every 10 time steps).

As a concluding remark, note that for a large number of particles per cell and for sufficiently small time and space discretizations, it is possible to show [25] that the solution provided by the DSMC method is equivalent to the one given by the Boltzmann equation.

Chapter 3

GPU programming with OpenACC

As mentioned in Chapter 1, current supercomputers feature heterogeneous architectures, combining multi-core CPUs with hardware accelerators such as GPUs, digital signal processors (DSPs) or field-programmable gate arrays (FPGAs), all characterized by a high level of parallelism.
Moreover, memories show hierarchical structures, where the main memory (typically DDR for CPUs) is accompanied by multiple layers of cache memory.
In fact, parallelism shows up at different levels, with varying degrees of coarseness [26, 27].

The Flynn's taxonomy

At hardware level, we can classify different architectures according to the Flynn's taxonomy

- Single Instruction Single Data (SISD)
- Single Instruction Multiple Data (SIMD)
- Multiple Instruction Single Data (MISD)
- Multiple Instruction Multiple Data (MIMD)

The simplest one, SISD, corresponds to computer architecture (the von Neumann architecture) in which a single uni-core processor executes a single instruction at a time and fetches or stores one data at a time.

On the other hand, SIMD describes computers with multiple processing units capable of performing the same operation on different data elements simultaneously. As a practical example, consider the situation where hardware registers are loaded with numbers, and a mathematical operation is performed on all registers simultaneously.

MIMD architectures, including multi-core superscalar processors and distributed systems, are multiple autonomous processors simultaneously executing different instructions on different data. They can be of either shared-memory (if block of Random Access Memory (RAM) can be accessed by several CPUs in a multiprocessor computer system) or distributed-memory (if each processor is coupled with its own memory and explicit communication between the various processor-memory pairs of the system is required).

Finally, applications for MISD architecture are much less common than MIMD and SIMD, as the latter two are often more appropriate for common data parallel techniques.

The role of GPUs

With this complex situation in mind, it is clear that programmers must choose a programming model that balances the need for performance with the need for portability, otherwise their applications may not be able to run on future architectures.

As far as performance is concerned, contemporary high-performance computing is typically associated with GPGPU. Actually, a significant milestone was the discovery in 2003 that GPU-based approaches for the solution of general linear algebra problems ran faster on GPUs than on CPUs.
However, at that time, GPU handled computations only for computer graphics, so these early efforts required reformulating scientific problems in terms of graphics primitives.

The situation changed in 2006 when NVIDIA released CUDA, allowing programmers to ignore such graphical paradigm in favor of common high-performance computing concepts. By adding a relatively small number of keywords to standard C, in order to take advantage of the CUDA architecture, NVIDIA was able to obtain orders-of-magnitude of performance improvement over the previous state-of-the-art implementations.

How was it possible?
The basic idea behind GPU architectures is rather simple: remove everything that makes a single instruction run fast on modern CPUs (such as out-of-order control logic, complex branch predictions, memory prefetching, etc.), instead invest the saved transistors into more copies of simple cores, that can process thousands of threads simultaneously.
Briefly, on one hand we have the CPU, designed following a latency-oriented vision and equipped with a large cache memory (saving data that is accessed frequently thus reducing the long latency) and a sophisticated logic control unit. On the other hand, we have the GPU, that is an example of throughput-oriented design, as it acts to maximize the throughput rather than investing in latency.
In other words, the CPU is great for control-intensive tasks, while the GPU is great for data-parallel computation-intensive tasks.

Therefore, a GPU works efficiently for algorithms that process large blocks of data in parallel, in the Single Instruction Multiple Threads (SIMT) execution model, where SIMD is combined with multi-threading. Actually, in the SIMT framework, each thread can access its own registers, can load and store from divergent addresses, and can follow divergent control paths, thus performing the same instruction on different data.
Notice that, currently, GPUs are not standalone platforms, but rather co-processors that operate in conjunction with CPUs, through PCI-Express buses. For this reason, the CPU is called the host and the GPU is called the device, see Figure 3.1.

The NVIDIA Volta architecture

In order to describe GPU architectures, let's focus on the NVIDIA Volta V100 released in 2017 and included in CINECA Marconi100 supercomputer. For details, see [28].

First of all, recall that GPUs are capable of executing multiple threads simultaneously. At hardware level, such execution is supported by an array of highly threaded Streaming Multiprocessors (SMs), each containing thousand of registers, several caches, warp schedulers, and floating-point execution cores. Figure 3.2 shows a full GV100 GPU.

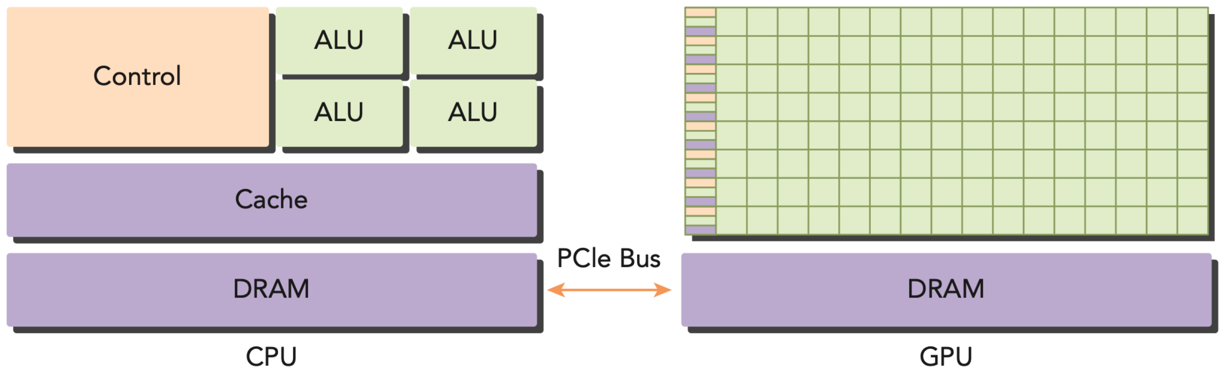

Figure 3.1: Representation of a heterogeneous system [26].

Actually, the NVIDIA Tesla V100 contains 80 SMs, each divided into 4 processing blocks consisting of: 8 FP64 Cores, 16 FP32 Cores, 16 INT32 Cores, two mixed precision Tensor Cores for deep learning and matrix arithmetic, 8 LD/ST units, one Special Function Units, one Warp Scheduler, one Dispatch Unit, L0 Instruction Cache, and 64 KB register file, see Figure 3.3.

With the impressive number of 21.1 billions transistors on a die of 815 mm^2, the delivered peak performances are

- 7.8 TFLOP/s of double precision floating-point (FP64) performance.
- 15.7 TFLOP/s of double precision floating-point (FP32) performance.
- 125 TFLOPS/s of mixed precision matrix-multiply-and-accumulate.

Each SM also features 128KB of L1 data cache combined with shared memory. The amount of cache dedicated to shared memory can be set at runtime with the range of available capacities going from 0 to 96 KB for SM.
Merging L1 and shared memory together allows L1 cache operations to benefit from shared memory performance. Actually, shared memory provides high bandwidth and low latency, but requires more coding effort since the programmer needs to explicitly manage this memory. Therefore, while shared memory remains the best choice for maximum performance, the Volta L1 design enables programmers to get excellent performance quickly. In addition, there is a 6144 KB unified L2 cache for data, instructions, and constant memory.

Each GPU currently comes with gigabytes DRAM, referred to as Global Memory. For a Tesla V100 the High Bandwidth Memory 2 (HBM2) global memory can be either 32 GB or 16 GB[1] with 900 GB/s of peak memory bandwidth.
Actually, GPUs are designed for stream or throughput computing, and device memory supports very high data bandwidth using a wide data path. This, in turns, comes with severe effective bandwidth degradation for strided accesses.

Every GPU is connected to a CPU (host) through a high-speed I/O bus. While the peak bandwidth guaranteed by PCIe is typically much lower than that of global memory, the newer generations support NVLink 2.0, a CPU-GPU or GPU-GPU interconnect that can deliver a

[1]The 16 GB configuration is the one hosted on Marconi100.

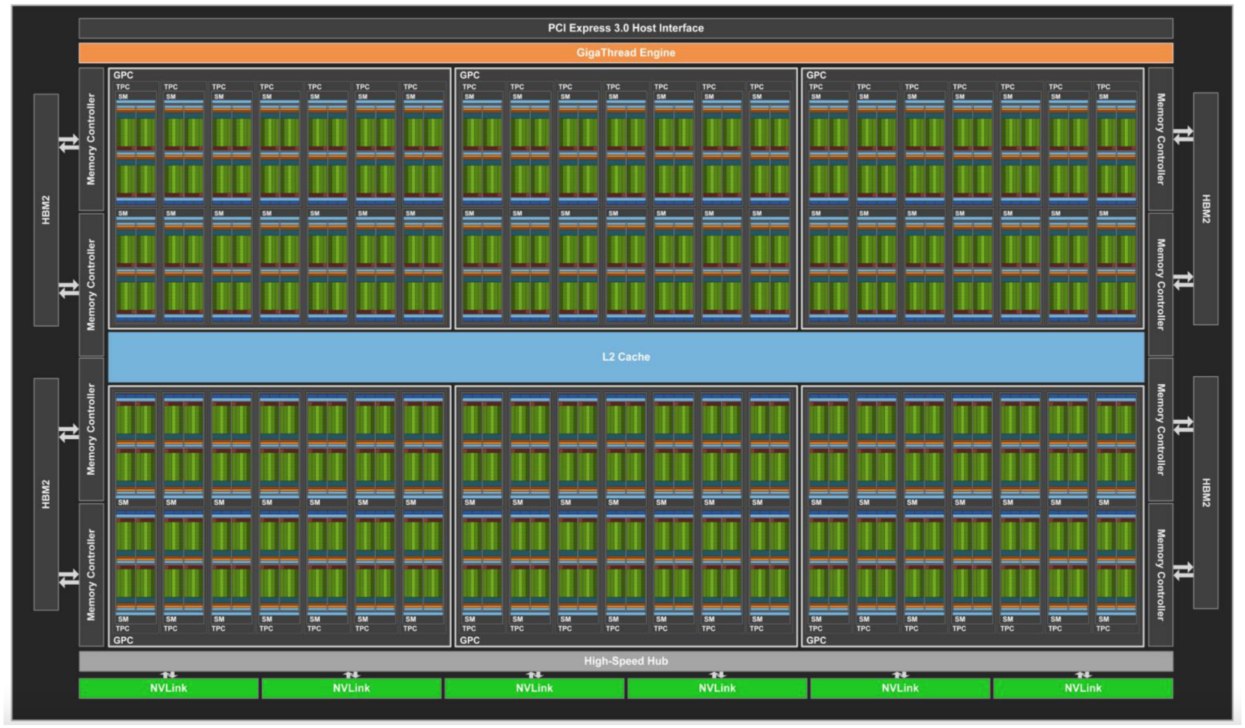

Figure 3.2: A Full NVIDIA GV100 Architecture [29].

unidirectional bandwidth of up to 25 GB/s.
Moreover, the introduction of NVSwitch, supporting up to 18 NVLinks, enables upto 900 GB/s bidirectional bandwidth between GPUs in high-end HPC facilities.

Finally, the thermal design power (TDP), i.e. the maximum amount of heat generated by the GPU that the cooling system is designed to dissipate under any workload, is 300 watts.

3.1 Programming models

As mentioned in the previous sections, taking advantage of hundreds of cores, GPUs can process thousands of software threads in parallel. However, to exploit such an extensive execution activity, an efficient organizational structure of the threads is needed.
This structure, in turns, may create several challenges for programmers, like mapping algorithms onto thread hierarchy as well as designing data, both in global memory and shared memory, to maximize coalesced memory access for the threads. For these reasons, porting programs to GPUs with low-level APIs (such as CUDA and OpenCL) may be difficult, time consuming, and typically needs extensive modifications to the original source code, decreasing code readability.

In the next sessions, we are first going to describe the most popular, low level, language based programming model for GPU, CUDA. Then, we are going to introduce the more high-level, open standard, directive-based OpenACC, that is becoming one of the most promising tool for developing or quick porting scientific applications to hybrid systems.

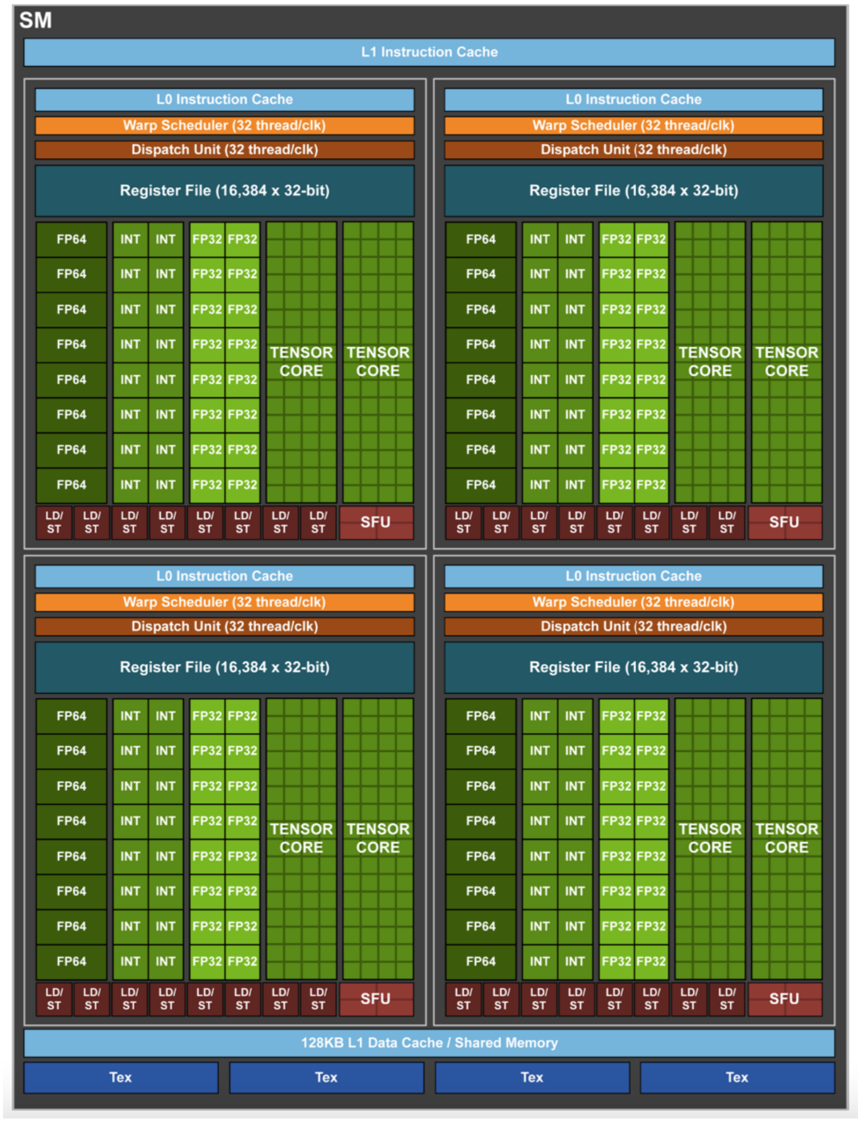
SM
L1 Instruction Cache
L0 Instruction Cache
Warp Scheduler (32 thread/clk)
Dispatch Unit (32 thread/clk)
Register File (16,384 x 32-bit)
FP64
INT
FP32
TENSOR CORE
LD/ ST
SFU
128KB L1 Data Cache / Shared Memory
Tex

Figure 3.3: The V100 streaming multiprocessor (SM)[29].

3.1.1 The CUDA programming model

CUDA is a general purpose parallel computing platform developed by NVIDIA to allow programmers to exploit the computing power of GPUs.
Basically, it provides a bridge between an application and its implementation on GPU hardware by means of an API representing an abstraction of the GPU architecture.
In fact, there are two very similar CUDA APIs, the driver API and the runtime API, that can be used interchangeably for the most part. While the runtime API makes device code management easier by providing implicit initialization, context management, and module management, the CUDA driver API allows for more extensive control of the GPU but requires more programming effort.

Let us see the main concepts of the CUDA programming model.
To execute any CUDA program, there are three main steps [30]:

- Copy the input data from host memory to device memory (host-to-device transfer).
- Load the GPU program and execute, caching data on-chip for performance.
- Copy the results from device memory to host memory (device-to-host transfer).

The portion of code that runs exclusively on the GPU is the kernel, where it is executed K times in parallel by K different CUDA threads.
As shown in Figure 3.4, CUDA threads are organized in a 2-level hierarchy: threads are grouped into CUDA blocks and CUDA blocks are grouped into grids. Actually, all threads created by a single kernel launch, collectively, form a grid.

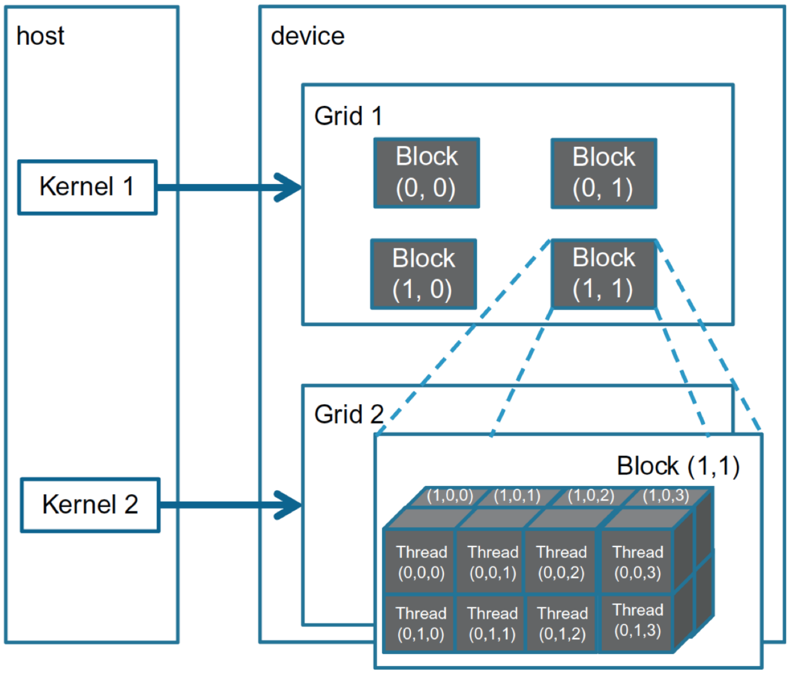

Figure 3.4: Threads hierarchy; 2D and 3D blocks grid [26].

CUDA defines built-in 3D variables for identifying threads and blocks. Specifically, each thread in the block has its own thread index called `threadIdx`. Similarly, blocks are also indexed using the in-built 3D variable called `blockIdx`. Threads belonging to the same block have the same `blockIdx` value. Notice that CUDA architectures, including Tesla V100, limit the numbers of threads per block (1024 threads per block limit).
Threads within a block can cooperate by sharing data through shared memory and can coordinate memory accesses by synchronizing their execution. However, thread blocks are required to execute independently. This need of independence means that the blocks are scheduled in any order through any number of cores.

A block is also divided into warps generally consisting of 32 threads, that are executed simultaneously by one SM. Basically, each thread in a warp executes the same instruction on different data, in a SIMT fashion. As thread blocks terminate, new blocks are launched on the vacated SMs. Actually, one SM can run several concurrent CUDA blocks depending on

the resources needed by CUDA blocks.
Moreover, since all threads in a warp must execute identical instructions on the same cycle, warp divergence can cause significant performance degradation if threads in the same warp take different paths. Indeed, the warp serially executes each branch path, disabling threads that don't take that path.
For details regarding the CUDA programming model, see for instance [31, 32, 33, 34].

3.1.2 The OpenACC programming model

As mentioned in Chapter 1, in this work we focus on OpenACC, that we are now going to briefly introduce, following [35, 36, 37, 38, 39].
To summarize the CUDA programming model, in GPU computing we use as many threads as data items/tasks we have to process. As a consequence, we need a rule to match a thread to a data item/task that this thread needs to process. Unfortunately, this is a common source of errors and frustration for CUDA developers.
It would be simpler if the compiler took care of this task by means of a collection of directives used to expose parallelism in the code, without programmers prior extensive knowledge on the specifics of the accelerator being used.

OpenACC is an open standard, initially developed in 2011 by CAPS, CRAY, PGI, and NVIDIA, providing such directive-based, portable programming model for accelerators.
Actually, in order to ensure portability to all available as well as future computing architectures, OpenACC defines an abstract model for accelerated computing, exposing multiple levels of parallelism together with a hierarchy of memories (with varying degrees of speed and addressability).
This is particularly relevant for the offloading of both computation and data from the host to an accelerator device. Actually, recall that on heterogeneous systems host and accelerators may have completely different architectures as well as different memory spaces. In this case, the compiler implicitly maps each component of this abstract model into the specific structure of the target architecture.

There are three layers of parallelism in OpenACC: vector threads, workers and gangs.
Vector threads represent the finest granularity, performing the same instruction on multiple data elements in a SIMT-fashion. Notice that, if there are fewer data than the length of the vector, the operation is performed on null data and the results are discarded.
Each vector is processed by a worker. All workers within a gang can share resources, such as cache memory.
A gang, comprised of one or multiple workers, represents the coarse-grained parallelism. The OpenACC model associates a cache memory for each gang that can be used by all workers and vectors within the gang. Actually, gangs work independently, do not share memory, and do not support synchronization.

In C and C++, directives take the form of a pragma. For example, in order to generate a compute region to be executed on the accelerator, we can can annotate the code with

```
#pragma acc parallel
{
  //
}
```

where the `parallel` directive is used for explicit parallelism, i.e. creating a number of parallel threads that execute the parallel region redundantly.
Alternatively, we can rely on compiler automatic parallelization techniques to identify operations that are safe to parallelize using the `kernels` directive

```
#pragma acc kernels
{
  //
}
```

Notice that we can execute serially a portion of code on the device using the directive

```
#pragma acc serial
{
  //
}
```

Upon the creation of such regions, the compiler also creates an implicit data region. All variables used inside parallel or kernels regions will be treated as implicit variables if they are not present in any `data` clauses, meaning that, by default, all the data required for the computation is copied from the host to the device once entering the region, and copied back at the end of the parallel region.

Remarkably, we can minimize and synchronize data movements between host and device memory with the `data` directive. Such directive facilitates the sharing of data between multiple parallel regions, thus optimizing data locality. In particular, we can control the data flow with the following clauses:

- `copy` allocates space for the specified variables on the device, copying data on the device at the beginning of the region. At the end of the region, it copies data back to the host, releasing the allocated memory on the device.

- `copyin` allocates space for the specified variables on the device, copying data on the device at the beginning of the region. At the end of the region, it releases the allocated memory on the device without copying the data back to the host.

- `copyout` allocates space for the specified variables on the device at the beginning of the region. At the end of the region, it copies them from the device to the host, releasing the allocated memory on the device.

- `create` allocates space for the specified variables on the device at the beginning of the region. At the end of the region, it releases the allocated memory on the device.

- `present` specifies that vars in var-list are in shared memory or are already present in the current device memory and should be used without any further redundant data movement.

- `deviceptr` is used to declare that the pointers in var-list are device pointers, allowing interaction with other APIs for handling the accelerator memory.

Notice that there are situations when structured data regions are not sufficient to tell the compiler the data layout. This is the case, for example, if complex structures, with dynamically allocated data members, are allocated and freed in different scopes. As another example, when using C++ classes, data transfer on the device (using the class constructors and destructors) and data usage (with the class methods) are performed in very different places.

To handle such cases OpenACC introduced unstructured data regions, where the `enter data` directive defines the start of an unstructured data region, supported by the `create` and `copyin` clauses to specify how data should be created on the device.
On the other hand, the `exit data` directive defines the end of an unstructured data region. The `copyout` and `delete` clauses can be used to identify precisely when data should be copied back or deallocated from the device.

In order to synchronize data between host and device memories, we can rely on the `update` directive. More specifically, we can move the specified data from device to host with the `device` clause and viceversa with the `self` clause.

In general, the main focus in parallel programming is represented by loops.
The OpenACC `loop` directive applies to the loop which immediately follows and can be used to provide the compiler with additional information using the following clauses:

- `collapse` specifies how many tightly nested loops are associated with the `loop`.
- `private` declares that a copy of each item in the var-list will be created for each gang.
- `firstprivate` declares that a copy of each item in the var-list will be created for each gang, and that the copy will be initialized with the value of that item on the local thread when a parallel or serial construct is encountered.
- `reduction` performs a reduction operation on a chosen variable by creating a private copy of the variable for each loop iteration returning the result of the combined result of the reduction at the end of the loop.
- `independent` tells the implementation that the loop iterations must be data independent, except for vars which appear in a reduction clause or which are modified in an atomic region. This allows the implementation to generate code to execute the iterations in parallel with no synchronization.
- `tile(N[,M,...])` specifies that the implementation should split each loop in the loop nest into two loops, with an outer set of tile loops and an inner set of element loops. Basically, it breaks the next one or more loops into tiles based on the provided dimensions.
- `seq` executes the loop or loops sequentially.

Moreover, we can specify how the compiler should map the loop iterations into hardware parallelism by using the `gang`, `worker` and `vector` clauses.
Mapping the high-level gang, worker, vector concepts to the loop constructs is vendor dependent. For example, considering the CUDA execution model, gangs correspond to CUDA blocks, vectors are mapped into threads and the number of workers can be fixed to match the warp size.

In OpenACC, we can overlap data movements with computation on the host, on the device, or both by means of the `async` clause. In this way, we can tell the compiler that, once a given operation has been sent to the accelerator, the host can asynchronously process the next instruction rather than waiting for its completion. The flow of asynchronous operations can be controlled using the `wait` directive to stop the execution until all operations are completed.

Finally, the OpenACC API can interoperate with other accelerator programming models, like CUDA or OpenCL. For example, suppose we create an array using OpenACC and we want to call a CUDA function performing some operations on such array. The `host_data use_device` construct makes the address of device data available on the host, so we can pass it to functions expecting CUDA device pointers. In other words, we can directly access the device copy of the arrays instead of the host copy.
This construct is also useful when passing in device pointers to MPI calls.

Porting Cycle

As a concluding remark, note that the best practice when porting scientific applications to GPUs with OpenACC is to proceed incrementally. First, it is important to understand the most time-consuming parts of the application. Then, we can use OpenACC directives to accelerate these regions on the target device. Finally, we can optimize data locality, to remove unnecessary data migrations between the host and device, as well loop optimizations to maximize performance on a given architecture.

Note that, after each modification, it is important to check the program results for correctness. In order to proceed incrementally, we can modify one routine at a time by updating data on device just before the function call, then performing the execution on device, and finally update back data on the host.

Chapter 4

DSMC on GPUs with OpenACC

In this chapter, we describe our implementation of a multi-GPUs DSMC solver.
Over the years, several DSMC codes have been developed and optimized for various architectures such as GPUs and many-threaded CPUs. Examples include DAC [40], SPARTA [41, 42], MGDS [43], dsmcFoam [44], Monaco [45, 46] or SMILE [47] to name but a few.
Our work builds on top of a simplified version of the *fm_dsmc* code developed in [14, 48, 49], written in C programming language and using the generic compiler/profiling library *ftmake* [50, 51], establishing a baseline for performance comparisons. However, several major modifications have been performed, since different data structure and different algorithms are needed to enable threaded parallelism.

Within this framework, we consider a fluid flow in a $L_x \times L_y \times L_z$ Cartesian grid, with the only boundary lying on the surface of the six outer-faces of the box, where we assume periodic boundary conditions. Moreover, in order to simplify MPI communications, each processor's grid is enveloped in a halo surface.

The program takes as input a `param.in` file, where we can specify the parameters listed in Table 4.1. The schematic algorithm of the main calls is shown in Algorithm 1.

Parameter	**Description**
`dsmc_sx`, `dsmc_sy`, `dsmc_sz`	Number of physical cells L_x, L_y, L_z

`dsmc_boundary_x_m`, `dsmc_boundary_x_p`
`dsmc_boundary_y_m`, `dsmc_boundary_y_p`
`dsmc_boundary_z_m`, `dsmc_boundary_z_p`
	Box size in the x direction
	Box size in the y direction
	Box size in the z direction

`dsmc_steps_segregate`	Number of time steps
`dsmc_dt`	Duration of each time step measured in s
`dsmc_steps_sampling`, `dsmc_steps_sampling_ndiag`	Control the sampling step
`dsmc_molecules_number_total`	Number of DSMC molecules
`dsmc_number_density`	Number density of the real molecules in m^{-3}
`dsmc_mass`, `dsmc_diameter`	Mass (Kg) and diameter (m) of real particles
`dsmc_omega`	Parameter ω for the VHS model
`dsmc_temperature`	Temperature in K

`dsmc_forced_decomposition_procx`
`dsmc_forced_decomposition_procy`
`dsmc_forced_decomposition_procz`
	Force a MPI decomposition along the x direction
	Force a MPI decomposition along the y direction
	Force a MPI decomposition along the z direction

Table 4.1: Parameters specified in the `param.in` file.

Algorithm 1 Schematics of the DSMC Algorithm

```
1:  Initialization
2:  Move grid-cells and particles data from host to device
3:  for istep_DSMC ← 0 to dsmc_steps_segregate do
4:      Deterministic ballistic motion
5:      Indexing
6:      Periodic boundary conditions
7:      Communication
8:      Collision
9:      if istep_DSMC ≥ dsmc_steps_sampling_ndiag then
10:         if istep_DSMC%dsmc_steps_sampling == 0 then
11:             Data Sampling
12:         end if
13:     end if
14: end for
15: Move grid-cells and particles data from device to host
16: Dump Data
17: Finalization
```

During the initialization, we start by handling GPUs affinity, and we rely on process placement with one rank per GPU, as shown in the snippet.

```
#if defined (DSMC_OPENACC_GPU)
 int ngpu = acc_get_num_devices(acc_device_nvidia);
 int igpu = me % ngpu;
 acc_set_device_num(igpu, acc_device_nvidia);
 acc_init(acc_device_nvidia);
 if(AMIROOT) fprintf(stdout, "NUM GPU: %d\n", ngpu);
 fprintf(stdout, "GPU ID: %d, PID: %d\n", igpu, me);
#endif /*DSMC_OPENACC_GPU*/
```

Then, we proceed with the allocation of the grid cells and the DSMC molecules both on host (using the C dynamic memory allocation function `malloc`) and device (using the `#pragma acc enter data create (...)` directive). After the initialization of cells and DSMC molecules on the host, we move grid cells and particles data to device by means of the `#pragma acc update device (...)` directive.

4.1 Data Layout

The basic structure for the DSMC code is the particle, characterized by the position $\mathbf{r} = (x, y, z)$ and velocity $\mathbf{v} = (v_x, v_y, v_z)$.
Clearly other information is needed such as the number of particles per each cell, possibly the mass of the particle and so on, but, for the moment, let's focus for simplicity just on positions and velocities represented by six double precision floating point numbers per particle.

For the DSMC algorithm, the canonical data structure that we would naturally use is the Array of Structures (AoS), which corresponds to a single array `dsmc_particle_aos` with one element per particle. Each particle is represented by a data structure used to store the six features $(\mathbf{r}_i, \mathbf{v}_i)$ associated to the particle with index i (i.e. `dsmc_particle_aos[i].x`, `dsmc_particle_aos[i].y`, `dsmc_particle_aos[i].z`, `dsmc_particle_aos[i].ux`, `dsmc_particle_aos[i].uy`, `dsmc_particle_aos[i].uz`).

Remarkably, while the AoS can represent a good choice for serial CPU implementations, as shown in [52] in the context of Lattice Boltzmann methods, the same is not true for GPU architectures due to poor memory access patterns.
In fact, within the AoS framework all the data associated with one particular particle are contiguous in memory, but the features (e.g. the x coordinate) of different particles, that we want to process in a SIMT fashion, are not contiguous, which results in non-unit strided access in memory, thus halting the SIMT instruction sets.
See 4.1 for a graphical representation of the memory organization of the AoS data structure.

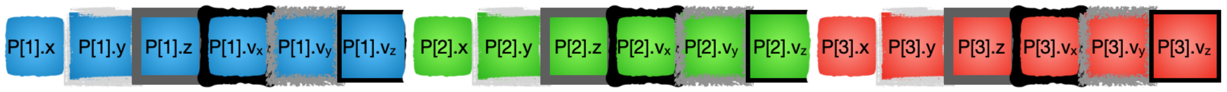

Figure 4.1: Graphical representation of the memory organization of the AoS data structure. In the Figure, we show three particles (described by different colors), each with six features (described by different borders). Contiguous blocks represent contiguous data in memory.

A different approach is represented by the Structure of Arrays (SoA) format.
As an example, in our simple case, `dsmc_particle_soa` consists of a structure with six arrays, each one corresponding to particles' coordinates and velocities (i.e. `dsmc_particle_soa.x[i]`, `dsmc_particle_soa.y[i]`, `dsmc_particle_soa.z[i]`, `dsmc_particle_soa.ux[i]`, `dsmc_particle_soa.uy[i]`, `dsmc_particle_soa.uz[i]`).
Actually, the SoA organization is the appropriate layout for SIMD and SIMT architectures since it guarantees better coalescence of global memory accesses and it is therefore the mandatory choice for exploiting GPU architectures [53, 54, 55].
See 4.2 for a graphical representation of the memory organization of the SoA data structure.

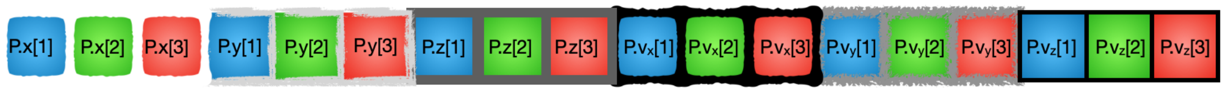

Figure 4.2: Graphical representation of the memory organization of the SoA data structure. In the Figure, we show three particles (described by different colors), each with six features (described by different borders). Contiguous blocks represent contiguous data in memory.

Besides the particles' positions and velocities, we need to store other information for the execution of the DSMC code. Actually, the grid is composed by several cells, and we need to access the particles residing in each cell to perform stochastic binary collisions.
As a consequence, in order to represent the particles, we consider the structure `dsmc_particle_type_soa`, with 2D arrays (one 2D array per feature), along with a 1D vector with counts of particles per cell.
Each 2D array is composed of the feature of the i-th particle in the cell (rows) versus the j-th cells in the grid (columns), as depicted in Figure 4.3.
Notice that, to avoid memory reallocation, we consider a fixed maximum number of particles per cell — `MAX_PARTICLES_PER_CELL`.

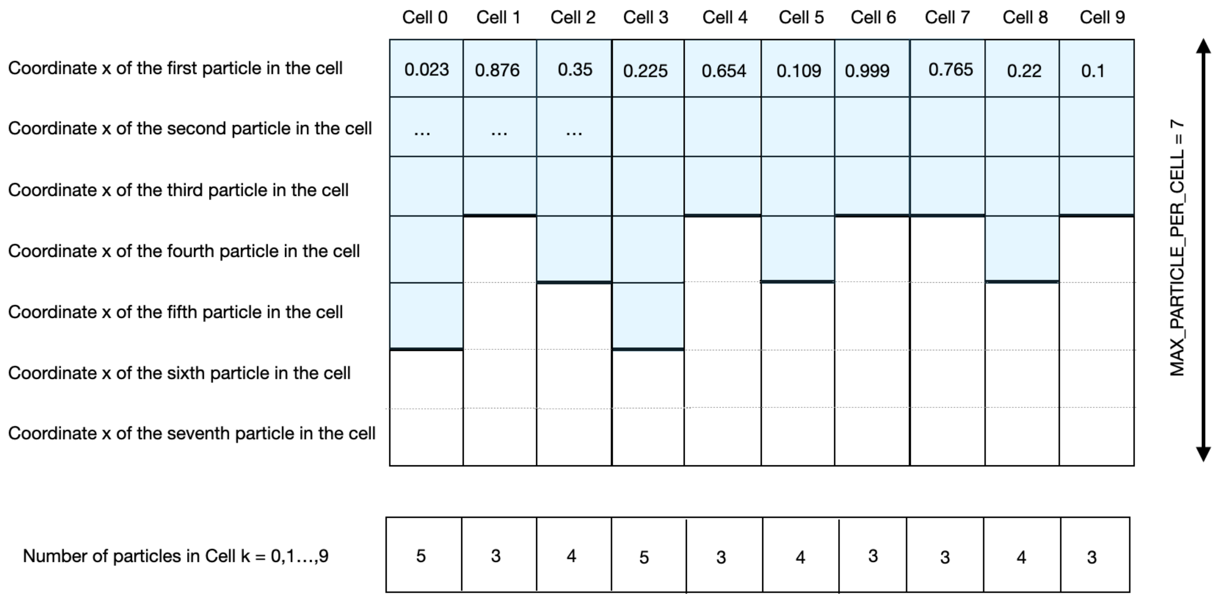
Cell 0
Cell 1
Cell 2
Cell 3
Cell 4
Cell 5
Cell 6
Cell 7
Cell 8
Cell 9
Coordinate x of the first particle in the cell
Coordinate x of the second particle in the cell
Coordinate x of the third particle in the cell
Coordinate x of the fourth particle in the cell
Coordinate x of the fifth particle in the cell
Coordinate x of the sixth particle in the cell
Coordinate x of the seventh particle in the cell
0.023
0.876
0.35
0.225
0.654
0.109
0.999
0.765
0.22
0.1
...
...
...
MAX_PARTICLE_PER_CELL = 7
Number of particles in Cell k = 0,1...,9
5
3
4
5
3
4
3
3
4
3

Figure 4.3: Visual representation of the `dsmc_particle_type_soa` data structure. Notice that here we have presented only the 2D array for the `x` coordinate. But, in reality, there is a 2D array for each feature (i.e. similar matrices also for `y`, `z`, `ux`, `uy`, `uz`, `dx`, `dy`, `dz`, . . .).

Moreover, as a result of the deterministic ballistic motion, the new coordinates of the particles might be associated to a different cell. As we will explain, information regarding the destination cell can be particularly useful in the *Indexing* step.
For this reason, we additionally store three integers for each particle i, implemented in the SoA format as arrays of integers `dsmc_particle_soa.dx[i]`, `dsmc_particle_soa.dy[i]`, `dsmc_particle_soa.dz[i]`.
For neighborhood cell displacement, `dx[i]`, `dy[i]`, `dz[i]` $\in \{-1, 0, 1\}$ (where `dx[i]` $=$ `dy[i]` $=$ `dz[i]` $= 0$ if the particle i remains in the same cell).
Finally, other information might be useful, such as the distance that the particles have traveled during the simulation (in order to compute the mean free path), or the mass of the particles (for instance if we have a mixture of gasses).
Therefore, the `dsmc_particle_type_soa` data structure is as follows:

```c
#define MAX_PARTICLES_PER_CELL 64
typedef struct {
  double *x, *y, *z;              /* molecule position */
  int *dx, *dy, *dz;              /* molecule cell variation */
  double *ux, *uy, *uz;           /* molecule velocity components */
  double *distance_traveled;      /* distance travelled by particle */
  double *mass;                   /* molecule mass */
  int *number_particle_per_cell;
} dsmc_particle_type_soa;
```

To access the particles, we perform the following loop

```c
#pragma acc parallel loop present (...) collapse(3)
for (mx = 0; mx < (NX+2); mx++){
  for (my = 0; my < (NY+2); my++){
    for (mz = 0; mz < (NZ+2); mz++){
      int cell_idx = mx * (NZ+2) * (NY+2) + my * (NZ+2) + mz;
      int n = dsmc_particle_soa.number_particle_per_cell[cell_idx];
      int i;
      #pragma acc loop seq
      for (i=0; i < n ; i++) {
        int part_idx = cell_idx + number_cells * i;
        /* For example: */
        dsmc_particle_soa.x[part_idx] += dsmc_particle_soa.ux[part_idx] * dt;
      }
    }
  }
}
```

Recall that the goal of the DSMC code is to evaluate flow macroscopic properties by means of statistical sampling. Such sampling procedure is performed by volume-averaged measurements at the cell level. For this reason, we need a second data structure to store information that pertains to each cell, such as the sampled hydrodynamic moments as well quantities used within the *Collision* step like $(\sigma_T|\mathbf{v}_r|)_{max}$.
Therefore, we define `dsmc_cell_type_soa` SoA data structure as follows:

```c
typedef struct {
  double *sigma_relativevelocity;
  /* product of collision cross-section and relative velocity */

  unsigned long long int *molecule_sum; /* sum of molecules in the cell */
  double *remainder;
  /* remainder when molecules selection number in each cell is rounded */

  double *sum_ux, *sum_uy, *sum_uz; /* sum of molecules velocity components */
  double *sum_ke_x, *sum_ke_y, *sum_ke_z; /* sum of molecules kinetic energy*/
  double *sum_uxux, *sum_uxuy, *sum_uxuz, *sum_uyuy, *sum_uyuz, *sum_uzuz;
  /* sum of molecules velocity components product with cumulative average */

  double *sum_uxsqu, *sum_uysqv, *sum_uzsqw;
  /* sum of molecules peculiar velocities multiplied by
     squared module of peculiar velocity */
} dsmc_cell_type_soa;
```

4.2 Streaming step

During the Streaming step, we need to update the particles' coordinates according to the Euler method, see Algorithm 2.

Algorithm 2 Deterministic ballistic motion

```
 1: for each cell $c$ do        ▷ #pragma acc parallel loop present(...)  collapse(3)
 2:     for each particle $p$ in cell $c$ do        ▷ #pragma acc parallel loop seq
 3:         Save the indices idx, idy, idz that identify the cell $c$ within the grid
 4:         $\texttt{x}(p) \leftarrow \texttt{x}(p) + \texttt{dt} * \texttt{ux}(p)$
 5:         $\texttt{y}(p) \leftarrow \texttt{y}(p) + \texttt{dt} * \texttt{uy}(p)$
 6:         $\texttt{z}(p) \leftarrow \texttt{z}(p) + \texttt{dt} * \texttt{uz}(p)$
 7:         Compute the indices $\texttt{idx}'$, $\texttt{idy}'$, $\texttt{idz}'$ of the cell $c'$ associated to $\texttt{x}(p)$, $\texttt{y}(p)$, $\texttt{z}(p)$
 8:         $\texttt{dx}(p) \leftarrow \texttt{idx}' - \texttt{idx}$
 9:         $\texttt{dy}(p) \leftarrow \texttt{idy}' - \texttt{idy}$
10:         $\texttt{dz}(p) \leftarrow \texttt{idz}' - \texttt{idz}$
11:     end for
12: end for
```

From the implementation perspective, we perform the deterministic ballistic motion with a kernel consisting in a loop over all cells, where each per-grid-cell thread updates the particles' coordinates within the cell $\mathbf{r}_i$, such that schematically $\mathbf{r}_i \leftarrow \mathbf{r}_i + \mathbf{u}_i \ \Delta t$.
At the same time, we record with three integers dx[i], dy[i], dz[i] whether the particle i remained in the same cell or moved in a different cell. Such information will be used in the *Indexing* step. In our implementation, if the particle i remained in the same cell we set $\texttt{dx[i]} = \texttt{dy[i]} = \texttt{dz[i]} = 0$, otherwise, for small enough Δt, we can assume that particles can move at most to neighboring cells.
In this case, we have $\texttt{dx[i]}, \texttt{dy[i]}, \texttt{dz[i]} \in \{-1, 0, 1\} \quad \forall i$, as shown in Figure 4.4.

4.3 Indexing step

As we have seen, during the deterministic ballistic motion, particles can move from one cell to the neighboring cells. Therefore, we now have to re-index these particles. In other words, in the *Indexing* step, we need to update the lists dsmc_particle_soa (of the dsmc_particle_type_soa type) storing the particles' features for each cell. However, particles assigned to different threads may end up in the same grid cell, as shown in Figure 4.5. In this case, threads have to coordinate to ensure safe updates of the arrays.
In fact, there are several possibilities to do so.
For example, we can use atomic operations [42]. However, atomics are typically a bottleneck, and we usually need to reduce such operations to increase application performance.
Actually, we have considered a different strategy.
The main idea is to coordinate the threads by first updating only the particles moving in one direction, say $(\texttt{dx}, \texttt{dy}, \texttt{dz}) = (-1, -1, -1)$.
Once all the threads have completed the update in the first direction, we proceed updating

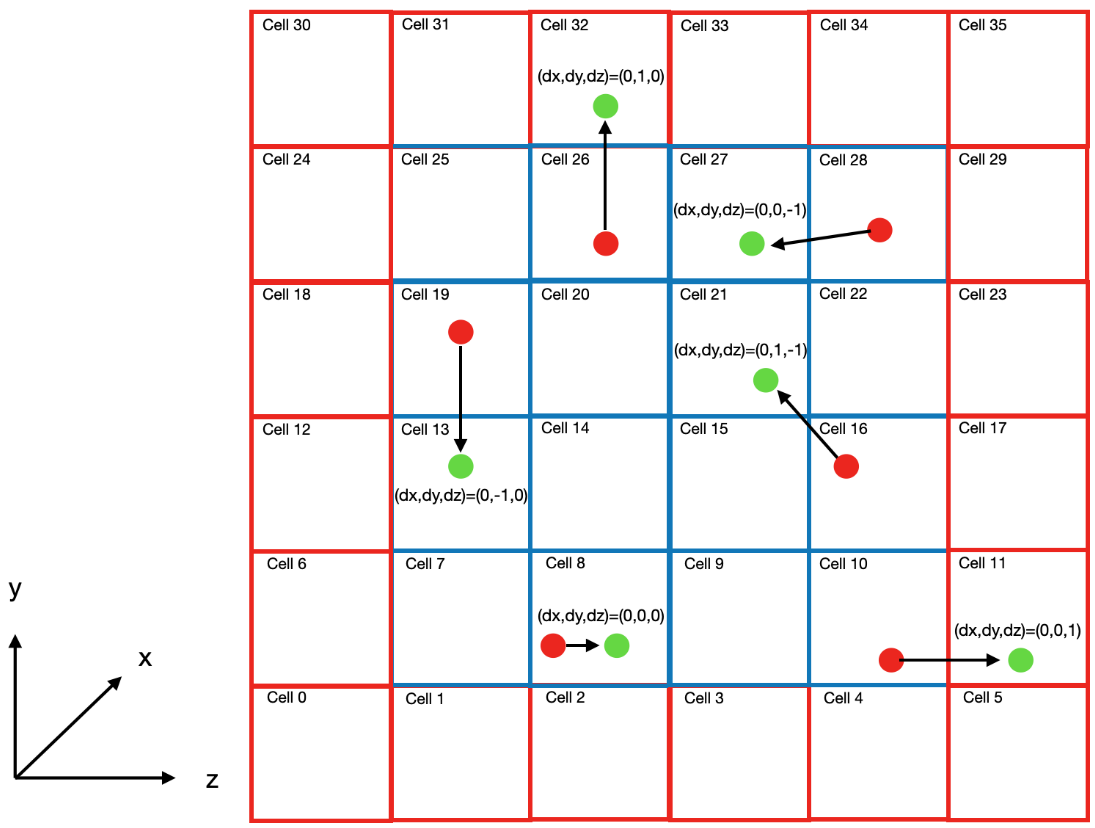

Figure 4.4: Particles configuration before (red dots) and after the deterministic ballistic motion (green dots). Physical cells have blue borders, whereas halo cells are identified with red borders. If the particle remains in the same cell we set $(\mathtt{dx}, \mathtt{dy}, \mathtt{dz}) = (0, 0, 0)$. If the particle move to a neighbor cell in the z direction we have $(\mathtt{dx}, \mathtt{dy}, \mathtt{dz}) = (0, 0, \pm 1)$. Similarly, for x and y. In total, for each cell, we have 26 neighboring cells.

the particles moving in the second direction, e.g. $(\mathtt{dx}, \mathtt{dy}, \mathtt{dz}) = (-1, -1, 0)$ and so on. In the end, we update all the 26 directions, one direction at a time.
More technically, we implement this operation by means of a for loop on the host running on the 26 directions. At each iteration, we open and close (thus creating an implicit synchronization) a parallel region performing the update.

In order to perform such updates, we copy the moving particles in an auxiliary buffer `dsmc_particle_soa_new` of the `dsmc_particle_type_soa` type.
In this way, threads always work in parallel, but we organize the updating procedure sequentially in the directions, as depicted in Figure 4.6.
Next, we add the particles that have remained within the same cell to the buffer `dsmc_particle_soa_new` array.
Finally, we swap `dsmc_particle_soa_new` and `dsmc_particle_soa`.

Once the particles have been indexed in the correct cells, we can update the coordinates imposing the periodic boundary conditions.

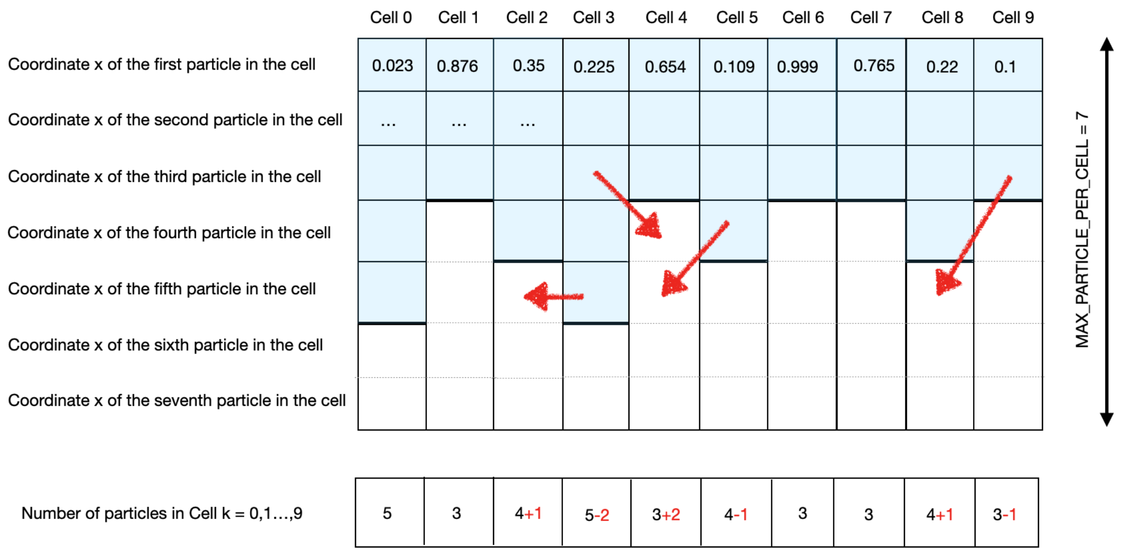
Cell 0
Cell 1
Cell 2
Cell 3
Cell 4
Cell 5
Cell 6
Cell 7
Cell 8
Cell 9
Coordinate x of the first particle in the cell
Coordinate x of the second particle in the cell
Coordinate x of the third particle in the cell
Coordinate x of the fourth particle in the cell
Coordinate x of the fifth particle in the cell
Coordinate x of the sixth particle in the cell
Coordinate x of the seventh particle in the cell
0.023
0.876
0.35
0.225
0.654
0.109
0.999
0.765
0.22
0.1
...
MAX_PARTICLE_PER_CELL = 7
Number of particles in Cell k = 0,1...,9
5
3
4+1
5-2
3+2
4-1
3
3
4+1
3-1

Figure 4.5: Representation of the `dsmc_particle_soa` 2D array after the ballistic motion. In our parallel pattern, each cell (i.e. each column) corresponds to a different thread. Particles moving to a different cell need to be re-indexed to the associated new column. Ensuring thread-safe updates of the arrays requires particular care, as more threads might have to update the same column.

4.4 Communication step

Recall that each processor also owns halo cells, i.e. copies of grid cells owned by other processors that overlap its extended bounding box.
Typically, after ballistic motion, the majority of the particles that a processor owns will remain within the physical cells of the the same processor. However, there is the possibility that some particles will move to cells owned by other processors. This means, in other words, that after the streaming some particles are indexed in the halo cells of a processor and need to be communicated to their new owning processor.

Now, a 3D Cartesian grid can be partitioned either across one, two or three dimensions, where higher-dimension partitions have the advantage of scaling the number of particles to be communicated over the processing nodes.
However, for simplicity, consider a 1D partitioning in the x-direction, that is, consider a ring topology consisting of N nodes, in which each node i exchanges data only with its neighbors $i-1$ and $i+1$. In this case, the *Communication* step requires three simple kernels:

i) We loop over the left and right halo cells in parallel, where each per-grid-cell thread buffers the particles residing in that halo cell.

ii) Once the particles have been moved into the right and left sending buffers, we perform MPI communication. More concretely, using `MPI_Sendrecv`, the `send_to_right` buffer is sent from processor i to $i+1$ where it is received in the receiving buffer `receive_from_left`. Similarly, the `send_to_left` buffer is sent from processor $i+1$ to i where it is received in the receiving buffer `receive_from_right`.

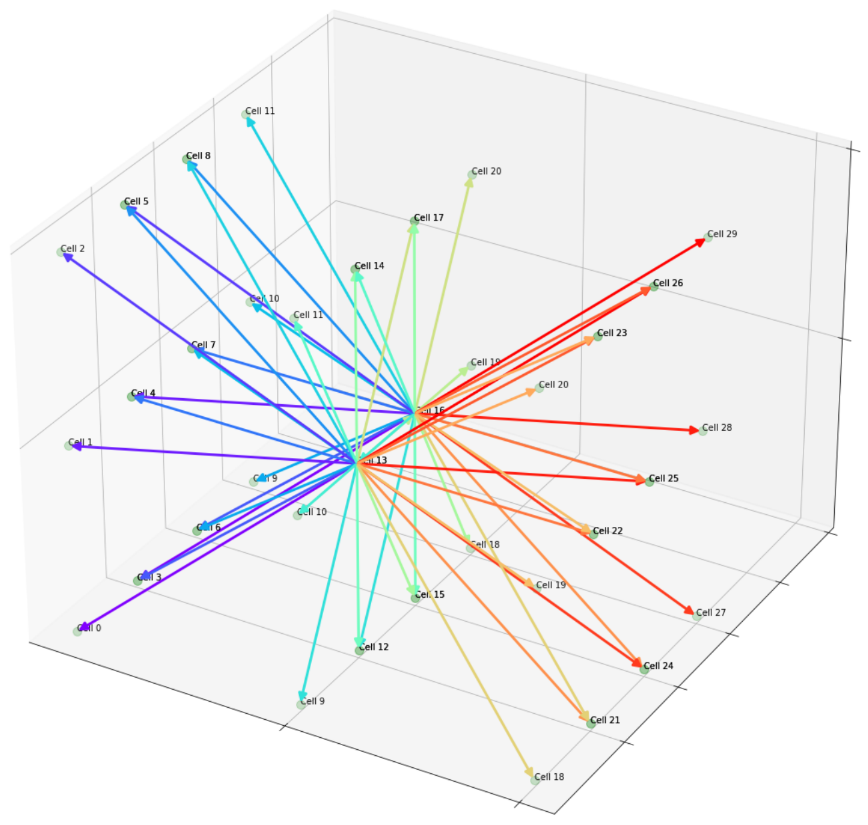

Figure 4.6: Visual representation of the *Indexing* update. Per-cell threads work in parallel updating the neighboring cell list one direction at a time, following the color gradient. Synchronization happens at the end of each direction. Since all threads update concordantly (without interfering) on the 26 directions, thread safety is guaranteed.

iii) Once the buffers have been received, we move the particles from the buffers to the corresponding physical cells, as in Figure 4.7.

Notice that some NVIDIA architectures, as CINECA Marconi100, support MPI CUDA-Aware [56] where the host and the device memory on a single node are combined into unique virtual address space. In this case, we can use the pointers to memory locations resident on the device in the MPI communications on the host side, increasing the efficiency by pipelining all the operations required in a data transfer and allowing RDMA communications [37].

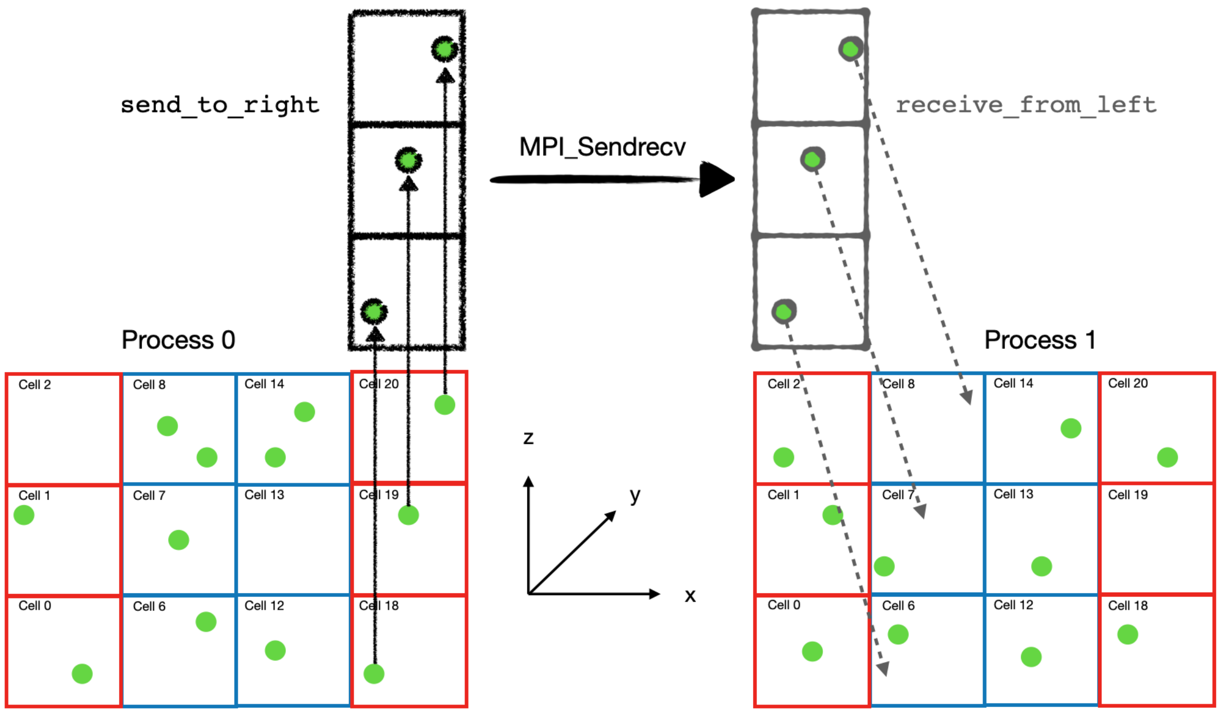

Figure 4.7: Schematic representation of the *Communication* step. Physical cells have blue borders, whereas halo cells are identified with red borders. After the *Streaming* step and the *Indexing* step, some particles (green dots) reside in the halo cells. In order to perform the *Communication* step, (*i*) each per-halo-cell thread (each thread is associated to a halo cell) moves their particles from the right halo to the `send_to_right` buffer. Using `MPI_Sendrecv`, data in the `send_to_right` buffer is sent from Process 0 and received by Process 1 in the `receive_from_left` buffer (*ii*). Finally, we move the particles from the `receive_from_left` buffer to the correspondingly physical cells (*iii*).

```
#ifdef DSMC_OPENACC_GPU
#pragma acc host_data use_device(send_to_right,recv_from_left)
#endif
MPI_Sendrecv(send_to_right,buffer_size,MPI_DOUBLE,pxp,3,recv_from_left,
             buffer_size,MPI_DOUBLE,pxm,3,MPI_COMM_ALONG_X,&status1);

#ifdef DSMC_OPENACC_GPU
#pragma acc host_data use_device(send_to_left,recv_from_right)
#endif
MPI_Sendrecv(send_to_left,buffer_size,MPI_DOUBLE,pxm,4,recv_from_right,
             buffer_size,MPI_DOUBLE,pxp,4,MPI_COMM_ALONG_X,&status2);
```

4.5 Collision step

In the *Collision* step, we perform stochastic binary collisions among particles within the same cell and compute the resulting post-collision velocities.
A technical aspect that we need to stress is the choice of the random number generator. For our purposes, we have considered a simplified linear congruential generator based on the APE random generator.
From the implementation point of view, we follow the Algorithm 3.

Algorithm 3 Collision step

```
 1: for each cell $c$ do      ▷ #pragma acc parallel loop present(...)  collapse(3)
 2:     compute $M_{cand}$
 3:     for $i \leftarrow 0$ to $M_{cand}$ do                ▷ #pragma acc parallel loop seq
 4:         randomly select two different particles $m$ and $\ell$
 5:         compute the pre-collision relative velocity $|\mathbf{v}_r|$ between $m$ and $\ell$
 6:         compute the new collision cross section $\sigma_T$
 7:         compute $P(\mathbf{v}_m, \mathbf{v}_\ell) = \frac{\sigma_T|\mathbf{v}_r|}{(\sigma_T|\mathbf{v}_r|)_{max}(c)}$
 8:         if the collision is accepted then
 9:             randomly sample the angles $\chi$ and $\phi$
10:             compute the post-collision velocities $\mathbf{v}_m^*$ and $\mathbf{v}_\ell^*$
11:         end if
12:     end for
13:     update $(\sigma_T|\mathbf{v}_r|)_{max}(c)$
14: end for
```

Actually, the collision kernel consists of a parallel loop over all cells, where each per-grid-cell thread first computes how many couples of particles M_{cand} might perform the collision, according to Equation 2.49.
Then, within the same parallel loop, each thread performs a sequential for loop, choosing randomly two candidate particles for each couple. In our implementation, all threads use the same seed for the random number generator during these random selections.

The attempted collision between two candidate particles is accepted with probability given by Equation 2.50.
Then, if the collision takes place, we need to sample the elevation and the azimuthal angles, as explained in Chapter 2.
Note that, for these latter operations, we choose a different strategy to improve the statistics. Specifically, for accepting the collision and sampling the angles, each thread has its own seed for the APE random number generator.

Once the collision is performed, the velocities of the two particles are updated with the post-collision values.
Finally, once all the candidate couples have been considered, each per-grid-cell thread updates the value of $(\sigma_T|\mathbf{v}_r|)_{max}$.

4.6 Sampling step

The last step is to sample flow properties and standard hydrodynamic moments.
From the implementation perspective, we consider a kernel with a parallel loop over all the cells, where each per-grid-cell thread computes and updates the averaging flow properties based on the particles residing in that cell.
For details see Algorithm 4, and recall the `dsmc_cell_type_soa` data structure defined in Section 4.1, as well as the definition of the pressure tensor in Equation 2.26.

Algorithm 4 Sampling step

```
 1: for each cell $c$ do      ▷ #pragma acc parallel loop present(...)  collapse(3)
 2:     for each particle $p$ in cell $c$ do             ▷ #pragma acc parallel loop seq
 3:         molecule_sum($c$) ← molecule_sum($c$) + 1
 4:         sum_u_i($c$) ← sum_u_i($c$) + u_i($p$)                          ▷ i = x,y,z
 5:         sum_ke_i($c$) ← sum_ke_i($c$) + (u_i($p$))$^2$                  ▷ i = x,y,z
 6:     end for
 7:     stream_i($c$) ← sum_u_i($c$)/molecule_sum($c$)                      ▷ i = x,y,z
 8:     for each particle $p$ in cell $c$ do             ▷ #pragma acc parallel loop seq
 9:         v_i($p$) ← u_i($p$) − stream_i($c$)                             ▷ i = x,y,z
10:         sum_u_iu_j($c$) ← sum_u_iu_j($c$) + v_i($p$) ⊙ v_j($p$)       ▷ i,j = x,y,z
11:     end for
12: end for
```

As mentioned in Chapter 2, the sampling operation shall not be performed at each time step, but rather every `dsmc_steps_sampling` steps (typically `dsmc_steps_sampling` ~ 10), and only after `dsmc_steps_sampling_ndiag` steps from the beginning of the simulations have completed, that is when the system has already reached a steady state.

4.7 Validation

Conservation laws

In order to validate the code, we first verify that the total number of particles N is conserved, as particles are not created nor destroyed at the boundaries.
Similarly, since we have imposed periodic boundary conditions, the momentum along the three directions as well as the total kinetic energy must also be conserved, namely

$$\begin{cases} P_x(t) = \sum_{k=1}^{N} m_k v_{xk} = \text{const.} \\ P_y(t) = \sum_{k=1}^{N} m_k v_{yk} = \text{const.} \\ P_z(t) = \sum_{k=1}^{N} m_k v_{zk} = \text{const.} \\ K(t) = \sum_{k=1}^{N} \frac{m_k}{2}(v_{xk}^2 + v_{yk}^2 + v_{zk}^2) = \text{const.} \end{cases} . \tag{4.1}$$

Relaxation towards equilibrium

Another possible check is to study how the system relaxes towards equilibrium.
For instance, this can be done by initializing the particles with velocities

$$\{v_x, v_y, v_z\} = \pm\sqrt{\frac{k_B T}{m}} \,, \tag{4.2}$$

where $\pm$ is a random number either $\{+1, -1\}$.
After performing several time steps $\mathcal{O}(10^4)$, we check whether the final probability distribution functions for the velocities are close to the theoretically predicted ones.
Actually, the probability of finding a particle with velocity in the infinitesimal element $[dv_x, dv_y, dv_z]$ about velocity $\mathbf{v} = [v_x, v_y, v_z]$ is [57]

$$f_{\mathbf{v}}\left(v_x, v_y, v_z\right)\, dv_x\, dv_y\, dv_z = f_v(v_x) f_v(v_y) f_v(v_z)\, dv_x\, dv_y\, dv_z \tag{4.3}$$

where the distribution for a single direction is

$$f_v(v_i) = \sqrt{\frac{m}{2\pi kT}} \exp\left(-\frac{mv_i^2}{2kT}\right) . \tag{4.4}$$

Integrating over solid angle, the probability distribution for $v = |\mathbf{v}|$ is given by

$$f(v) = \left(\frac{m}{2\pi kT}\right)^{3/2} 4\pi v^2 e^{-\frac{mv^2}{2kT}} . \tag{4.5}$$

Typical results are depicted in Figure 4.8.

Mean free path

Finally, we can compare the mean free path measured in the DSMC simulation with the theoretical one λ_{th}. Specifically, for the VHS model, we have [14]

$$\lambda_{th} = \frac{1}{\sqrt{2}\pi n d_{ref}^2 \left(\frac{T_{ref}}{T}\right)^{\omega - 1/2}} \tag{4.6}$$

For example, we consider the scaling relation $\lambda(T)$ by varying the temperature T from 100 K to 500 K and keeping fixed $\omega = 0.81$, the reference temperature $T_{ref} = 273$ K, number density $n = 2 \times 10^{20}$ m^{-3} and diameter $d_{ref} = 4.092 \times 10^{-10}$ m. The comparison between the theoretical mean free path λ_{th} and the measured one after 30000 time step is shown in Figure 4.9.

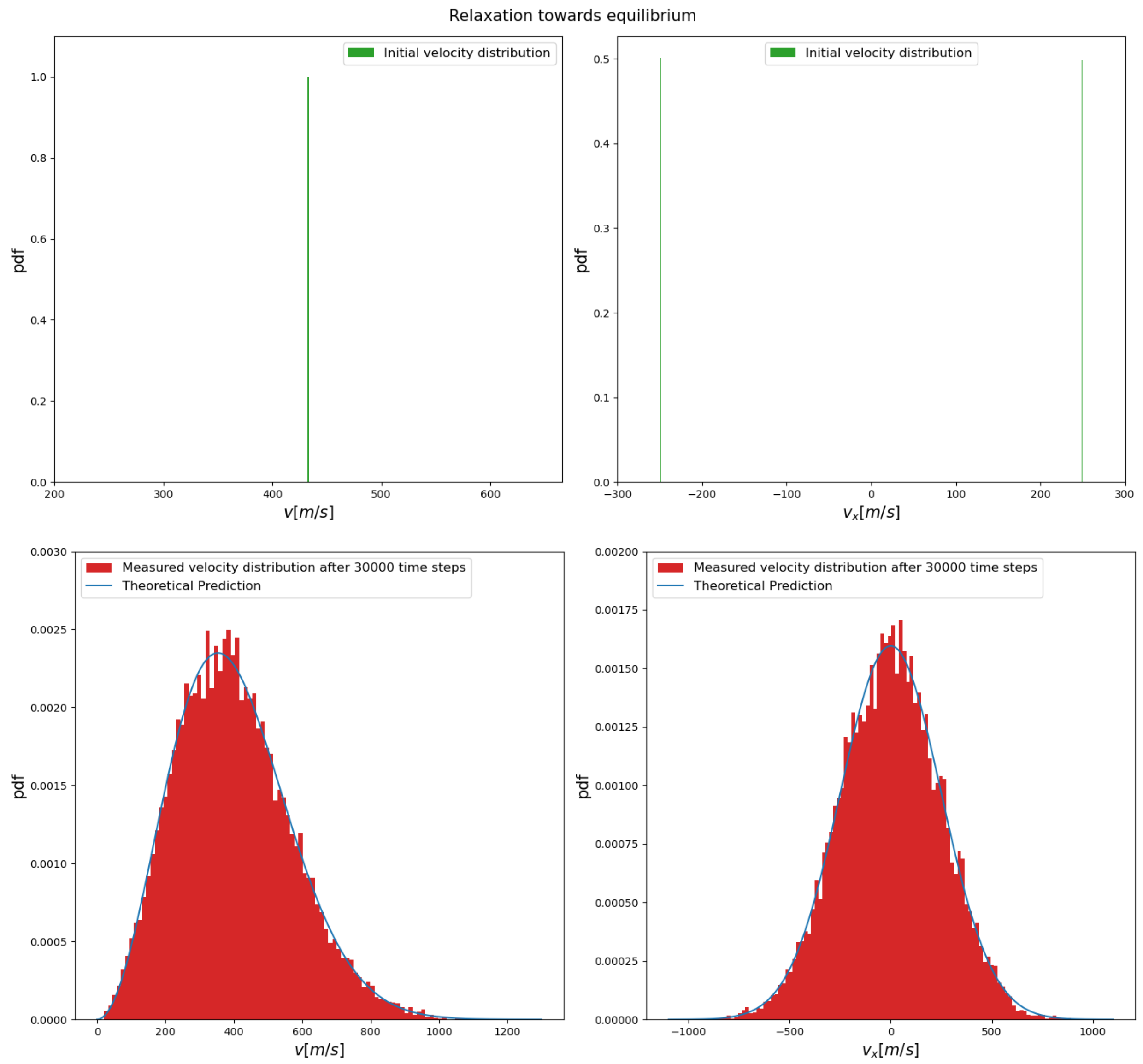

Figure 4.8: We consider a simulation with 16384 DSMC molecules ($8 \times 8 \times 8 = 512$ cells with 32 particles per cell) representing particles with mass $m = 6.63 \times 10^{-26}$ Kg and diameter $d_{ref} = 4.092 \times 10^{-10}$ m at temperature $T = 300$ K and number density $n = 2 \times 10^{20}$ m^{-3} in a box of 1 m^3. For the collisions, we consider the VHS with parameter $\omega = 0.81$. The upper panel shows the initial velocities distributions for $v = |\mathbf{v}|$ and v_x as in Equation 4.2. The lower panel shows the final velocities distributions after 30000 time steps each of duration $\Delta t = 10^{-6}$ s, as well as the theoretical predictions given by Equation 4.5 and Equation 4.4.

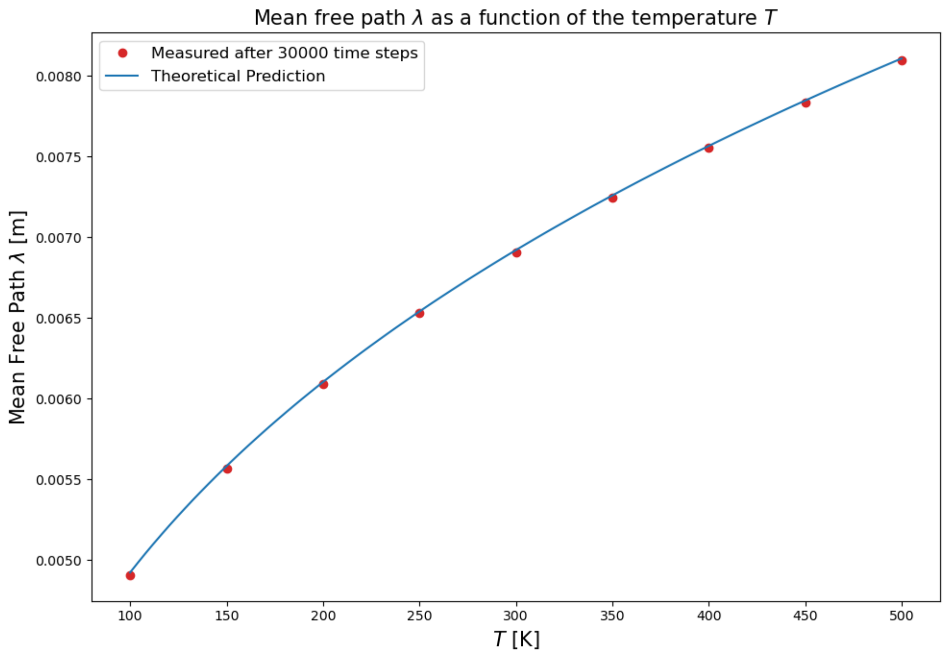

Figure 4.9: Theoretical (according to Equation 4.6) and measured (within the DSMC simulation) mean free path as a function of the temperature.

Chapter 5

Results

The benchmark problem used here to assess performance is a variable hard-sphere gas in a box with periodic boundary conditions.
We initialize the gas with initial velocity as in Equation 4.2, then molecular collisions lead the gas to eventually reach thermodynamic equilibrium.
For these benchmark runs, we simulate Argon-like gas at a temperature of 273.15 K and number density of 7.1×10^{22} m^{-3} with an average of 20 DSMC particles per grid cell (setting `MAX_PARTICLES_PER_CELL` $= 64$).
The gas is composed by molecules of mass $m = 6.63 \times 10^{-26}$ Kg and reference diameter $d_{ref} = 4.17 \times 10^{-10}$ m, with the VHS parameter $\omega = 0.81$.
We consider simulations of 1000 time steps, each during 7.0×10^{-9} s.
Moreover, in order to have a homogeneous MPI topology, we consider a 1D decomposition of the domain along the x direction.

For benchmark timings, in general, there are several metrics of interest when studying the performance of the DSMC code.
Here, we mainly focus on Average Time per Call (AVG $\times$ Call), which is the total time spent on the kernel divided by the number of calls to that kernel,

$$\text{AVG} \times \text{Call} = \frac{\text{Total Time}}{\text{Num Calls}} , \tag{5.1}$$

where the Calls refer to the function in the time loop, namely *Ballistic motion*, *Indexing*, *Periodic boundary conditions*, *Communication*, *Collision* and *Data Sampling*.

An important aspect that we need to stress is that performances strongly depend on the size of the problem, expressed in terms of the number of DSMC particles in the simulation. Recall that we keep the number of particles per cell fixed (specifically 20 particles per cell). As a consequence, when we vary the total number of particles, also the number of grid cells changes. Since we consider a per-grid-cell thread parallelism, this implies that the total number of DSMC particles is associated to the number of threads, and thus related to the performances of the GPUs.

The performance analysis of the code we have implemented refers to simulations running on Marconi100[1], one of the HPC clusters hosted by CINECA.
The Table 5.1 lists the one node specification for Marconi100.

[1]Marconi100 cluster: https://www.hpc.cineca.it/hardware/marconi100

Specification	Marconi100	Marconi
CPU model	IBM POWER9 AC922	Intel Xeon 8160 CPU (Skylake)
Architecture	ppc64le	x86_64
Sockets	2	2
Cores/socket	16	24
Threads/core	4	2
Total physical cores	32	48
Total logical cores	128	96
Processor Base Frequency	2.60 GHz	2.10 GHz
CPU Max frequency	3.8 GHz	3.70 GHz
L1d cache/core	32KB	32KB
L1i cache/core	32KB	32KB
L2 cache/core	512KB	1024KB
L3 cache	10240KB	33792KB
RAM	256 GB DDR4	192 GB DDR4
Memory channels per socket	8	6
GPU (on specific nodes)	4 $\times$ Tesla V100-SXM2 16GB	

Table 5.1: Hardware specification of the one node of Marconi100 and Marconi (A3) clusters. Marconi100 runs on a Mellanox IB EDR DragonFly++ (100Gb/s) high-performance network. Marconi runs on a Intel OmniPath (100Gb/s) high-performance network.

Moreover, the source code was compiled using the Nvidia HPC SDK (`hpc-sdk/2021`) compiler with the following flags `-fast -O3 -Minline -Msafeptr -ta=tesla:cc70`.

As an example, it is interesting to study the AVG $\times$ Call for one full node (with 4 MPI processes associated to the 4 GPUs of the Marconi100 node) as a function of the simulated DSMC particles. Results are shown in Figure 5.1 where on the x-axis we vary the number of particles p from $p \approx 500$ K (thousand) to $p \approx 125$ M (million) (corresponding to $c \approx 25$ K to $c \approx 6.25$ M cells, respectively).

Remarkably, from the plot we see one of the main problem of our implementation, namely the *Indexing* function scales only for large simulations. In other words, we observe performance degradation when the number of particles per process is less than $\approx$ 4 M. Further investigation is needed to fully understand and possibly improve such behavior, but few considerations are in order. First of all, the *Indexing* function does not perform any floating point operations, but rather memory accesses, that is not what GPUs are optimized for. The second aspect is that, in the *Indexing* function, 26 kernels are created to sequentially update the particles moving in one of the 26 directions at a time. Even though we tried to minimize data movements using the `present` clause, the compiler might need to perform data transfer operations.

Clearly, this does not affect just small size simulations, but, importantly, also large simulations as we increase the number of processors, that is the strong scaling.

As already mentioned, this work is based on a simplified version of the fm_dsmc code developed in [14, 48, 49]. In this original fm_dsmc code, parallelization is achieved by distributing cells and particles among MPI processes, in a pure distributed memory approach (e.g. no shared memory paralellization with OpenMP threads).

While porting the code to multi-GPUs required major modifications, such as different data structures and different algorithms, it is still useful to compare the original code and our

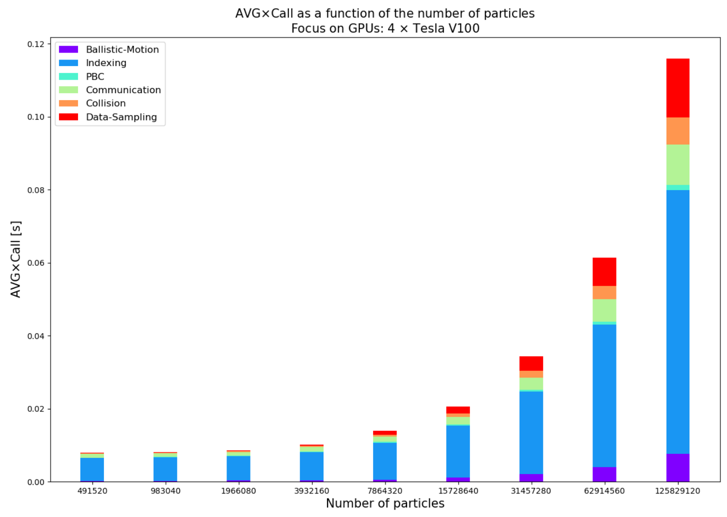

Figure 5.1: AVG×Call as a function of the number of particles for one full node of Marconi100 (4 × Tesla-V100-SXM2-16GB), with 20 particles per cell.

implementation with GPU offloading, to fully understand pros and cons.
Also notice that the original code takes into account the possibility that collision cells differ from sampling cells. More precisely, for computational reasons, each sampling cell is divided into n_{sc} collision subcells.
In order to compare the original code and our new implementation, we have set $n_{\text{sc}} = 1$, but such comparison is slightly biased due to additional branching of the original code.

Simulations with the original code have been performed both on Marconi100 and Marconi[2] clusters. The Table 5.1 lists the one node specification also for Marconi.
On Marconi100 we compile the original code with IBM XL C compiler (`xl/16.1.1`) and Spectrum-MPI parallel library (`spectrum_mpi/10.4.0`), using the `-O3` optimization flag.
On the other hand, on Marconi we compile the original code with the INTEL Compiler (`intel/pe-xe-2018`) and IntelMPI (`intelmpi/2018`), using the `-O3 -xCORE-AVX512` optimization flag, where the `-xCORE-AVX512` flag generates AVX-512 instructions, optimised for the Skylake processor.

5.1 Strong scaling

In this section, we analyze the performances and timings for strong scaling, defined as the variation of the time taken to complete the task with increasing number of parallel processes while keeping the problem size constant.
Specifically, we consider a problem of $p \approx 125$ M particles and $c \approx 6.25$ M cells (precisely $p = 125829120$ DSMC molecules and $c = 6291456$ grid-cells, with 20 particles per cell), running on $1, 2, 4, 8, 16, 32, 64$ nodes.

[2]Marconi cluster: https://www.hpc.cineca.it/hardware/marconi

Moreover, for simulations with GPU offloading, we consider 4 MPI processes per node. In this way, each process is associated to one of the 4 Tesla V100 within the Marconi100 node. For the original code simulations running on Marconi100, with Power9 (P9) processors, we consider 32 MPI processes per node, each associated to one of the 32 physical cores within the node (no multi-threading). Similarly, for simulations running on Marconi with Skylake (SKL) processors, we consider 48 MPI processes per node, each associated to one of the 48 physical cores within the node (no multi-threading).
In Figure 5.2, we show the AVG $\times$ Call as a function of the number of nodes.

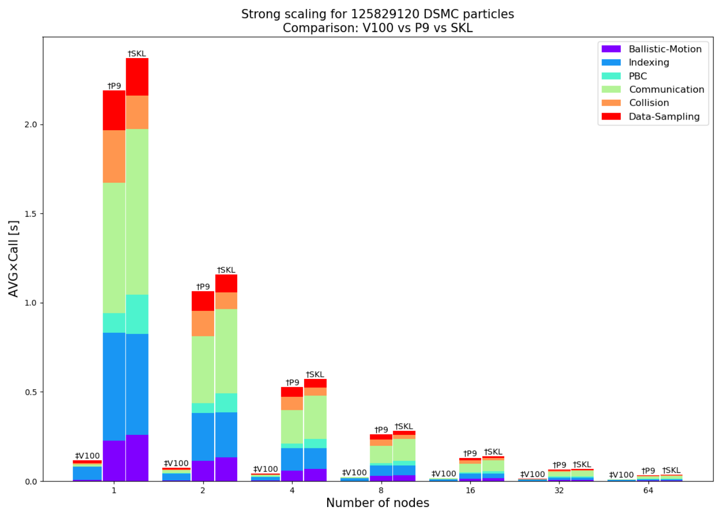

Figure 5.2: Strong scaling for 125829120 DSMC molecules, with 20 particles per cell. In the plot, we show the AVG$\times$Call as a function of the number of nodes for our implementation ‡ with GPU offloading running on 4 Tesla V100 per node (‡$V100$) as well as the original code † running on 2×16 physical cores Power9 per node (†$P9$) and 2×24 physical cores Skylake per node (†SKL).

Let's focus on the pros and cons of our code with GPU offloading.
Remarkably, for the size under consideration, running on one single Marconi100 node, the code with GPU offloading is significantly more performing compared to the original code. Another way to see this is to notice that we need 16 nodes for the original code to achieve performances comparable with one full GPU node.
Moreover, note that, for the problem size under consideration, the original code running on Power9 processors shows better performances compared to the Skylake processors, especially in the *Periodic Boundary Conditions* routine and in the creation of the buffer for the *Communication* function. A possible reason for this behavior might be related to the higher memory bandwidth of the Power9.
In fact, in the *Collision* function, where most of the floating point operations are performed, the Skylake processor shows better performances.
Conversely, for simulation running on 64 nodes, the aforementioned GPU performance

improvement is severely reduced. The reason can be traced back to the poor scaling of our code, as we can see in Figure 5.3.
In the plot, we show the speedup as a function of the number of nodes n defined by

$$\text{Speedup}(n) = T_p(1)/T_p(n) \tag{5.2}$$

where $p = 125829120$ is the number of particles, $T_p(1)$ is the time for solving the problem with p particles and one node, while $T_p(n)$ is the time for solving the problem with p particles and n nodes.

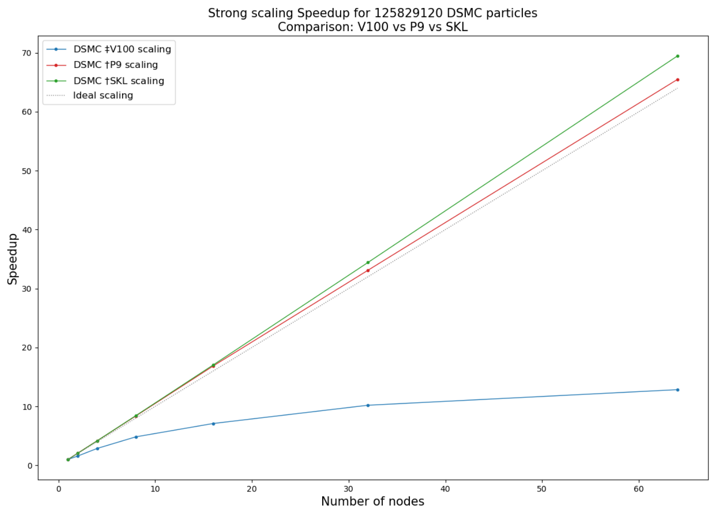

Figure 5.3: Strong scaling Speedup, defined in Equation 5.2, for $p = 125829120$ DSMC molecules, with 20 particles per cell. In the plot, we show the Speedup as a function of the number of nodes for our implementation ‡ with GPU offloading running on 4 Tesla V100 per node (‡$V100$) as well as the original code † running on 2×16 physical cores Power9 per node (†$P9$) and 2×24 physical cores Skylake per node (†SKL).

Remarkably, on one hand, we see that the original code has an ideal scaling, Speedup(n) $\simeq$ n. On the other hand, starting from 8 nodes (corresponding to 32 MPI processes) we have less than $\approx$ 4 M particles per process, precisely the regime where the *Indexing* function stops scaling, as we can see focusing on the GPU strong scaling in Figure 5.4.

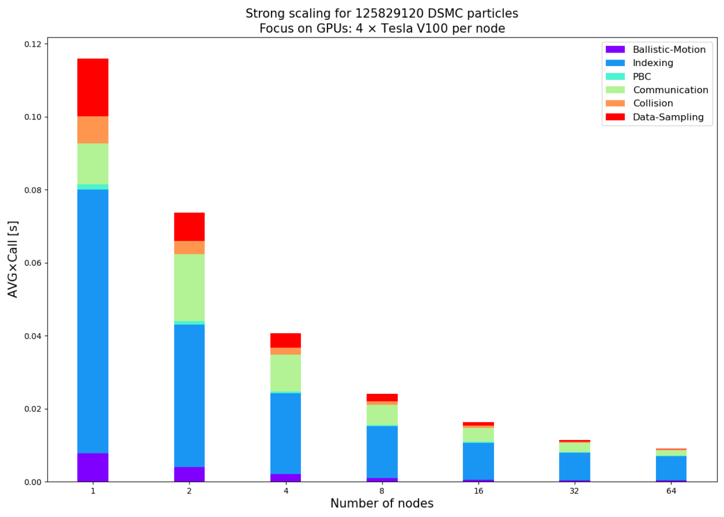

Figure 5.4: Strong scaling for $p = 125829120$ DSMC molecules, with 20 particles per cell. In the plot, we focus on the AVG×Call as a function of the number of nodes for our implementation, with GPU offloading, running on 4 Tesla V100 per node. As we can see, for more than 8 nodes the *Indexing* function stops scaling.

5.2 Weak scaling

As we have seen in the previous sections, the code with GPU offloading, developed in this work, stops scaling when we have less than ≈ 4 M particles per process.
Therefore, in order to avoid such performance degradation, we have to keep a large number of DSMC particle per process, that is, we need to consider weak scalability.
Actually, weak scaling can be defined as the variation in time taken to complete the task with increasing number of processes while keeping workload per process constant.
In particular, the weak scaling results for $p \approx 32$ M (precisely $p = 31457280$) DSMC molecules per node are shown in Figure 5.5.

Moreover, in Figure 5.6 we show the efficiency as a function of the nodes, defined as

$$\text{Efficiency}(n) = \frac{T_p(1)}{T_{n \cdot p}(n)} \ , \tag{5.3}$$

where $T_{n \cdot p}(n)$ is the time for solving the problem with $n \cdot p$ particles and n nodes.

Note that the original code shows an almost ideal efficiency, Efficiency$(n) \simeq 1$, with a slow degradation as the number of nodes increases, due to the *Communication* step. Conversely, the efficiency of our implementation drops significantly moving from 1 node to 2 nodes; then for $n \geq 2$ we see the slow decay as the number of nodes increases, as before.
In order to understand what is happening, let's focus on the weak scaling for our code with GPU offloading, as shown in Figure 5.7. Remarkably, the *Communication* time moving from intra-node to inter-node visibly increases (it more than doubles). Afterwards, the *Communication* time slowly increases as more and more nodes are considered.
The Avg $\times$ Call for the other functions remain constant.

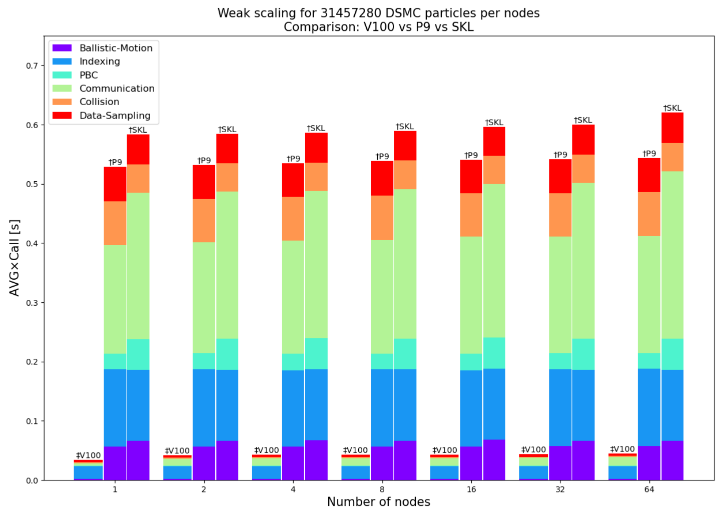

Figure 5.5: Weak scaling for $p = 31457280$ DSMC molecules per process, with 20 particles per cell. In the plot, we show the AVG×Call as a function of the number of nodes for our implementation ‡ with GPU offloading running on 4 Tesla V100 per node (‡$V100$) as well as the original code † running on 2×16 physical cores Power9 per node (†$P9$) and 2×24 physical cores Skylake per node (†SKL).

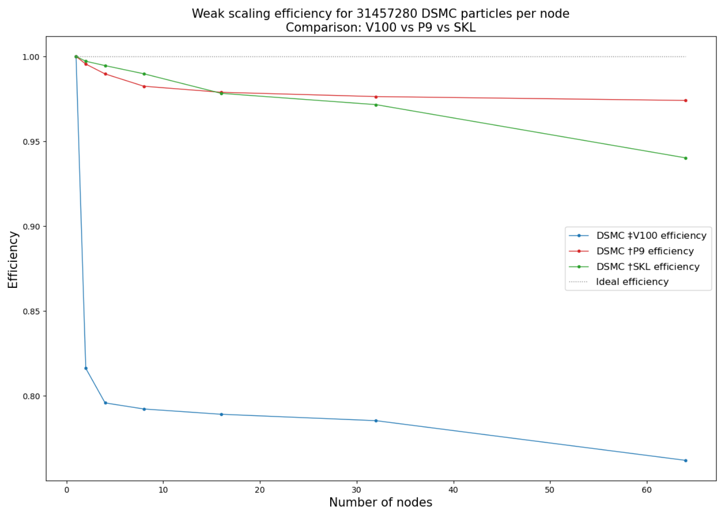

Figure 5.6: Weak scaling Efficiency, defined in Equation 5.3, for $p = 31457280$ DSMC molecules per process, with 20 particles per cell. In the plot, we show the Efficiency as a function of the number of nodes for our implementation ‡ with GPU offloading running on 4 Tesla V100 per node (‡$V100$) as well as the original code † running on 2×16 physical cores Power9 per node (†$P9$) and 2×24 physical cores Skylake per node (†SKL).

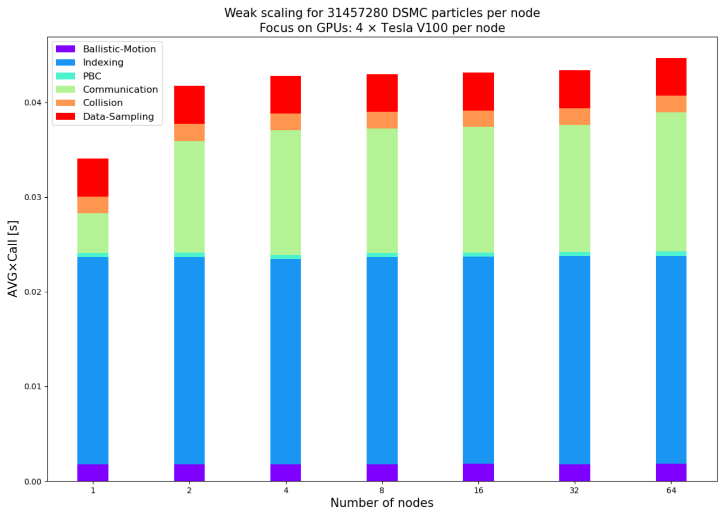

Figure 5.7: Weak scaling for $p = 31457280$ DSMC molecules per process, with 20 particles per cell. In the plot, we focus on the AVG×Call as a function of the number of nodes for our implementation, with GPU offloading, running on 4 Tesla V100 per node. As we can see, the *Communication* time moving from intra-node (1 node) to inter-node (≥ 2 nodes) visibly increases.

Chapter 6

Conclusions

In this work, we have studied the offloading to accelerators, using OpenACC, of a simplified version of the *fm_dsmc* code, developed at FLOW Matters Consultancy B.V., a start-up company supporting this project.
Despite possible performance degradation, compared to native programming languages such as CUDA, OpenACC provides a very interesting and promising tool to express parallelism in DSMC applications, both for its simplicity and portability. Actually, offloading kernels to accelerators with OpenACC is quick and simple, while, at the same, preserving the code readability and the maintainability.

However, when porting the DSMC code to multi-GPUs, different data structures and different algorithms are needed to enable threaded parallelism and take advantage of the GPUs computing power.
Actually, on one hand, exploiting GPU architectures requires the SoA data organization rather than the traditional AoS format, as it guarantees better coalescence of global memory accesses. On the other hand, *Indexing* particles after the *Streaming* step requires dedicated algorithms or specific expedients to ensure thread safety.
In particular, by carefully choosing the duration of the time step Δt, we could safely assume that, after the *Streaming* step, particles can move at most to neighboring cells. In this way, thread safety is guaranteed by updating particles moving in one of the 26 direction at a time, thus avoiding atomic operations.

Performances strongly depend on the problem size and the available resources.
Actually, for simulations with $p \approx 125$ M DSMC particles, our application with GPU offloading running on one full Marconi100 node (4 MPI processes for 4 NVIDIA Tesla V100) shows about 18 times speedup when compared to the original code running on the same node (32 MPI processes for 2×16 IBM Power AC922 physical cores).
Unfortunately, the situation completely changes as we increase the number of nodes, that is, when we consider strong scalability.
For instance, consider simulations with $p \approx 125$ M particles running on 64 Marconi100 nodes. In this case, the performance speedup of our application with GPU offloading drops to about 3 times when compared to the original code.
Similar results can be obtained with the original code running on Marconi supercomputer (with Intel Xeon 8160 Skylake processors).
The reason can be traced back to the poor scaling of our *Indexing* function when we have

less than ≈ 4 M particles per MPI process (i.e. per GPU).
However, in a weak scaling test, keeping fixed $p \approx 32$ M particles per node, we have measured about 12 times speedup up to 64 nodes.
Moreover, on Marconi100, for simulations with $p \approx 125$ M DSMC particles, our application with GPU offloading running on 8 nodes shows better performances then the original code running on 64 nodes. This result can be relevant for efficiency in energy consumption, as well as for quality of service and fare usage of HPC facilities with heterogeneous architectures.

Clearly, future developments need to be focused on improving the scaling of the *Indexing* function, either by careful profiling and optimizing the existing algorithm, and/or by asynchronously performing some operations on the host. Finally, another possibility is to look for new algorithms ensuring thread safety as well as better scaling performances.

Appendix A

AoS vs SoA: a toy-model example

In this Appendix, we consider an illustrative comparison between the AoS and SoA organization. Here, we are not providing a rigorous proof that the SoA format is the best one for our DSMC code, but rather giving some intuition with a simple toy model.
Specifically, consider two very simple kernels (one for each data struct), representing a simplified version of the streaming step, where we simply update N particles' coordinates according to the Euler method:

$$\mathbf{r}_i(t + \Delta t) = \mathbf{r}_i(t) + \mathbf{v}_i \; \Delta t \tag{A.1}$$

To have more statistics, we loop over many time steps, performing `max_iter` iterations.
For simplicity, the initial positions and velocities are randomly initialized. Moreover, since this is a simple toy model, we consider an infinitely large box, without any collision grid structure. Concretely, we define the following data structure

```c
typedef struct {
  double x, y, z;              /* molecule position */
  double ux, uy, uz;           /* molecule velocity components */
} dsmc_basic_type_aos;

typedef struct {
  double *x, *y, *z;           /* molecule position */
  double *ux, *uy, *uz;        /* molecule velocity components */
} dsmc_basic_type_soa;
```

and the following kernels, where we take into account two different possibilities (to be selected at compile time with the `RANDOM` flag).

```
// AoS simple streaming kernel
for (iter = 0; iter < max_iter ; ++iter) {
 #if defined (_OPENACC)
   #pragma acc parallel loop present (dsmc_particle_aos[0:N],dt,N,RND[0:N]) \
   independent
  #elif defined (_OPENMP)
   #pragma omp parallel for
  #endif
  for (i = 0 ; i < N ; ++i) {
    #ifdef RANDOM
      int l = (int) (randf64(&(RND[i])) * ( N-0.001));
    #else
      int l = i;
    #endif
     dsmc_particle_aos[l].x += dt * dsmc_particle_aos[l].ux;
     dsmc_particle_aos[l].y += dt * dsmc_particle_aos[l].uy;
     dsmc_particle_aos[l].z += dt * dsmc_particle_aos[l].uz;
  }
}
```

```
// SoA simple streaming kernel
for (iter = 0; iter < max_iter ; ++iter) {
  #if defined (_OPENACC)
   #pragma acc parallel loop present (dsmc_particle_soa,dt,N,RND[:N], \
 dsmc_particle_soa.x[:N],dsmc_particle_soa.y[:N],dsmc_particle_soa.z[:N],\
 dsmc_particle_soa.ux[:N],dsmc_particle_soa.uy[:N],dsmc_particle_soa.uz[:N])\
  independent
  #elif defined (_OPENMP)
   #pragma omp parallel for
  #endif
  for (i = 0 ; i < N ; ++i) {
    #ifdef RANDOM
      int l = (int) (randf64(&(RND[i])) * ( N-0.001));
    #else
      int l = i;
    #endif
     dsmc_particle_soa.x[l] += dsmc_particle_soa.ux[l] * dt;
     dsmc_particle_soa.y[l] += dsmc_particle_soa.uy[l] * dt;
     dsmc_particle_soa.z[l] += dsmc_particle_soa.uz[l] * dt;
  }
}
```

For the random memory accesses option, we use the same random number generator as in our DSMC code. Moreover, we consider per-particle thread parallelism where each particle (thread) has a different seed.

Compiling the OpenACC code with the `nvc` compiler (`hpc-sdk/2021`) and the OpenMP code with the `gcc` compiler (`gnu/8.4.0`) using the following flags

```
          GPU_ACC = -fast -O3 -acc -ta=tesla:cc70 -acc=noautopar
   GPU_ACC_RANDOM = -fast -O3 -acc -ta=tesla:cc70 -acc=noautopar -DRANDOM
    MULTICORE_ACC = -fast -O3 -acc -ta=multicore -Mvect=simd:256 -acc=noautopar
    MULTICORE_OMP = -O3 -fopenmp -lm
```

and running on one node of CINECA Marconi100 using only one NVIDIA Volta V100 GPU (there is no MPI communication in this toy model) and 32 cores IBM POWER9 AC922 (carefully setting the environment variable `ACC_NUM_CORES` = 32), we get the results shown in Figure A.1.

Few considerations are in order.
Let's start focusing on the GPU performances which are our primary concern.
Remarkably, the GPU with the SoA data structure with contiguous memory accesses (GPU_ACC SoA) is clearly the most performing one.
However, the situation completely changes when compiling with the `RANDOM` flag, where on one hand we generally observe a severe, but expected, performance degradation.
On the other hand, for strided memory accesses, the GPU shows better performances with the AoS rather than the SoA data structure.

A second consideration regards a similar behavior for both the OpenMP and OpenACC multicore CPUs. Actually, while the SoA data struct performs better than the AoS one for $N = 10^6 - 10^7$, we observe the opposite for very large simulations $N = 10^8$.
A full explanation of these results is beyond the purposes of this Appendix and requires further investigation.

In conclusion, we have a reassuring confirmation that, for contiguous memory accesses (i.e. the typical memory accesses in the DSMC algorithm), the SoA data structure is the most performing one for codes with GPU offloading.
In fact, we perform strided memory accesses only in the *Collision* step, where the number of such random accesses is limited by the fact that there are only few particles per cell and only a fraction of these particles (M_{cand}) might perform a collision.

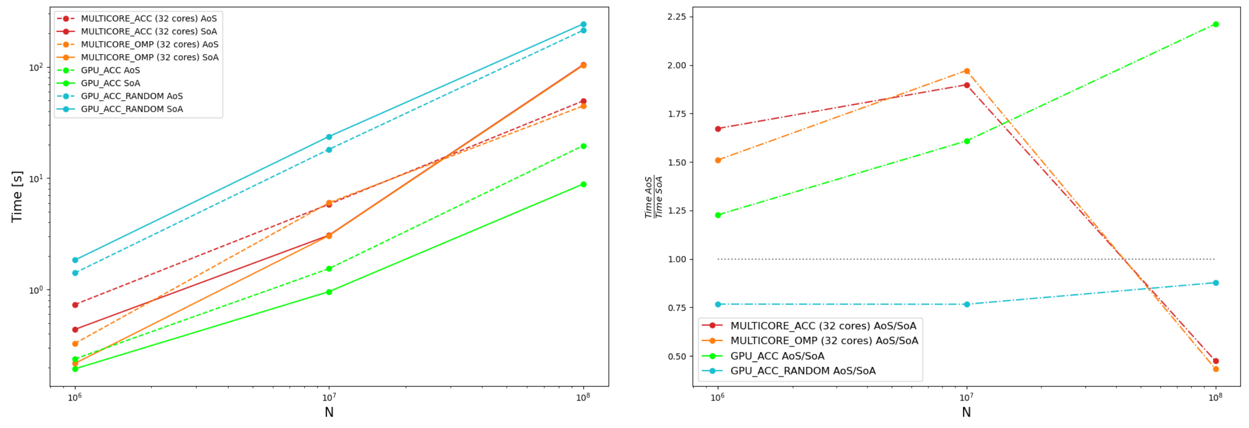

Figure A.1: On the left, we show the time (in seconds) it takes for the kernels to perform `max_iter` = 1000 iterations as a function of the number of particles $N \in \{10^6, 10^7, 10^8\}$. On the right, we plot the ratio $\frac{\text{Time AoS}}{\text{Time SoA}}$ with the same compilation settings. The SoA data structure is more performing than the AoS one for data points above the horizontal grey line.

Bibliography

[1] The green500 project. `https://www.top500.org/lists/green500/2022/11/`, 2022.

[2] Paul Carpenter, Uwe-Haus Utz, Sai Narasimhamurthy, and Estela Suarez. Heterogeneous high performance computing, February 2022.

[3] The top500 project. `https://www.top500.org/lists/top500/2022/11/`, 2022.

[4] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with cuda. In *ACM SIGGRAPH 2008 Classes*, SIGGRAPH '08, New York, NY, USA, 2008. Association for Computing Machinery.

[5] NVIDIA. *Thrust, the CUDA C++ template library.*

[6] Khronos Group. *The Open Standard for Parallel Programming of Heterogeneous Systems.*

[7] Rohit Chandra, Leo Dagum, David Kohr, Ramesh Menon, Dror Maydan, and Jeff McDonald. *Parallel programming in OpenMP*. Morgan kaufmann, 2001.

[8] Openacc 3.2 specification. `https://www.openacc.org/sites/default/files/inline-images/Specification/OpenACC-3.2-final.pdf`. Accessed: November, 2021.

[9] Openacc programming and best practices guide. `https://www.openacc.org/sites/default/files/inline-files/openacc-guide.pdf`. Accessed: April 2022.

[10] Enrico Calore, Alessandro Gabbana, Jiri Kraus, Sebastiano Fabio Schifano, and Raffaele Tripiccione. Performance and portability of accelerated lattice boltzmann applications with openacc. *Concurrency and Computation: Practice and Experience*, 28(12):3485–3502, 2016.

[11] G. A. Bird. Approach to translational equilibrium in a rigid sphere gas. *The Physics of Fluids*, 6(10):1518–1519, 1963.

[12] W. Steckelmacher. Molecular gas dynamics and the direct simulation of gas flows. *Vacuum*, 47(9):1140–, 1996.

[13] G.A. Bird. *Molecular Gas Dynamics and the Direct Simulation of Gas Flows.* The Oxford engineering science series. Clarendon Press, 1994.

[14] G. Di Staso. *Hybrid discretizations of the Boltzmann equation for the dilute gas flow regime.* PhD thesis, Applied Physics, December 2018. Proefschrift.

[15] Carlo Cercignani. *The Boltzmann Equation*, pages 40–103. Springer New York, New York, NY, 1988.

[16] H. Struchtrup. *Macroscopic Transport Equations for Rarefied Gas Flows: Approximation Methods in Kinetic Theory.* Interaction of Mechanics and Mathematics. Springer Berlin Heidelberg, 2006.

[17] D. Tong. Lectures on kinetic theory lecture notes for the cambridge mathematics course. `https://www.damtp.cam.ac.uk/user/tong/kinetic.html`, 2012.

[18] K. Huang. *Statistical Mechanics.* John Wiley and Sons, 2000.

[19] M. Kardar. *Statistical Physics of Particles.* Cambridge University Press, 2007.

[20] Pareschi, Lorenzo and Russo, Giovanni. An introduction to monte carlo method for the boltzmann equation. *ESAIM: Proc.*, 10:35–75, 2001.

[21] Hong Liu and Paolo Glorioso. Lectures on non-equilibrium effective field theories and fluctuating hydrodynamics. *PoS*, TASI2017:008, 2018.

[22] G. A. Bird. The velocity distribution function within a shock wave. *Journal of Fluid Mechanics*, 30(3):479–487, 1967.

[23] S. K. Stefanov. On the basic concepts of the direct simulation monte carlo method. *Physics of Fluids*, 31(6):067104, 2019.

[24] Stefan K. Stefanov. On dsmc calculations of rarefied gas flows with small number of particles in cells. *SIAM Journal on Scientific Computing*, 33(2):677–702, 2011.

[25] Wolfgang Wagner. A convergence proof for bird's direct simulation monte carlo method for the boltzmann equation. *Journal of Statistical Physics*, 66(3):1011–1044, 1992.

[26] David B. Kirk and Wen-mei W. Hwu. *Programming Massively Parallel Processors, Third Edition: A Hands-on Approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3rd edition, 2016.

[27] Peter Pacheco. *An Introduction to Parallel Programming.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2011.

[28] Zhe Jia, Marco Maggioni, Benjamin Staiger, and Daniele P. Scarpazza. Dissecting the nvidia volta gpu architecture via microbenchmarking, 2018.

[29] Luke Durant, Olivier Giroux, M. H., and N. Stam. Nvidia tesla v100 gpu architecture the world's most advanced data center gpu., 2017.

[30] P. Gupta. Cuda refresher: The cuda programming model. `https://developer.nvidia.com/blog/cuda-refresher-cuda-programming-model/`, 2020.

[31] G. Ruetsch and M. Fatica. *CUDA Fortran for Scientists and Engineers: Best Practices for Efficient CUDA Fortran Programming*. Elsevier Science, 2013.

[32] J. Han and B. Sharma. *Learn CUDA Programming: A beginner's guide to GPU programming and parallel computing with CUDA 10.x and C/C++*. Packt Publishing, 2019.

[33] J. Cheng, M. Grossman, and T. McKercher. *Professional CUDA C Programming*. EBL-Schweitzer. Wiley, 2014.

[34] Jimmy Vianello. Development of a multi-gpu navier-stokes solver, 2020.

[35] S. Chandrasekaran and G. Juckeland. *OpenACC for Programmers: Concepts and Strategies*. Pearson Education, 2017.

[36] Rob Farber. *Parallel Programming with OpenACC*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2016.

[37] Alessandro Gabbana. Accelerating the d3q19 lattice boltzmann model with openacc and mpi, 2015.

[38] Saeid Aliei. Porting of the lbe3d to gpu using openacc, 2020.

[39] Alessandro Gabbana. *Lattice Boltzmann Methods for Fluid-Dynamics in Relativistic Regimes*. PhD thesis, Wuppertal U., 2019.

[40] G.J LeBeau and F.E Lumpkin III. Application highlights of the dsmc analysis code (dac) software for simulating rarefied flows. *Computer Methods in Applied Mechanics and Engineering*, 191(6):595–609, 2001. Minisymposium on Methods for Flow Simulation and Modeling.

[41] Michael A. Gallis, John R. Torczynski, Steven J. Plimpton, Daniel J. Rader, and Timothy Koehler. Direct simulation monte carlo: The quest for speed. *AIP Conference Proceedings*, 1628(1):27–36, 2014.

[42] S. J. Plimpton, S. G. Moore, A. Borner, A. K. Stagg, T. P. Koehler, J. R. Torczynski, and M. A. Gallis. Direct simulation monte carlo on petaflop supercomputers and beyond. *Physics of Fluids*, 31(8), 8 2019.

[43] Chonglin Zhang and Thomas E. Schwartzentruber. Robust cut-cell algorithms for dsmc implementations employing multi-level cartesian grids. *Computers & Fluids*, 69:122–135, 2012.

[44] T.J. Scanlon, E. Roohi, C. White, M. Darbandi, and J.M. Reese. An open source, parallel dsmc code for rarefied gas flows in arbitrary geometries. *Computers and Fluids*, 39(10):2078–2089, 2010.

[45] Stefan Dietrich and Iain D. Boyd. Scalar and parallel optimized implementation of the direct simulation monte carlo method. *J. Comput. Phys.*, 126(2):328–342, jul 1996.

[46] Jose F. Padilla. *Comparison of DAC and MONACO DSMC codes with flat plate simulation.* NASA/TM- ; 2010-216835. National Aeronautics and Space Administration, Langley Research Center, Hampton, Va, 2010.

[47] Mikhail S. Ivanov, Alexander V. Kashkovsky, Sergey F. Gimelshein, Gennady Markelov, Alina A. Alexeenko, Yevgeny A. Bondar, G. A. Zhukova, S. B. Nikiforov, and P. V. Vaschenkov. Smile system for 2 d / 3 d dsmc computations. In *Proceedings of 25th International Symposium on Rarefied Gas Dynamics, St. Petersburg, Russia*, 2007.

[48] G. Di Staso, H.J.H. Clercx, S. Succi, and F. Toschi. Dsmc–lbm mapping scheme for rarefied and non-rarefied gas flows. *Journal of Computational Science*, 17:357–369, 2016. Discrete Simulation of Fluid Dynamics 2015.

[49] G. Di Staso, S. Srivastava, E. Arlemark, H. J.H. Clercx, and F. Toschi. Hybrid lattice boltzmann-direct simulation monte carlo approach for flows in three-dimensional geometries. *Computers and Fluids*, 172:492–509, August 2018.

[50] Ivan Girotto, Sebastiano Fabio Schifano, Enrico Calore, Gianluca Di Staso, and Federico Toschi. Performance and energy assessment of a lattice boltzmann method based application on the skylake processor. *Computation*, 8(2), June 2020. This Paper Is an Extended Version of Our Paper Published in the Proceedings of the ParCo2019: Mini-Symposium on Energy-Efficient Computing on Parallel Architectures, Prague, Czech Republic, 10–13 September 2019.

[51] Ivan Girotto, Sebastiano Fabio Schifano, Enrico Calore, Gianluca Di Staso, and Federico Toschi. Computational performances and energy efficiency assessment for a lattice boltzmann method on intel knl. In Ian Foster, Gerhard R. Joubert, Ludek Kucera, Wolfgang E. Nagel, and Frans Peters, editors, *Parallel Computing*, volume 36 of *Advances in Parallel Computing*, pages 605–613, Netherlands, 2020. IOS Press.

[52] Jens Wilke, Thomas Pohl, Markus Kowarschik, and Ulrich Rüde. Cache performance optimizations for parallel lattice boltzmann codes. In Harald Kosch, László Böszörményi, and Hermann Hellwagner, editors, *Euro-Par 2003 Parallel Processing*, pages 441–450, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[53] Christian Obrecht, Frédéric Kuznik, Bernard Tourancheau, and Jean-Jacques Roux. A new approach to the lattice Boltzmann method for graphics processing units. *Computers and Mathematics with Applications*, 2010.

[54] Christian Feichtinger, Johannes Habich, Harald Köstler, Georg Hager, Ulrich Rüde, and Gerhard Wellein. A flexible patch-based lattice boltzmann parallelization approach for heterogeneous gpu–cpu clusters. *Parallel Computing*, 37(9):536–549, 2011. Emerging Programming Paradigms for Large-Scale Scientific Computing.

[55] Mark J. Mawson and Alistair J. Revell. Memory transfer optimization for a lattice boltzmann solver on kepler architecture nVidia GPUs. *Computer Physics Communications*, 185(10):2566–2574, oct 2014.

[56] J. Kraus. An introduction to cuda-aware mpi. `http://devblogs.nvidia.com/parallelforall/introduction-cuda-aware-mpi/`, 2013.

[57] Harald J W Müller-Kirsten. *Basics of Statistical Physics*. WORLD SCIENTIFIC, 2nd edition, 2013.