# Vlasov on GPU (VOG Project)

Luca Marradi, Bedros Afeyan, Michel Mehrenberger, Nicolas Crouseilles, Christophe Steiner, Eric Sonnendrücker

▶ **To cite this version:**

Luca Marradi, Bedros Afeyan, Michel Mehrenberger, Nicolas Crouseilles, Christophe Steiner, et al.. Vlasov on GPU (VOG Project). ESAIM: Proceedings, 2013, 43, p. 37-58. 10.1051/proc/201343003 . hal-00908498

**HAL Id: hal-00908498**

**https://hal.science/hal-00908498v1**

Submitted on 5 Apr 2022

**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.

Distributed under a Creative Commons Attribution 4.0 International License

**ESAIM: PROCEEDINGS**, **December 2013, Vol. 43, p. 37-58**
S. Descombes, B. Dussoubs, S. Faure, L. Gouarin, V. Louvet, M. Massot, V. Miele, Editors

# VLASOV ON GPU (VOG PROJECT) [*, **, ***]

M. Mehrenberger[1], C. Steiner[2], L. Marradi[3], N. Crouseilles[4], E. Sonnendrücker[5] et B. Afeyan[6]

**Résumé.** Ce travail concerne la simulation numérique du modèle de Vlasov-Poisson à l'aide de méthodes semi-Lagrangiennes, sur des architectures GPU. Pour cela, quelques modifications de la méthode traditionnelle ont dû être effectuées. Tout d'abord, une reformulation des méthodes semi-Lagrangiennes est proposée, qui permet de la réécrire sous la forme d'un produit d'une matrice circulante avec le vecteur des inconnues. Ce calcul peut être fait efficacement grâce aux routines de FFT. Actuellement, le GPU n'est plus limité à la simple précision. Néanmoins, la simple précision reste intéressante pour des raisons de performance et de mémoire disponible. Afin de contourner le problème de la simple précision, une méthode de type $\delta f$ est alors utilisée. Ainsi, un code Vlasov-Poisson GPU permet de simuler et de décrire avec un haut degré de précision (grâce à l'utilisation de reconstructions d'ordre élevé et d'un grand nombre de points de l'espace des phases) des cas tests académiques mais aussi des phénomènes physiques pertinents, comme la simulation des ondes KEEN.

**Abstract.** This work concerns the numerical simulation of the Vlasov-Poisson equation using semi-Lagrangian methods on Graphics Processing Units (GPU). To accomplish this goal, modifications to traditional methods had to be implemented. First and foremost, a reformulation of semi-Lagrangian methods is performed, which enables us to rewrite the governing equations as a circulant matrix operating on the vector of unknowns. This product calculation can be performed efficiently using FFT routines. Nowadays GPU is no more limited to single precision; however, single precision may still be preferred with respect to performance and available memory. So, in order to be able to deal with single precision, a $\delta f$ type method is adopted which only needs refinement in specialized areas of phase space but not throughout. Thus, a GPU Vlasov-Poisson solver can indeed perform high precision simulations (since it uses very high order of reconstruction and a large number of grid points in phase space). We show results for more academic test cases and also for physically relevant phenomena such as the bump on tail instability and the simulation of Kinetic Electrostatic Electron Nonlinear (KEEN) waves.

[*] *Thanks to Edwin Chacon-Golcher, Philippe Helluy, Guillaume Latu, Pierre Navaro for fruitful discussions and helps*

[**] *Thanks to the CEMRACS organizers and participants for the nice stay*

[***] *This work was carried out within the framework the European Fusion Development Agreement and the French Research Federation for Fusion Studies. It is supported by the European Communities under the contract of Association between Euratom and CEA. The views and opinions expressed herein do not necessarily reflect those of the European Commission.*

[1] IRMA, Université de Strasbourg, 7, rue René Descartes, F-67084 Strasbourg & INRIA-Nancy Grand-Est, projet CALVI, e-mail : mehrenbe@math.unistra.fr.

[2] IRMA, Université de Strasbourg, 7, rue René Descartes, F-67084 Strasbourg & INRIA-Nancy Grand-Est, projet CALVI, e-mail : steiner@math.unistra.fr.

[3] LIPHY, Université Joseph Fourier, 140, avenue de la Physique, F-38402 Saint Martin d'Hères, e-mail : luca.marradi@ujf-grenoble.fr.

[4] INRIA-Rennes Bretagne Atlantique, projet IPSO & IRMAR, Université de Rennes 1, 263 avenue du général Leclerc, F-35042 Rennes, e-mail : nicolas.crouseilles@inria.fr.

[5] Max-Planck Institute for plasma physics, Boltzmannstr. 2, D-85748 Garching, e-mail : sonnen@ipp.mpg.de.

[6] Polymath Research Inc., 827 Bonde Court, Pleasanton, CA 94566, e-mail : bedros@polymath-usa.com.

© EDP Sciences, SMAI 2013

Article published online by EDP Sciences and available at http://www.esaim-proc.org or http://dx.doi.org/10.1051/proc/201343003

## Introduction

At the one body distribution function level, the kinetic theory of charged particles interacting with electrostatic fields and ignoring collisions, may be described by the Vlasov-Poisson system of equations. This model takes into account the phase space evolution of a distribution function $f(t,x,v)$ where $t \geq 0$ denotes time, $x$ denotes space and $v$ is the velocity. Considering one-dimensional systems leads to the $1D \times 1D$ Vlasov-Poisson model where the solution $f(t,x,v)$ depends on time $t \geq 0$, space $x \in [0,L]$ and velocity $v \in \mathbb{R}$. The distribution function $f$ satisfies

$$\partial_t f + v\partial_x f + E\partial_v f = 0, \tag{1}$$

where $E(t,x)$ is an electric field. Poisson's law dictates that the charged particle distribution must be summed over velocity to render the self-consistent electric field as a solution to the Poisson equation:

$$\partial_x E = \int_{\mathbb{R}} f dv - 1. \tag{2}$$

To ensure the uniqueness of the solution, we impose to the electric field a zero mean condition $\int_0^L E(t,x)dx = 0$. The Vlasov-Poisson system (1)-(2) requires an initial condition $f(t=0,x,v) = f_0(x,v)$. We will restrict our attention to periodic boundary conditions in space and vanishing $f$ at large velocity.

Due to the nonlinearity of the self-consistent evolution of two interacting fields, in general it is difficult to find an analytical solution to (1)-(2). This necessitates the implementation of numerical methods to solve it. Historically, progress was made using particles methods (see [4]) which consist in advancing in time macro-particles through the equations of motion whereas the electric field is computed on a spatial mesh. Despite the inherent statistical numerical noise and their low convergence, the computational cost of particle methods is very low even in higher dimensions which explains their enduring popularity. On the other hand, Eulerian methods, which have been developed more recently, rely on the direct gridding of phase space $(x,v)$. Eulerian methods include finite differences, finite volumes or finite elements. Obviously, these methods are very demanding in terms of memory, but can converge very fast using high order discrete operators. Among these, semi-Lagrangian methods try to retain the best features of the two approaches: the phase space distribution function is updated by solving backward the equations of motion (*i.e.* the characteristics), and by using an interpolation step to remap the solution onto the phase space grid. These methods are often implemented in a split-operator framework. Typically, to solve (1)-(2), the strategy is to decompose the multi-dimensional problem into a sequence of $1D$ problems. We refer to [2,6,9,12,14,15,19,24] for previous works on the subject.

The main goal of this work is to use recent GPU devices for semi-Lagrangian simulations of the Vlasov-Poisson system (1)-(2). Indeed, looking for new algorithms that are highly scalable in the field of plasmas simulations (like tokamak plasmas or particle beams), it is important to mimic plasma devices more reliably. Particle methods have already been tested on such architectures, and good scalability has been obtained as in [5,30]. We mention a recent precursor work on the parallelization in GPU in the context of a gyrokinetic eulerian code GENE [13]. Semi-Lagrangian algorithms dedicated to the simplified setting of the one-dimensionnal Vlasov-Poisson system have also recently been implemented in the CUDA framework (see [22,25]). In the latter two works, in which the interpolation step is based on cubic splines, one can see that the speedup can reach a factor of $\times 80$ in certain cases. Here, we use higher complexity algorithms, which are based on the Fast Fourier Transform (FFT). We will see that our GPU simulations will directly benefit from the huge acceleration obtained for the FFT on GPU. They are thus also very fast enabling us to test and compare different interpolation operators (very high order Lagrangian or spline reconstructions) using a large number of grid points per direction in phase space.

To achieve this task, flexibility is required to switch easily from one representation of an operator to another. Here, semi-Lagrangian methods are reformulated in a framework which enables the use of existing optimized Fast Fourier Transform routines. This formulation gives rise to a matrix which possesses the circulant property, which is a consequence of the periodic boundary conditions. Let us emphasize that such boundary conditions are used not only in $x$ but also in $v$; this is made possible by taking the velocity domain $[-v_{\max}, v_{\max}]$, with $v_{\max}$

big enough. Note also that the proof of convergence of such numerical schemes can be obtained following [3,7]. Due to the fact that such matrices are diagonalizable in a Fourier basis, the matrix vector product can be performed efficiently using FFT. In this work, Lagrange polynomials of various odd degrees $(2d+1)$ and B-spline of various degree $k$ have been tested and compared. Another advantage of the matrix-vector product formulation is that the numerical cost is almost insensitive to the order of the method. Finally, since single precision computations are preferable to get maximum performance out of a GPU, other improvements have to be made to the standard semi-Lagrangian method. To achieve the accuracy needed to observe relevant physical phenomena, two modifications are proposed: the first is to use a $\delta f$ type method following [22]. The second is to impose a zero spatial mean condition on the electric field. Since the response of the plasma is periodic, this is always satisfied.

The rest of the paper is organized as follows. First, the reformulation of the semi-Lagrangian method using FFT is presented for the numerical treatment of the doubly periodic Vlasov-Poisson model. Then, details of the GPU implementation are given, highlighting the particular modifications that were necessary in order to use GPUs with single precision. We then move on to show numerical results. These involve several comparisons between the different methods and orders of numerical approximation and their performances on GPU and CPU on three canonical test problems.

## 1. FFT implementation

In this section, we give an explicit formulation of semi-Lagrangian schemes for the solution of the Vlasov-Poisson system of equations in the doubly periodic case using circulant matrices. First, the classical directional Strang splitting (see [9,27]) is recalled. Then, the problem is reduced to a sequence of one-dimensional constant advections; a circulant-matrix formulation is proposed, for which the use of Fast Fourier Transform is very well suited; it can be applied for many methods, with arbitrary order of interpolation.

### 1.1. Strang-splitting

For the Vlasov-Poisson set of equations (1)-(2), it is natural to split the transport in the $x$-direction from the transport in the $v$-direction. Moreover, this also corresponds to a splitting of the kinetic and electrostatic potential part of the Hamiltonian $|v|^2/2+\phi(t,x)$ where the electrostatic potential $\phi$ is related to the electric field through $E(t,x)=-\partial_x\phi(t,x)$.

For plasmas simulations, even when high order splittings are possible (see [11] and references therein), the second order Strang splitting is a good compromise between accuracy and simplicity, which explains its popularity. Due to filamentation, even if high order scheme and fine grid is used in space, the error is generally more important in space than in time. On the other hand, the use of high order splitting is a possible option, which can be managed easily and will probably have the capability of enhancing the results and/or diminishing the cost of the simulation.

Starting from time $t_n=n\Delta t$ and assuming that $f^n\simeq f(t_n,\cdot,\cdot)$ and $E^n\simeq E(t_n,\cdot)$ are known, the Strang splitting is composed of three steps plus an update of the electric field before the advection in the $v$-direction

(1) Transport in $v$ over $\Delta t/2$: compute $f^{n,\star}(x,v)=g(\Delta t/2,x,v)$ by solving

$$\partial_t g(t,x,v)+E^n(x)\partial_v g(t,x,v)=0,$$

with the initial condition $g(0,x,v)=f^n(x,v)$.

(2) Transport in $x$ over $\Delta t$: compute $f^{n,\star\star}(x,v)=g(\Delta t,x,v)$ by solving

$$\partial_t g(t,x,v)+v\partial_x g(t,x,v)=0,$$

with the initial condition $g(0,x,v)=f^{n,\star}(x,v)$.
Update of electric field $E^{n+1}(x)$ by solving $\partial_x E^{n+1}(x)=\int f^{n,\star\star}(x,v)dv-1$.

(3) Transport in $v$ over $\Delta t/2$: compute $f^{n+1}(x,v) = g(\Delta t/2, x, v)$ by solving

$$\partial_t g(t,x,v) + E^{n+1}(x)\partial_v g(t,x,v) = 0,$$

with the initial condition $g(0,x,v) = f^{n,\star\star}(x,v)$.

One of the main advantages of this splitting is that the algorithm reduces to solving a series of one-dimensional constant coefficient advections. Indeed, considering the transport along the $x$-direction, for each fixed $v$, one faces a constant advection. The same is true for the $v$-direction since for each fixed $x$, $E^n$ does not depend on the advected variable $v$. We choose to start with the advection in $v$, which permits to get the electric field at integer multiples of time steps. The third step of the $n^{th}$ iteration could be merged with step (1) of the $(n+1)^{th}$ iteration, but we do not resort to this short cut here.

## 1.2. Constant advection

In this part, a reformulation of semi-Lagrangian methods is proposed, in the case of constant advection equations with periodic boundary conditions. Let us consider $u = u(t,x)$ to be the solution of the following equation for a given $c \in \mathbb{R}$:

$$\partial_t u + c\partial_x u = 0, \qquad u(t=0,x) = u_0(x),$$

where periodic boundary conditions are assumed in $x \in [0,L]$. The continuous solution satisfies for all $t, s \geq 0$ and all $x \in [0,L]$: $u(t,x) = u(s, x - c(t-s))$. Let us mention that $x - c(t-s)$ has to be understood *modulo* $L$ since periodic boundary conditions are being considered.

Let us consider a uniform mesh within the interval $[0,L]$: $x_i = i\Delta x$ for $i = 0,\dots,N$ and $\Delta x = L/N$. We also introduce the time step $\Delta t = t_{n+1} - t_n$ for $n \in \mathbb{N}$. Note that we have $u_0^n = u_N^n$. By setting

$$u^n = \begin{pmatrix} u_0^n \\ \vdots \\ \vdots \\ u_{N-1}^n \end{pmatrix}, \quad u_i^n \approx u(t_n, x_i), \tag{3}$$

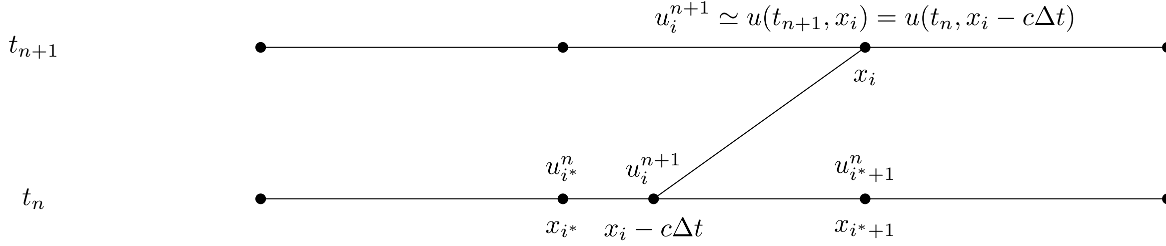
the semi-Lagrangian scheme reads $u_i^{n+1} = \pi u^n(x_i - c\Delta t)$ where $\pi$ is a piecewise polynomial function which interpolates $u_i^n$ for $i = 0,\dots,N-1$: $\pi(x_i) = u_i^n$. This can be reformulated into $u^{n+1} = Au^n$ where $A$ is the matrix defining the interpolation. Periodic boundaries imply that the matrix $A$ is circulant:

$$A = \mathcal{C}(a_0, a_1, ..., a_{N-1}) := \begin{pmatrix} a_0 & a_1 & \dots & \dots & a_{N-1} \\ a_{N-1} & a_0 & a_1 & \dots & a_{N-2} \\ \ddots & \ddots & \ddots & \ddots & \ddots \\ \ddots & \ddots & \ddots & \ddots & \ddots \\ a_1 & \dots & \dots & a_{N-1} & a_0 \end{pmatrix} \tag{4}$$

Obviously, this matrix depends on the choice of the polynomial reconstruction $\pi$. In the following, some explicit examples are shown.

**Examples of various methods and orders of interpolation**

We have to evaluate $\pi u^n(x_i - c\Delta t)$. Let $\beta := -c\Delta t/\Delta x$ be the normalized displacement which can be written in a unique way as $\beta = b + b^\star$ with $(b, b^\star) \in \mathbb{Z} \times [0,1[$. This means that the feet of the characteristics $(x_i - c\Delta t)$ belong to the interval $[x_{i^\star}, x_{i^\star+1}[$ with $i^\star + b^\star = i + \beta$, or $i^\star = i + b$.




(1) *Lagrange* 1. The nonvanishing terms of the matrix $A$ are:

$$a_b = 1 - b^\star, \qquad a_{b+1} = b^\star.$$

(2) *Lagrange* $2d+1$ (with $2d+1 \leq N-1$). The nonvanishing terms of matrix are :

$$\forall j \in \{-d, \dots, d+1\}, \quad a_{b+j} = \prod_{k=-d,\ k\neq j}^{d+1} \frac{b^\star - k}{j-k}.$$

(3) *B-Spline of degree k.*
We define $B_i^k(x)$ the B-spline of degree $k$ on the mesh $(x_i)_i$ by the following recurrence:

$$B_i^0(x) = \mathbb{1}_{[x_i, x_{i+1}[}(x), \qquad B_i^k(x) = \frac{x - x_i}{k\Delta x} B_i^{k-1}(x) + \left(1 - \frac{x - x_{i+1}}{k\Delta x}\right) B_{i+1}^{k-1}(x).$$

Then, in this case, the nonvanishing terms of the matrix $A$ are:

$$A = M \times \mathcal{C}(\underbrace{0, \dots, 0}_{N-k}, \underbrace{B_0^k(x_1), B_0^k(x_2), \dots, B_0^k(x_k)}_{k})^{-1},$$

where the nonvanishing terms of the circulant matrix $M$ are:

$$\forall j \in \{0, \dots, k\}, \qquad m_{b-j} = B_0^k(x_{j+b^\star}).$$

Now, starting from this reformulation, the algorithm reduces to a matrix vector product at each time step. Since the matrices are circulant, this product can be performed using FFT. Indeed, circulant matrices are diagonalizable in Fourier space [18] so that

$$A = UDU^\star,$$

where $\frac{1}{\sqrt{N}}U$ is unitary ($U^\star$ denotes the adjoint matrix of $U$) and $D$ is diagonal. They are given by

$$\begin{aligned} U_{m,k} &= e^{-2i\pi mk/N}, \;\; m,k = 0 \dots N-1, \\ D_{m,m} &= \sum_{k=0}^{N-1} a_k e^{-2i\pi mk/N}, \;\; m = 0, ..., N-1. \end{aligned}$$

The product of $U$ by a vector $v \in \mathbb{R}^N$ can then be obtained performing the Fast Fourier Transform of $v$. In the same way, $U^\star v$ can be obtained by computing the inverse Fourier Transform of $v$.

The product matrix vector $Au^n = UDU^\star u^n$ is then computed following the algorithm:

(1) Compute $U^\star u^n$ by calculating $\tilde{u} = \mathrm{FFT}^{-1}(u^n)$.
(2) Compute $D$ by calculating $\mathrm{FFT}(a)$.
(3) Compute $w = DU^\star u^n$ by calculating $D\tilde{u}$.

(4) Compute $Au^n$ by calculating FFT$(w)$.

The complexity of the algorithm is then $\mathcal{O}(N \log N)$, independently of the degree of the polynomial reconstruction.

## 2. CUDA GPU implementation

From the Strang-splitting and the constant advection, we can easily define the $2D$ algorithm. The unknowns are

$$f_{i,j}^n \simeq f(t_n, x_i, v_j),\ x_i = x_{\min} + i\Delta x,\ v_j = v_{\min} + j\Delta v,\ i = 0, \dots, N_x - 1,\ j = 0, \dots, N_v - 1,$$

with $\Delta x = (x_{\max} - x_{\min})/N_x,\ \Delta v = (v_{\max} - v_{\min})/N_v$, and $N_x,\ N_v \in \mathbb{N}^*$. We use kernels on GPU by using existing NVIDIA routines for FFT, transposition and scalar product. Note that such a choice has also been made in the more difficult context [13]. We would have liked to use OPENCL (as done in [10]) in order not be attached to NVIDIA cards; but we had difficulties to get the friendly well-documented features of NVIDIA, especially for the FFT.

FFTs are computed using the `cufft` library. For transposition, different possible algorithms are provided from CUDA samples (see http://docs.nvidia.com/cuda/cuda-samples/index.html#matrix-transpose). For this step, the condition $N = N_x = N_v$ is always required We also have that $N$ is a power of 2, for the FFT step. In order to compute charge density

$$\rho(t, x) = \int f(t, x, v) dv \simeq \Delta v \sum_{j=0}^{N_v - 1} f(t, x, v_j),$$

we adapt the `ScalarProdGPU` routine from CUDA samples (see http://docs.nvidia.com/cuda/cuda-samples/index.html#scalar-product), since we have

$$\sum_{j=0}^{N_v - 1} f(t, x, v_j) = \langle u, v \rangle,\ \text{with } u = (f(t, x, v_0), \dots, f(t, x, v_{N_v - 1})),\ v = (1, \dots, 1),$$

and $\langle \cdot, \cdot \rangle$ is the scalar product.

We also write a kernel on GPU for computing coefficients of the $A$ matrix. An analytical formula is used for each coefficient $a_i$. In the case of Lagrange interpolation of degree $2d + 1$, the complexity switches from $O(Nd)$ to $O(Nd^2)$ operations because of a rewritten CPU divided differences based algorithm which cannot be parallelized.

The main steps of the algorithm are :

- Initialization: the initial condition computed on CPU and transferred to GPU
- Computation of initial charge density $\rho$ on GPU by using `ScalarProd`
- Transfer of $\rho$ to CPU
- Computation of the electric field $E$ on CPU
- Time loop
  1. $\Delta t/2$ advection in $v$ with FFT on GPU
  2. Transposition in order to pass into the $x$-direction on GPU
  3. $\Delta t$ advection in the $x$ direction with FFT on GPU
  4. Transposition in order to pass into the $v$-direction on GPU
  5. Computation of $\rho$ on GPU by using `ScalarProd`
  6. Transfer of $\rho$ to CPU
  7. Computation of the electric field $E$ on CPU
  8. $\Delta t/2$ advection in $v$ with FFT on GPU

**Remarque 2.1.** The electric field is computed on CPU, for simplicity; we have not made the effort to translate the code in GPU. We expect not to have a real improvement on the performance by computing the electric field on GPU; the amount to transfer is negligible (the size is $O(N)$) compared to the advection (the size is $O(N^2)$).

### Some details on the implementation

We list here the CUDA kernel calls for information and give some descriptions of the variables.

```
//for transposition
  transposeNoBankConflicts<<<grid, threads>>>(d_odata, f_d, N, N, 1);
  copy<<<grid, threads>>>(f_d, d_odata, N, N, 1);

//for advection
  //for going to complex data
  real_to_complex<<<grid, threads>>> (f_d, f_complex_d, N);
  //for matrix computation: from alpha_d and  i0_d returns w_d
  compute_coefficients<<<grid, threads>>> (w_d, alpha_d, i0_d, N, degree);
  cufftExecZ2Z (plan1d, w_d, w_d, CUFFT_FORWARD);
  //forward fft
  cufftExecZ2Z (plan1d, f_complex_d, f_complex_d, CUFFT_FORWARD);
  //for multiplication in Fourier space
  mult_complex_array<<<grid, threads>>> (f_complex_d, w_d, N);
  //backward fft
  cufftExecZ2Z (plan1d, f_complex_d, f_complex_d, CUFFT_INVERSE);
  //for going back to real data and multiply by scale factor
  complex _to_real_scaled<<<grid, threads>>> (f_complex_d, f_d, N, scale);

//for computation of charge density
  compute_sum_v<<<grid_rho, threads_rho>>> (rho_d, f_d, N);
```

The variables `f_d, d_odata` are real arrays of size $N^2$; `f_d` represents the distribution function $f$.
The variables `w_d, f_complex_d` are complex arrays of size $N^2$; `w_d` represents the matrix of coefficients. In Fourier space, we then only need to make the multiplication terms by terms with `f_complex_d`, the complex Fourier transform of $f$.
The variable `i0_d` (resp. `alpha_d`) is an integer (resp. real) array of size $N$; its elements are $b$ (resp. $b^*$), for each of the $N$ constant advections.
The variable `degree` is an integer, that represents $d$, that stands here for the Lagrange interpolation of degree $2d+1$.
The variable `scale` is a real number that is set to $1/N$ for FFT scaling purpose.
The variable `plan1d` is a type that initializes the FFT, so that the FFT is applied $N$ times for vectors of size $N$, which are stored contiguously in memory. For this, we just have to make the following call once for all:

```
cufftPlan1d( &plan1d, N, CUFFT_Z2Z, N);
```

This explains why the transposition of the data are necessary.
The variable `rho_d` is a real array of size $N$, which stores the charge density $\rho$.
The variables `grid, threads` and `grid_rho, threads_rho` are initialized as follows:

```
dim3 grid(N/TILE_DIM, N/TILE_DIM), threads(TILE_DIM,BLOCK_ROWS);
dim3 grid_rho(N/TILE_DIM, 1), threads_rho(TILE_DIM,1);
```

Here we have set `TILE_DIM=16` and `BLOCK_ROWS=16`. These values are typical GPU parameters which may be changed (according to some constraints) for better performances; but we have here not made changes with regards to these parameters. For the `compute_sum_v` routine, as already told, we have adapted the `scalarProdGPU` routine. There a variable `ACCUM_N` was set to 1024. We have set it here to 32.

## 3. Questions about single precision

In principle, computations on GPU can be performed using either single or double precision. However, the numerical cost becomes quite high when one deals with double precision (we will see in our case, that the cost is generally a factor of two) and is not always easily available across all platforms. Note that in [13] and [25], only double precision was used. Discussions about single precision have already been presented in [22]. Hereafter, we propose two slight modifications of the semi-Lagrangian method which enable the use of single precision computations while at the same time recovering the precision reached by a double precision CPU code.

### 3.1. $\delta f$ method

The $\delta f$ method consists on a scale separation between an equilibrum and a perturbation so that we decompose the solution as

$$f(x,v) = \delta f(x,v) + f_{\rm eq}(v), \quad f_{\rm eq}(v) = \frac{1}{\sqrt{2\pi}}\exp(-v^2/2).$$

Then, we are interested in the time evolution of $\delta f$ which satisfies

$$\partial_t \delta f + v\partial_x \delta f + E\partial_v[f_{\rm eq} + \delta f] = 0.$$

The Strang splitting presented in subsection 1.1 is modified since we advect $\delta f$ instead of $f$. Since $f_{\rm eq}$ only depends on $v$, advections in $x$ are not modified. Now we can rewrite the $v$-advection as

$$\partial_t[f_{\rm eq} + \delta f] + E^n\partial_v[f_{\rm eq} + \delta f] = 0,$$

with the initial condition $f_{\rm eq} + \delta f^{n,\star}$. This means that $(f_{\rm eq} + \delta f)$ is preserved along the characteristics $(f_{\rm eq} + f^{n,\star\star})(x,v) = (f_{\rm eq} + f^{n,\star})(x, v - \Delta t E^n(x))$. We then deduce that

$$\delta f^{n,\star\star}(x,v) = \delta f^{n,\star}(x, v - \Delta t E^n(x)) + f_{\rm eq}(v - \Delta t E^n(x)) - f_{\rm eq}(v).$$

which provides the update of $\delta f$ for the $v$-advection. Note that $f_{\rm eq}(v - \Delta t E^n(x))$ is an evaluation and not an interpolation.

**Remarque 3.1.** We use here the standard Gaussian $f_{\rm eq}$, because in our test cases, we are not far from this equilibrium. In the bump on tail test case, at initial time, we are nearer of another (unstable) equilibrium: there is another Gaussian, the bump, which is however small and does not remain constant in time; thus we have not found worth enough to adapt $f_{\rm eq}$ to that equilibrium. It may be interesting to look for situations, where we are not far from another equilibrium (which may even evolve in time) and see how to adapt the procedure. Note that we explicitely use here that $f_{\rm eq}$ does not depend on $x$.

### 3.2. The zero mean condition

The electric field is computed from (2). Note that the right hand side of (2) has zero mean, and the resulting electric field has also zero mean. This is true at the continuous level; however when we deal with single precision, a systematic cumulative error could occur here. In fact, it is also true in the double precision case, but the influence is quite less significative, as we will see on the numerical results. In order to prevent this cumulative error phenomenon, we can enforce the zero mean condition on the discrete grid numerically: from $\rho_k^n \simeq \rho(t^n, x_k) = \int_{\mathbb{R}} f(t^n, x_k, v)dv,\ k = 0, \ldots, N-1$, we compute the mean

$$M = \frac{1}{N}\sum_{k=0}^{N-1} \rho_k^n,$$

and then subtract this value to $\rho_k^n$:

$$\tilde{\rho}_k^n = \rho_k^n - M,\ k = 0, \ldots, N-1,$$

so that $\tilde{\rho}_k^n \simeq \rho(t^n, x_k) - 1$ is of zero mean numerically, whereas $\rho_k^n - 1$ is only approximatively of zero mean.

We repeat this same procedure once the electric field is computed: from a given computed electric field $\tilde{E}_k^n,\ k = 0, \ldots, N-1$, which may not be of zero mean, we compute $\tilde{M} = \frac{1}{N}\sum_{k=0}^{N-1} \tilde{E}_k^n$, and set

$$E_k^n = \tilde{E}_k^n - \tilde{M},\ k = 0, \ldots, N-1.$$

For computing the electric field, we use the trapezoidal rule:

$$\tilde{E}_{k+1}^n = \tilde{E}_k^n + \Delta x \frac{\tilde{\rho}_k^n + \tilde{\rho}_{k+1}^n}{2},\ k = 0, \ldots, N-1, \text{ with } \tilde{E}_0^n \text{ set arbitrarily to zero,}$$

or Fourier (with FFT). Note that in the case of Fourier, the zero mean is automatically satisfied numerically, since the mode 0 which represents the mean is set to 0.
We will see that this zero mean condition is of great importance in the numerical results. It has to be satisfied with enough precision. It can be viewed as being related to the "cancellation problem" observed in PIC simulations [20]. Note also, that by dealing with $\delta f$, which is generally of small magnitude, a better resolution of the zero mean condition is reached.

## 4. Numerical results

This section is devoted to the presentation of numerical results obtained by the following methods: the standard semi-Lagrangian method (with various different interpolation operators), including the $\delta f$ and zero mean condition modifications. Comparisons between CPU and GPU simulations and discussions about the performance will be given on three test cases: Landau damping, bump on tail instability, and KEEN waves. As interpolation operator, we will use use by default LAG17, the Lagrange $2d+1$ interpolation with $d = 8$. Similarly, LAG3 stands for $d = 1$ and LAG9 for $d = 4$. We will also show simulations with standard cubic splines (for comparison purposes), which correspond to B-splines of degree $k$ with $k = 4$. We will use several machines for GPU: MacBook, irma-gpu1 and hpc. See subsection 4.4 for details.

### 4.1. **Landau Damping**

For this first standard test case [21], the initial condition is taken to be:

$$f_0(x, v) = \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{v^2}{2}\right)(1 + \alpha \cos(x/2)), \quad (x, v) \in [0, 4\pi] \times [-v_{\max}, v_{\max}],$$

with $\alpha = 10^{-2}$. We are interested in the time evolution of the electric energy $\mathcal{E}_e(t) = (1/2)\|E(t)\|_{L^2}^2$ which is known to be exponentially decreasing at a rate $\gamma = 0.1533$ (see [29]). Due to the fact that the electric energy decreases in time, this test emphasizes the difference between single and double precision computations.

Numerical results are shown on Figure 1 (top and middle left). We use LAG17, $N = 2048$, $v_{\max} = 8$ as default values.

In the single precision case (top left), we see the benefit of using the zero mean modifications (plots 6 and 7: *efft nodelta* and *trap zero mean nodelta*): the two results are similar (we use the trapezoidal rule for the electric field or Fourier and we recall that in both cases, the zero mean is satisfied) and improved with respect to the case where the zero mean is not enforced in the trapezoidal case (plot 8: *trap nodelta*). We have counted 23 right oscillations until time $t = 50$ for plots 6 and 7 (the two last oscillations are however less accurately described), whereas we have only 16 right oscillations until time $t = 34.8$ for plot 8, before saturation. If we

use the $\delta f$ method, we observe a further improvement (plots 1 to 5): we gain 4 oscillations (that is we have 27 oscillations in total) until time $t = 60$, and the electric field is below $6 \cdot 10^{-6} < e^{-12}$. Note that in the case where we use the $\delta f$ method, adding the zero mean modification has no impact here; on the other hand, results with the $\delta f$ method are better than results with the zero mean modification on this picture. We have also added a result on an older machine (plot 9: *MacBook*), which leads to very poor results (only 9 oscillations until time $t = 19$ for the worst method). The use of an older version of CUDA and non conforming IEEE floating point standard may explain this behaviour; this should be corrected with new versions of CUDA. Also the results, which are not shown here, due to space limitations, were different by applying the modifications; as an example, we got 28 right oscillations by using the $\delta f$ method with zero mean modification. Floating point standard may not have been satisfied there which could explain the difference in the results.

In the double precision case (top right), we can go to higher precision results. By using $\delta f$ method or zero mean modification (the difference between both options is less visible), we get 92 right oscillations until time $t = 206$ (the last oscillation is not good resolved hat the end), the electric field goes under $6 \cdot 10^{-13} < e^{-28}$, and we guess that we could add 11 more oscillations until time $t = 231$ (we see that grid size effects pollute the result), to obtain 103 oscillations and with electric field below $6 \cdot 10^{-14} < e^{-30}$, but we are limited here in the double precision case to $N = 2048$. A CPU simulation with $N = 4096$ confirms the results. We also see the effect of the grid (runs with $N = 1024$) and the velocity (runs with $v_{\max} = 6$). Note that the plot 6 (*trap nodelta 1024 v6*) has lost a lot of accuracy compared to the other plots: the grid size is too small ($N = 1024$), the velocity domain also ($v_{\max} = 6$) and above all there is no zero mean or $\delta f$ method. In that case, we only reach time $t = 100$. We refer to [17,31] for other numerical results and discussions and to the seminal famous work [23] for theoretical results. In [31], it was already mentionned that we have to take the velocity domain large enough and to take enough grid points. Concerning GPU and single precision, the benefit of a $\delta f$ method was also already treated in [22]: 29 right oscillations were obtained in the single precision case with a $\delta f$ modification, 13 right oscillations without the modification and the time $t = 100$ was reached in the CPU case ($N$ was set to 1024 and $v_{\max}$ to 6).

On Figure 1 middle left, we plot the error of mass, which is computed as $|\hat{\rho}_0 - 1|$. We clearly see the impact between the conservation of the mass and the former results. We can also note that, the zero mean modification does not really improve the mass conservation (just a slight improvement at the end, plots 2,3,4), but has a benefic effect on the electric field: the bad behaviour of the mass conservation is not propagated to the electric field. On the other hand, the $\delta f$ method clearly improves the mass conservation. We also see the effect of taking a too small velocity domain, in the double precision case.

### 4.2. **Bump on tail**

For this second standard test case, the initial condition is considered as a spatial and periodic perturbation of two Maxwellians (see [27])

$$f_0(x, v) = \left( \frac{9}{10\sqrt{2\pi}} \exp\left(-\frac{v^2}{2}\right) + \frac{2}{10\sqrt{2\pi}} \exp(-2(v - 4.5)^2) \right) (1 + 0.03\cos(0.3x)), \quad (x, v) \in [0, 20\pi] \times [-9, 9].$$

The Vlasov-Poisson model preserves physical quantities with time like Casimir functions, which will be used to compare the different implementations. Particulary, we look at the time history of the total energy $\mathcal{E}$ of the system, which is the sum of the kinetic energy $\mathcal{E}_k$ and the electric energy $\mathcal{E}_e$

$$\mathcal{E}(t) = \mathcal{E}_k(t) + \mathcal{E}_e(t) = \int_0^{20\pi} \int_{\mathbb{R}} f(t, x, v) \frac{v^2}{2} dv dx + \frac{1}{2} \int_0^{20\pi} E^2(t, x) dx.$$

As in the previous case, the time evolution of the electric energy is chosen as a diagnostics.

Results are shown on Figure 1 (middle right and bottom) and on Figure 2.

We see on Figure 1 middle right the evolution in time of the electric field. Single and double precision results are compared. In the single precision case, the $\delta f$ method with FFT computation of the electric field (plot

3: *single delta*) is the winner and the basic method without modifications and trapezoidal computation of the electric field (plot 7: *trap single no delta*) leads to the worst result. Double precision computations lead to better results and differences are small: plots 1 (*double delta*) and 2 (*double no delta*) are undistinguishable and plot 8 (*trap double no delta*) is only different at the end. Thus, such modifications are not so mandatory in the double precision case. We then see for the same runs, the evolution of the error of mass (bottom left) and of the first mode of $\rho$ in absolute value (bottom right). We notice that the error of mass linearly accumulates in time. Here no error coming from the velocity domain is seen, because $v_{\max}$ is large enough ($v_{\max} = 9$ in all the runs). The evolution of the first mode of $\rho$ is quite instructive: we see that it exponentially grows from round off errors and the different runs lead to quite different results. The loss of mass can become critical in the single precision case (no real impact in the double precision case are detected) and implementations without mass error accumulation would be desirable. The $\delta f$ method improves the results, but the error of mass still accumulates much more than in the double precision case.

On Figure 2, we see the same diagnostics in the double precision case. We make vary the number of grid points, the degree of the interpolation and the time step. By taking smaller time step, we can increase the time before the merge of two vortices among three which leads to a breakdown of the electric field. Higher degree interpolation lead to better results (in the sense that the breakdown occurs later), for not too high grid resolution. When $N = 2048$, lower order interpolation seems to be better, since it introduces more diffusion, whereas high order schemes try to capture the small scales, which are then sharper and more difficult to deal with in the long run. Adaptive methods and methods with low round-off error in the single precision case may be helpful to get better results.

### 4.3. **KEEN Waves**

In this last and most intricate test, instead of considering a perturbation of the initial data, we add an external driving electric field $E_{\text{app}}$ to the Vlasov-Poisson equations:

$$\partial_t f + v\partial_x f + (E - E_{\text{app}})\partial_v f = 0, \ \partial_x E = \int_{\mathbb{R}} f dv - 1,$$

where $E_{\text{app}}(t,x)$ is of the form $E_{\text{app}}(t,x) = E_{\max} k a(t)\sin(kx - \omega t)$, where

$$a(t) = \frac{0.5(\tanh(\frac{t-t_L}{t_{wL}}) - \tanh(\frac{t-t_R}{t_{wR}})) - \epsilon}{1-\epsilon}, \ \epsilon = 0.5(\tanh(\frac{t_0 - t_L}{t_{wL}}) - \tanh(\frac{t_0 - t_R}{t_{wR}}))$$

is the amplitude, $t_0 = 0,\ t_L = 69,\ t_R = 307,\ t_{wL} = t_{wR} = 20$, $k = 0.26$, $\omega = 0.37$ and $E_{\max} = 0.2$. The initial condition is

$$f_0(x,v) = \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{v^2}{2}\right), \quad (x,v) \in [0, 2\pi/k] \times [-6, 6].$$

See [1, 28] for details about this physical test case. Its importance stems from the fact that KEEN waves represent new non stationary multimode oscillations of nonlinear kinetic plasmas with no fluid limit and no linear limit. They are states of plasma self-organization that do not resemble the (single mode) way in which the waves are initiated. At low amplitude, they would not be able to form. KEEN waves can not exist off the dispersion curves of classical small amplitude waves unless a self-sustaining vortical structure is created in phase space, and enough particles trapped therein, to maintain the self-consistent field, long after the drive field has been turned off. For an alternate method of numerically simulating the Vlasov-Poisson system using the discontinuous Galerkin approximation, see [8] for a KEEN wave test case.
As diagnostics, we consider here different snapshots of $f - f_0$ at different times: $t = 200, t = 300, t = 400, t = 600$ and $t = 1000$.

We first consider the time $t = 200$ (upper left in Figure 3). At this time, all the snapshots are similar so we present only one (GPU single precision and a grid of $1024^2$ points). The five others graphics of this figure are taken at time t=300. We show that, at this time, there is again convergence because the graphic on the middle

right (GPU single precision, $N = 4096$ and $\Delta t = 0.1$) is identical to the bottom left one (GPU single precision, $N = 4096$ and $\Delta t = 0.01$).

Figure 4 presents different snapshots at times $t = 400$ and $t = 600$. At time $t = 400$, the upper left graphic (GPU single precision, $N = 2048$, $\Delta t = 0.1$) is similar to the upper right one (CPU, $N = 2048$, $\Delta t = 0.05$), that shows that the CPU and the GPU codes give the same results. With 4096 points (on middle left), we observe a little difference with the 2048 points case. Between the snapshots at time $t = 400$ and those at time $t = 600$, we observe the emergence of diffusion.

The time $t = 1000$ is considered on Figure 5. We see that there is no more convergence at this time: there is a lag, but the structure remains the same. We compare also different interpolators (cubic splines, LAG 3, LAG 9, LAG 17). If the order of the interpolation is high (graphic at the top right : CPU, LAG 17, $\Delta t = 0.05, N_x = 512, N_v = 4096$) there is appearance of finer structures. At this time, one sees little difference between CPU results (graphic at the middle right : CPU, LAG 3, $\Delta t = 0.05, N = 4096$) and GPU results (graphic at the bottom left : GPU, LAG 3, $\Delta t = 0.05, N = 4096$), but there is no lag.

Figure 6 (at time $t = 1000$) shows the differences between single and double precision when the value of $N$ is changed. The two graphs above show the case $N = 1024$, the left is single precision while the right one is in double precision. We see that there are very few differences. When $N = 2048$, the results are different in single precision (graphic on middle left) and double precision (graphic on middle right). When $N = 4096$, the code does not work in double precision so we compared the results for single precision GPU with $\Delta t = 0.05$ (bottom left graphic) and $\Delta t = 0.01$ (bottom right graphic). There are also differences due to the non-convergence. Moreover, we see that there are more filamentations when $N$ increases.

Figure 7 shows the time evolution of the absolute value of the first Fourier modes of $\rho$. We see that single precision can modify the results on the long time (top left). The GPU code is validated in double precision (top right). We clearly see the benefit of the $\delta f$ method in the GPU single precision (middle left), where it has no effect in the double precision case (middle right). Further plots are given with $N = 4096$ (bottom left and right). With smaller time steps, some small oscillations appear with single precision GPU code (bottom right). In all the plots, we see no difference at the beginning; differences appear in the long run as it was the case for the plots of the distribution function.

### 4.4. **Performance results**

**Characteristics.** We have tested the code on different computers with the following characteristics:

- GPU
  - (1) = irma-gpu1 : NVIDIA GTX 470 1280 Mo; Cuda version 5.0
  - (2) = hpc : GPU NVIDIA TESLA C2070; Cuda version 5.0
  - (3) = MacBook : NVIDIA GeForce 9400M; Cuda version 4.2
- CPU
  - (4) = MacBook : Intel Core 2 Duo 2.4 GHz
  - (5) = irma-hpc2: Six-Core AMD Opteron(tm) Processor 8439 SE
  - (6) = irma-gpu1: Intel Pentium Dual Core 2.2 Ghz 2Gb RAM
  - (7) = MacBook : Intel Core i5 2.4 GHz

We measure in the GPU codes the proportion of FFT which consists in: transform $1D$ real data to complex, computing the FFT, making the complex multiplication, computing the inverse FFT, transforming to real data (together with addition of $\delta f$ modification, if we use the $\delta f$ method). We add a diagnostic for having the proportion of time in the `cufftExec` routine; we note that this extra diagnostic can modify a little the time measures (when this is the case; new measures are given in brackets, see on Table 1).
When the number of cells grows, the proportion of FFT time in total time grows, as shown on Table 1 (KEEN wave test case with $\delta f$ modification) or Table 2 (KEEN wave test case without $\delta f$ modification). Note that the initialisation time and the 2d-diagnostic time are not included in total time.

The results with a CPU code (vlaso) without OpenMP are given on Table 3, top; in that code, the Landau test case is run with $\Delta t/2$ advection in $x$, followed by $\Delta t$ advection in $v$ and $\Delta t/2$ advection in $x$ and the last advection in $x$ of iteration $n$ is merged with the first advection in $x$ of iteration $n+1$.

The results with Selalib (Table 3, bottom) [26] are obtained with OpenMP. We use 2 threads for (4), 24 threads for (5), 2 threads for (6) and 4 threads for (7).

In order to compare the performances, we introduce the number MA which represents the number of millions of point advections made per second : $MA = \frac{N_{step} \times N_{adv} \times N^2}{10^6 \times \text{Total time}}$ and the number of operations per second (in GigaFLOPS) given by :

$$GF = \frac{N_{step} \times N_{adv} \times (2N \times 5N \log(N) + 6N^2)}{10^9 \times \text{Total time}} \quad \text{with complex data (GPU)}$$
$$GF = \frac{N_{step} \times N_{adv} \times (N \times 5N \log(N) + 3N^2)}{10^9 \times \text{Total time}} \quad \text{with real data (CPU)}$$

where $N_{step}$ refers to the number of time steps and $N_{adv}$ represents the number of advections made in each time step ($N_{adv} = 3$ in GPU and Selalib codes; $N_{adv} = 2$ in vlaso code). In each advection, we compute $N$ times (GPU in complex data) or $N/2$ times (CPU in real data) :

- A forward FFT and backward FFT with approximately $5N \log(N)$ operations for each FFT computation
- A complex multiplication that requires $6N$ operations.

The speedup in Table 1 and 2 are computed, by taking the fastest and slowest CPU simulation of Table 3. The comparison between Table 1 and Table 2 shows that the cost of the method $\delta f$ is not too important but not negligible. This cost could be optimized. We clearly benefit of the huge acceleration of the FFT routines in GPU and we thus gain a lot to use this approach. Most of the work is on the FFT, which is optimized for CUDA in the `cufft` library, and is transparent for the user. Note that we are limited here to $N = 4096$ in single precision and $N = 2048$ in double precision; also we use complex Fourier transform; optimized real transforms may permit to go even faster. The merge of two velocity time steps can also easily improve the speed. Higher order time splitting may be also used. Also, a better comparison with CPU parallelized codes can be envisaged (here, we used a basic OpenMP implementation which only scales for 2 processors). We can also hope to go to higher grids, since `cufft` should allow grid sizes of 128 millions elements in double precision and 64 millions in single precision (here we use $2^{24} \simeq 16.78 \cdot 10^6$ elements in single precision and $2^{22} \simeq 4.2 \cdot 10^6$ elements in double precision; so we should be able to run with $N = 8192$ in single precision and $N = 4096$ in double precision). Higher complexity problems (as $4D$ simulations) will probably need multi-gpu which is another story, see [13] for such a work.

## 5. Conclusion

We have shown that this approach works. Most of the load is carried by the FFT routine, which is optimized for CUDA in the `cufft` library, leading to huge speedups and is invisible to the user. Thus, the overhead of implementation time which can be quite significant in other contexts is here reduced, since we use largely built-in routines which are already optimized. The use of single precision is made harmless thanks to a $\delta f$ method. We however are not able to get as precise results as in the case of double precision. The test cases we chose are quite sensitive to single precision round off errors. We point out also that the electric field has to satisfy a zero mean condition with enough accuracy on a discrete grid. For the moment, we are limited to same sizes in $x$ and $v$ (needed here for the transposition step) and to $N = 2048$ in double precision ($N = 4096$ in single precision). We hope to implement a four dimensional $(2x, 2v)$ version of this code, next, including weak collisions.

| | | Single precision | | | Double precision | | |
|---|---|---|---|---|---|---|---|
| | $N_x$ | Time (ms) (**speedup**) | MA | FFT (cufftExec) | Time (ms) (**speedup**) | MA | FFT (cufftExec) |
| (1) | 256 | 703 (**2.8-8.5**) | 279.6 | 0.635 (0.36) | 1304 (**1.5-4.6**) | 150.7 | 0.767 (0.61) |
| | 512 | 1878 (**4.3-17**) | 418.7 | 0.759 (0.46) | 3516 (**2.3-8.8**) | 223.6 | 0.839 (0.67) |
| | 1024 | 6229 (**9.6-20**) | 505.0 | 0.841 (0.51) | 11670 (**5.1-11**) | 269.5 | 0.889 (0.71) |
| | 2048 | 21908 (**13-27**) | 574.3 | 0.861 (0.50) | 49925 (**5.7-12**) | 252.0 | 0.916 (0.75) |
| | 4096 | 90093 (**15-52**) | 558.6 | 0.888 (0.54) | - | - | - |
| (2) | 256 | 1096 [1378] (**1.8-5.5**) | 179.3 | 0.471 [0.59 (0.37)] | 1653 (**1.2-3.6**) | 118.9 | 0.637 (0.5) |
| | 512 | 2125 [2550] (**3.8-15**) | 370.0 | 0.654 [0.69 (0.48)] | 3896 (**2.1-8.0**) | 201.8 | 0.777 (0.66) |
| | 1024 | 5684 [6001] (**11-22**) | 553.4 | 0.775 [0.79 (0.59)] | 12127 (**4.9-10**) | 259.3 | 0.866 (0.76) |
| | 2048 | 19871 [20284] (**14-29**) | 633.2 | 0.825 (0.62) | 45753 (**6.3-13**) | 275.0 | 0.897 (0.80) |
| | 4096 | 81943 (**17-57**) | 614.2 | 0.859 (0.66) | - | - | - |
| (3) | 256 | 5783 (**0.3-1.0**) | 33.9 | 0.773 (0.65) | - | - | - |
| | 512 | 19936 (**0.4-1.6**) | 39.4 | 0.780 (0.66) | - | - | - |
| | 1024 | 87685 (**0.68-1.4**) | 35.8 | 0.813 (0.71) | - | - | - |

TABLE 1. Performance results for GPU, nbstep=1000, LAG17, KEEN wave test case with $\delta f$ modification: total time, speedup, MA, proportion FFT/total time (and cufftExec/total time).

| | | Single precision | | | | Double precision | | | |
|---|---|---|---|---|---|---|---|---|---|
| | $N_x$ | Time (ms) **speedup** | MA | GF | FFT | Time (ms) **speedup** | MA | GF | FFT |
| (1) | 256 | 570 (**3.5-11**) | 344.9 | 29.6 | 0.573 | 1183 (**1.7-5.1**) | 166.1 | 14.2 | 0.754 |
| | 512 | 1421 (**5.6-22**) | 553.4 | 53.1 | 0.702 | 3121 (**2.6-10**) | 251.9 | 24.1 | 0.826 |
| | 1024 | 4516 (**13-28**) | 696.5 | 73.8 | 0.787 | 10221 (**5.9-12**) | 307.7 | 32.6 | 0.876 |
| | 2048 | 15189 (**19-38**) | 828.4 | 96.0 | 0.802 | 44244 (**6.5-13**) | 284.3 | 32.9 | 0.906 |
| | 4096 | 63310 (**22-73**) | 795.0 | 100.1 | 0.842 | - | - | - | - |
| (2) | 256 | 1000 (**2.0-6.0**) | 196.6 | 16.9 | 0.520 | 1569 (**1.3-3.8**) | 125.3 | 10.7 | 0.657 |
| | 512 | 2000 (**4.0-15**) | 393.2 | 37.7 | 0.635 | 3750 (**2.1-8.3**) | 209.7 | 20.1 | 0.782 |
| | 1024 | 5067 (**12-25**) | 620.8 | 65.8 | 0.762 | 11749 (**5.1-11**) | 267.7 | 28.3 | 0.865 |
| | 2048 | 17692 (**16-33**) | 711.2 | 82.5 | 0.805 | 44446 (**6.5-13**) | 283.1 | 32.8 | 0.895 |
| | 4096 | 73488 (**19-63**) | 684.8 | 86.2 | 0.843 | - | - | - | - |
| (3) | 256 | 5513 (**0.36-1.1**) | 35.6 | 3.0 | 0.763 | - | - | - | - |
| | 512 | 18805 (**0.43-1.6**) | 41.8 | 4.0 | 0.769 | - | - | - | - |
| | 1024 | 83312 (**0.72-1.5**) | 37.7 | 4.0 | 0.804 | - | - | - | - |

TABLE 2. Performance results for GPU, nbstep=1000, LAG17, KEEN wave test case without $\delta f$ modification: total time, speedup, MA, GFlops and proportion FFT/total time.

| | (4) | | | (5) | | | (6) | | | (7) | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $N_x$ | Total time | MA | GF | Total time | MA | GF | Total time | MA | GF | Total time | MA | GF |
| 256 | 4s | 27.4 | 1.1 | 4s | 28.8 | 1.2 | 6s | 21.4 | 0.9 | 3s | 38.8 | 1.6 |
| 512 | 27s | 19.2 | 0.9 | 18s | 28.8 | 1.3 | 31s | 16.5 | 0.7 | 15s | 34.7 | 1.6 |
| 1024 | 1min52s | 18.7 | 0.9 | 2min4s | 16.8 | 0.8 | 2min7s | 16.4 | 0.8 | 1min18s | 26.7 | 1.4 |
| 2048 | 8min16s | 16.9 | 0.9 | 9min31s | 14.6 | 0.8 | 9min42s | 14.4 | 0.8 | 5min36s | 24.9 | 1.4 |
| 4096 | 41min05s | 13.6 | 0.8 | 48min16s | 11.5 | 0.7 | 52min20s | 10.6 | 0.6 | 28min28s | 19.6 | 1.2 |
| 256 | 3s | 58.0 | 2.4 | 4s | 43.9 | 1.8 | 3s | 54.3 | 2.3 | 2s | 72.6 | 3.1 |
| 512 | 19s | 39.6 | 1.9 | 8s | 90.6 | 4.3 | 22s | 35.0 | 1.6 | 13s | 58.7 | 2.8 |
| 1024 | 1min25s | 36.8 | 1.9 | 1min21s | 38.5 | 2.0 | 1min35s | 32.9 | 1.7 | 1min0s | 52.1 | 2.7 |
| 2048 | 6min41s | 31.3 | 1.8 | 7min46s | 27.0 | 1.5 | 8min47s | 28.3 | 1.6 | 4min47s | 43.7 | 2.5 |
| 4096 | 34min39s | 24.2 | 1.5 | 25min33s | 32.8 | 2.0 | 77min31s | 10.8 | 0.6 | 23min09s | 36.2 | 2.2 |

TABLE 3. Performance results for CPU vlaso code, nbstep=1000, LAG 17, Landau test case (top): total time, MA and GFlops. Performance results for CPU Selalib code, nbstep=1000, LAG 17, KEEN test case without $\delta f$ modification (bottom): total time, MA and GFlops.

# References


[1] B. Afeyan, K. Won, V. Savchenko, T. Johnston, A. Ghizzo, and P. Bertrand. *Kinetic Electrostatic Electron Nonlinear (KEEN) Waves and their Interactions Driven by the Ponderomotive Force of Crossing Laser Beams.*, Proc. IFSA 2003, 213, 2003, and arXiv:1210.8105, http://arxiv.org/abs/1210.8105.

[2] T. D. Arber, R. G. Vann, *A critical comparison of Eulerian-grid-based Vlasov solvers*, JCP, **180** (2002), pp. 339-357.

[3] N. Besse, M. Mehrenberger, *Convergence of classes of high-order semi-lagrangian schemes for the Vlasov-Poisson system,* Mathematics of Computation, 77, 93–123 (2008).

[4] C. K. Birdsall, A. B. Langdon, *Plasma Physics via Computer Simulation*, Adam Hilger, 1991.

[5] K. J. Bowers, B. J. Albright, B. Bergen, L. Yin, K. J. Barker, D. J. Kerbyson, 0.374 *pflop/s trillion-particle kinetic modeling of laser plasma interaction on roadrunner*, Proc. of Supercomputing. IEEE Press, 2008.

[6] J. P. Boris, D. L. Book, *Flux-corrected transport. I: SHASTA, a fluid transport algorithm that works*, J. Comput. Phys. **11** (1973), pp. 38-69.

[7] F. Charles, B. Després, M. Mehrenberger, *Enhanced convergence estimates for semi-lagrangian schemes Application to the Vlasov-Poisson equation,* SIAM J. Numer. Anal., 51(2), 840–863 (2013).

[8] Y. Cheng, I. M. Gamba, P. J. Morrison, *Study of conservation and recurrence of Runge-Kutta discontinuous Galerkin schemes for Vlasov-Poisson systems*,arXiv:1209.6413v2, 17 Dec 2012, http://arxiv.org/abs/1209.6413.

[9] C. Z. Cheng, G. Knorr, *The integration of the Vlasov equation in configuration space*, J. Comput. Phys. **22** (1976), pp. 330-3351.

[10] A. Crestetto, P. Helluy, *Resolution of the Vlasov-Maxwell system by PIC Discontinuous Galerkin method on GPU with OpenCL*, http://hal.archives-ouvertes.fr/hal-00731021

[11] N. Crouseilles, E. Faou, M. Mehrenberger, *High order Runge-Kutta-Nyström splitting methods for the Vlasov-Poisson equation*, inria-00633934, version 1, http://hal.inria.fr/IRMA/inria-00633934.

[12] N. Crouseilles, M. Mehrenberger, E. Sonnendrücker, *Conservative semi-Lagrangian schemes for Vlasov equations*, J. Comput. Phys. **229** (2010), pp. 1927-1953.

[13] T. Dannert, *GENE on Accelerators*, 4th Summer school on numerical modeling for fusion, 8-12 October 2012, IPP, Garching near Munich, Germany, http://www.ipp.mpg.de/ippcms/eng/for/veranstaltungen/konferenzen/su_school/.

[14] E. Fijalkow, *A numerical solution to the Vlasov equation*, Comput. Phys. Commun. **116** (1999), pp. 329-335.

[15] F. Filbet, E. Sonnendrücker, P. Bertrand, *Conservative numerical schemes for the Vlasov equation*, J. Comput. Phys. **172** (2001), pp. 166-187.

[16] F. Filbet, E. Sonnendrücker, *Comparison of Eulerian Vlasov solvers*, Comput. Phys. Comm. **151** (2003), pp. 247-266.

[17] F. Filbet *Numerical simulations avalaible online at* http://math.univ-lyon1.fr/~filbet/publication.html

[18] R.M. Gray, *Toeplitz and circulant matrices: a review*, Now Publishers Inc, Boston-Delft (2005).

[19] Y. Guclu, W. N. G. Hitchon, Szu-Yi Chen, *High order semi-lagrangian methods for the kinetic description of plasmas,* Plasma Science (ICOPS), 2012 Abstracts IEEE, vol., no., pp.5A-5, 8-13 July 2012, doi: 10.1109/PLASMA.2012.6383976.

[20] R. Hatzky, *Global electromagnetic gyrokinetic particle-in-cell simulation*, 4th Summer school on numerical modelling for fusion, 8-12 October 2012, IPP, Garching near Munich, Germany, http://www.ipp.mpg.de/ippcms/eng/for/veranstaltungen/konferenzen/su_school/.

[21] N.A. Krall, A.W. Trivelpiece, *Principles of Plasma Physics*, McGraw-Hill, New York (1973).

[22] G. Latu, *Fine-grained parallelization of Vlasov-Poisson application on GPU*, Euro-Par 2010, Parallel Processing Workshops, Springer (New York, 2011).

[23] C. Mouhot, C. Villani, *On Landau damping*, Acta Mathematica, volume 207, number 1, pages 29-201.

[24] J.M. Qiu, C. W. Shu, *Conservative semi-Lagrangian finite difference WENO formulations with applications to the Vlasov equation*, Comm. Comput. Phys. **10** (2011), pp 979-1000.

[25] T. M. Rocha Filho, *Solving the Vlasov equation for one-dimensional models with long range interactions on a GPU*, Comput. Phys. Comm. 184 Issue 1, pp. 34-39 (2013).

[26] *Selalib, a semi-Lagrangian library,* http://selalib.gforge.inria.fr/

[27] M. Shoucri, *Nonlinear evolution of the bump-on-tail instability*, Phys. Fluids **22** (1979), pp. 2038-2039.

[28] E. Sonnendrücker , N. Crouseilles , B. Afeyan, *BP8.00057: High Order Vlasov Solvers for the Simulation of KEEN Wave Including the L-B and F-P Collision Models*, 54th Annual Meeting of the APS Division of Plasma Physics Volume 57, Number 12, Monday-Friday, October 29–November 2 2012; Providence, Rhode Island, http://meeting.aps.org/Meeting/DPP12/SessionIndex2/?SessionEventID=181483.

[29] E. Sonnendrücker, *Approximation numérique des équations de Vlasov-Maxwell*, Master lectures, http://www-irma.u-strasbg.fr/~sonnen/polyM2VM2010.pdf.

[30] G. Stantchev, W. Dorland, N. Gumerov, *Fast parallel particle-to-grid interpolation for plasma PIC simulations on the GPU*, J. Parallel Distrib. Comput., **68**(10), pp. 1339-1349, (2008).

[31] T. Zhou, Y. Guo, C.W. Shu, *Numerical study on Landau damping*, Physica D 157 (2001), 322–333.

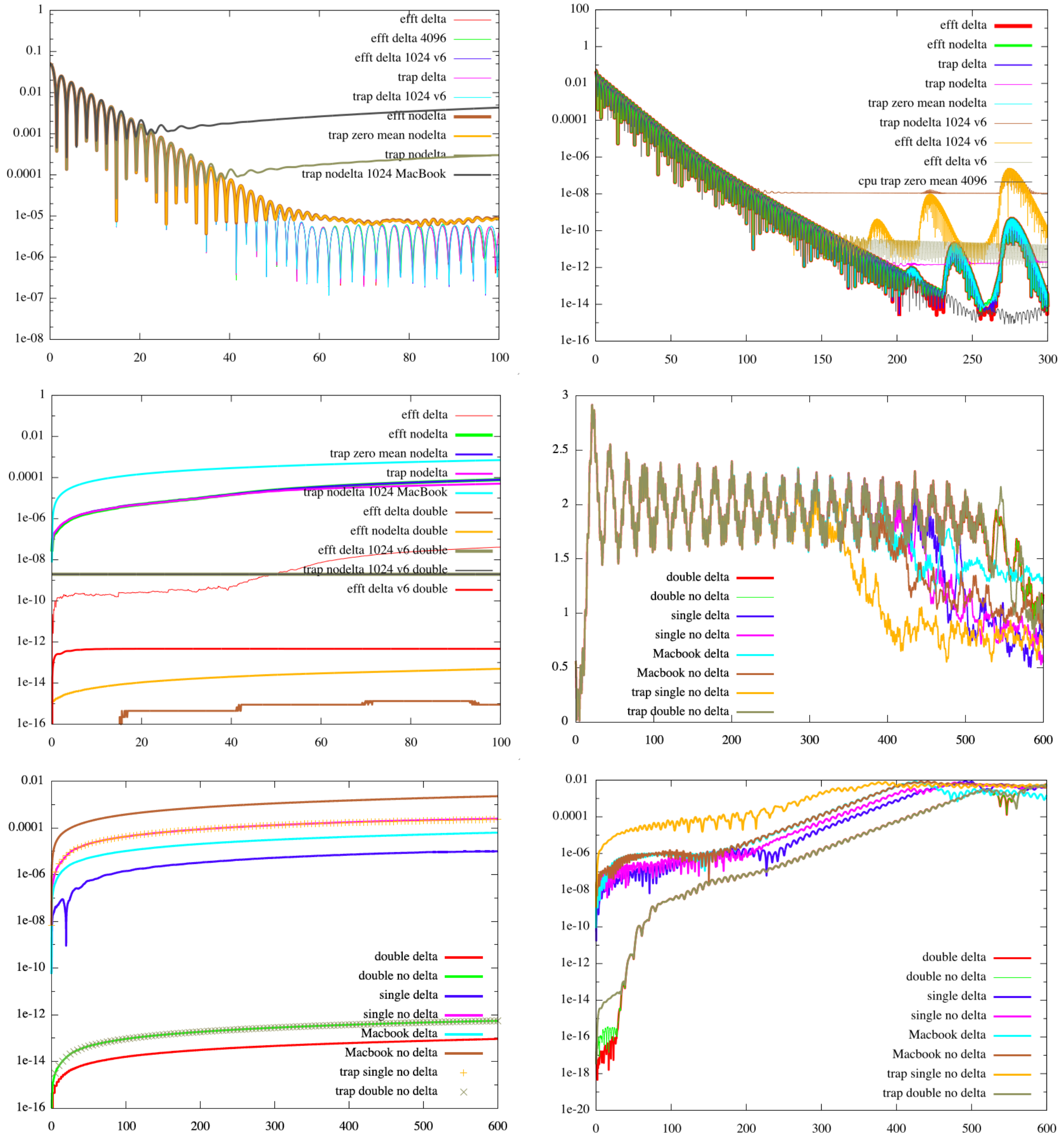



FIGURE 1. Linear Landau damping. $N = 2048$, $\Delta t = 0.1$, $v_{\max} = 8$, LAG17, irma-gpu1 on GPU as default. Evolution in time of electric energy in single/double precision (top left/right). Error of mass $|\hat{\rho}_0 - 1|$ with single precision as default (middle left). Bump on tail test case. $N = 1024, \Delta t = 0.05$, LAG9, irma-gpu1 on GPU as default. Evolution in time of the electric energy/ error of mass/ first Fourier mode of $\rho$, $|\hat{\rho}_1|$ (middle right/bottom left/bottom right). [ for details, see the legends. efft:electric field compute with FFT; delta= $\delta f$ method; no delta= without the $\delta f$ method; single=single precision; double:double precision; trap:electric field computed with trapezoidal method; zero mean:zero mean modification for the electric field; cpu: cpu code used; 1024: $N = 1024$; 4096: $N = 4096$; v6: $v_{\max} = 6$; Macbook: MacBook GPU is used].

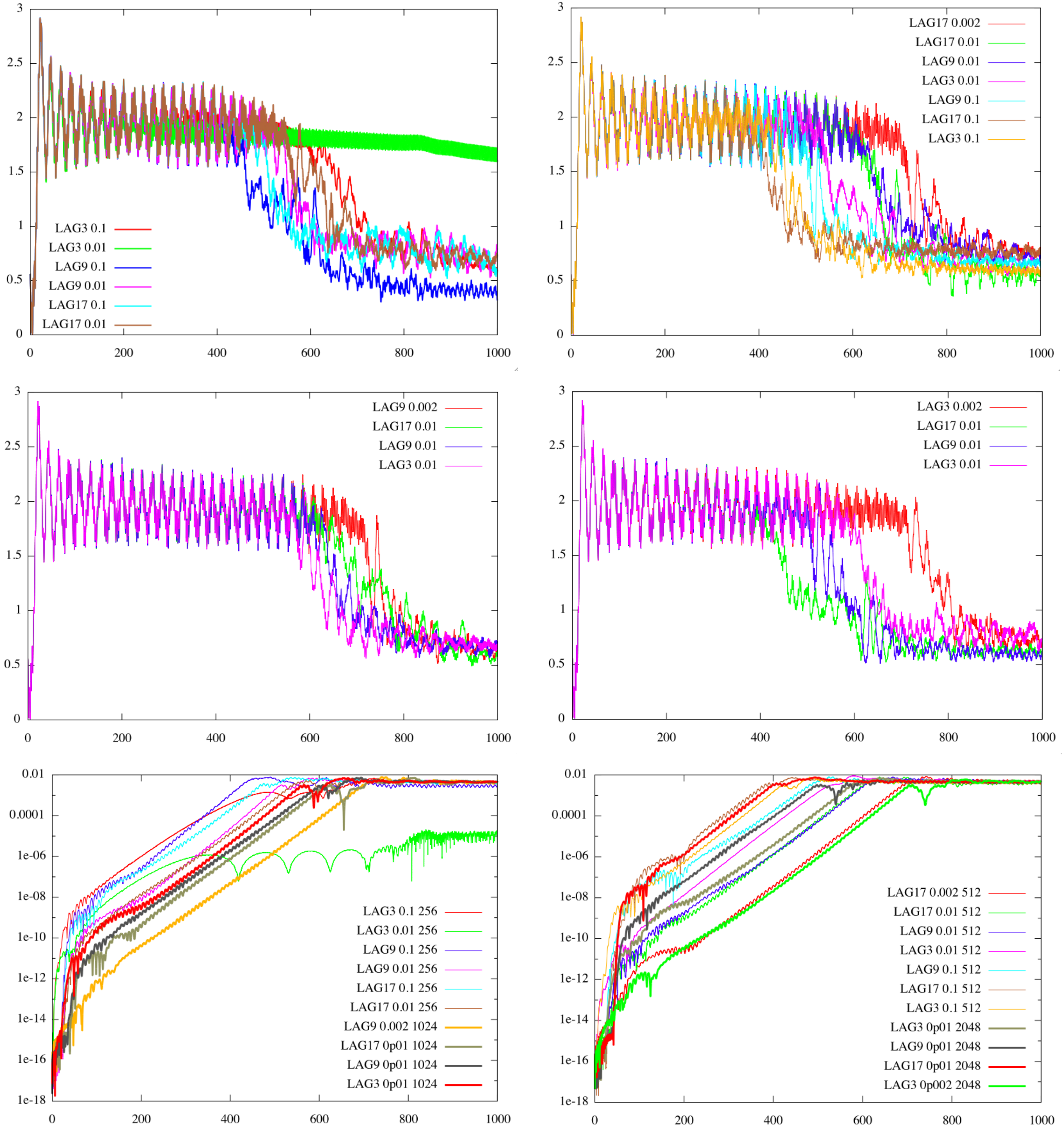



Figure 2. Bump on tail test case. Double precision is used, irma-gpu1 on GPU. Evolution in time of the electric energy for $N = 256, 512, 1024, 2048$ (top left, top right, middle left, middle right), with LAG3, LAG9, LAG17 recosntructions and various time steps $(0.1, 0.01, 0.002)$. Evolution in time of the first Fourier mode , $|\hat{\rho}_1|$ for $N = 256$ and $N = 1024$ (bottom left), and for $N = 512$ and $N = 2048$ (bottom left), with the same reconstructions and time steps.

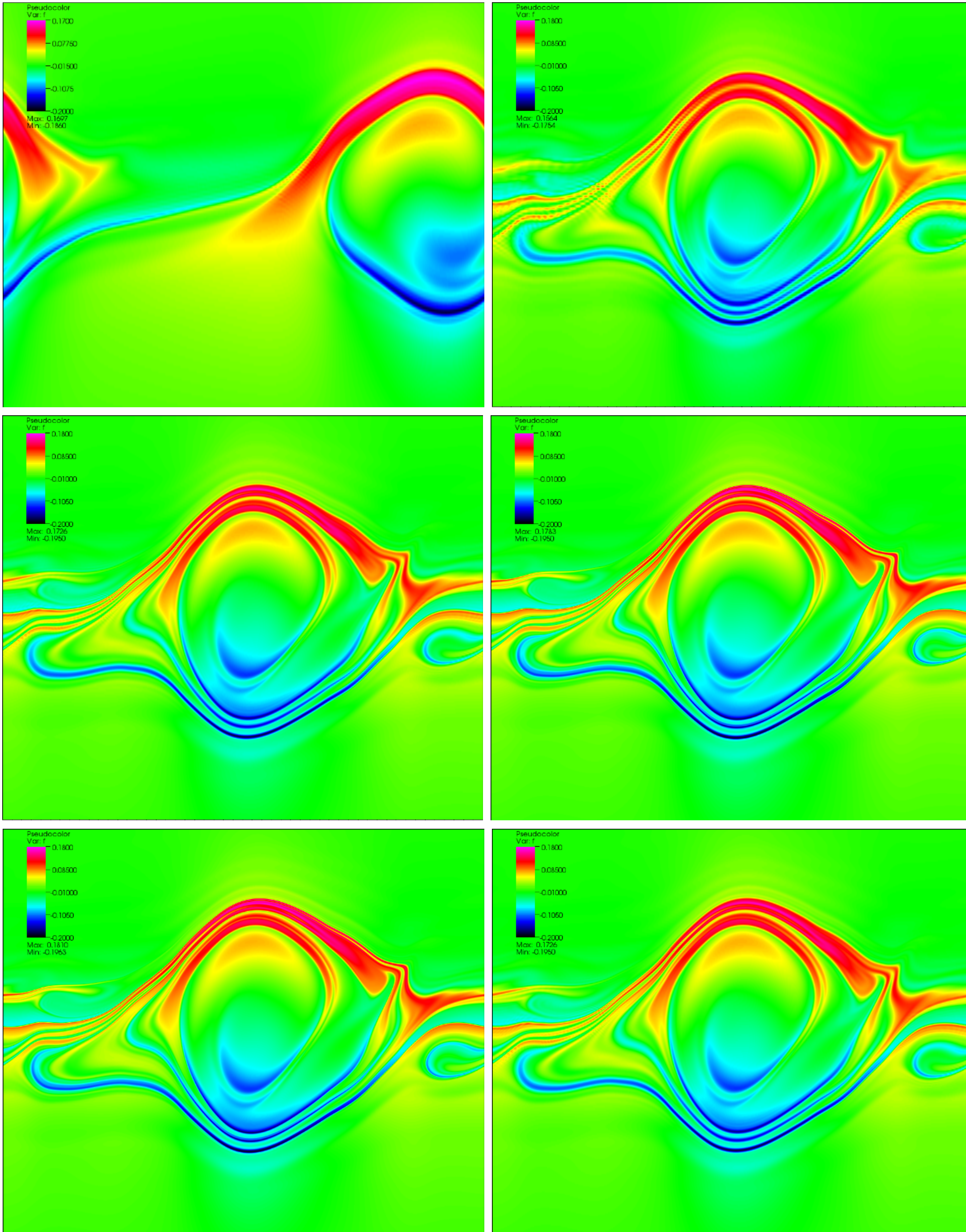


Figure 3. KEEN wave test case (LAG17): $f(t, x, v) - f_0(x, v)$. At time $t = 200$, GPU single precision $N = 1024$ (top left). At time $t = 300$, GPU single precision $N = 1024, 2048, 4096$ and $\Delta t = 0.1$ (top right, middle left, middle right). $N = 4096$ and $\Delta t = 0.01$ (bottom left). CPU $N = 2048, \Delta t = 0.1$ (bottom right). $(x, v) \in [0, 2\pi/k] \times [0.18, 4.14]$. If not changed, from one picture to another (from top left to bottom right), parameters are not restated.

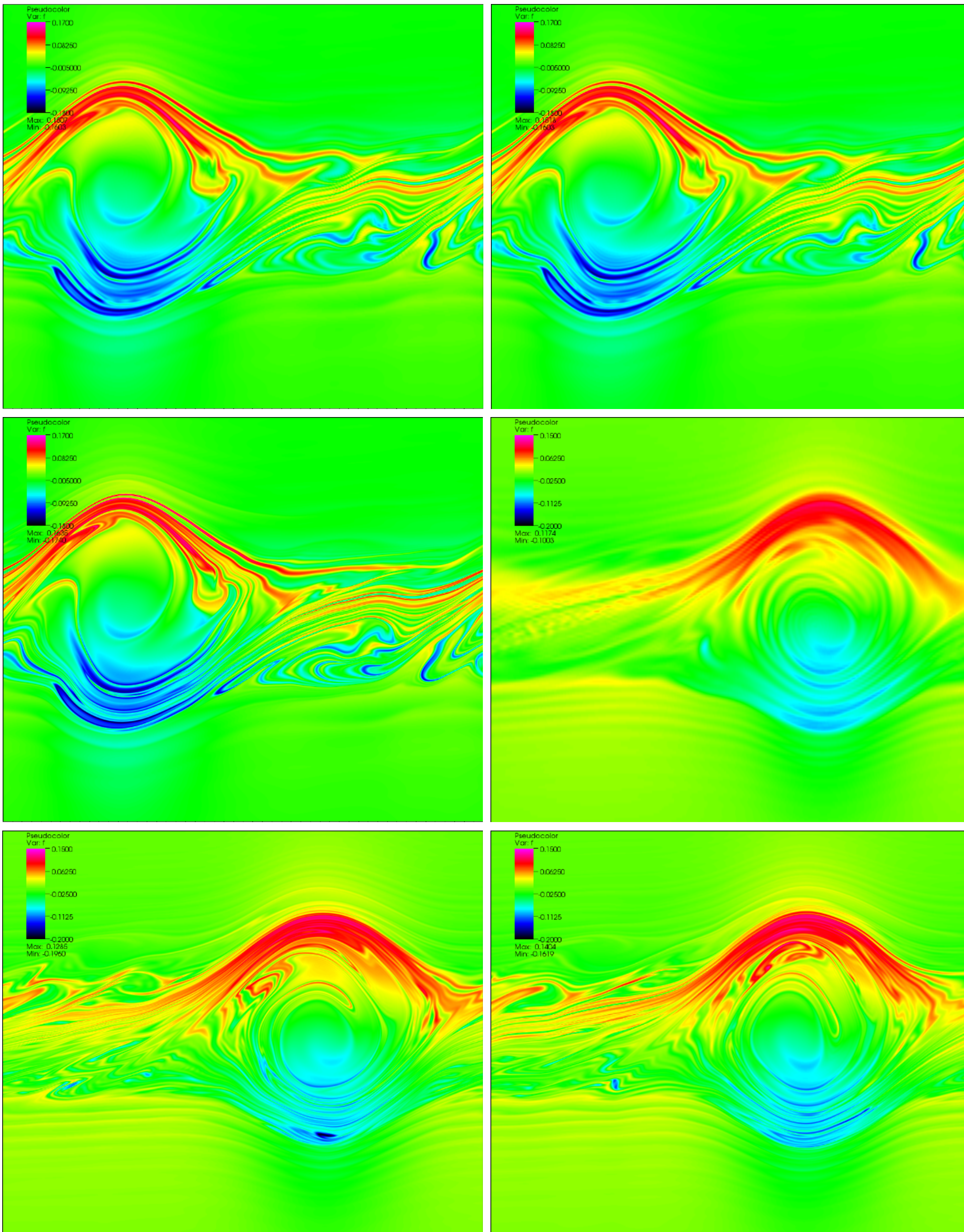


FIGURE 4. KEEN wave test case (LAG17): $f(t,x,v) - f_0(x,v)$. At time $t = 400$, GPU single precision $N = 2048, \Delta t = 0.1$ (top left). CPU $\Delta t = 0.05$ (top right). GPU single precision $N = 4096, \Delta t = 0.1$, at time $t = 600$ (middle left). GPU single precision $N = 1024$ (middle right). $\Delta t = 0.01, N = 4096$ (bottom left). CPU $N_x = 512, N_v = 4096$ (bottom right). $(x,v) \in [0, 2\pi/k] \times [0.18, 4.14]$. If not changed, from one picture to another (from top left to bottom right), parameters are not restated.

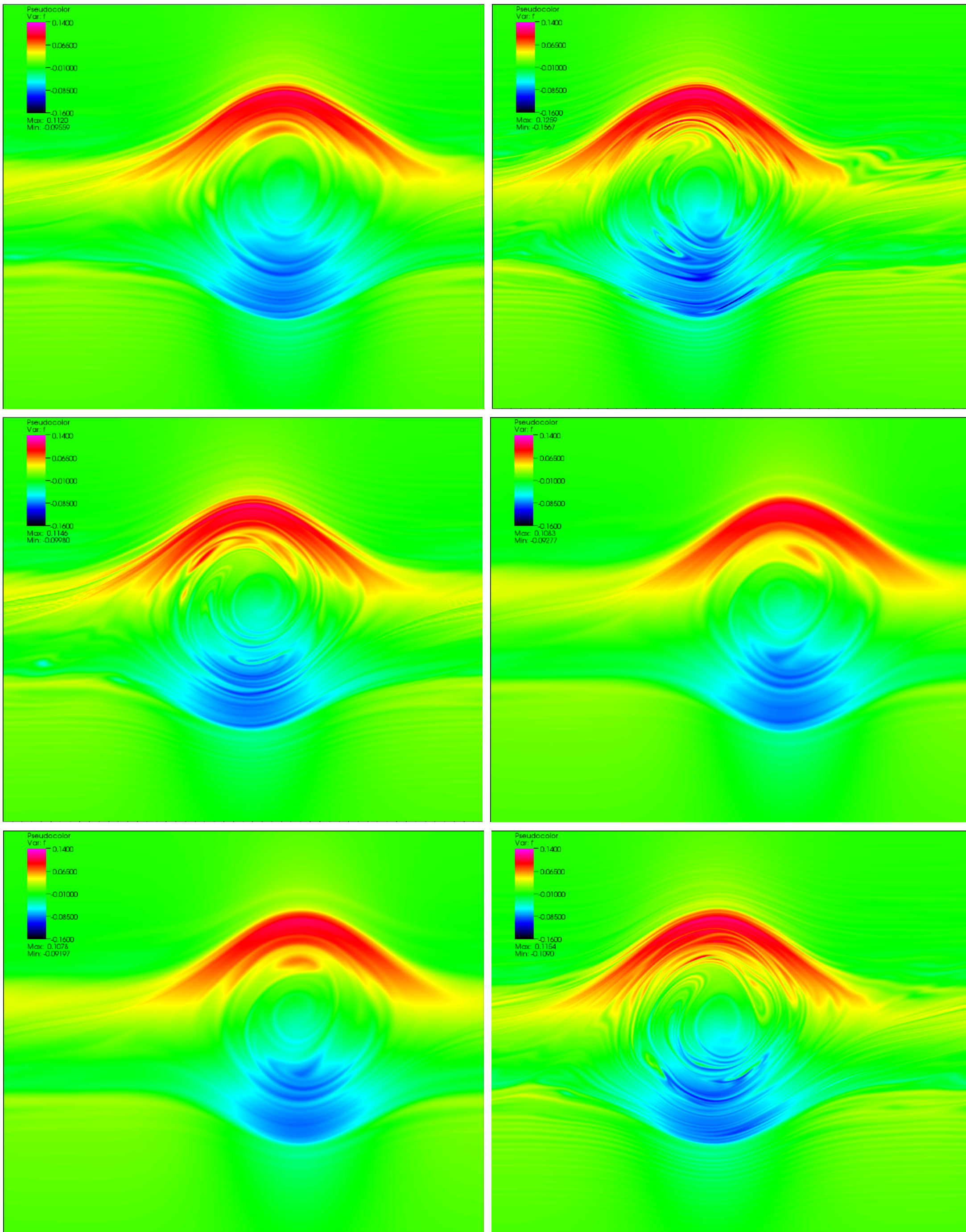


FIGURE 5. KEEN wave test case: $f(t,x,v) - f_0(x,v)$ at time $t = 1000$. CPU cubic splines, $\Delta t = 0.05, N_x = 512, N_v = 4096$ (top left). LAG17 (top right). $N = 4096$ and cubic splines (middle left). LAG3 (middle right). GPU single precision (top left). LAG9 (bottom right). $(x,v) \in [0, 2\pi/k] \times [0.18, 4.14]$. If not changed, from one picture to another (from top left to bottom right), parameters are not restated.

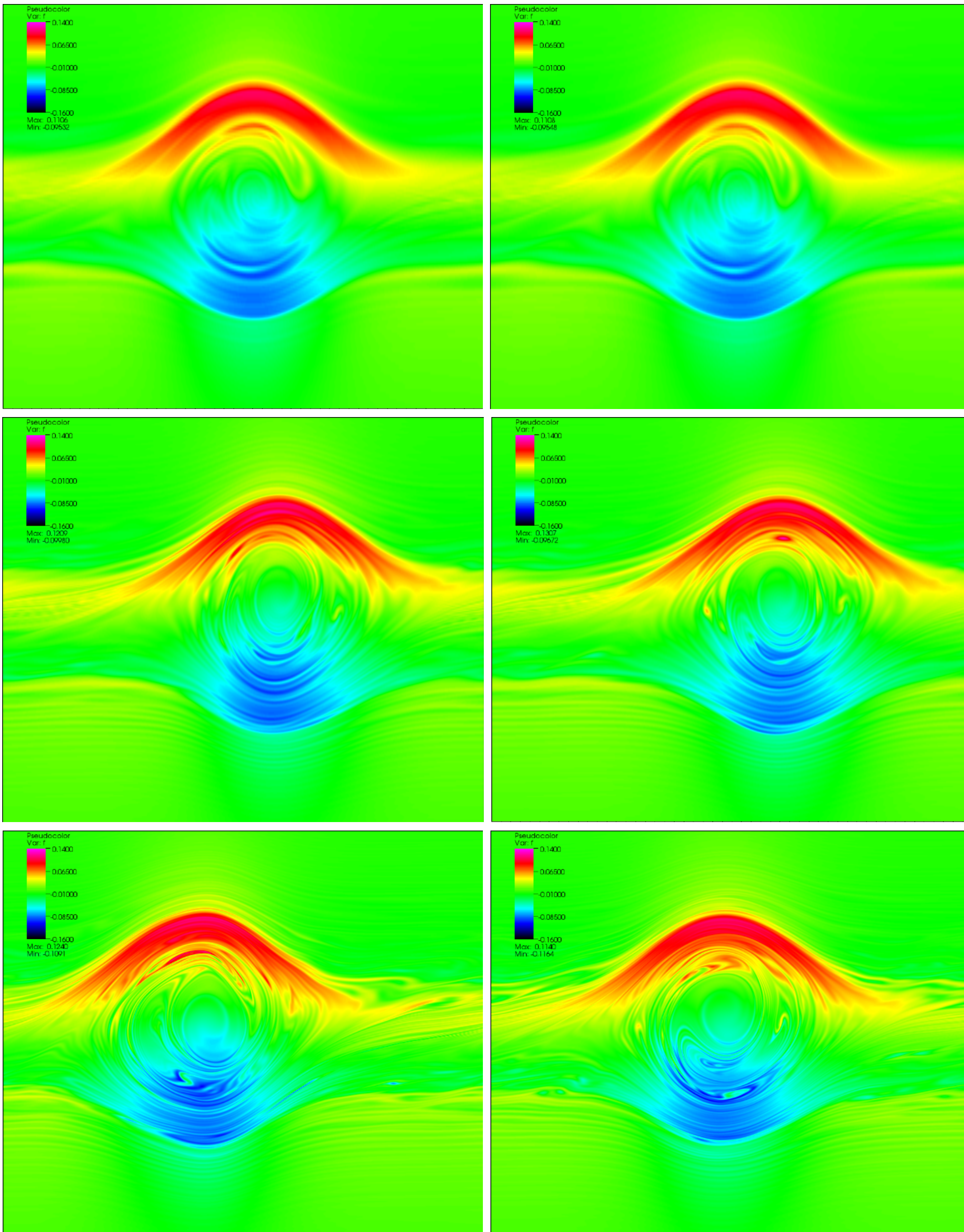



FIGURE 6. KEEN wave test case (LAG17): $f(t,x,v)-f_0(x,v)$ at time $t=1000$, $\Delta t=0.05$, $N=1024$ as default. GPU single/double precision (top left/right). $N=2048$, GPU single/double precision (middle left/right). $N=4096$, GPU single precision (bottom left). $\Delta t=0.01$ (bottom right). $(x,v)\in[0,2\pi/k]\times[0.18,4.14]$. If not changed, from one picture to another (from top left to bottom right), parameters are not restated.

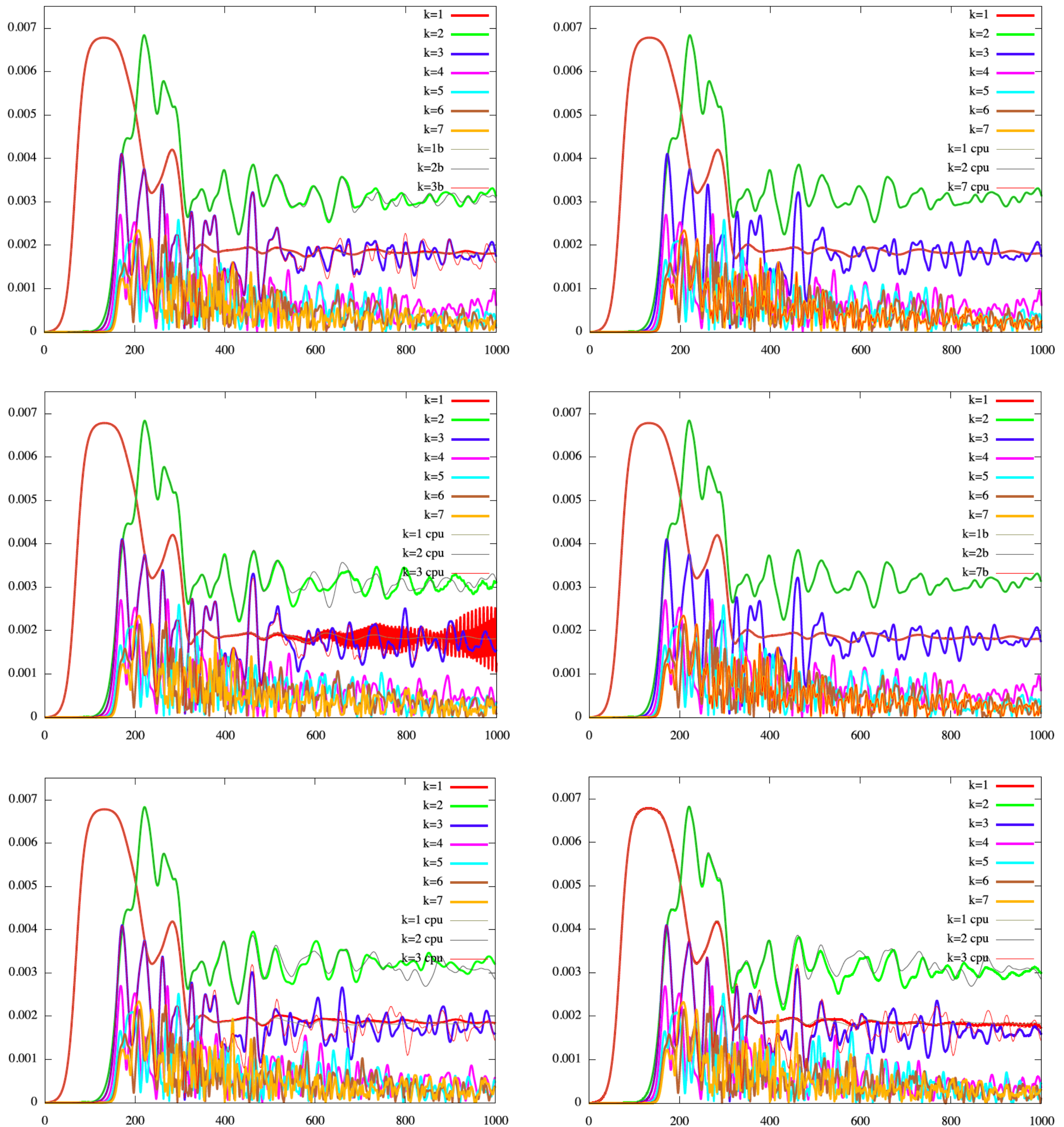



FIGURE 7. KEEN wave test case (LAG17): Absolute values of the first Fourier modes of $\rho$ (from mode $k = 1$ to mode $k = 7$) vs time. $\delta f$ method, with $N = 2048$ $\Delta t = 0.05$ GPU, double and single precision (1b,2b,3b) (top left). double GPU and double CPU (top right). Full version GPU in single precision and $\delta f$ version CPU (middle left). Full version and $\delta f$ version, in double precision (middle right). $N = 4096$, GPU and CPU (bottom left). GPU with $\Delta t = 0.01$ and CPU with $\Delta t = 0.05$ (bottom right). If not changed, from one picture to another (from top left to bottom right), parameters are not restated.