Efficient parallelization for 3D-3V sparse grid Particle-In-Cell: Single GPU architectures

Fabrice Deluzet, G. Fubiani, Laurent Garrigues, Clément Guillet, Jacek Narski

▶ **To cite this version:**

Fabrice Deluzet, G. Fubiani, Laurent Garrigues, Clément Guillet, Jacek Narski. Efficient parallelization for 3D-3V sparse grid Particle-In-Cell: Single GPU architectures. Computer Physics Communications, 2023, 480, 10.1016/j.jcp.2023.112022 . hal-03773226v2

HAL Id: hal-03773226
https://hal.science/hal-03773226v2

Submitted on 17 Apr 2023

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Efficient parallelization for 3D-3V sparse grid Particle-In-Cell: Single GPU architectures

Fabrice Deluzet

Université de Toulouse; UPS, INSA, UT1, UTM,
Institut de Mathématiques de Toulouse,
CNRS, Institut de Mathématiques de Toulouse UMR 5219,
F-31062 Toulouse, France

Gwenael Fubiani

LAPLACE, Université de Toulouse, CNRS, INPT, UPS,
118 Route de Narbonne, 31062 Toulouse, France

Laurent Garrigues

LAPLACE, Université de Toulouse, CNRS, INPT, UPS,
118 Route de Narbonne, 31062 Toulouse, France

Clément Guillet*

Université de Toulouse; UPS, INSA, UT1, UTM,
Institut de Mathématiques de Toulouse,
CNRS, Institut de Mathématiques de Toulouse UMR 5219,
F-31062 Toulouse, France

LAPLACE, Université de Toulouse, CNRS, INPT, UPS,
118 Route de Narbonne, 31062 Toulouse, France

Jacek Narski

Université de Toulouse; UPS, INSA, UT1, UTM,
Institut de CNRS, Institut de Mathématiques de Toulouse UMR 5219,
F-31062 Toulouse, France

Abstract

In the present paper, an efficient General Purpose Graphical Processing Unit (GPGPU)-based implementation of sparse grid Particle-In-Cell (PIC) methods is proposed. The parallelization, implementing novel strategies specific to sparse-PIC methods and tailored to GPU architectures, provides speed-ups* as large as 100 on a single Tesla V100 GPU, with respect to sequential Computing Processing unit (CPU) execution; and a four order of magnitude reduction of the computational time in comparison with a standard PIC sequential CPU simulation. In addition, the simple implementation of the parallelization with the OpenACC framework offers portability to a large class of accelerators.

Keywords. Plasma physics, Particle-In-Cell (PIC), sparse grids, sparse grid combination technique, parallelization , GPU , OpenACC

*GPU speed-ups are established with respect to a sequential execution on a CPU corresponding to the same generation than the GPU.

*Corresponding author

Email addresses: fabrice.deluzet@math.univ-toulouse.fr (Fabrice Deluzet), gwenael.fubiani@laplace.univ-tlse.fr (Gwenael Fubiani), laurent.garrigues@laplace.univ-tlse.fr (Laurent Garrigues), clement.guillet@math.univ-toulouse.fr (Clément Guillet), jacek.narski@math.univ-toulouse.fr (Jacek Narski)

1. Introduction

Particle-In-Cell (PIC) schemes [1, 2, 15, 16, 21, 28, 33, 39] are widespread numerical methods used for the simulation of kinetic plasma problems. The method is based on a coupling between a Lagrangian discretization for the Vlasov equation, based on the integration of numerical particle trajectories and a mesh-based discretization of Poisson's equation for the computation of the self-consistent electrostatic field. For years, strategies have been developed to mitigate the statistical error originating from the sampling of the distribution function by a limited number of numerical particles, representing the major weakness of the method. In this context, noise reduction strategies such as variance reduction methods [18], filtering methods [24], or more recently sparse grid techniques [16, 22, 23, 35, 40] arouse a great interest. Sparse grids have been developed to interpolate high dimensional functions [3, 4, 5, 27], then, extended to the approximation of partial differential equations [25, 26, 42, 41]. Lately, sparse grids have been introduced in PIC framework, offering a significant mitigation of the statistical noise and a reduction of the grid operation complexity (*e.g.* the resolution of Poisson equation).

In the last decades, the emergence of General Purpose Graphics Processing Units (GPGPU) has dramatically disrupted the High Performance Computing (HPC) domain with the appearance of accelerators with thousands of compute cores achieving performance in the range of several TFLOP/s (10^{12} instructions per second). Most of supercomputers now employ up to thousands of Graphical Processing Units (GPU), resulting in a total of millions of compute cores. Therefore, an increasing interest of GPGPU for massively parallel applications has emerged in past decades. For instance, several GPU implementations of 1D and 2D PIC simulations have been proposed, demonstrating speed-ups in the range of 20-100 [8, 12, 13, 14, 6, 30, 20, 44, 31].

Recently, sparse-PIC method optimization and parallelization have been investigated in a shared memory architecture framework [17]. On the one hand, sparse-PIC methods have been proven to be particularly memory efficient, with respect to the size of the grids used to accumulate the density and compute the electric field, as well as the storage of the particles properties. The number of those numerical particles is significantly reduced, compared to standard methods, owing to the better control of the statistical noise. This limited memory footprint calls for the development of parallel implementations on a single GPU architecture, handling the memory requirements of sparse-PIC methods. On the other hand, sparse-PIC has demonstrated substantial speed-ups both for sequential and parallel implementation with respect to the standard PIC method. The sequential computational time of the standard method is reduced by two orders of magnitude with the sparse-PIC approach for an equivalent amount of statistical noise between the two simulations.

The present paper extends the works of [16, 17] to the parallelization of 3D-3V sparse-PIC methods on GPU architectures. Although GPUs offer a limited amount of memory in comparison to CPUs, this is not an issue for sparse-PIC implementations thanks to the substantial gains upon the memory footprint. The purpose of the present paper is to propose a first efficient GPU implementation of the sparse-PIC method with parallelization strategies tailored specifically for accelerator architectures. The sparse grid reconstructions require operations of each particle with numerous (tens) anisotropic grids, with coarse discretizations and different sparsity patterns. These grids are referred to as *component grids*, the set of nodes of all the component grids is being designated by the *sparse grid* terminology.

The efficient implementation introduced in this paper is compared to another one, which is merely the extension to GPU of sparse-PIC strategies introduced for shared memory CPU architecture in [17]. The novel implementation combines two level of parallelism for the charge density accumulation: a coarse-grain parallelism based on a particle population decomposition and a Single Instruction Multiple Data (SIMD) parallelism based on a component grid work sharing. This two-level parallelism exploits the architecture of the GPU decomposed into independent Streaming Multiprocessors (SM), each containing compute cores. The particle population is divided into a large number of particle clusters which are distributed onto the SMs. Within a cluster, each particle contribution is computed for all the component grids at once, enabling Single Instruction Multiple Threads (SIMT) fashion for the GPU. In addition, this approach offers a better management of the GPU cache memory (L2-cache) and helps to mitigate the randomness of the memory access, detrimental to GPU efficiency. One key element of the implementation is the absence of memory transfers between the host and the device during the simulation since all data fit on the GPU memory.

As a result of the large number of component grids involved, sparse-PIC methods offer a trade-off between memory access and computational instructions. This feature is capital for PIC implementations on GPU since PIC methods are globally memory bounded (limited by the memory accesses) and GPUs are designed to maximize the number of

instructions treated at once.

The organization of the paper is the following. In section 2, the sparse-PIC method and its major features are presented. The section 3 is devoted to the GPU parallelization of the method. A brief overview of the hardware architecture and usual GPU programming is provided with the aim to outline the algorithmic issues to achieve performance. Then, two implementations of the sparse-PIC method are presented: an extension to GPU of the parallel version designed for shared memory architectures, which has been introduced in [17], and a version tailored specifically for GPUs. Finally in section 4, the efficiency of the implementations are assessed on three dimensional classical test cases: the non-linear Landau damping and the diocotron instability using two different hardware: a laptop GPU (Quadro T2000) and a Tesla V100.

2. Sparse grid reconstructions for Particle-In-Cell methods

2.1. Notations

Let us introduce some notations for the following of the paper. Let d be the dimension of the problem considered, $\mathbf{l} = (l_1, ..., l_d) \in \mathbb{N}^d$ a multi-index denoting the level, *i.e.* the discretization resolution in a multivariate sense and $\mathbf{i} = (i_1, ..., i_d) \in \mathbb{N}^d$ be a multi-index denoting spatial positions. In the following, the index $\mathbf{l} = (l_1, l_2, l_3)$ is also represented $\mathbf{l} = (k, l, m)$. We define order relations on multi-index by:

$$\mathbf{k} \leq \mathbf{l} \;\Leftrightarrow\; \forall j \in \{1, ...d\}\; k_j \leq l_j, \tag{1}$$

$$\mathbf{k} < \mathbf{l} \;\Leftrightarrow\; \mathbf{k} \leq \mathbf{l} \text{ and } \exists\, j \in \{1, ..., d\}\; s.t.\; k_j < l_j, \tag{2}$$

and discrete l^1 norm and l^∞ norm by:

$$|\mathbf{l}|_1 := \sum_{j=1}^{d} |l_j|, \quad \|\mathbf{l}\|_\infty := \max_{j \in \{1,...,d\}} |l_j|. \tag{3}$$

2.2. Mathematical background

Let $f_s(\mathbf{x}, \mathbf{v}, t)$ be the phase-space distribution function attached to a species s (ion, electron, etc.), q_s, m_s the corresponding charge and mass, $\mathbf{E}$ the electric field and ρ the charge density. The non-relativistic Vlasov-Poisson system with external magnetic field $\mathbf{B}$ is considered:

$$\begin{cases} \dfrac{\partial f_s}{\partial t} + \mathbf{v} \cdot \nabla_{\mathbf{x}} f_s + \dfrac{q_s}{m_s}(\mathbf{E} + \mathbf{v} \times \mathbf{B}) \cdot \nabla_{\mathbf{v}} f_s = 0, \\ \nabla \cdot \mathbf{E} = \dfrac{\rho}{\varepsilon_0}, \qquad \mathbf{E} = -\nabla \Phi, \end{cases} \tag{4}$$

The charge density is obtained from the phase-space distribution of each species according to the relation:

$$\rho(\mathbf{x}, t) = \sum_s \rho_s(\mathbf{x}, t) = \sum_s q_s \int f_s(\mathbf{x}, \mathbf{v}, t) d\mathbf{v}. \tag{5}$$

The particle distribution f_s is numerically represented by a collection of N particles whose positions and velocities are denoted $(\mathbf{x}_p, \mathbf{v}_p)$, for $p \in [\![1, N]\!]$.

2.3. Sparse-PIC method

The main difference between the sparse grid PIC schemes and the standard PIC scheme [10, 11] is the underlying mesh of the method. Unlike standard methods, based on an unique fine Cartesian grid, the sparse grid PIC method is based on a collection of component grids upon which the grid operations (accumulation of the density, field solver, etc.) are performed. Let us consider the family of d-dimensional anisotropic grids $\Omega_{h_\mathbf{l}}$ on the unit interval $\Omega = [0, 1]^d$ called component grids:

$$\Omega_{h_\mathbf{l}} := \left\{ \mathbf{i} h_\mathbf{l} \mid \mathbf{i} \in I_{h_\mathbf{l}} \right\}, \quad I_{h_\mathbf{l}} := [\![0, h_{l_1}^{-1}]\!] \times ... \times [\![0, h_{l_d}^{-1}]\!] \subset \mathbb{N}^d, \tag{6}$$

and parameterized by the multi-index level $\mathbf{l} \in L$:

$$L := \bigcup_{j \in [\![0, d-1]\!]} L_j, \quad L_j := \{\mathbf{l} \in \mathbb{N}^d \mid |\mathbf{l}| = n + d - 1 - j,\ \mathbf{l} \geq \mathbf{1}\}, \tag{7}$$

where $h_{\mathbf{l}} := (h_{l_1}, ..., h_{l_d}) := 2^{-\mathbf{l}}$ is called the grid discretization. The number of component grid, denoted N_{sg}, is given by:

$$N_{sg} := \mathrm{Card}(L) = \sum_{j=1}^{d-1} \binom{n + d - 2 - j}{d - 1}, \tag{8}$$

which falls down to $\frac{(n+1)n}{2} + \frac{n(n-1)}{2} + \frac{(n-1)(n-2)}{2}$ for three dimensional computations. The Cartesian grid, denoted $\Omega_{h_n}^{(\infty)}$, and corresponding to a component grid of level $\mathbf{n} = n \cdot \mathbf{1}$ with discretization h_n in all directions, is also introduced.

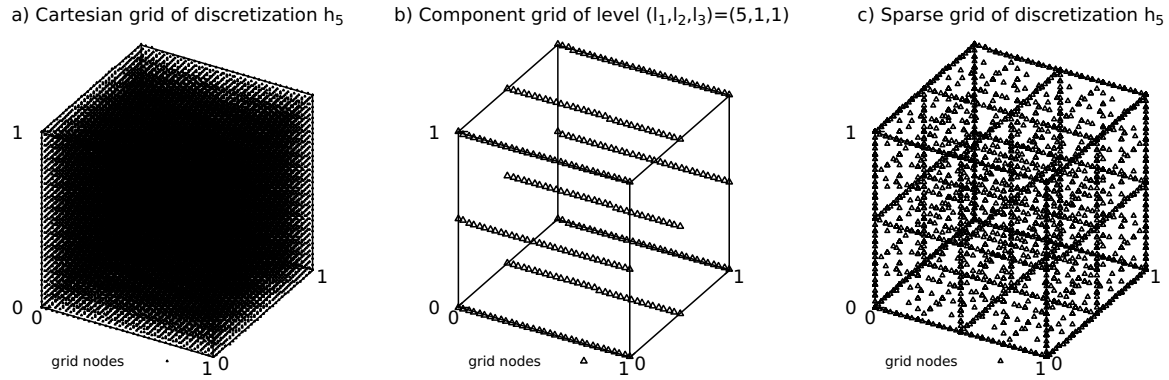

Figure 1: The discretization parameter is $n = 5$: Panel a) represents the Cartesian grid of discretization h_n; panel b) represents a component grid of level $\mathbf{l} = (5, 1, 1)$; panel c) represents the sparse grid, composed of all the nodes of all component grids.

Particle-In-Cell methods are based on a coupling between particles and a mesh-based discretization. Therefore, interactions between the particle population and the mesh, i.e. the component grids, shall be defined for the sparse-PIC methods. The charge density accumulation consists in a reconstruction of the charge density at the component grid nodes from the particles population. For each component grid $\Omega_{h_{\mathbf{l}}}$, an estimator of the density, denoted $\hat{\rho}_{h_{\mathbf{l}}}$, is constructed:

$$\hat{\rho}_{h_{\mathbf{l}}}(\mathbf{x}) = \frac{Q}{N} \sum_{p=1}^{N} \mathcal{S}_{h_{\mathbf{l}}} \left(\mathbf{x} - \mathbf{x}_p\right), \quad \mathcal{S}_{h_{\mathbf{l}}} = \bigotimes_{j=1}^{d} \left(h_{l_j}^{-1} \varphi\left(h_{l_j}^{-1} \cdot\right)\right), \quad \varphi(x) = \max\left(1 - |x|, 0\right). \tag{9}$$

The local error between the density and its statistical estimator can be decomposed into the bias of the estimator, referred to as the grid-based error and denoted $\mathrm{Bias}(\hat{\rho}_{h_{\mathbf{l}}})$, and a centered random variable, denoted $\mathcal{V}(\hat{\rho}_{h_{\mathbf{l}}})$ corresponding to the error stemming from the variance of the sampling with a finite number of particles:

$$(\hat{\rho}_{h_{\mathbf{l}}} - \rho)(\mathbf{x}) = \underbrace{\mathrm{Bias}(\hat{\rho}_{h_{\mathbf{l}}})(\mathbf{x})}_{\text{grid-based error}} + \underbrace{\mathcal{V}(\hat{\rho}_{h_{\mathbf{l}}})(\mathbf{x})}_{\text{particle sampling error}}, \tag{10}$$

where the grid-based error and the particle sampling error are defined by:

$$\mathrm{Bias}(\hat{\rho}_{h_{\mathbf{l}}}) = (\hat{\rho}_{h_{\mathbf{l}}} - \mathbb{E}(\hat{\rho}_{h_{\mathbf{l}}})), \quad \mathcal{V}(\hat{\rho}_{h_{\mathbf{l}}}) = (\mathbb{E}(\hat{\rho}_{h_{\mathbf{l}}}) - \rho), \tag{11}$$

and verify:

$$\text{Bias}(\hat{\rho}_{h_\mathbf{l}})(\mathbf{x}) = O\left(h_{l_1}^2, ..., h_{l_d}^2\right), \qquad \mathbb{V}(\mathcal{V}(\hat{\rho}_{h_\mathbf{l}}))^{\frac{1}{2}} = O\left((Nh_n)^{-\frac{1}{2}}\right). \tag{12}$$

The particle sampling error is the cause of PIC methods statistical noise [16].

A Poisson problem is considered on each component grid and approximated with finite difference method:

$$\mathbf{E}_{h_\mathbf{l}} = -\nabla_{h_\mathbf{l}}\Phi_{h_\mathbf{l}}, \quad \Delta_{h_\mathbf{l}}\Phi_{h_\mathbf{l}} = -\frac{\hat{\rho}_{h_\mathbf{l}}}{\varepsilon_0}, \tag{13}$$

where $\nabla_{h_\mathbf{l}}$ and $\Delta_{h_\mathbf{l}}$ are finite difference operators defined on $\Omega_{h_\mathbf{l}}$ (see [16]). From these component grid contributions of the electric potential, one shall reconstruct a precise solution on the refined Cartesian grid. Let $\mathbf{i} \in I_{h_\mathbf{l}}$, and consider basis functions defined by tensor products of one-dimensional hat functions as follows:

$$\varphi_{h_\mathbf{l},\mathbf{i}}(\mathbf{x}) := \left(\bigotimes_{j=1}^{d} \varphi_{h_{l_j},i_j}\right)(\mathbf{x}), \quad \varphi_{h_{l_i},i_j}(x) := \varphi\left(h_{l_j}^{-1}(x_j - i_j h_{l_j})\right), \tag{14}$$

where φ is defined in equation (9). The space of d-dimensional hat functions with respect to the component grid $\Omega_{h_\mathbf{l}}$, denoted $V_{h_\mathbf{l}}$, is defined by:

$$V_{h_\mathbf{l}} := \text{span}\{\varphi_{h_\mathbf{l},\mathbf{i}} \mid \mathbf{i} \in I_{h_\mathbf{l}}\}. \tag{15}$$

The sparse-PIC reconstruction of the electric potential is then defined by a linear combination of the component grid contributions interpolated onto $V_{h_\mathbf{l}}$:

$$\Phi_{h_n}^c = \sum_{\mathbf{l}\in L} c_\mathbf{l} I_{V_{h_\mathbf{l}}} \Phi_{h_\mathbf{l}}, \tag{16}$$

where $c_\mathbf{l} := (-1)^j \dfrac{(d-1)!}{j!(d-1-j)!}$ for $\mathbf{l} \in L_j$ are called the combination coefficients and $I_{V_{h_\mathbf{l}}}$ defines the interpolation onto the space $V_{h_\mathbf{l}}$. In this paper, we consider an interpolation in the hierarchical basis of the space $V_{h_\mathbf{l}}$, where a transformation from the nodal basis to a hierarchical basis is performed (see [17] for details). The rest of the scheme is similar to standard PIC methods and the sparse-PIC scheme is summarized in algorithm 1. It has been proved in [16] that the error between the solution and its sparse grid reconstruction can be decomposed into a grid-based error depending on the grid discretization and a particle sampling error depending on the mean number of particles per cell.

Proposition 2.1. *Let Φ and ρ be sufficiently smooth functions, then the grid-based error of the function $\Phi_{h_\mathbf{l}}$ defined by equation (13) verifies the following point-wise error expression for each $\mathbf{l} \in L$:*

$$\text{Bias}(\Phi_{h_\mathbf{l}})(\mathbf{x}) - \Phi(\boldsymbol{x}) = \sum_{m=1}^{d} \sum_{\substack{\{j_1,...,j_m\} \\ \subset\{1,...,d\} \\ j_i\neq j_k}} a_{j_1,...,j_m}(\boldsymbol{x}; h_{l_{j_1}}, ..., h_{l_{j_m}}) h_{l_{j_1}}^2 ... h_{l_{j_m}}^2 \tag{17}$$

with bounded $\|a_{j_1,...,j_m}(\cdot; h_{l_{j_1}}, ..., h_{l_{j_m}})\|_\infty \leq \kappa$. The local error between the sparse grid reconstruction $\Phi_{h_n}^c$ and the solution is recast into:

$$\Phi_{h_n}^c - \Phi = \underbrace{\text{Bias}(\Phi_{h_n}^c)}_{\textit{grid-based error}} + \underbrace{\mathcal{V}(\Phi_{h_n}^c)}_{\textit{particle sampling error}}, \tag{18}$$

and the grid-based error and the particle sampling error verify:

$$\left\|\text{Bias}(\Phi_{h_n}^c)\right\|_\infty \leq O\left(h_n^2 \cdot |\log h_n|^{d-1}\right), \quad \left\|\mathbb{V}\left(\mathcal{V}(\Phi_{h_n}^c)\right)^{\frac{1}{2}}\right\|_\infty \leq O\left((Nh_n)^{-\frac{1}{2}} |\log h_n|^{d-1}\right) \tag{19}$$

Algorithm 1 PIC-Sg scheme

Require: Particle positions and velocities $(\mathbf{x}_p, \mathbf{v}_p)$, time step Δt, external magnetic field $\mathbf{B}$.
for each time step Δt **do**
for each component grid $\Omega_{h_\mathbf{l}}$ of index $\mathbf{l} \in L$ **do**
Accumulate the charge density onto $\Omega_{h_\mathbf{l}}$ according to (9)
Compute the electric potential from the charge density on $\Omega_{h_\mathbf{l}}$ with finite differences according to (13)
Interpolate the electric potential onto $V_{h_\mathbf{l}}$ (15).
end for
Combine the electric potential onto $\Omega_{h_n}^{(\infty)}$ (16).
Differentiate the electric potential on $\Omega_{h_n}^{(\infty)}$.
Interpolate the electric field at the particle positions.
Update the particle positions and velocities:

$$\frac{d\mathbf{x}_p}{dt} = \mathbf{v}_p, \qquad \frac{d\mathbf{v}_p}{dt} = \frac{q_s}{m_s}(\mathbf{E} + \mathbf{v}_p \times \mathbf{B})|_{\mathbf{x}=\mathbf{x}_p}. \tag{20}$$

end for

Proof. We refer to [16] for the proof of this proposition. □

A capital feature of the sparse-PIC methods is the sparsity of the component grids upon which the density is accumulated and the Poisson problems are solved, resulting in a significant reduction of the statistical noise, as well as a mitigation of the linear system resolution computational effort. As a result, the particle sampling error is significantly reduced for the sparse-PIC method in comaprison to the standard approach (see table 2).

Table 2: Comparison between standard and sparse-PIC error components for the potential Φ.

method	grid-based error	particle sampling error
standard	$O\left(h_n^2\right)$	$O\left((N h_n^d)^{-\frac{1}{2}}\right)$
sparse-PIC	$O\left(h_n^2 \cdot \lvert \log h_n \rvert^{d-1}\right)$	$O\left((N h_n)^{-\frac{1}{2}} \lvert \log h_n \rvert^{d-1}\right)$

In order to control the statistical noise in the simulation, it is common for standard PIC methods to choose the number of numerical particle N accordingly to the mean number of particle per cell of the grid, referred to as P_c:

$$N = P_c * h_n^{-d}, \tag{21}$$

where h_n is the discretization of the Cartesian grid. A similar estimation is provided for the sparse-PIC methods, derived from the mean number of particles per cell of all the component grids:

$$\begin{aligned} N &= P_c * \left(\sum_{\mathbf{l} \in L} |c_\mathbf{l}| h_{l_1}^{-1} \dots h_{l_d}^{-1} \right) \\ &= P_c * h_n^{-1} * \left(\frac{9}{2} n^2 - \frac{3}{2} n + 1 \right), \quad \text{for } d = 3. \end{aligned} \tag{22}$$

It follows that an equivalent amount of statistical noise as in the standard scheme is obtained for much less particles (see [17, 16]), drastically reducing memory requirements of three dimensional simulations. This feature is capital for the sparse-PIC methods since memory requirements is often limiting for a lot of 3D-3V computations with standard methods.

The reduction of the number of particles in sparse-PIC methods comes along with an increase of the number of grid-particle operations. For instance, the charge of one particle shall be accumulated onto all the component grids, instead of one Cartesian grid for the standard method. The ranking of the step computational costs is therefore reversed

such that the charge accumulation consumes the largest part of the execution time of the algorithm [17] (around 90% of the time iteration, depending on the configurations). A particular attention shall then be payed to the optimization and parallelization of the charge accumulation, hence we mainly focus on this step in the present paper.

3. Implementations on Graphical Processing Units (GPU)

3.1. GPU architecture and programming

The hardware architecture of a GPU differs from that of a Computing Processing Unit (CPU) in some key aspects, the differences being inherited from the initial field of application of GPUs (realtime graphics) where the same instruction has to be applied to a large amount of data [43].

A common CPU is optimized to minimize memory latency, since fetching data from the (off chip) main memory is a very time consuming operation. Therefore, CPU cores have a complex structure and involve out-of-order execution, branch prediction, memory pre-fetching and cache hierarchy, the purpose of all these optimizations being to improve the performance in a Single Instruction Single Data (SISD) fashion. By contrast, GPUs are optimized to maximize throughput, i.e. allowing to execute as many tasks as possible at once. In order to achieve this, a large number of cores, as simple as possible, is required, thus removing all logic that boosts single instruction stream performance but gaining the ability to put more cores on a chip.

A single GPU device consists of multiple Streaming Multiprocessors (SM). The streaming processors can be operated independently. One SM contains tens (*e.g.* 32) compute cores working in a Single Instruction Multiple Thread (SIMT) fashion, meaning that all instructions in all threads are executed in lock-step. A SM contains thousand of registers in order to run a large number of threads simultaneously and perform fast "context switching" between different warps. Typically a SM is assigned 8 thread blocks, consisting of tens of warps, a warp being composed of 32 threads (from the compute capability 2.x, the number of threads in a thread block may exceed 1024).

The SM is oversubscribed with thousands of threads (for tens of cores) in order to hide the memory latency: the warp scheduler of the SM switch quickly from warps stalled due to memory latency to resume a warp for which the data are ready; hence the need of rapid context switching.

The accesses to the GPGPU main memory are coalesced within the threads of a warp. The load and store from and to the main memory are organized in chunks of contiguous memory addresses, these chunks being aimed at feeding all the threads in a warp. To minimize the number of transactions with the main memory, memory accesses within the block shall be ordered to maximize performance: the k^{th} thread in a warp shall access the k^{th} element in the memory chunk.

Several programming languages are available for programmers who want to use GPU resources to accelerate general computational applications, the most widespread being the Compute Unified Device Architecture (CUDA) [37, 9, 7], specific to Nvidia GPUs. However, with the appearance of modern GPUs from competitive brands such as AMD or Intel the restriction to Nvidia GPUs could reveal limiting. Thus, other languages such as OpenMP 4., OpenCL [34] or OpenACC [29] have recently become popular. In the present paper, we implement our code with OpenACC which is an Application Programming Interface (API) written in C, C++, Fortran. OpenACC is a directive-based host driven language, meaning that the host (CPU) is responsible for launching every operations executed on the device (GPU) including execution of kernels (code running on a GPU), allocation of memory and data transfers. OpenACC is not a low-level programming language like CUDA, it allows more portability of the code thanks to an abstraction of the hardware. Though the fine tuning of the compute kernels to the GPU architecture is not in the scope of this API, OpenACC exposes the three levels of parallelism available on an accelerator, namely coarse-grain, fine-grain and SIMD. Gang parallelism is the highest level of parallelism (coarse-grain), equivalent to CUDA thread block, gangs being executed independently of each other without synchronization. The worker level (fine-grain) involves workers, similar to CUDA warps, that share data within a gang. The innermost level of parallelism, the vector parallelism, equivalent to CUDA thread concept, is based on SIMT execution model: an instruction is executed on vector of data.

3.1.1. Memory hierarchy

The memory architecture of a GPU also differs from that of a CPU in several aspects. The Nvidia Tesla V100 memory architecture is considered as a representative example in the following. Within GPGPU applications, the device (GPU) does not operate on the host (CPU) main memory but is connected to its own off-chip memory (DRAM).

The data have to be transferred from the host global memory to the GPU specific memory. The bandwidth between the device (GPU) and its specific memory is much higher (897 GB/s) than the bandwidth between host memory (CPU) and device memory (16 GB/s)[1]. Hence, for best overall performance, it is capital to minimize data transfers between the host and the device, even if that means running kernels on the GPU that do not demonstrate any speedup compared to running them on the host CPU. This is the policy followed within this work.

Compared to a CPU, a GPU works with fewer, and relatively small, memory cache layers. The memory hierarchy is sketched in the following lines:

- The main memory (DRAM) consisting of 16GB accessed with a theoretical peak bandwidth of 897GB/s. Global memory can be read and written by all the threads on the GPU.
- The constant memory (64 KB read-only memory) which is faster than global memory because it is cached. Constant memory can be read by all the threads on the GPU.
- The L2 cache of 6.3MB shared by all SMs with a theoretical peak bandwidth of 4.1TB/s can be read and written by all threads.
- The L1 data cache of 128KB per SM, made of shared memory and texture memory. Shared memory provides high bandwidth and low latency but can only be read and written by the threads belonging to the same thread block (in CUDA terminology). It has a theoretical peak bandwidth of approximately 14TB/s.
- The register file of 256KB for each SM (16,384 32-bit registers on each processing block, 4 per SM). It allows fast read-write operations to the data stored in it. Registers are private to one thread and can only be accessed by the owning threads.

As mentioned before, the memory architecture is designed to optimize coalesced data transfers with the main memory. Consecutive threads from the same warp should access consecutive blocks of memory addresses in order to optimally exploit the memory bandwidth.

As a summary, in order to conceive an efficient GPU algorithm, the following policies shall be respected:

1. Limit the transfers between host (CPU) and device (GPU).
2. Favor the locality of the data.
3. Encourage coalesced data accesses with the memory.
4. Create as many independent task as possible to mask the memory latency.

The GPGPU-specific algorithm proposed in the present article is compliant with 1 and 4, and to some extent 2, but not to 3 because of the nature of the particle-grid operations. These points are outlined in the sequel.

3.2. Data management

Sparse-PIC methods dramatically reduce the memory footprint of PIC computations thanks to the significant diminution of the number of particles necessary to maintain an appropriate statistical noise. Let us investigate the memory requirements of both the standard and the Sparse PIC methods. The amount of data in bytes [B] to handle resulting from N numerical particles is given by:

$$DataParticles\ [B] = N * 3 * (SizePositionData + SizeVelocityData + SizeAccelerationData). \tag{23}$$

[1]For a node of the supercomputer OLYMPE from CALMIP, equipped with a Nvidia Tesla V100 associated to an Intel® Skylake CPU with a PCI GEN3 16X bus of 15.8 GB/S.

Position, velocity and acceleration data are double precision, requiring 8 Bytes [B]. The data size for a contribution (charge density or electric potential) of all the component grids can be bounded by:

$$DataComponentGrids\ [B] = SizeData * \sum_{\mathbf{l}\in L}\left(\prod_{j=1}^{3}(2^{l_j}+1)\right)$$
$$\leq SizeData * N_{sg} * 9 * (2^n+1). \tag{24}$$

For instance for n ranging from 7 to 10, corresponding to configurations equivalent to 128^3 and 1024^3 grids with respect to the standard method ($h_7 = 1/128$ to $h_{10} = 1/1024$), the data size of the component grid ranges from 594KB to 10MB. It results in a significant reduction compared to the Cartesian grid of the PIC method:

$$DataCartesianGrid\ [B] = SizeData * (2^n+1)^3, \tag{25}$$

which requires 17MB to 8GB for n ranging from 7 to 10. A comparison of the particle data requirements, relative to the number of particle per cell (defined in equations (21), (22)) and the grid data sizes is provided on figure 4.

This outlines one main advantage of Sparse PIC method over standard ones when porting to GPGPU: the whole data necessary during the simulation fit into the device memory of most accelerators (tens of GB capacity), even for configurations equivalent to 1024^3 grids. Therefore our data management strategy shall capitalize on this observation. In order to avoid as much as possible data transfers between the host and the device, the data shall stay on the device memory throughout the whole simulation. One unique data transfer is realized at the initialization to send all the data on the device.

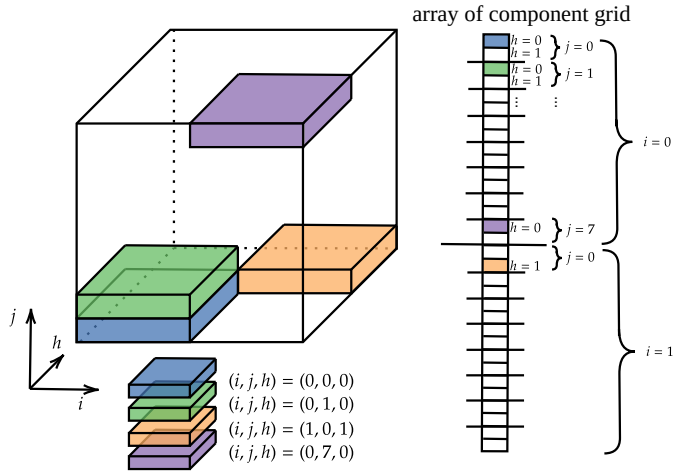

Figure 3: 1-dimensional structure array of a component grid. The cells of the three-dimensional grid are arranged in a one-dimensional array where the z-dimension is the fast axis and the x-dimension is the slow axis.

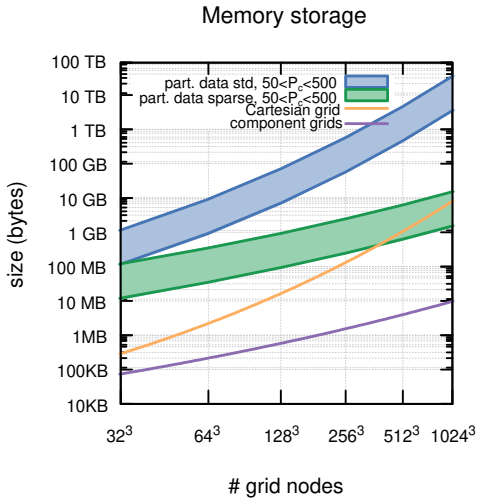

Figure 4: The storage size of data (particle data, Cartesian grid data and component grid data) is represented as a function of the number of grid nodes and particles per cell.

3.3. Data structure

The state of each numerical particle is described by its position, velocity and acceleration (corresponding to the electric field component) and is represented by the tuple $\langle x, y, z, v_x, v_y, v_z, a_x, a_y, a_z\rangle$. The position, velocity and acceleration coordinates are represented by double-precision floats. The particle data are represented by three arrays of N rows and three columns:

$$double\ \mathbf{x}_p[1:N,1:3], \mathbf{v}_p[1:N,1:3], \mathbf{a}_p[1:N,1:3];$$

where N is the number of particles.

In order to represent the set of component grids, a three dimensional array (standing for one component grid) is required for all the grids. Since the size of the component grids differs from one to another, a more complex data structure than a four-dimensional array is required. In this paper, two different data structures policies representing the set of component grids are considered. The component grids are either represented by an Array of Structure (AoS) or a two-dimensional array.

The first policy, being the most natural and consisting in an AoS, has been introduced in [17] specifically for the parallelization of sparse-PIC methods on shared memory (CPUs) architecture. With this policy, a data structure is created to store the dimensions and contributions (charge density, electric potential) of a component grid:

$$
\begin{aligned}
&type\ \ component\_grid\\
&\quad integer :: k, l, m;\\
&\quad double :: \rho(0:2^k, 0:2^l, 0:2^m), \Phi(0:2^k, 0:2^l, 0:2^m);\\
&end\ \ type
\end{aligned}
$$

where k, l, m are the dimension of the component grid and ρ, Φ are the arrays containing the charge density and the electric potential contributions. All the information of the component grids are stored in a single array whose elements are component grid data structures:

$$type(component\_grid) \ :: comp\_grid[1:N_{sg}],$$

where N_{sg} is the number of component grids, given by equation (8). Let us recall that for sparse-PIC parallelization strategies designed for CPUs, the particle properties are accumulated onto one component grid for all the particles, this repeated for each component grid (see [17] for details). This strategy, together with the AoS data structure, entails locality of the data. For the deposition of the particle density onto a component grid, the array used to store one component grid being small enough to be nursed in L1 cache of a CPU core. However, in the following, a GPU implementation based on the converse strategy, consisting in the accumulation of one particle property onto all the component grids, repeated for each particle, is introduced. Therefore, in order to preserve a good locality of the data between the component grids, a specific data structure is introduced. The rationale for this strategy is proposed in the next section.

This second policy is based on a transformation of the component grids from a three-dimensional structure into a one-dimensional structure. Each component grid is reshaped into a one-dimensional array (see figure 3) and all the one-dimensional component grid contributions are stored in an array of N_{sg} rows and $9(2^n + 1) + 1$ columns:

$$double \ :: \rho[1:N_{sg}, 1:9*(2^n+1)], \Phi[1:N_{sg}, 1:9*(2^n+1)].$$

This data structure is based on an upper estimation of the total number of component grid nodes. Indeed, the largest (in term of number of grid nodes) of the component grids are the most anisotropic ones, *i.e.* the ones corresponding to the levels $\mathbf{l} = (n, 1, 1); (1, n, 1); (1, 1, n)$, each containing $9 * (2^n + 1)$ grid nodes.

3.4. GPGPU charge deposition

3.4.1. Charge density accumulation algorithm

The accumulation of the particle properties onto the component grids accounts for more than 90% of a sparse-PIC iteration (for a sequential execution on CPU). This step is therefore anticipated to be the key point to obtain an efficient parallel implementation on GPGPU. It consists mainly in reading one particle coordinates, computing the grid cell it is contained in, and accumulate the contribution onto the 8 nodes of this cell. These operations are repeated for each grid. During the procedure, two consecutive particles (with regard to their indices in the particle coordinate array) may contribute to different cells in the density array. Therefore either a contiguous memory access in the particle array or in the density (grid) array occurs. Standard implementations usually entail random memory accesses in the density array. In addition, the paralellization of the density accumulation may lead to race conditions when threads associated to different particles add their contributions to the same grid cell, writing at the same memory address. This issue is usually bypassed with either private copies of grid arrays and reduction operations or atomic operations.

3.4.2. Why sparse-PIC parallel implementations designed for shared memory (CPU) architectures are not efficient on GPUs

Let us first recall the main features of the sparse-PIC parallel implementation designed for CPUs, then, in the following of the section, we introduce the extension of the implementation to GPUs, named CPU-inherited implementation and finally investigate the limitations of the algorithm. The implementation details of the CPU-inherited algorithm are provided in algorithm 2.

sparse-PIC parallelization efficiency on CPU is chiefly based on cache memory reuse. The non contiguous memory accesses to the grid data are mitigated by the large number of L1-cache hits. The cache reuse is maximized by considering the following charge density accumulation policy: the interactions of all the particles are computed with one component grid and repeated as many times as the number of component grids. Therefore, since each component grid fits in the L1-cache, the number of grid data cache misses is dramatically reduced, though the irregular (non contiguous) accesses.

sparse-PIC parallelization designed for CPUs takes benefit from the natural parallelism offered by the deposition onto the different component grids. The operations attached to two different component grids are independent and can therefore be processed concurrently (by different CPU-cores). Nonetheless this only level of parallelism is not sufficient to provide a good load balance for tens of cores. Therefore, a second level of parallelism is exploited: the particle sampling work sharing. It amounts to decompose the particle population into clusters, distributed onto the different threads. Each thread operates on its own sample of particles, accumulating the particle properties onto a private array. These private copies of the grid data are mandatory to avoid race conditions between threads assigned to different particles. Finally a reduction operation between the private copies is performed to gather the different contributions.

The hurdles of an extension to GPGPU are of different nature. First, the number of threads initiated within a SM (thousands) is large compared to CPUs (tens). This is particularly important for particle-grid operations, the memory accesses being genuinely non contiguous and non coalesced. Therefore, in order to mask the memory latency of the data transfer and achieve the best performance on GPU, one shall run a large number of instruction streams, *i.e.* a large number of gangs in OpenACC terminology. Nonetheless the number of gangs cannot be chosen as large as it shall be to maximize the performance of the CPU-inherited implementation because of the reduction operation requiring copies of the grids for each gang. The memory capacity, as well as the number of reduction operations to operate the array copies may be a limiting feature for a large number of gangs with this implementation. In addition, the method does not provide an effective memory access pattern of the component grid structure. The randomness of the spatial distribution for two consecutive particles in the particle array results in non-coalesced (random) memory accesses in the component grid array. The cumulative features of non efficient memory accesses and poor number of gangs (being the result of the reduction operation) may limit the efficiency of CPU-inherited implementation on GPU.

Second, the L1-cache is private to one CPU core while it is shared by all the SIMD processors within a SM. Each CPU core owns a private L1-cache of usually 32KB whereas GPU cores within a SM share a L1-cache of 128KB on the Tesla V100. As a result, based on the maximum number of threads within a SM for the Tesla V100, which is 2048, each thread has roughly 62B of available L1-cache memory which is significantly less than the 32KB L1-cache memory of the CPU cores. The private L1-cache being at the core of the efficiency of the CPU implementation of the sparse-PIC charge deposition (see [17]), it explains why an efficient GPGPU parallel implementation can not be conceived as a CPU parallel implementation run with thousand of threads rather than tens.

A third difficulty shall be pointed out here: OpenACC only provides a weak control of the cache hierarchy. On top of that, in the CPU parallel implementation, a reduction gathering all the particle contributions is performed for each component grids on a three-dimensional array ρ corresponding to the AoS component grid data structure (see section 3.3). OpenACC only offers reduction on the coarsest grain parallelism (gang level loop) via the "REDUCTION" clause, restricting the parallelization wihtin the CPU-inherited policy (*i.e* the interactions of all the particles are computed with one component grid and repeated as many times as the number of component grids). It results from this implementation a number of independent kernels equal to the number of component grids (N_{sg}). Therefore the two-level parallelization strategy, efficient on CPUs, is limited on GPUs.

3.4.3. Why CPU and GPU implementations of standard PIC methods are not suitable for sparse-PIC methods

The main objective of an efficient parallelization strategy for the charge density accumulation within PIC methods is to mitigate the randomness of the data access (see section 3.4.1) and entail contiguous (CPU) or coalesced (GPU)

Algorithm 2 GPGPU implementation of the CPU-inherited charge accumualtion algorithm.

Require: Particle position array $\mathbf{x}_p[1:N, 1:3]$, AoS $comp\_grid[1:N_{sg}]$, weight of particles ω.
Variables: Integer: i_x, i_y, i_z, real: $i_{xr}, i_{yr}, i_{zr}, s_{x1}, s_{x2}, s_{y1}, s_{y2}, s_{z1}, s_{z2}$
for each component grid $i \in [\![1, N_{sg}]\!]$ **do**
$k \leftarrow comp\_grid[i]\%k$; $l \leftarrow comp\_grid[i]\%l$;$m \leftarrow comp\_grid[i]\%m$
!$ACC PARALLEL NUM_GANGS() VECTOR_LENGHT()
!$ACC LOOP REDUCTION (+:$\rho$) //Parallelism on the SMs and the cores of the SMs
for each particle $i_p \in [\![1, N]\!]$ **do**
// Read particle data
$i_{xr} \leftarrow \mathbf{x}_p[ip, 1]/2^{n-k}$;
$i_{yr} \leftarrow \mathbf{x}_p[ip, 2]/2^{n-l}$;
$i_{zr} \leftarrow \mathbf{x}_p[ip, 3]/2^{n-m}$;
// Determine grid cell containing particle
$i_x \leftarrow i_{xr}$; $i_{xr} \leftarrow i_{xr} - i_x$;
$i_y \leftarrow i_{yr}$; $i_{yr} \leftarrow i_{yr} - i_y$;
$i_z \leftarrow i_{zr}$; $i_{zr} \leftarrow i_{zr} - i_z$;
// Determine charge contribution of particle
$s_{x1} \leftarrow (1 - i_{xr}) * 2^k$; $s_{x2} \leftarrow i_{xr} * 2^k$;
$s_{y1} \leftarrow (1 - i_{yr}) * 2^l$; $s_{y2} \leftarrow i_{yr} * 2^l$;
$s_{x1} \leftarrow (1 - i_{zr}) * 2^m$; $s_{z2} \leftarrow i_{zr} * 2^m$;
// Add contribution to the grid
$comp\_grid[i]\%\rho[i_x, i_y, i_z] \leftarrow comp\_grid[i]\%\rho[i_x, i_y, i_z] + s_{x1} * s_{y1} * s_{z1} * \omega$;
$comp\_grid[i]\%\rho[i_x + 1, i_y, i_z] \leftarrow comp\_grid[i]\%\rho[i_x + 1, i_y, i_z] + s_{x2} * s_{y1} * s_{z1} * \omega$;
$comp\_grid[i]\%\rho[i_x, i_y + 1, i_z] \leftarrow comp\_grid[i]\%\rho[i_x, i_y + 1, i_z] + s_{x1} * s_{y2} * s_{z1} * \omega$;
$comp\_grid[i]\%\rho[i_x + 1, i_y + 1, i_z] \leftarrow comp\_grid[i]\%\rho[i_x + 1, i_y + 1, i_z] + s_{x2} * s_{y2} * s_{z1} * \omega$;
$comp\_grid[i]\%\rho[i_x, i_y, i_z + 1] \leftarrow comp\_grid[i]\%\rho[i_x, i_y, i_z + 1] + s_{x1} * s_{y1} * s_{z2} * \omega$;
$comp\_grid[i]\%\rho[i_x + 1, i_y, i_z + 1] \leftarrow comp\_grid[i]\%\rho[i_x + 1, i_y, i_z + 1] + s_{x2} * s_{y1} * s_{z2} * \omega$;
$comp\_grid[i]\%\rho[i_x, i_y + 1, i_z + 1] \leftarrow comp\_grid[i]\%\rho[i_x, i_y + 1, i_z + 1] + s_{x1} * s_{y2} * s_{z2} * \omega$;
$comp\_grid[i]\%\rho[i_x + 1, i_y + 1, i_z + 1] \leftarrow comp\_grid[i]\%\rho[i_x + 1, i_y + 1, i_z + 1] + s_{x2} * s_{y2} * s_{z2} * \omega$;
end for
!$ACC END LOOP
!$ACC END PARALLEL
end for

memory accesses.

The most natural strategy to deal with randomness of data accesses is inspired from parallelization strategies designed for CPUs and consists in a sorting of the particle population and distribution into clusters. Usually, on CPUs, the particle population is sorted so that most of the time consecutive particles write their contributions in the same density array cells, enabling efficient cache memory reuse.

CPU particle sorting is not applicable to sparse-PIC computations on GPUs though, because, in order to optimize memory transfer, one shall fetch data from the device memory in a coalesced fashion. Therefore, an effective sorting shall result in a particle array where consecutive particle correspond to consecutive grid cells, *i.e.* one sorted particle array for each component grid, which is a way too cumbersome data constraint.

The most efficient GPGPU parallelization strategy to reduce the effects of the irregular memory accesses is to consider the shared memory of the device [30, 6, 20, 44]. With this approach, a private copy of the density array is created in the shared memory of each SM (actually each gang). The accumulation of the particles properties onto the grid is performed with shared-memory atomic operations for threads from the same gang. Finally, a global reduction operates between the different copies of the density array from each gang. This strategy is usually performed along with a particle cluster work sharing strategy, where the particle population is divided into clusters of particles, each assigned to a thread block. In order to ensure that all particles in a cluster are stored contiguously and can deposit to the accumulating density array in the shared memory, a sorting based on the cluster index every time step may be necessary [31, 13].

Whereas CUDA provides a simple and effective way to use the shared memory of the GPU, this features is not available in a transparent and straightforward way on OpenACC. The "CACHE" directive allows to access the shared memory but only for simple memory access patterns and does not suit the algorithm specificity .

3.4.4. Sparse-PIC implementation for GPGPUs

The extension to GPU of the implementation designed for CPUs reveals an inefficient use of the computational capacities since the kernels (each dedicated to the accumulation of the particle properties onto one component grid) are executed in sequential, one after the others. This is the result of the CPU-inherited policy and OpenACC restrictions. In addition, the CPU-inherited implementation does not take advantage of the cache memory as it does on CPUs. We therefore propose a second implementation of the accumulation step for GPGPUs, named GPGPU-specific implementation, taking advantage both of the independent computations between the component grids and the large L2-cache memory capacity.

This strategy is based on a component grid work sharing principle in a SIMT pattern, *i.e.* the operations on the component grids are realized in a SIMT fashion: each thread within a gang operates on a different component grid. This strategy is coupled with a particle work sharing strategy at the gang level, where the particle population is divided into clusters and distributed into the gangs similarly to the CPU-inherited implementation (see figure 5):

- The first level (coarse-grain) of parallelism is based on the distribution of the particle population into clusters. The gangs (or thread blocks) are associated to the clusters and distributed to the SMs in a Single Program Multiple Data (SPMD) fashion.
- The second level (SIMT) of parallelism is based on the component grid work sharing principle. Within each gang, the threads operate on the component grid at the same time in a SIMT fashion. *E.g.* if there are 32 threads in a gang, the particle properties of the particle cluster are accumulated onto the first 32 component grids simultaneously by the threads.

The number of clusters is expected to be large (a lot larger than the number of SMs) so that a large number of thread blocks is enabled and the interleave stream strategy shall efficiently mask the non coalesced memory accesses.

The goal of the GPGPU algorithm is to exploit the GPU architecture so that the device is fed with a large number of similar instructions on multiple data (accumulation of a particle from one cluster onto all the component grids at once). Nonetheless, in order to avoid the limitations resulting from the reduction operation, an array shared between the gangs is considered along with ATOMIC operations. The details of the implementation are given in algorithm 3.

Unlike the CPU-inherited implementation, the interaction of one particle is computed with all the component grids at once, and repeated for all the particles. It provides an unique particle data memory access for all the component grids, reducing therefore the number of data transfer with the device memory. The two-dimensional component grid

data structure is considered (see section 3.3) to ensure that threads from the same gang access consecutive memory addresses.

One of the benefits of the method concerns the cache memory reuse. Thanks to the shared status of the density array, it is mutual to all gangs and thus it can benefit from the large size of the L2 cache to mitigate the cost due to the random accesses to the grid array, the data size of the component grid ranging from 594KB to 10MB for $n = 7$ up to $n = 10$.

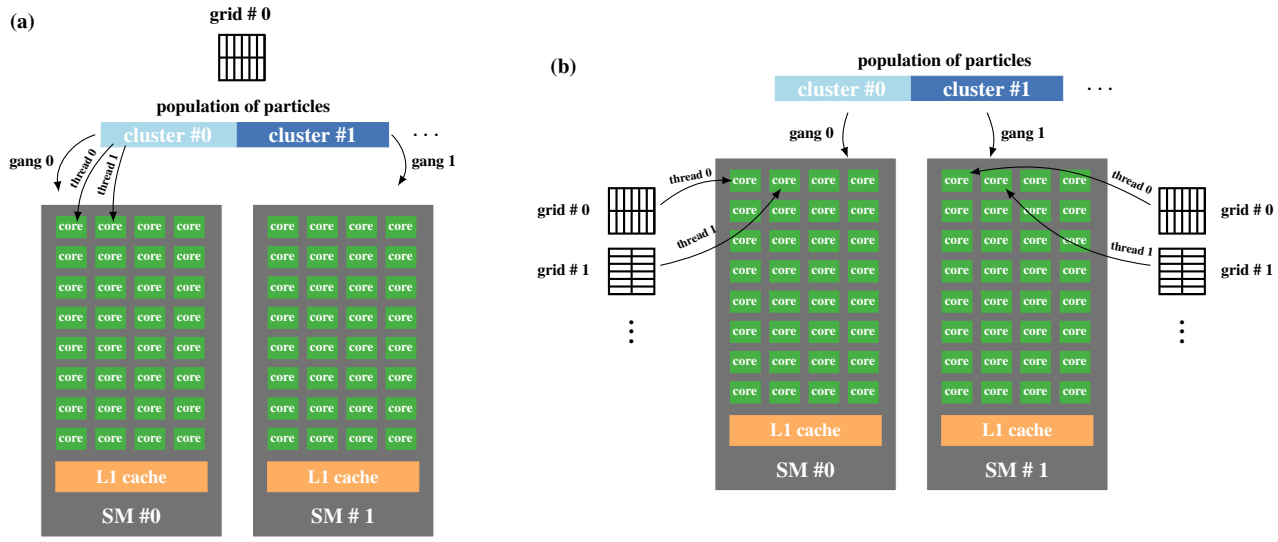

Figure 5: CPU-inherited (panel a) and GPGPU implementations (panel b) of the charge deposition. The operation is repeated for all the component grids wihtin the CPU-inherited method. In both implementations, the particle population is divided into clusters of particles and distributed onto the SMs (associated to the gangs/thread blocks). In the CPU-inherited algorithm (of the first component grid), the clusters are again divided ad distributed to the threads of the SM (gangs/thread blocks). In the GPGPU algorithm, the threads from the same gang operate simultaneously on the component grids in a SIMT fashion.

3.4.5. Resolution of Poisson equation, field interpolation, particle pusher, etc. algorithms

Sparse-PIC methods offer a significant alleviation of the grid operations with respect to the standard methods, resulting from a diminution of the grid nodes constituting the mesh of the method. Grid quantities are computed on each component grids, the operations being independent from one grid to another. It results in several independent linear systems to solve for the resolution of Poisson equation and necessitates novels parallelization strategies. It exists GPU-based libraries offering tools for the resolution of linear system such as AMGx, MAGMA, and CUDA libraries (CuSolver, CuBlas, CuSparse).

Different strategies may be considered: the first strategy consists in solving the linear systems issued from the dicretization of the Poisson problem on the component grids one after the other. The advantage of this strategy lies in the very small size of the linear systems. The second strategy consists in gathering all these problems into a single (by block) linear system. Solving this single system is a more computational expensive task, however it can better benefit from the computational capacity of the GPU.

For the field interpolation and the particle pusher, a straightforward parallelization, based on a decomposition of the particle population and distribution onto the threads, is proposed. No competitive memory access between the threads (attached to different particles) leading to race conditions is involved. For the combination, a similar strategy to the one introduced in [17] for the dehierarchization principle (transformation from the hierarchical basis to the nodal basis), is applied both to the hierarchization (transformation from the nodal basis to the hierarchical basis) and the dehierarchization. It exploits the tensor product structure of the basis functions. One-dimensional operations are performed on a collection of two-dimensional poles (see [17]), which are independent and therefore can be paralllelized. This parallelization strategy is not optimal but the combination step usually counts for thousandths of one iteration and therefore a finer parallelization has not proven to be mandatory to obtain a good efficiency.

Algorithm 3 GPU-based algorithm of charge density accumulation.

Require: Particle position array $\mathbf{x}_p[1:N, 1:3]$, 2d-structure $\rho[1:N_{sg}, 1:9*(2^n+1)]$, weight of particles ω.
Variables: Integer: $i_x, i_y, i_z, i_1, i_2, i_3, i_4, i_5, i_6, i_7, i_8$, real: $i_{xr}, i_{yr}, i_{zr}, s_{x1}, s_{x2}, s_{y1}, s_{y2}, s_{z1}, s_{z2}, x_p, y_p, z_p$
!$ACC PARALLEL NUM_GANGS() VECTOR_LENGHT()
!$ACC LOOP GANG //Coarse-grain parallelism on SMs, Single Program Multiple Data (SPMD) fashion
for each particle $i_p \in [\![1, N]\!]$ **do**
// Read particle data from device memory
$x_p \leftarrow \mathbf{x}_p[ip, 1]$;
$y_p \leftarrow \mathbf{x}_p[ip, 2]$;
$z_p \leftarrow \mathbf{x}_p[ip, 3]$;
!$ACC LOOP VECTOR //Fine-grain parallelism on the cores of the SM, SIMT fashion
for each component grid $i \in [\![1, N_{sg}]\!]$ **do**
$k \leftarrow comp\_grid[i]\%k$; $l \leftarrow comp\_grid[i]\%l$;$m \leftarrow comp\_grid[i]\%m$;
// Adapt particle data to the grid
$i_{xr} \leftarrow x_p/2^{n-k}$;
$i_{yr} \leftarrow y_p/2^{n-l}$;
$i_{zr} \leftarrow z_p/2^{n-m}$;
// Determine grid cell containing particle
$i_x \leftarrow i_{xr}$; $i_{xr} \leftarrow i_{xr} - i_x$;
$i_y \leftarrow i_{yr}$; $i_{yr} \leftarrow i_{yr} - i_y$;
$i_z \leftarrow i_{zr}$; $i_{zr} \leftarrow i_{zr} - i_z$;
// Determine charge contribution of particle
$s_{x1} \leftarrow (1 - i_{xr}) * 2^k$; $s_{x2} \leftarrow i_{xr} * 2^k$;
$s_{y1} \leftarrow (1 - i_{yr}) * 2^l$; $s_{y2} \leftarrow i_{yr} * 2^l$;
$s_{x1} \leftarrow (1 - i_{zr}) * 2^m$; $s_{z2} \leftarrow i_{zr} * 2^m$;
// Determine cell of the 2d-structure
$i_1 \leftarrow i_z + i_y * (2^m + 1) + i_x * (2^m + 1) * (2^l + 1)$;
$i_2 \leftarrow i_1 + (2^m + 1) * (2^l + 1)$;
$i_3 \leftarrow i_1 + 2^m + 1$; $i_4 \leftarrow i_2 + 2^m + 1$;
$i_5 \leftarrow i_1 + 1$; $i_6 \leftarrow i_2 + 1$; $i_7 \leftarrow i_3 + 1$; $i_8 \leftarrow; i_4 + 1$
// Add contribution to the grid
!$ACC ATOMIC UPDATE
$\rho[i, i_1] \leftarrow \rho[i, i_1] + s_{x1} * s_{y1} * s_{z1} * \omega$;
!$ACC ATOMIC UPDATE
$\rho[i, i_2] \leftarrow \rho[i, i_2] + s_{x1} * s_{y2} * s_{z1} * \omega$;
...
!$ACC ATOMIC UPDATE
$\rho[i, i_8] \leftarrow \rho[i, i_8] + s_{x2} * s_{y2} * s_{z2} * \omega$;
end for
!$ACC END LOOP
end for
!$ACC END LOOP
!$ACC END PARALLEL

The differentiation is also straightforward to parallelize. The nodes of the Cartesian grid are distributed to the threads and gangs in a way decided by the compiler.

4. Numerical results

4.1. Setup and performance metrics

The domain is a periodic cube $\Omega = (\mathbb{R}/L\mathbb{Z})^3$, of dimension L. Dimensionless variables are considered, the reference length and time units being the Debye length $\lambda_D = (\varepsilon_0 T_e/q_e n_0)^{\frac{1}{2}}$ and the plasma period $\omega_p^{-1} = (q_e n_0/m_e\varepsilon_0)^{-\frac{1}{2}}$. The electrons are immersed in a uniform, immobile, background of ions ($\rho_i = Q_e/\int d\mathbf{x}$). Electron mass, temperature and charge are normalized to one. In this paper we consider two classical 3D-3V test cases:

- The 3D-3V non-linear Landau damping [40]: the evolution in time of a perturbation known as the Landau damping [32] is considered. A perturbation in a maxwellian equilibrium state of the distribution is considered:

$$f_e(\mathbf{x},\mathbf{v}) = \frac{1}{2\pi}\prod_{i=1}^{3}\left(1+\alpha_i\cos\left(\frac{\beta_i 2\pi x_i}{L}\right)\right)e^{\frac{-\|\mathbf{v}\|_2^2}{2}}, \tag{26}$$

  where $\|\mathbf{v}\|_2^2 = v_1^2+v_2^2+v_3^2$, α_i is the magnitude and β_i is the period of the perturbation in the i^{th} dimension. Let $\alpha_i = 0.15$, $\beta_i = 3$, $i = 1,2,3$ in equation (26), let $L = 160$, $\Delta t = \frac{1}{20}$. The system is observed at time $T = 6$.

- An extension to 3D-3V of the 2D-2V diocotron instability [40] is considered: a 2d hollow profile is initialized in the electron distribution, confined by a magnetic field $\mathbf{B}$ [38]. Let $L = 22$ and consider the following initial Maxwellian distribution of electrons :

$$f_e(\mathbf{x},\mathbf{v},0) = f_v^0(\mathbf{v})\frac{\gamma}{L0.02(2\pi)^2}e^{-\frac{\left(\sqrt{(x-11)^2+(y-11)^2}-5.5\right)^2}{2(L0.02)^2}}, \quad \gamma\ s.t.\ \iint_{\Omega\times\mathbb{R}^3} f(\mathbf{x},\mathbf{v})d\mathbf{x}d\mathbf{v} = 1 \tag{27}$$

  The external magnetic field is constant and aligned with the z-axis $\mathbf{B}(z) = (0,0,B_z)$ ($B_z = 15$). Its magnitude is large enough so that the electron dynamics is dominated by advection in the self-consistent $\mathbf{E}\times\mathbf{B}$ field. The electrons are immersed in a uniform, immobile, background of ions ($\rho_i = Q_e(\int d\mathbf{x})^{-1}$). Let the time discretization step be $\Delta t = \frac{1}{10}$, the system is observed at time $T = 50$.

To assess the performance of the GPU implementation strategies, we consider two different platforms (see table 6) consisting of a single GPU and an associated host CPU:

- The first hardware is a laptop equipped with a Quadro T2000 with 32 (FP64) CUDA cores, at a base clock frequency of 1.575GHz. The size of the DRAM memory is 4.1 GB. The cache memory is divided into a first level (L1) of 64KB per SM and a second level cache (L2) of 1.024MB. The memory bandwidth between the device and its specific memory is 112.1GB/s. The GPU is associated to an Intel® Core™ i9-10885H CPU (host) with 8 cores @2.40 GHz.

- The second GPU considered belongs to a node of the supercomputer OLYMPE from CALMIP: the Tesla V100 with 2,560 (FP64) CUDA cores, and a base clock frequency of 1.245GHz. The size of the memory is 16GB. The cache memory is divided into a first level (L1) of 128KB per SM and a second level cache (L2) of 6MB. The memory bandwidth between the device and its specific memory is 897GB/s; the L2 cache memory bandwidth is 4.2TB/s and the L1 cache memory bandwidth is 14TB/s. It is associated to an Intel® Skylake CPU with 18 cores @2,3 GHz.

The speedups of the algorithms run on GPGPUs are measured against the most efficient sequential implementation run on one core of the CPU host (Intel® core i9 for the Quadro T2000 and the Intel® Skylake for the Tesla V100, alternatively one core of a bisocket AMD EPYC™ 7713*Milan* may be considered).

The first hardware operated with the compiler Nvidia nvhpc version 22.3 compiler and the flags *-fast -acc -ta=tesla* and CUDA Driver version 11.6. The second hardware operates with the nvhpc version 22.1and the flags *-fast -acc*

Table 6: Performance characteristics of the GPU hardware.

GPU	FP64 cores	Instruction throughput (FP64)	DRAM bandwidth	L2-cache bandwidth	L1-cache bandwidth
Quadro T2000	32	103.7 GFLOP/s	112.1 GB/s	×	×
Tesla V100	2560	7,800 GFLOP/s	897 GB/s	4.2 TB/s	14 TB/s

-Minfo=all -ta=tesla -Mx,231,0x1, the last flag allowing to correct bugs with atomic and reduction operations within the 22.1 version, and CUDA Driver version 11.5. The Poisson equation is solved with the CuSolver [37] library for the first hardware and either with CuSolver or AMGx [36] for the second one. We have noticed a bug in the Fortran random number generator "*random number()*", providing not enough randomness in the particle distribution when used along with the nvhpc compiler. Therefore, the CUDA random number generator CuRand is considered in the following to initialize the particles.

The roofline performance model [19, 45] is a method for determining the maximum performance of an algorithm running on a given hardware. It is used to asses the performance of the different implementations, it is elaborated on metrics provided by the Nvidia Nsight Compute profiler. The theoretical instruction throughput performance is obtained from the number of cores, the clock frequency of the GPU and the number of instructions per cycle (1 operation for multiplication or addition and 2 operations per cycles for Fused Multiply-Add (FMA) instructions):

$$IntructionThroughput[GFlop/s] = nb.\,of\ cores * instruction\ per\ cycle * clock frequency. \tag{28}$$

It provides the upper bound of the computing capacity of the hardware. Nonetheless, in most cases the performance of the hardware is limited by its memory bandwidth. Therefore, the theoretical peak performance of the hardware is defined by:

$$PeakPerformance[GFlop/s] = \min(IntructionThroughput, Bandwidth * ArithmeticIntensity), \tag{29}$$

where the arithmetic intensity (AI), a metric characteristics of the algorithm, is defined by:

$$ArithmeticIntensity = \frac{nb.\,of\ arithmetic\ operations[FLOP]}{DataRead[B] + DataWritten[B]}. \tag{30}$$

The program is said to be compute-bounded when the minimum in the right side of equation (29) is the instruction throughput. In this case, peak performance of the hardware can be achieved. Otherwise the program is memory-bounded, that is the theoretical performance of the hardware is deteriorated by the memory accesses and relies upon the memory bandwidth efficiency of the hardware (see figure 7)[2]. The peak performance metric shall be compared to the effective performance of the algorithm, defined by:

$$EffectivePerformance[GFlop/s] = \frac{nb.\,of\ arithmetic\ operations[FLOP]}{time\ of\ execution[s]}. \tag{31}$$

A new metric is introduced computed as the ratio of the measured FLOP related to the measured Bytes transferred to and from the different memories: DRAM, L2 and L1 caches. Due to the analogy of the Arithmetic Intensity

[2]These metrics are available with the following command line:

$$\begin{aligned} >ncu --metrics\ & dram\_\_bytes.sum, lts\_\_t\_bytes.sum, l1tex\_\_t\_bytes.sum, \\ & sm\_\_sass\_thread\_inst\_executed\_op\_dadd\_pred\_on.sum, \\ & sm\_\_sass\_thread\_inst\_executed\_op\_dfma\_pred\_on.sum, \\ & sm\_\_sass\_thread\_inst\_executed\_op\_dmul\_pred\_on.sum, \end{aligned}$$

where $dram\_\_bytes.sum$ provides the data transfer with the device memory, $lts\_\_t\_bytes.sum$ the data transfer with the L2 cache and $l1tex\_\_t\_bytes.sum$ the data transfer with the L1 cache. The last three metrics provide the total number of double precision floating point operations (addition, multiplication and fused multiply-add operations).

characterizing the algorithm, the new metrics are referred to as DRAM-AI, L2-AI and L1-AI. *E.g.* the DRAM-AI is defined by:

$$DRAM\text{–}AI = \frac{nb.\,of\ arithmetic\ operations[FLOP]}{DataReadFromDRAM[B] + DataWrittenIntoDRAM[B]}. \tag{32}$$

Based on this metric, a theoretical peak performance for each memory can be defined; *e.g.* the DRAM Peak Performance (DRAM-PP) is defined by:

$$DRAMPeakPerformance[GFlop/s] = \min(IntructionThroughput, Bandwidth * DRAM\text{–}AI), \tag{33}$$

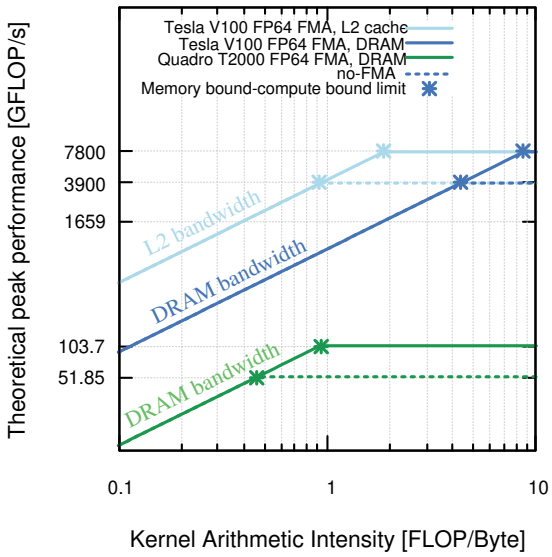

Figure 7: DRAM and L2 cache roof-line performance of the Tesla V100 and Quadro T2000.

4.2. Performance analysis of charge density accumulation algorithms

The charge density accumulation is the most predominant step in sparse-PIC methods with a computational cost counting for roughly 90% of the simulation (for a sequential execution on CPU). Therefore an efficient parallelization of the accumulation is capital to achieve significant performance on accelerators. In section 3.4, two different GPU implementations have been proposed for the charge deposition. These implementations shall now be investigated and compared. Let us consider the sparse-PIC method for the 3D-3V Landau damping in a configuration equivalent to a 128^3 Cartesian grid (*i.e.* $n = 7$) and 500 particles per cell (P_c) with respect to the standard method, amounting to 1.3×10^7 particles (N). As a comparison, the standard method on the Cartesian grid would require 10^9 particles.

4.2.1. Impact of the number of gangs and vector size on the CPU-inherited and GPGPU-specific impelemntations

In this section, the gang, worker and vector length configurations are investigated. The number of workers is usually advised to be set to one [29], as Nvidia compilers do. The vector size shall be a multiple of the number of compute cores within a SM (32 for Nvidia hardware). The computational time of the charge accumulation step on the Quadro T2000 with different configurations of gangs and vector size is provided in figure 8 and the middle panel of figure 10. The same trend is observed on the second platform (Tesla V100). The number of gangs (or thread blocks in CUDA terminology) is a capital tuning to achieve good performance. For the CPU-inherited implementation, the number of gangs is optimal when it is equal to the number of SMs. The performance drastically decreases when a too large number of gangs is used. This is the consequence of the reduction operation performed on the three-dimensional

grid. For the GPGPU-specific implementation introduced in this paper, the performance increases with the number of gangs, facilitating the interleave stream strategy to mask the memory latency. Therefore the number of gangs for the GPGPU specific implementation may be delegated to the compiler. The execution time increases significantly when the vector length is superior to 128 but no significant differences have been observed for vector size equal to 128 and below, so this parameter is delegated to the compiler (usually 128) in the following of the paper.

Implementation	Gangs	Vector length	Time [s]	Time per particle [s]
CPU-inherited	16	128	6.22	4.6×10^{-7}
*	32	128	7.19	⋮
*	64	128	8.60	
*	128	128	11.22	8.3×10^{-7}
GPGPU-specific	16	128	1.49	1.1×10^{-7}
*	128	128	0.92	⋮
*	1024	128	0.88	
*	65,535	128	0.87	6.4×10^{-8}
*	65,535	256	1.0	⋮
*	65,535	1024	1.42	
*	65,535	64	0.85	

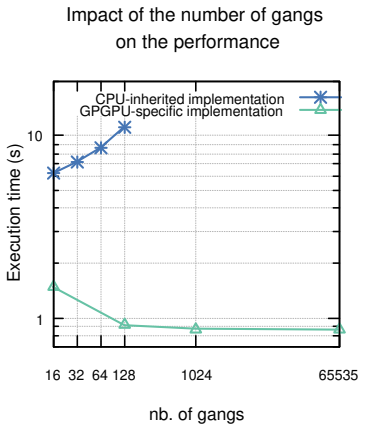

Figure 8: Impact of the number of gangs (thread blocks) and vector size (thread) on the performance of the charge accumulation algorithm for the Quadro T2000.

4.2.2. Memory management and kernel analysis

Let us now investigate specifically the kernels of the CPU-inherited and the GPGPU-specific implementation. The optimal configuration of thread blocks (gangs) and threads within a thread block (vector size) for each method is chosen according to the following policy: the number of gangs is set to the number of SM for the CPU-inherited implementation; while this choice is delegated the compiler for the GPGPU implementation. The number of threads (vector size) is set to 128 for both methods. In the CPU-inherited implementation, one kernel (corresponding to the accumulation onto one component grid) is considered, *e.g.* the kernel associated to the first component grid (of level $\mathbf{l} = (1, 1, 7)$). The same conclusions can be drawn for all kernels by extension. The kernel is compared to the kernel of the GPGPU specific algorithm, implementing the accumulation of the particle properties onto all the component grids.

First, the arithmetical intensity (AI), the peak and effective performances of the kernels involved in both the CPU-inherited and the GPGPU-specific implementations are investigated on table 9. The AI is determined with the total number of FLOP relative to the total amount of data to be read and written (see equation 32). The peak performance is determined by the Roofline model from the AI and the DRAM bandwidth of the device. It is compared to the effective performance measured thanks to the profiler. Although the kernel AI of the two implementations are equivalent, we observe a significant discrepancy in the effective performance, the effective performance of the GPGPU implementation being roughly twice its theoretical peak performance. This surprising result is the consequence of the cache memory reuses occurring in the GPGPU implementation.

Table 9: Charge accumulation algorithm performance on Tesla V100 with 128^3 grids and $P_c = 500$. Kernel CPU-inheirted corresponds to the kernel of the first component grid ($\Omega_{h_{(1,1,7)}}$) for the CPU-inherited algorithm. Kernel GPGPU-specific corresponds to the kernel of all the component grids for the GPGPU-specific algorithm.

Kernel	AI	Peak performance	Effective performance (%)
CPU-inherited	0.12	107.6 GFLOP/s	58.6 GFLOP/s (54%)
GPGPU-specific	0.14	125.5 GFLOP/s	213 GFLOP/s (170%)

Let us now consider a more thorough analysis of the kernels to better understand the memory managements of

the methods. In the following we consider the metrics based on the different memories and introduced previously as DRAM-AI, L2-AI and L1-AI. These metrics are computed as the ratio of the total number of FLOP related to the effective amount of data handled by a specific memory at the execution (DRAM data, L2-cache data on table 11). All the characteristics and performance of the charge accumulation algorithms are provided on the table 11. The relative amount of L1-cache and L2-cache hits are represented on the left panel of figure 10.

First, it is manifest that the GPGPU-specific kernel has a better L2-cache efficiency (more than 99% of L2-cache hit) than the CPU-inherited kernel (roughly 30% of L2-cache hit). We observe a non efficient use of the cache memory for the CPU-inherited method since more than one half of the L1 cache requests has to be fetched from the DRAM memory (and less than a third has to be fetched in the L2-cache). The amount of data transferred between the device and the DRAM memory is dramatically reduced for the GPGPU implementation, emphasizing the success of the strategy targeted by these implementation. However, the number of L1-cache hits remains significantly low for the GPGPU-specific implementation, this being the result of the irregular and non coalesced memory accesses genuine to the method.

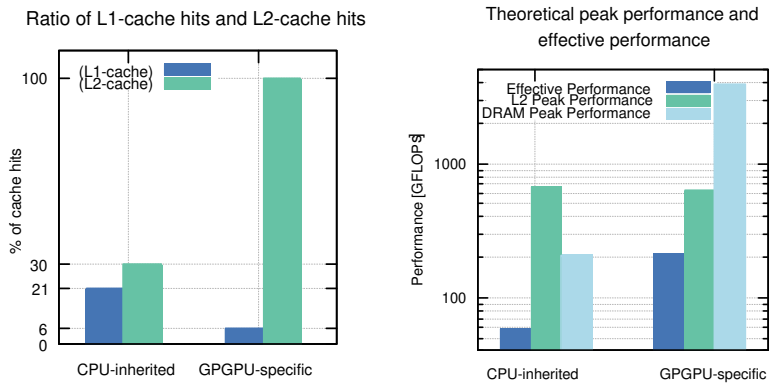

Figure 10: Relative amount of L1-cache and L2-cache hits (left panel) and theoretical/effective performance (right panel) for kernels running on the Tesla V100. Kernel CPU-inherited corresponds to the kernel of the component grid $\Omega_{h_{(1,1,7)}}$ for the CPU-inherited algorithm. Kernel GPGPU-specific corresponds to the kernel of all the component grids for the GPGPU-specific algorithm.

The theoretical peak performance specific to the DRAM (DRAM-PP) can be determined from the DRAM-AI of the kernel and compared to the effective performance. A high DRAM-PP compared to the theoretical Peak Performance means that a lot of data is loaded from the cache memory. For the Tesla V100, the kernel of the CPU-inherited algorithm is memory-bounded according to this metric, with a DRAM peak performance of roughly 206 GFLOP/s (see table 11) and effective performance of 58.6 GFLOP/s. The kernel of the GPGPU algorithm is compute-bounded according to this metric, with a DRAM peak performance of 3,900 GLOP/s (since no FMA operation is performed) and an effective performance of 213 GFLOP/s. The DRAM-PP is roughly 30 times higher than the theoretical Peak Performance, emphasizing on the good cache memory management of the method. The kernel is actually far from being compute-bounded and most of the memory transfers take place in the L2-cache and not the DRAM. Indeed, most of the data necessary in the kernel is fetched from the caches (L1 and mostly L2 layers) and a significantly small proportion of data is transferred from the main memory (The number of DRAM memory accesses represent 0.1% of those of L2-cache memory). For this reason, the DRAM-AI and DRAM-PP metrics are not relevant for the GPGPU-specific implementation. For the Quadro T2000, the same conclusions can be drawn. Nonetheless, the effective performance of the GPGPU implementation is closer to the theoretical one (72%).

Let us now investigate the roofline model based on the L2-cache data transfers. The L2-AI of the two implementations are equivalent (of roughly 0.15) and so does the L2-Peak Performance. The effective performance is about 33% of the L2-PP for the GPGPU implementation and 8% for the CPU-inherited implementation. Because the peak performance is not reached in both cases, the implementations are rather latency-bound and the performance is limited by the irregular and non coalesced memory accesses.

Despite the randomness of the memory accesses increasing the latency between the device and its memory, a

significant speed-up is achieved both on the Quadro T2000 and the Tesla V100. Although the computing capacity of the Quadro T2000 is poor, a speed-up close to 12 is reached in double precision, meaning that a GPGPU execution is more efficient than parallelization with the 8 cores Intel® Core™ i9-10885H of the laptop host. The Tesla V100 achieves a speed-up of more than 100 on the charge deposition in double precision, which is significant for such an inexpensive hardware. As a comparison, in [17] a speed-up of 126 is achieved for the charge accumulation on a hardware consisted of 2 sockets AMD EPYC™ 7713 *Milan* with 128 cores.

In conclusion, the detrimental effects of the irregular non coalesced memory accesses are mitigated by a good locality of the data, resulting in an efficient L2-cache memory management, as well as an efficient latency masking strategy based on a large number of gangs associated to clusters of particles.

Table 11: Charge accumulation algorithm characteristics and performance (DRAM and L2 cache Roofline model) on the Tesla V100 (a) and the Quadro T2000 (b), with a 128^3 grid and P_c = 500. Kernel CPU corresponds to the kernel of the component grid $\Omega_{h_{(1,1,7)}}$ for the CPU-inherited algorithm. Kernel GPU corresponds to the kernel of all the component grids for the GPGPU-specific algorithm. DRAM, L2 cache and L1 cache data correspond to the amount of data transferred with respectively the main memory (DRAM), the L2 cache or the L1 cache during the execution; *e.g.*, the amount of L2 cache hit data is $L2\ cache\ data - DRAM\ data$. DRAM-AI and L2-AI correspond to metrics of the kernel based on the measured Bytes transferred to and from the different memories (DRAM, L2). The peak performance is defined with the DRAM-AI and L2-AI metrics and the effective performance is defined by equation (31).

(a)

Kernel	Gangs (blockidx%x)	Vector length (threadidx%x)	DRAM data	L2-cache data	L1-cache data
CPU	80	128	4.57GB	6.59GB	8.43GB
GPU	65,535	128	326MB	215GB	229GB

Kernel	L2-cache hit (%)	L1-cache hit (%)
CPU	2.02GB (30%)	1.84GB (21%)
GPU	214.6GB(99.8%)	14GB (6%)

Kernel	FLOP	DRAM-AI	DRAM-PP	Effective performance	Execution time (all grids)	Speed-up
CPU	1.05×10^9	0.23	206 GFLOP/s	58.6 GFLOP/s (28%)	0.0179 s (1.4 s)	11.7
GPU	3.28×10^{10}	101	3,900 GFLOP/s	213 GFLOP/s (5%)	0.1540 s	106

Kernel	L2-AI	L2-PP	Effective performance
CPU	0.16	672 GFLOP/s	58.6 GFLOP/s (8%)
GPU	0.15	630 GFLOP/s	213 GFLOP/s (33%)

(b)

Kernel	Gangs (blockidx%x)	Vector length (threadidx%x)	DRAM data	L2-cache data	L1-cache data
CPU	16	128	4.45GB	5.92GB	7.5GB
GPU	65,535	128	328MB	215GB	229GB

Kernel	FLOP	DRAM-AI	DRAM-PP	Effective performance	Execution time (all grids)	Speed-up
CPU	1.0×10^9	0.226	25.4 GFLOP/s	12.3 GFLOP/s (48%)	0.082 s (7.04 s)	1.47
GPU	3.28×10^{10}	100	51.85 GFLOP/s	37.4 GFLOP/s (72%)	0.874 s	11.8

4.3. Investigation of the whole scheme

Let us consider the parallelization of all the steps of the sparse-PIC scheme. The 3D-3V Landau damping is run on two platforms, both with a 128^3 grid: on the one hand the Intel® Core™ i9-10885H with 1 core is compared to the Quadro T2000 run for 100 particles per cell; on the other hand the Intel® Skylake with 1 core is compared to the Tesla V100 run for 500 particles per cell. In the following, the GPGPU implementation is considered for runs on GPUs.

First, the different strategies for the resolution of the Poisson equation are investigated, namely: either sovling the linear systems issued from the discretization of the problem on the component grids one after the other or gathering all the problems into a single (by block) linear system. The first strategy is considered with the CuSolver dense library and the second one with the AMGX library. The computational time of both strategies is represented on figure 13 for different grid discretization (ranging from a 32^3 grid to a 512^3 grid). For coarse discretizations, the first option is the most efficient despise the dense format of the solver. This is caused by the very small size of the linear system, which are roughly of one-dimension complexity. When the grid is refined, the second option along with the AMGX solver becomes less time consuming as a result of the increasing work load favorable to the GPU.

The mean computational time out of 5 iterations is represented on figure 12 for the two platforms, namely the Quadro T2000 associated to the Intel® Core™ i9-10885H CPU and the Tesla V100 associated to the Intel® Skylake CPU. The speed-ups obtained on the GPU devices with respect to the host runs for each step, as well as the total

iteration, are provided in table 14. The GPU execution time is compared to a sequential CPU execution with the MUMPS library together with a GNU compiler (see [17] for more details).

The field interpolation and the particle pusher have the most significant speed-ups (of roughly 130 on the Tesla V100). Indeed, these steps, consisting in basic operations working on a large amount of data are well suited for SIMT processing of GPUs. The speed-ups of the combination, differentiation and Poisson solver are less substantial, mostly because of the insufficient kernel arithmetic intensity. The operations within these steps are made on a very small amount of data (component grid nodes) and the sequential computational time is insignificant, even on 1 core (the three steps represent less than 0.3% on the Intel® Skylake run with 1 core). The charge density accumulation, which is the most time consuming operations of the scheme have a significant speed-up, of roughly 100 on the Tesla V100. It results in a total speed-up for the whole iteration of about 12 on the Quadro T2000 and 95 on the Tesla V100.

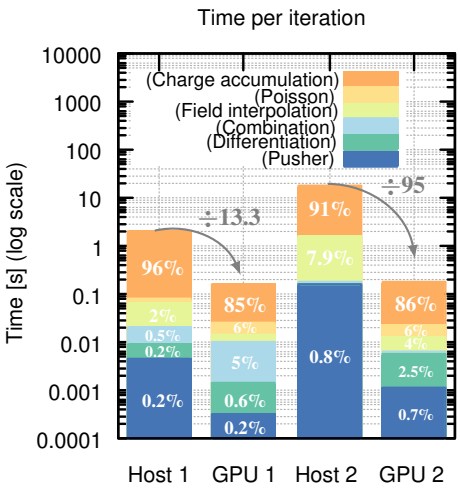

Figure 12: Time per iteration of the CPU and GPU sparse-PIC implementation. Host 1 and GPU 1 run with 100 particles per cells while host and GPU 2 run with 500 particles per cell. Host 1 is the Intel® Core™ i9-10885H with 1 core; GPU 1 is the Quadro T2000; Host 2 is the Intel® Skylake with 1 core; GPU 2 is the Tesla V100.

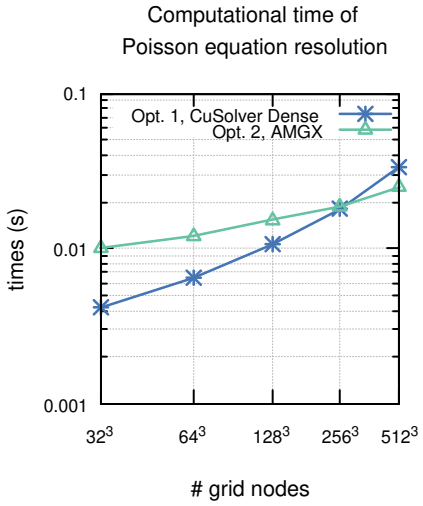

Figure 13: Computational time of the Poisson equation resolution as a function of the grid discretization. In option 1, all the systems issued from the discretization of the Poisson equation are solved one after the other. In option 2, all the problems are gathered into a single (by blocks) linear system.

Table 14: Speed-ups (and relative time) of the steps for the Quadro T2000 (GPU 1) and Tesla V100 (GPU 2). GPU 1 runs with 100 particles per cells while GPU 2 runs with 500 particles per cell. Poisson speed-up is provided with respect to a sequential CPU execution with MUMPS library.

	Hardware	Projection	F. Interpolation	Pusher	Combination	Diff.	Poisson	Total
Host	Intel® Core™ i9	1 (96%)	1 (2%)	1 (0.2%)	1 (0.5%)	1 (0.2%)	1 (1.1%)	1 (100%)
Device	Quadro T2000	14.1 (85%)	12.33 (3.2%)	13.7 (0.2%)	1.2 (5%)	4.44 (0.6%)	1.4 (6%)	12.5 (100%)
Host	Intel® Skylake	1 (91%)	1 (7.9%)	1 (0.8%)	1 (0.2%)	1 (0.1%)	1 (0.1%)	1 (100%)
Device	Tesla V100	107 (86%)	139 (4%)	133 (0.7%)	21.8 (0.8%)	29.8 (2.5%)	1 (6%)	95 (100%)

4.4. Qualitative results of the 3D-3V test cases

The sparse-PIC GPGPU-specific implementation are compared to a sequential CPU implementation of the standard scheme. The architecture considered for the standard PIC simulations consists of a single computational server equipped with two AMD EPYC™ 7713 *Milan* CPUs and RAM memory of 512GB. The standard scheme runs with a particle population sorted beforehand, i.e. the particle sorting time is not taken into account for the total execution time, which provides an upper bound of the method efficiency. As a first investigation, let us consider first the 3D-3V Landau damping on a standard PIC method configuration of a 128^3 grid with 128 particles per cell, which is considered as a reference. Sparse-PIC discretizations carried out on a 128^3 grid (resp. a 512^3 grid) with 128 (resp. 500) particles

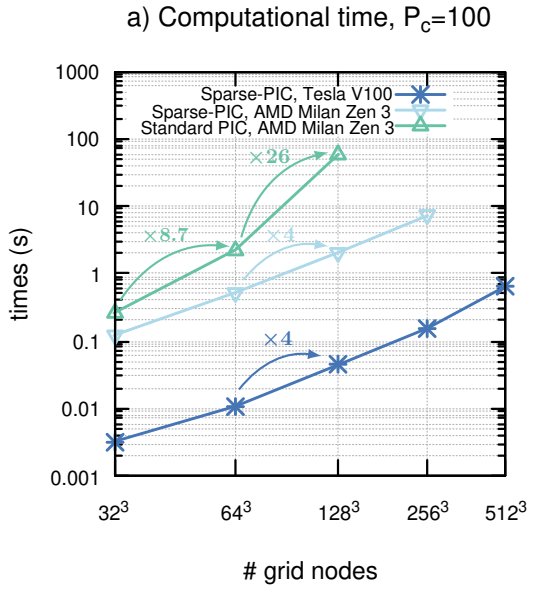

Figure 15: Computational time of the standard and sparse-PIC schemes for grids ranging from 32^3 to 512^3. Standard simulation is performed on the AMD Zen 3 core @2 GHz. Sparse-PIC simulations are performed on the AMD Zen 3 core and on the Tesla V100.

Table 16: Configurations and results of the 3D-3V Landau damping test case. AMD EPYC™ 7713 *Milan* with AMD ZEN 3 core @2.0 GHz and Intel® *Skylake* core @2,3 GHz are considered.

Method	Figure	Grid/particle per cell (P_c)	Particles (N)	Memory footprint	Time (1 iter.)
Standard (CPU AMD *Milan*)	17 (left)	$128^3/128$	2.6×10^8	19GB	139.6 s
Sparse-PIC (CPU Intel® *Sylake*)		$128^3/128$	3.45×10^6	248MB	5.4 s (÷25.8)
Sparse-PIC (CPU AMD *Milan*)		$128^3/128$	3.45×10^6	248MB	2.6 s (÷53.7)
Sparse-PIC (GPU Tesla V100)	17 (right)	$128^3/128$	3.45×10^6	248MB	0.05 s (÷2792)
Sparse-PIC (GPU Tesla V100)		$512^3/500$	9.01×10^7	6.48GB	1.739 s

per cell are considered as a comparison. It results in a substantial reduction of the number of particles (two orders of magnitude) and thus of the memory footprint (from GB to MB). Besides the significant gain on the memory footprint offered by the sparse-PIC method, the computational time is also reduced (by one order of magnitude in that configuration, see table 16). Let $\Delta t = \frac{1}{20}$ and $T = 6$. The three dimensional representation of the electron density is provided for the two methods on figure 17.

The computational time of the standard method on a single CPU core, the sparse-PIC method on a single CPU core and the GPGPU implementation are represented as a function of the grid discretization on figure 15. In the following, the sequential executions of the sparse-PIC method are provided for the most efficient algorithm on CPU, which is the algorithm 2. When the mesh is refined in the standard method, the number of particles and the number of Cartesian grid nodes are multiplied by 8 and therefore so does the computational time. Nonetheless, a significantly larger increase is observed when the mesh is refined from a 64^3 grid to a 128^3 grid. Actually, it is explained by the non-locality of the data: the simulation data for the 128^3 simulation no longer fit on a single NUMA node of the AMD EPYC computational server. The computational time of both the CPU and GPU sparse-PIC simulations is multiplied by 4 when the mesh is refined, which is twice as much as the standard method.

In [16, 17, 35], new sparse grid reconstructions have been introduced to improve the approximations of solutions with localized support and fine structures. As an illustration, we investigate the three-dimensional diocotron test in which instabilities caused by a magnetic field lead to the formation of a discrete number of vortices exhibiting the weaknesses of the method. This test case is particularly unfavorable to the sparse grid methods implementing the hierarchization (see [17]), because the number of particles per cells of the Cartesian grid is very low (*e.g.* five particles per cells here). Indeed, in the present paper, the electric field is reconstructed onto the Cartesian grid before being interpolated onto the particles. This choice is arbitrary, the projection of the electric field onto a Cartesian grid is not

mandatory. It may be recombined from the component grids onto the particle positions, similarly to density projection. This choice is made here, to investigate the performance of the hierarchization on GPGPU architectures. The cost and the scalability of the electric field interpolation from the component grids to the particle positions would be similar to the charge accumulation step. Let $\Delta t = \frac{1}{10}$ and $T = 50$. The three dimensional representation of the electron density is represented on figure 18 at $T = 100$, with a 128^3 grid and $P_c = 5$ for the standard scheme (top left panel), with a 256^3 grid and $P_c = 5$ for the sparse grid scheme without (top right panel) and with the offset combination technique (for the parameters $(\tau_0, \tau_1) = (3, 6)$) (bottom panel). The offset combination technique [16] and truncated combination technique [35] are derivations of the classical combination technique, consisting in the elimination of the most anisotropic grids from the combination. With the offset combination technique, there are less, but more refined, component grids in the combination. As a result, more particles are required to obtain a good statistical resolution, therefore the grid operations such as the resolution of Poisson equation are more costly. Though the grid is more refined ($h_n = 2^{-8}$), the sparse grid scheme with the classical combination technique fails to reproduce the fine structures of the density. One can see that the sparse grid reconstruction has a tendency to flatten the steep gradients of the solution. The offset combination technique provides a significant improvement of the sparse grid reconstructions and a mitigation of the statistical noise in comparison to the standard approach despite a significantly reduced number of particles. Even for such an unfavorable test case, sparse-PIC methods embedding offset combination technique provides a gain over the standard method on memory footprint, as well as computational time.

Conclusions

In this paper, we proposed an efficient GPGPU implementation of the sparse-PIC methods based on parallelization strategies specific to sparse grid reconstructions and tailored to GPU architectures.

In [17], a first parallelization dedicated to shared memory CPU architecture for sparse-PIC methods has been introduced. The present paper proposed the extension of this implementation to GPGPU, designated by the CPU-inherited implementation, which has proven to not fully exploit the potential of GPUs. Indeed, the CPU implementation is mainly based on L1 cache memory reuse which cannot operate on GPUs because of their memory architecture differences with respect to CPUs. In addition, we have encountered difficulties with the OpenACC standard such that a coarse/fine grain parallelism cannot be applied.

Therefore we conceived a GPGPU implementation, based on GPU memory architectures, in order to take advantage of the potential of GPUs. Coarse-grain and SIMT parallelisms are entailed with the GPGPU implementation thanks to particle sampling and component grid work sharing strategies. The algorithm is also designed to benefit from the large L2 cache memory of the GPU, mitigating the negative impact of the non-coalesced memory accesses.

Sparse grid reconstructions within PIC methods offer a significant better control of the statistical noise in simulations and allow to dramatically reduce the number of particle mandatory to reach an appropriate amount of noise. Therefore sparse-PIC methods benefit from a substantial reduction of the memory footprint with respect to the standard PIC methods. Our implementation takes advantage of this feature as all the data lie on the device (a single GPU) throughout the simulation and the only data transfers between the host and the device occur at the initialization.

The sparse-PIC GPU implementation achieves speed-ups close to 100 on a Tesla V100 for 3D-3V PIC simulations, resulting in a computational time diminution of four orders of magnitude with respect to a single core CPU standard PIC execution. As a result, 3D-3V PIC Simulations with a 512^3 grid and 500 particles per cell have been performed on a single Tesal V100 GPU device.

Acknowledgements

Clément Guillet benefits from a Université de Toulouse/Région Occitanie PhD grant.
This work has been carried out within the framework of the EUROfusion Consortium, funded by the European Union via the Euratom Research and Training Programme (Grant Agreement No 101052200 — EUROfusion). Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Commission. Neither the European Union nor the European Commission can be held responsible for them.

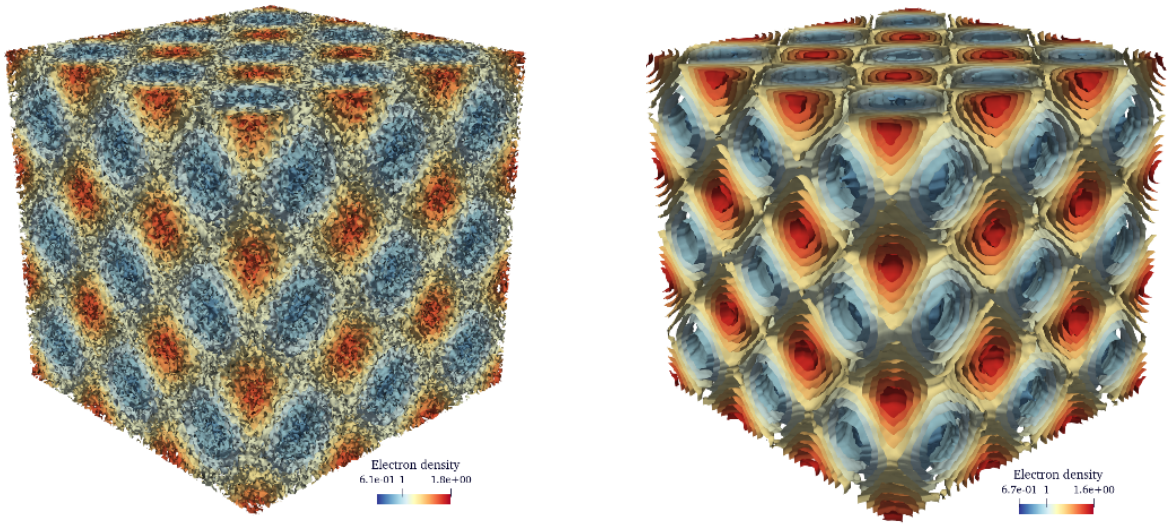

Figure 17: Representation of the electron density for the 3D-3V Landau damping simulation with $h_n = 2^{-7}$, $P_c = 128$. Three dimensional representation of the standard method (left) and the sparse-PIC method (right) after two oscillations, 120 time steps, $t = 6$. Figure a) requires 585 s per iteration on a single core Intel® Skylake. Figure b) requires 0.05 s on a Tesla V100. See table 16 for configurations.

This work has been supported by a public grant from the "Laboratoire d'Excellence Centre International de Mathématiques et d'Informatique" (LabEx CIMI) overseen by the French National Research Agency (ANR) as part of the "Investissements d'Avenir" program (reference ANR-11-LABX-0040) in the frame of the PROMETEUS project (PRospect of nOvel nuMerical modEls for elecTric propulsion and low tEmperatUre plaSmas).
Support from the FrFCM (Fédération de recherche pour la Fusion par Confinement Magnétique) in the frame of the SPARCLE project (SParse grid Acceleration for the paRticle-in-CelL mEthod) is also acknowledged.
Jacek Narski was supported by the ANR project MUFFIN (ANR-19-CE46-0004).
This work was granted access to the HPC resources of CALMIP supercomputing center under the allocation 2022-2022-1125.
We thank Atos and its Application Experts for their knowledgeable support.

References

[1] Charles K Birdsall and Dieter Fuss. Clouds-in-clouds, clouds-in-cells physics for many-body plasma simulation. *Journal of Computational Physics*, 3(4):494–511, April 1969.
[2] C.K. Birdsall and A.B Langdon. *Plasma Physics via Computer Simulation*. CRC Press, 0 edition, October 1985.
[3] Hans-Joachim Bungartz. *Finite elements of higher order on sparse grids*. Berichte aus der Informatik. Shaker, Aachen, als ms. gedr edition, 1998.
[4] Hans-Joachim Bungartz and Stefan Dirnstorfer. Higher Order Quadrature on Sparse Grids. In Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Dough Tygar, Moshe Y. Vardi, Gerhard Weikum, Marian Bubak, Geert Dick van Albada, Peter M. A. Sloot, and Jack Dongarra, editors, *Computational Science - ICCS 2004*, volume 3039, pages 394–401. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004. Series Title: Lecture Notes in Computer Science.
[5] Hans-Joachim Bungartz and Michael Griebel. Sparse grids. *Acta Numerica*, 13:147–269, May 2004.
[6] G. Chen, L. Chacón, and D.C. Barnes. An efficient mixed-precision, hybrid CPU-GPU implementation of a nonlinearly implicit one-dimensional particle-in-cell algorithm. *Journal of Computational Physics*, 231(16):5374–5388, June 2012.
[7] John Cheng. *Professional Cuda C programming*. Wrox programmer to programmer. John Wiley and Sons, Inc, Indianapolis, IN, 2014.
[8] J. Claustre, B. Chaudhury, G. Fubiani, M. Paulin, and J. P. Boeuf. Particle-In-Cell Monte Carlo Collision Model on GPU – Application to a Low-Temperature Magnetized Plasma. *IEEE Trans. Plasma Sci.*, 41(2):391–399, February 2013.
[9] Shane Cook. *CUDA Programming: a Developer's Guide to Parallel Computing with GPUs*. Elsevier Science, Saint Louis, 2014. OCLC: 1045075687.

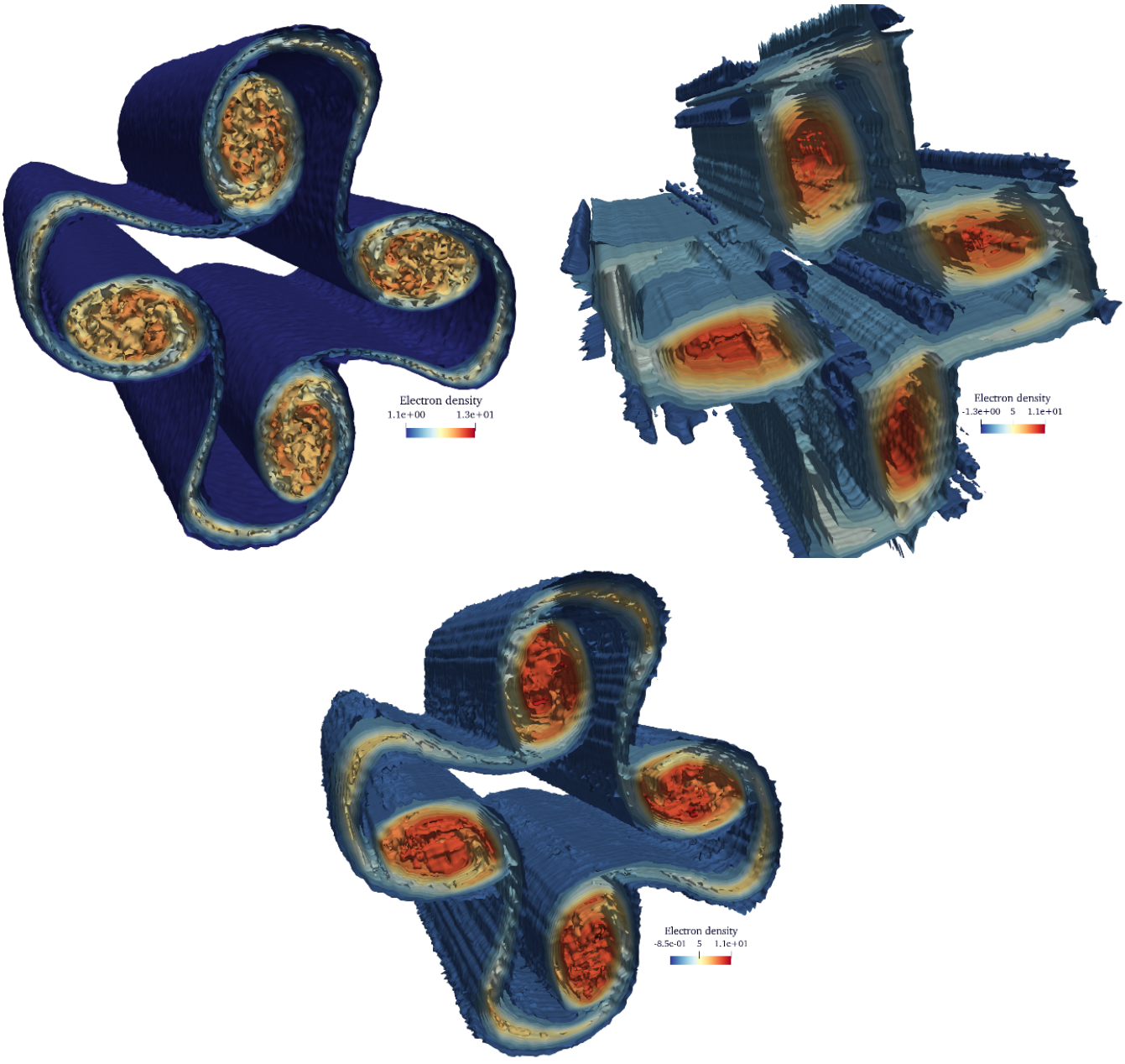

Figure 18: Representation of the electron density for the 3D-3V diocotron simulation. Three dimensional representation after 500 time steps, $t = 100$. Panel top left (standard PIC) requires 7.02 s per iteration on a single core Intel® Skylake ($N = 1.05 \times 10^7$). Panel top right (sparse-PIC) requires 0.046 s per iteration on a Tesla V100 (3.54×10^5). Panel bottom (sparse-PIC offset) requires 0.76 s per iteration on a Tesla V100 ($N = 8.68 \times 10^6$).

[10] G.-H. Cottet and P.-A. Raviart. Particle Methods for the One-Dimensional Vlasov-Poisson Equations. *SIAM J. Numer. Anal.*, 21(1):52–76, February 1984.
[11] G. H. Cottet and P. A. Raviart. On particle-in-cell methods for the Vlasov-Poisson equations. *Transport Theory and Statistical Physics*, 15(1-2):1–31, February 1986.
[12] Viktor K. Decyk and Charles D. Norton. UCLA Parallel PIC Framework. *Computer Physics Communications*, 164(1-3):80–85, December 2004.
[13] Viktor K. Decyk and Tajendra V. Singh. Adaptable Particle-in-Cell algorithms for graphical processing units. *Computer Physics Communications*, 182(3):641–648, March 2011.
[14] Viktor K. Decyk and Tajendra V. Singh. Particle-in-Cell algorithms for emerging computer architectures. *Computer Physics Communications*, 185(3):708–719, March 2014.
[15] Pierre Degond, Fabrice Deluzet, and David Doyen. Asymptotic-preserving Particle-In-Cell methods for the Vlasov-Maxwell system near quasi-neutrality. *arXiv:1509.04235 [physics]*, September 2015. arXiv: 1509.04235.
[16] F. Deluzet, G. Fubiani, L. Garrigues, C. Guillet, and J. Narski. Sparse grid reconstructions for Particle-In-Cell methods. *ESAIM: M2AN*, 56(5):1809–1841, September 2022.
[17] Fabrice Deluzet, Gwenael Fubiani, Laurent Garrigues, Clément Guillet, and Jacek Narski. Efficient parallelization for 3d-3v sparse grid Particle-In-Cell: Shared memory architectures. *Journal of Computational Physics*, 480:112022, May 2023.
[18] R.E. Denton and M. Kotschenreuther. {delta}f Algorithm. Technical Report DOE/ET/53088–629, IFSR–629, 10105190, nov 1993.
[19] Nan Ding and Samuel Williams. An Instruction Roofline Model for GPUs. In *2019 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, pages 7–18, Denver, CO, USA, November 2019. IEEE.
[20] Shahab Fatemi, Andrew R. Poppe, Gregory T. Delory, and William M. Farrell. AMITIS: A 3D GPU-Based Hybrid-PIC Model for Space and Plasma Physics. *J. Phys.: Conf. Ser.*, 837:012017, May 2017.
[21] L. Garrigues, G. Fubiani, and J.P. Boeuf. Negative ion extraction via particle simulation for fusion: critical assessment of recent contributions. *Nucl. Fusion*, 57(1):014003, January 2017.
[22] L. Garrigues, B. Tezenas du Montcel, G. Fubiani, F. Bertomeu, F. Deluzet, and J. Narski. Application of sparse grid combination techniques to low temperature plasmas particle-in-cell simulations. I. Capacitively coupled radio frequency discharges. *Journal of Applied Physics*, 129(15):153303, April 2021.
[23] L. Garrigues, B. Tezenas du Montcel, G. Fubiani, and B. C. G. Reman. Application of sparse grid combination techniques to low temperature plasmas Particle-In-Cell simulations. II. Electron drift instability in a Hall thruster. *Journal of Applied Physics*, 129(15):153304, April 2021.
[24] Salimou Gassama, Eric Sonnendrücker, Kai Schneider, Marie Farge, and Margarete O. Domingues. Wavelet denoising for postprocessing of a 2D Particle - In - Cell code. *ESAIM: Proc.*, 16:195–210, 2007.
[25] Michael Griebel. Adaptive sparse grid multilevel methods for elliptic PDEs based on finite differences. *Computing*, 61(2):151–179, June 1998.
[26] Michael Griebel and Jan Hamaekers. Sparse grids for the Schrödinger equation. *ESAIM: M2AN*, 41(2):215–247, March 2007.
[27] M. Hegland. Adaptive sparse grids. *ANZIAMJ*, 44:335, April 2003.
[28] R Hockney and J Eastwood. *Computer Simulation Using Particles*. Taylor & Francis, January 1988.
[29] Guido Juckeland and Sunita Chandrasekaran. *OpenACC for programmers: concepts and strategies*. 2018. OCLC: 1047846209.
[30] Zoltan Juhasz, Ján Ďurian, Aranka Derzsi, Štefan Matejčík, Zoltán Donkó, and Peter Hartmann. Efficient GPU implementation of the Particle-in-Cell/Monte-Carlo collisions method for 1D simulation of low-pressure capacitively coupled plasmas. *Computer Physics Communications*, 263:107913, June 2021.
[31] Xianglong Kong, Michael C. Huang, Chuang Ren, and Viktor K. Decyk. Particle-in-cell simulations with charge-conserving current deposition on graphic processing units. *Journal of Computational Physics*, 230(4):1676–1685, February 2011.
[32] Nicholas A. Krall, Alvin W. Trivelpiece, and Robert A. Gross. Principles of Plasma Physics. *American Journal of Physics*, 41(12):1380–1381, December 1973.
[33] P. C. Liewer, V. K. Decyk, J. M. Dawson, and G. C. Fox. A universal concurrent algorithm for plasma particle-in-cell simulation codes. In *Proceedings of the third conference on Hypercube concurrent computers and applications* -, volume 2, pages 1101–1107, Pasadena, California, United States, 1988. ACM Press.
[34] A. Munshi, B. Gaster, T.G. Mattson, and D. Ginsburg. *OpenCL Programming Guide*. OpenGL. Pearson Education, 2011.
[35] Sriramkrishnan Muralikrishnan, Antoine J. Cerfon, Matthias Frey, Lee F. Ricketson, and Andreas Adelmann. Sparse grid-based adaptive noise reduction strategy for particle-in-cell schemes. *Journal of Computational Physics: X*, 11:100094, June 2021.
[36] Maxim Naumov, M. Arsaev, Patrice Castonguay, Jonathan M. Cohen, Julien Demouth, Joe Eaton, Simon K. Layton, N. Markovskiy, István Z. Reguly, Nikolai Sakharnykh, V. Sellappan, and Robert Strzodka. Amgx: A library for gpu accelerated algebraic multigrid and preconditioned iterative methods. *SIAM J. Sci. Comput.*, 37, 2015.
[37] NVIDIA, Péter Vingelmann, and Frank H.P. Fitzek. Cuda, release: 10.2.89, 2020.
[38] J. Petri. Non-linear evolution of the diocotron instability in a pulsar electrosphere: 2D PIC simulations. *A&A*, 503(1):1–12, August 2009. arXiv: 0905.1076.
[39] Alexander A. Philippov and Anatoly Spitkovsky. AB INITIO PULSAR MAGNETOSPHERE: THREE-DIMENSIONAL PARTICLE-IN-CELL SIMULATIONS OF AXISYMMETRIC PULSARS. *ApJ*, 785(2):L33, April 2014.
[40] L F Ricketson and A J Cerfon. Sparse grid techniques for particle-in-cell schemes. *Plasma Phys. Control. Fusion*, 59(2):024002, February 2017.
[41] Stephen Russell and Niall Madden. An Introduction to the Analysis and Implementation of Sparse Grid Finite Element Methods. *Computational Methods in Applied Mathematics*, 17(2):299–322, April 2017.
[42] Jie Shen and Haijun Yu. Efficient Spectral Sparse Grid Methods and Applications to High-Dimensional Elliptic Problems. *SIAM J. Sci. Comput.*, 32(6):3228–3250, January 2010.
[43] S. Soller. *GPGPU Origins and GPU Hardware Architecture*. Hochschule der Medien, 2011.
[44] Junya Suzuki, Hironori Shimazu, Keiichiro Fukazawa, and Mitsue Den. Acceleration of PIC Simulation with GPU. *Plasma and Fusion*

Research, 6:2401075–2401075, 2011.

[45] Charlene Yang, Thorsten Kurth, and Samuel Williams. Hierarchical Roofline analysis for GPUs: Accelerating performance optimization for the NERSC-Perlmutter system. *Concurrency Computat Pract Exper*, 32(20), October 2020.