The Journal of Supercomputing
https://doi.org/10.1007/s11227-023-05189-0

Efficient GPU implementation of a Boltzmann-Schrödinger-Poisson solver for the simulation of nanoscale DG MOSFETs

Francesco Vecil[1] · José Miguel Mantas[2] · Pedro Alonso-Jordá[3]

Accepted: 10 March 2023

Abstract

A previous study by Mantas and Vecil (Int J High Perform Comput Appl 34(1): 81–102, 2019) describes an efficient and accurate solver for nanoscale DG MOSFETs through a deterministic Boltzmann-Schrödinger-Poisson model with seven electron–phonon scattering mechanisms on a hybrid parallel CPU/GPU platform. The transport computational phase, i.e. the time integration of the Boltzmann equations, was ported to the GPU using CUDA extensions, but the computation of the system's eigenstates, i.e. the solution of the Schrödinger-Poisson block, was parallelized only using OpenMP due to its complexity. This work fills the gap by describing a port to GPU for the solver of the Schrödinger-Poisson block. This new proposal implements on GPU a Scheduled Relaxation Jacobi method to solve the sparse linear systems which arise in the 2D Poisson equation. The 1D Schrödinger equation is solved on GPU by adapting a multi-section iteration and the Newton-Raphson algorithm to approximate the energy levels, and the Inverse Power Iterative Method is used to approximate the wave vectors. We want to stress that this solver for the Schrödinger-Poisson block can be thought as a module independent of the transport phase (Boltzmann) and can be used for solvers using different levels of description for the electrons; therefore, it is of particular interest because it can be adapted to other macroscopic, hence faster, solvers for confined devices exploited at industrial level.

Keywords Semiconductor physics · Deterministic mesoscopic models · Parallel heterogeneous systems · GPU computing · Schrödinger-Poisson system · Parallelization of numerical algorithms

✉ José Miguel Mantas
jmmantas@ugr.es

Extended author information available on the last page of the article

1 Introduction

This paper comes as a completion of the work described in [27], in which a deterministic and physically accurate solver for Double-Gate Metal Oxide Field-Effect Transistors (DG MOSFETs) was implemented on a high-performance platform in order to alleviate the computational weight of such a high-dimensional model. Nanoscale DG MOSFETs are a key element in modern integrated circuits, and their modeling and simulation aim at contributing to their downscaling following Moore's law. Figure 1 sketches the geometry and spatial dimensions of the particular 2D DG-MOSFET device.

The deterministic model consists of a set of collisional Boltzmann equations to describe electron transport inside the structure, and a 1D Schrödinger–2D Poisson block to compute the eigenstates, which read, in its dimensionless form (after a cartesian-to-ellipsoidal change of variables in the impulsion space) as:

$$\frac{\partial \Phi_{\nu,p}}{\partial t} + \frac{\partial}{\partial x}\left[a^1_\nu \Phi_{\nu,p}\right] + \frac{\partial}{\partial w}\left[a^2_{\nu,p} \Phi_{\nu,p}\right] + \frac{\partial}{\partial \phi}\left[a^3_{\nu,p} \Phi_{\nu,p}\right] = \mathcal{Q}_{\nu,p}[\Phi]\, s_\nu(w) \tag{1}$$

$$-\frac{1}{2}\frac{\mathrm{d}}{\mathrm{d}z}\left(\frac{1}{m_{z,\nu}}\frac{\mathrm{d}\psi_{\nu,p}}{\mathrm{d}z}\right) - \left(V + V_c\right)\psi_{\nu,p} = \epsilon_{\nu,p}\,\psi_{\nu,p} \tag{2}$$

$$-\nabla\cdot\left(\varepsilon_{\mathrm{R}}\nabla V\right) = -\left(N - N_D\right). \tag{3}$$

where $z \in [0,1]$ is the electron confinement dimension (transversal dimension) and $x \in [0,1]$ is the electron transport dimension (longitudinal dimension), $w \in [0,\infty[$ is a dimensionless energy, $\phi \in [0,2\pi[$ is the azimuthal angle, $\nu \in \{0,1,2\}$ indexes the valley (we consider three valleys in the silicon band structure) and $p \in \{0,\dots,5\}$ indexes the subband (energy level).

Here, $\Phi_{\nu,p}(t,x,w,\phi)$ is the probability of finding an electron of the ν^{th} valley, p^{th} subband, at time t, at position x, with energy-angle (w,ϕ) in the 2D impulsion space.

The presence of several valleys inside the Si band structure, plus the confinement due to the oxide layers make that we have as many Boltzmann Transport

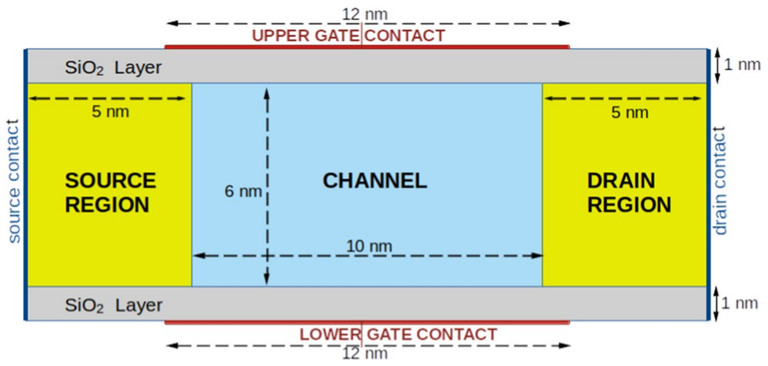

Fig. 1 Geometry and spatial dimensions of the nanoscale 2D DG-MOSFET

Equations (BTEs) (1) as (ν, p)-pairs; for each BTE, the electrons are advected through the fluxes given by

$$\begin{aligned}
a^1_\nu(w,\phi) &= \frac{\sqrt{2w(1+\alpha_\nu w)}\cos(\phi)}{\sqrt{m_{x,\nu}}(1+2\alpha_\nu w)},\\
a^2_{\nu,p}(x,w,\phi) &= -\frac{\partial \epsilon_{\nu,p}}{\partial x}(x)\, a^1_\nu(w,\phi)\\
a^3_{\nu,p}(x,w,\phi) &= \frac{\partial \epsilon_{\nu,p}}{\partial x}(x)\, \frac{1}{\sqrt{2w(1+\alpha_\nu w)}}\, \frac{\sin(\phi)}{\sqrt{m_{x,\nu}}}
\end{aligned}$$

where $\epsilon_{\nu,p}(x)$ are the energy levels, α_ν is the Kane's non-parabolicity factor for the ν^{th} valley, and $m_{x,\nu}$ is the electron effective mass along dimension x for the ν^{th} valley (see Appendix A for the details about α_ν and $m_{x,\nu}$).

The scattering operator $\mathcal{Q}_{\nu,p}[\Phi]$ describes the electron–phonon interactions and $s_\nu(w)$ is a given function due to the change of variables in the impulsion space. Refer [27, 38] for the details about these terms.

In the Schrödinger equations (2), which describe the confinement, $\psi_{\nu,p}(x,z)$ are the wave functions and $V(x, z)$ is the electrostatic potential. Additionally, $V_c(z)$ represents the MOSFET's confinement potential and $m_{z,\nu}$ is the electron effective mass along dimension z for the ν^{th} valley (see Appendix A for the details about $V_c(z)$ and $m_{z,\nu}$). Since dimension x acts only as a parameter, we have to solve as many eigenproblems as Si valleys times the discretization points along the x-dimension.

In the Poisson equation (3), the divergence and the gradient operators are meant for both the transport and the confinement dimensions (for (x, z)). The surface density $\varrho_{\nu,p}$ and the volume density N in (3) are given by:

$$\begin{aligned}
\varrho_{\nu,p}(x) &= \int_{w'=0}^{+\infty}\int_{\phi'=0}^{2\pi} \Phi_{\nu,p}(w',\phi')\,\mathrm{d}\phi'\,\mathrm{d}w',\\
N(x,z) &= \sum_{\nu,p} \varrho_{\nu,p}\cdot \left|\psi_{\nu,p}\right|^2.
\end{aligned}$$

In (3), ε_{R} represents the dielectric constant and $N_D(x,z)$ is the doping profile which takes into account the injected impurities in the semiconductor lattice (see Appendix A for the details about ε_{R} and $N_D(x,z)$).

The numerical solver described in [27] fully ports onto GPU the transport phase (called *BTE* phase) where the Boltzmann Transport Equations (BTEs) (1) are solved, while the goal of the present paper is to describe how we fully port onto GPU the phase corresponding to the solution of the Schrödinger-Poisson block (2)-(3) (called *iter* phase). We hence achieve a twofold improvement:

– to exploit the higher computational power of modern GPUs to accelerate this computational phase and

– to avoid definitively costly data transfer between the host and the device RAM in the heterogeneous platform.

In order to solve the Schrödinger-Poisson block (2)-(3), whose input is the surface densities $\varrho_{\nu,p}(x)$ and whose outputs are the energy levels $\epsilon_{\nu,p}(x)$, the wave functions $\psi_{\nu,p}(x,z)$ and the electrostatic potential V, a Newton-Raphson iterative algorithm is used, as was the case in the previous works (we address the reader to [27] and references therein for more details). An iteration in the Newton-Raphson algorithm consists of two main computational phases, which will be described separately in the following (see Fig. 2):

a) Updating of the guess for the potential V through a Poisson-like equation (unlike the Poisson equation (3) it contains an additional non-local term). The linear system deriving from the Poisson-like equation, and whose solution is the update for the guess on the potential V, is solved by means of a Scheduled Relaxation Jacobi (SRJ) scheme [2, 3, 39]: it consists of a sequence of relaxed Jacobi schemes with different relaxation factors, constructed in such a way to boost convergence to the solution.
b) Updating of the eigenstates $\{\epsilon_{\nu,p}(x)\}$ and $\{\psi_{\nu,p}(x,z)\}$ through the Schrödinger equation (2). The computation of the energy levels $\{\epsilon_{\nu,p}(x)\}$, i.e. the eigenvalues of the Schrödinger matrix, is achieved by using a multi-section algorithm [24] in the initial time step and a Newton-Raphson iterative algorithm in the following steps. Once the energy levels have been computed, the wave-vectors $\{\psi_{\nu,p}(x,z)\}$, which are the eigenvectors of the Schrödinger matrix, are computed by means of the Inverse Power Iterative Method (IPIM) [16], which in turn exploits the Thomas Algorithm [40] for the solution of the tridiagonal linear systems appearing at each iteration.

The parallel implementation of the numerical solution for the Schrödinger-Poisson block to simulate semiconductor devices has been tackled using different approaches and programming technologies. Initially, numerical solvers for shared-memory parallel architectures were derived using OpenMP [10]. In this way, an OpenMP implementation of a numerical solver for a drift-diffusion-Schrödinger-Poisson model is described in [33] and a 2D multi-subband ensemble Monte Carlo simulator of 2D MOSFET devices which solves the Poisson-Schrödinger block is described in [37]. Subsequently, versions of solvers of the Poisson-Schrödinger block for distributed-memory machines were obtained using the Message Passing Interface (MPI) to describe the interprocessor communication. Thus, the development of the nanoelectronics modeling tool NEMO5 [35] includes a Schrödinger-Poisson simulation and the parallelization of the simulations in NEMO5 is based on geometric partitioning techniques using MPI and several portable open-source packages. A parallel 1D Schrödinger-3D Poisson solver is implemented with a Gummel iterative method [17] using MPI and the PETSC library [5, 6] in [20]. In [22], a parallel implementation to simulate a metal-oxide-semiconductor (MOS) device, where a set of 1D Schrödinger-Poisson equations are solved, is described.

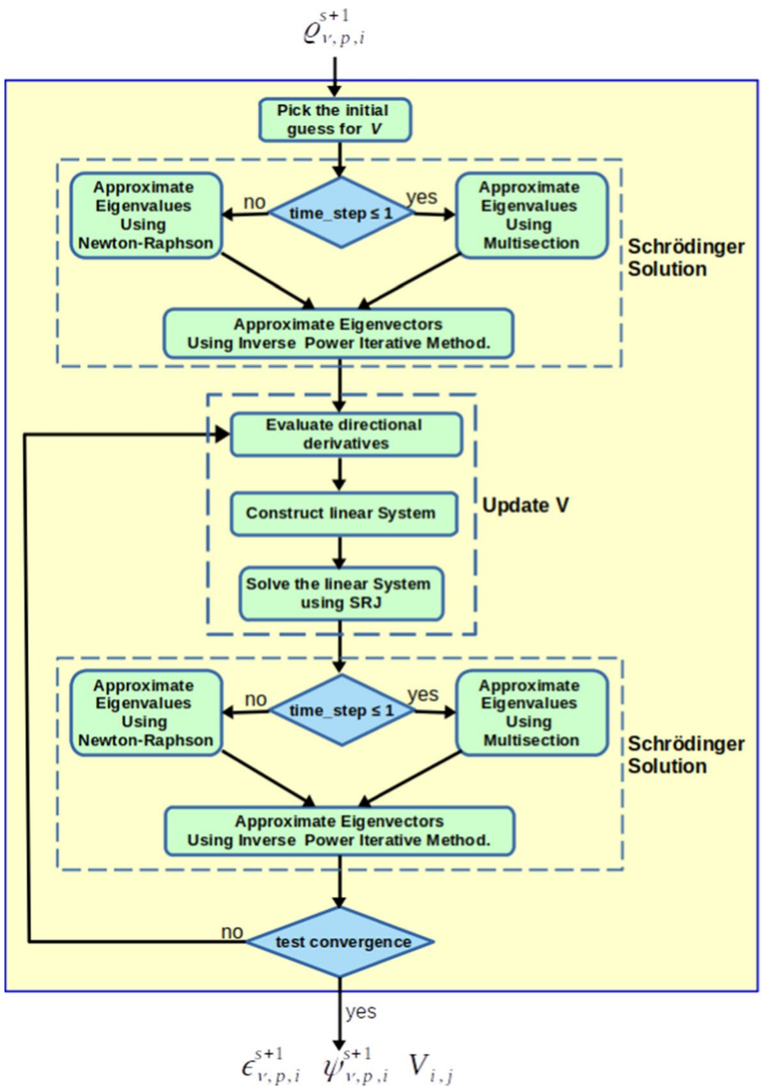
$\varrho^{s+1}_{\nu,p,i}$
Pick the initial guess for V
time_step ≤ 1
no
yes
Approximate Eigenvalues Using Newton-Raphson
Approximate Eigenvalues Using Multisection
Approximate Eigenvectors Using Inverse Power Iterative Method.
Schrödinger Solution
Evaluate directional derivatives
Construct linear System
Solve the linear System using SRJ
Update V
time_step ≤ 1
no
yes
Approximate Eigenvalues Using Newton-Raphson
Approximate Eigenvalues Using Multisection
Approximate Eigenvectors Using Inverse Power Iterative Method.
Schrödinger Solution
test convergence
no
yes
$\epsilon^{s+1}_{\nu,p,i}$ $\psi^{s+1}_{\nu,p,i}$ $V_{i,j}$

Fig. 2 Structure of the iterative solver for the Schrödinger-Poisson block. Two main phases appear: the update of the electrostatic potential *V*, and the diagonalization of the Schrödinger matrix to keep consistency with *V*

In this implementation, a parallel divide-and-conquer algorithm is developed to solve the Schrödinger equation while the Poisson equation is solved with a parallelization of a monotone iterative method. Additionally, an MPI implementation of a resolution scheme of 2D Schrödinger equation-based corrections compatible with an existing parallel drift-diffusion model was derived in [14] to simulate 3D semiconductor devices in the simulation framework VENDES [34].

The present work is of interest also for other kinds of solvers which also require the solution of the Schrödinger-Poisson block but using a less accurate description of the carriers in nanoscale semiconductors. The solver for the Schrödinger-Poisson equations, seen as a blackbox, receives as input the surface electron densities and returns as output the eigenstates, and in particular the force field that drives the electrons along the device thanks to the applied voltage. Therefore, this machinery and its efficient implementation on CUDA-enabled platforms, can be adapted to macroscopic models, that are in general preferred in industrial simulations because of their lower computational cost, like drift-diffusion solvers [13, 19, 29, 33], Monte Carlo solvers [12, 32, 37], solvers based on the maximum-entropy-principle energy transport model [8, 26] and Spherical Harmonics Expansion (SHE) solvers [21].

The paper is organized as follows: in Sect. 2, we summarize the model and the equations on which we focus; in Sect. 3, we describe the solvers and the strategy implemented to achieve a solution of the Poisson-like equation on GPU; in Sect. 4, we describe the solvers and the procedure employed to compute the eigenstates on GPU; in Sect. 5, we show the numerical results we have obtained on a dual processor server equipped with powerful modern GPUs; in Sect. 6, we draw some conclusions and sketch the future work in this promising research line.

2 The Schrödinger-Poisson solver

From an algorithmical point of view, the Schrödinger-Poisson block (2)-(3) receives as entry the surface densities $\{\varrho_{\nu,p}\}$ and returns as result the energy levels $\left\{\epsilon_{\nu,p}\right\}$, the wave vectors $\left\{\psi_{\nu,p}\right\}$ and the electrostatic potential V [7, 27, 38], such as it is shown in Fig. 2. In this figure, $\nu \in \{0, 1, 2\}$ denotes the valley, $p \in \{0, \dots, N_{\text{sbn}} - 1\}$ denotes the subband (we consider $N_{\text{sbn}} = 6$), $i = 0, \dots, N_x - 1$ denotes the index for a discretization point in the longitudinal dimension (x) of the physical 2D device, being N_x the number of discretization points in that dimension, $j = 0, \dots, N_z - 1$ represents an index for a discretization point in the transversal dimension (confined) of the device (N_z is the number of discretization points in that dimension) and s denotes the particular stage ($s = 0, 1, 2$) of the third-order Total-Variation Diminishing Runge–Kutta method [9] used for time integration.

From now on, we refer to the energy levels $\left\{\epsilon_{\nu,p}\right\}$ as the eigenvalues (of the Schrödinger matrix) and the wave vectors $\left\{\psi_{\nu,p}\right\}$ as the eigenvectors.

Equations (2)-(3) have to be seen as a block because:

– The 1D steady-state Schrödinger equation (2) takes as entry the potential $\{V_{i,j}\}$ and returns as many eigenvalues $\left\{\epsilon_{\nu,p}\right\}$ and corresponding eigenvectors $\left\{\psi_{\nu,p}\right\}$ as needed for the sake of precision, and this must be done for each fixed position x_i and each fixed band $\nu \in \{0, 1, 2\}$. As an example, in our solver, by using $N_x = 65$ and $N_{\text{sbn}} = 6$, this means that we have to compute 1170 eigenvalues and eigenvectors.

– The 2D Poisson equation (3) receives as input the eigenvectors $\{\psi_{v,p,i,j}\}$ and provides as output the potential $\{V_{i,j}\}$.

So, as can be seen, the output of (2) is the input of (3) and vice versa. In the following we describe the strategy to solve this block.

The idea is to restate (3) as seeking for the zero of functional

$$P[V] := -\nabla \cdot \left(\varepsilon_{\mathrm{R}} \nabla V\right) + \sum_{v,p} \varrho_{v,p}(x) \cdot \left|\psi_{v,p}\right|^2 - N_D \tag{4}$$

under the constraints of the Schrödinger equation (2) via a Newton-Raphson iterative scheme:

$$\begin{aligned} &V^{(0)} \text{ is given} \\ &P\left[V^{(k)}\right] + \mathrm{d}P\left(V^{(k)}, V^{(k+1)} - V^{(k)}\right) = 0 \qquad \text{for } k \geq 0. \end{aligned} \tag{5}$$

Obviously, stage $k+1$ is a refinement of the previous stage k. The derivative is meant in a directional sense (Fréchet derivative). Details of the computations can be found in [7].

The scheme is sketched in Fig. 2: starting from an initial guess, we refine the guess on the potential, and keep consistency with the eigenstates.

From a computational point of view, this means that we have to be prepared for an alternate solution of the Schrödinger eigenproblem (2) and the linear system (5). The strategies to deal with this process are described in the following.

2.1 Schrödinger diagonalization

We can rewrite the steady-state Schrödinger equation in terms of the V-dependent linear operator $\mathcal{L}$:

$$S[V](\Psi) = -\frac{1}{2}\frac{\mathrm{d}}{\mathrm{d}z}\left(\frac{1}{m_{z,v}}\frac{\mathrm{d}\Psi}{\mathrm{d}z}\right) - \left(V + V_c\right)\Psi =: \mathcal{L}(\Psi).$$

We wish to compute the first N_{sbn} eigenvalues and relative eigenvectors (we recall they will be equivalently referred to as *energy levels* and *wave functions*).

In order to do this, we take into account the uniform grid described in [27] for the spatial dimensions (x and z) and discretize the operator using finite differences. As a result, a symmetric tridiagonal matrix of order $\mathrm{n} := N_z - 2$ is obtained:

$$\mathcal{L}_{v,i} = \begin{pmatrix} d_0 & e_0 & & & & & \\ e_0 & d_1 & e_1 & & & & \\ & e_1 & d_2 & e_2 & & & \\ & & e_2 & d_3 & e_3 & & \\ & & & \ddots & \ddots & \ddots & \\ & & & & e_{\mathrm{n}-3} & d_{\mathrm{n}-2} & e_{\mathrm{n}-2} \\ & & & & & e_{\mathrm{n}-2} & d_{\mathrm{n}-1} \end{pmatrix} \tag{6}$$

being: for $j = 1, \dots, N_z - 2$

$$d_j := \left(\frac{\frac{1/4}{m_{z,\nu,i,j-1}} + \frac{1/2}{m_{z,\nu,i,j}} + \frac{1/4}{m_{z,\nu,i,j+1}}}{\Delta z^2} - V_{i,j} \right)$$

the elements in the diagonal, and for $j = 1, N_z - 3$

$$e_j := \left(-\frac{\frac{1/4}{m_{z,\nu,i,j}} + \frac{1/4}{m_{z,\nu,i,j+1}}}{\Delta z^2} \right)$$

the elements in the sub-diagonal (and the super-diagonal).

The values of the effective masses $m_{z,\nu}$, for the particular case of the DG MOSFET device, depend on the material:

$$m_{z,\nu,i,j} = \begin{cases} 0.5 & \text{if } (i,j) \text{ is in the SiO}_2 \text{ region} \\ 0.19 & \text{if } \nu < 2 \text{ and } (i,j) \text{ is in the Si region} \\ 0.98 & \text{if } \nu = 2 \text{ and } (i,j) \text{ is in the Si region.} \end{cases} \tag{7}$$

From this matrix we extract by some method the first (lowest) N_{sbn} eigenvalues $\left\{\epsilon_{\nu,p,i}\right\}_{p\in\{0,\dots,N_{\text{sbn}}-1\}}$ and relative eigenvectors $\left\{\psi_{\nu,p,i,j}\right\}_{(p,j)\in\{0,\dots,N_{\text{sbn}}-1\}\times\{0,\dots,N_z-1\}}$.

We take into account the boundary condition

$$\psi_{\nu,p,i,0} = \psi_{\nu,p,i,N_z-1} = 0$$

and the normalization of the eigenvectors

$$\left(\psi_{\nu,p,i,j} \longleftarrow \frac{\psi_{\nu,p,i,j}}{\sqrt{\Delta z \sum_{j'=1}^{N_z-2} \left|\psi_{\nu,p,i,j'}\right|^2}} \right)_{j=1,\dots,N_z-2}.$$

2.2 Evaluation of the directional derivative and construction of the linear system

One stage of the Newton-Raphson scheme on (4) translates into solving (5). (More details about the derivation can be found in [7].) This scheme boils down to the linear system on $V^{(k+1)}$

$$L^{(k)}\, V^{(k+1)} = R^{(k)}, \tag{8}$$

where

$$\begin{aligned} L^{(k)}\,V^{(k+1)} &= -\operatorname{div}\left[\varepsilon_{\mathrm{R}}\,\nabla V^{(k+1)}\right] + \int \mathcal{A}^{(k)}(x,z,\zeta)\,V^{(k+1)}(x,\zeta)\,\mathrm{d}\zeta \\ R^{(k)} &= -N^{(k)}(x,z) + \int \mathcal{A}^{(k)}(x,z,\zeta)\,V^{(k)}(x,\zeta)\,\mathrm{d}\zeta, \end{aligned} \tag{9}$$

being $\mathcal{A}^{(k)}(x,z,\zeta) := \mathcal{A}[V^{(k)}](x,z,\zeta)$ basically the directional derivative of the density $N^{(k)} := N[V^{(k)}]$ [15].

2.2.1 Evaluation of the directional derivative (Fréchet derivative)

The evaluation of $\mathcal{A}^{(k)}(x,z,\zeta)$ at the grid points reads:

$$\mathcal{A}^{(k)}_{i,j,j'} = 2\sum_{\nu,p}\sum_{p'\neq p}\frac{\varrho^{s+1}_{\nu,p,i}-\varrho^{s+1}_{\nu,p',i}}{\epsilon^{(k)}_{\nu,p',i}-\epsilon^{(k)}_{\nu,p,i}}\times\psi^{(k)}_{\nu,p,i,j'}\,\psi^{(k)}_{\nu,p',i,j'}\,\psi^{(k)}_{\nu,p',i,j}\,\psi^{(k)}_{\nu,p,i,j}. \tag{10}$$

We recall that, here, the surface densities $\{\varrho^{s+1}_{\nu,p,i}\}$ are the entry for the whole Schrödinger-Poisson block, seen as a blackbox, where s indexes the external Runge–Kutta stage governed by the time integrator, while index k refers to the Newton-Raphson stage.

2.2.2 Construction of the linear system

The Laplacian in the linear operator (9) reads

$$\operatorname{div}\left[\varepsilon_{\mathrm{R}}\,\nabla V^{(k+1)}\right] = \frac{\partial}{\partial x}\left(\varepsilon_{\mathrm{R}}\,\frac{\partial V^{(k+1)}}{\partial x}\right) + \frac{\partial}{\partial z}\left(\varepsilon_{\mathrm{R}}\,\frac{\partial V^{(k+1)}}{\partial z}\right)$$

and is discretized using the following finite-difference approximation:

$$\begin{aligned} \left(\operatorname{div}\left[\varepsilon_{\mathrm{R}}\nabla V^{(k+1)}\right]\right)_{i,j} =& \left(\frac{\frac{1}{2}(\varepsilon_R)_{i-1,j}+\frac{1}{2}(\varepsilon_R)_{i,j}}{x^2}\right)V^{(k+1)}_{i-1,j} + \left(\frac{\frac{1}{2}(\varepsilon_R)_{i,j-1}+\frac{1}{2}(\varepsilon_R)_{i,j}}{z^2}\right)V^{(k+1)}_{i,j-1} \\ &- \left(\frac{\frac{1}{2}(\varepsilon_R)_{i-1,j}+(\varepsilon_R)_{i,j}+\frac{1}{2}(\varepsilon_R)_{i+1,j}}{x^2} + \frac{\frac{1}{2}(\varepsilon_R)_{i,j-1}+(\varepsilon_R)_{i,j}+\frac{1}{2}(\varepsilon_R)_{i,j+1}}{z^2}\right)V^{(k+1)}_{i,j} \\ &+ \left(\frac{\frac{1}{2}(\varepsilon_R)_{i,j}+\frac{1}{2}(\varepsilon_R)_{i,j+1}}{z^2}\right)V^{(k+1)}_{i,j+1} + \left(\frac{\frac{1}{2}(\varepsilon_R)_{i,j}+\frac{1}{2}(\varepsilon_R)_{i+1,j}}{x^2}\right)V^{(k+1)}_{i+1,j}. \end{aligned} \tag{11}$$

The integral is discretized by means of trapezoid rule

$$\left(\int \mathcal{A}^{(k)}(x,z,\zeta)\,V^{(k+1)}(x,\zeta)\,\mathrm{d}\zeta\right)_{i,j} = \frac{\Delta z}{2}\left[\sum_{j'=0}^{N_z-2}\mathcal{A}^{(k)}_{i,j,j'}\,V^{(k+1)}_{i,j'} + \sum_{j'=1}^{N_z-1}\mathcal{A}^{(k)}_{i,j,j'}\,V^{(k+1)}_{i,j'}\right]. \tag{12}$$

For the right hand side $R^{(k)}$, the integral is computed in a similar way to (12), and the density is simply

$$N_{i,j}^{(k)} = 2 \sum_{\nu,p} \sum_{p' \neq p} \varrho_{\nu,p,i}^{s+1} \left| \psi_{\nu,p,i,j}^{(k)} \right|^2 .$$

As for the boundary conditions, Dirichlet is imposed at metallic contacts (source, drain and the two gates), while homogeneous Neumann is taken elsewhere.

As a remark, these Dirichlet conditions at the source and drain contacts represent the potential applied through the device, and the Dirichlet conditions applied at the gates represent the control on the opening and closing of the channel, thus switching the device between the *on* and the *off* phases.

3 Highly-parallel methods for the linear system

The matrix $L^{(k)}$ representing the linear system (8) is of order $N_x \times N_z$, and contains N_x square blocks of size N_z on the diagonal.

An approach to solve this linear system is to employ strategies to significantly accelerate the convergence of Jacobi method without losing its simplicity and locality [2, 30, 39]. Following this approach, in this work, we have implemented on GPU a Scheduled Relaxation Jacobi (SRJ) [2, 39] method to solve efficiently this type of systems. SRJ methods extend the Jacobi method for linear systems which result from elliptic PDEs and present several important advantages for our particular case:

– they exhibit excellent convergence behaviour while preserving the simplicity and the straightforward parallelization of Jacobi method,
– they are particularly suitable for linear systems which result from discretizing Poisson-like PDEs and
– they do not require advanced preconditioning (we can use inverse diagonal as in the Jacobi iteration).

An alternative approach would be to use Krylov subspace iterative methods such as the conjugate gradient (CG) and Generalized Minimal Residual (GMRES) methods [31]. However, these methods have a more complex implementation than the Jacobi method, and require the use of effective preconditioners to ensure fast convergence, where the preconditioners usually increase notably the computational cost and may involve significant effort for parallelization. Moreover, in [30] it is shown that approaches based on accelerating the Jacobi iteration can be an efficient alternative to the Krylov subspace methods.

3.1 The Scheduled Relaxation Jacobi (SRJ) method

The Jacobi method for the solution of a linear system provides poor convergence rate but exhibits a high concurrency degree, as each value of the vector solution can be updated totally independently from all the other values of the vector solution.

Suppose, we have to solve the system $\boldsymbol{Au} = \boldsymbol{b}$, where $\boldsymbol{A} = \left(a_{ij}\right)_{N\times N}$ $(i = 0, ..., N-1, j = 0, ..., N-1)$ is a square matrix of order N, $\boldsymbol{b}$ a vector of size N and $\boldsymbol{D}$ is the diagonal component of $\boldsymbol{A}$ ($\boldsymbol{D} = \mathrm{diag}\left(a_{00}, a_{11}, \dots, a_{N-1N-1}\right)$).

A classical Jacobi iteration can be rewritten in vector form [1] in order to exploit the matrix–vector product operation:

$$\begin{aligned}&\text{take } \boldsymbol{u} \text{ as initial guess}\\ &\text{repeat } \boldsymbol{u} \longleftarrow \boldsymbol{u} + \boldsymbol{D}^{-1}(\boldsymbol{b} - \boldsymbol{A}\,\boldsymbol{u}) \text{ until convergence.}\end{aligned}$$

A significant acceleration of the Jacobi algorithm can be obtained by applying the Scheduled Relaxation Jacobi (SRJ) method. The SRJ method extends the classical Jacobi method by introducing P different relaxation factors $\omega_i > 0, i = 1, \dots, P$. In the SRJ method, one relaxed Jacobi step with parameter ω_i has the following form:

$$\boldsymbol{u} \longleftarrow \boldsymbol{u} + \omega_i \boldsymbol{D}^{-1}(\boldsymbol{b} - \boldsymbol{A}\,\boldsymbol{u}). \tag{13}$$

In SRJ, we complete several cycles until reaching convergence. At each cycle, we perform M relaxed Jacobi steps as (13) where

$$M = \sum_{i=1}^{P} q_i,$$

being q_i the number of times we apply the parameter ω_i.

Therefore, a SRJ cycle consists in defining sequences of M relaxed Jacobi steps. In our experiments, we have obtained good results with $P = 7$ cycles with $M = 93$, using the following relaxation parameters:

$$\begin{aligned}&(\omega_1, q_1) = (370.035, 1) \quad (\omega_2, q_2) = (167.331, 2)\\ &(\omega_3, q_3) = (51.1952, 3) \quad (\omega_4, q_4) = (13.9321, 7)\\ &(\omega_5, q_5) = (3.80777, 13) \quad (\omega_6, q_6) = (1.18727, 26) \quad (\omega_7, q_7) = (0.556551, 41).\end{aligned}$$

In [2], one can obtain optimal parameters for $\omega_i, i = 1, \dots, P$ for several values of both the number of steps P and the number of grid points (taking into account a discretization using 2nd-order central differences of a 2D Laplace equation on a uniform grid). In particular, we have used the values for the case $P = 7$ steps and $N = 32$ points (N must be less than $max(N_x, N_z)$), for which we have experimentally obtained very good convergence results. The parameters $q_i, i = 1, \dots, P$ and M are easily inferred from the parameters $\beta_i, i = 1, \dots, P$ (also shown in [2]) describing the proportion of iterations in which a given weight ω_i is applied over the total number of iterations of each cycle.

Algorithm 1 Implementation of the SRJ method.

```
Input: Narrow-banded sparse matrix $\boldsymbol{A}$, vector $\boldsymbol{b}$, $tolerance$
Output: $\boldsymbol{u_0}$ such that $\frac{|\boldsymbol{b}-\boldsymbol{A}\boldsymbol{u_0}|_\infty}{|\boldsymbol{u_0}|_\infty} \leq tolerance$
 1: Declare arrays of N doubles: $\boldsymbol{u_0}$, $\boldsymbol{u_1}$ and $\boldsymbol{x}$
 2: Declare double: $res = 10^{36}$
 3: Initialize $\boldsymbol{u_0}$ with the last known value for the potential vector $V$
 4: while $res > tolerance$ do
 5:     for $k = 1, \ldots, M$ do
 6:         Select the next index $i \in 1, \ldots, P$
 7:         Launch CUDA kernel to compute $\boldsymbol{x} = \boldsymbol{A} \cdot \boldsymbol{u_0}$
 8:         Launch CUDA kernel to compute $\boldsymbol{x} = \boldsymbol{b} - \boldsymbol{x}$ and $\boldsymbol{u_1} = \boldsymbol{u_0} + \omega_i \boldsymbol{x}$
 9:         Swap $\boldsymbol{u_1}$ and $\boldsymbol{u_0}$
10:     end for
11:     Launch CUDA kernel to compute $|\boldsymbol{x}|_\infty$ and $|\boldsymbol{u_0}|_\infty$ for each CUDA block
12:     Copy partial result vectors for $|\boldsymbol{x}|_\infty$ and $|\boldsymbol{u_0}|_\infty$ from device to host
13:     $res = \frac{|\boldsymbol{x}|_\infty}{|\boldsymbol{u_0}|_\infty}$
14: end while
```

3.2 Implementation details

Algorithm 1 describes the CPU-GPU implementation of the SRJ method. As initial value for vector $\boldsymbol{u_0}$, we use the last known value for the potential vector V (obtained in the previous Newton-Raphson iteration or in the previous Runge-Kutta stage).

The selection of the next ω_i in a SRJ cycle does not follow the natural sequence of ascending order where ω_1 is applied q_1 times, then ω_2 is applied and so on, but the over-relaxation Jacobi steps (with $\omega_i > 1$) are evenly spaced over the SRJ cycle to avoid overflow in the numerical experiments (see [39] for more details).

To implement each SRJ step (13) in CUDA we need, among others, a CUDA kernel to perform the sparse matrix–vector product

$$\boldsymbol{x} = \boldsymbol{A} \cdot \boldsymbol{u_0}. \tag{14}$$

This CUDA kernel uses one-dimensional CUDA blocks and takes into account the narrow-banded structure of the sparse matrix $\boldsymbol{A}$. In this kernel, the computation of the i-th element of the vector $\boldsymbol{x}$ (by performing the dot product of the i-th row of the sparse matrix $\boldsymbol{A}$ by the vector $\boldsymbol{u_0}$) is computed by a different CUDA warp (see Fig. 3). We store the matrix $\boldsymbol{A}$ in global memory as a rectangular array whose row dimension is equal to the bandwidth of $\boldsymbol{A}$. We use one-dimensional CUDA blocks where each CUDA block computes $\frac{B}{32}$ elements of $\boldsymbol{x}$, being B the block size. Initially, all the warps in a CUDA block cooperate to read, in a coalescent way, the required values of $\boldsymbol{u_0}$ and load them in a shared-memory array ***s_u***. Then, the j-th warp in the k-th CUDA block read the corresponding non-zero values in the row $t = \frac{kB}{32} + j$ of $\boldsymbol{A}$ and the affected values ***s_u*** in order to compute the t-th element of $\boldsymbol{x}$. For this, each

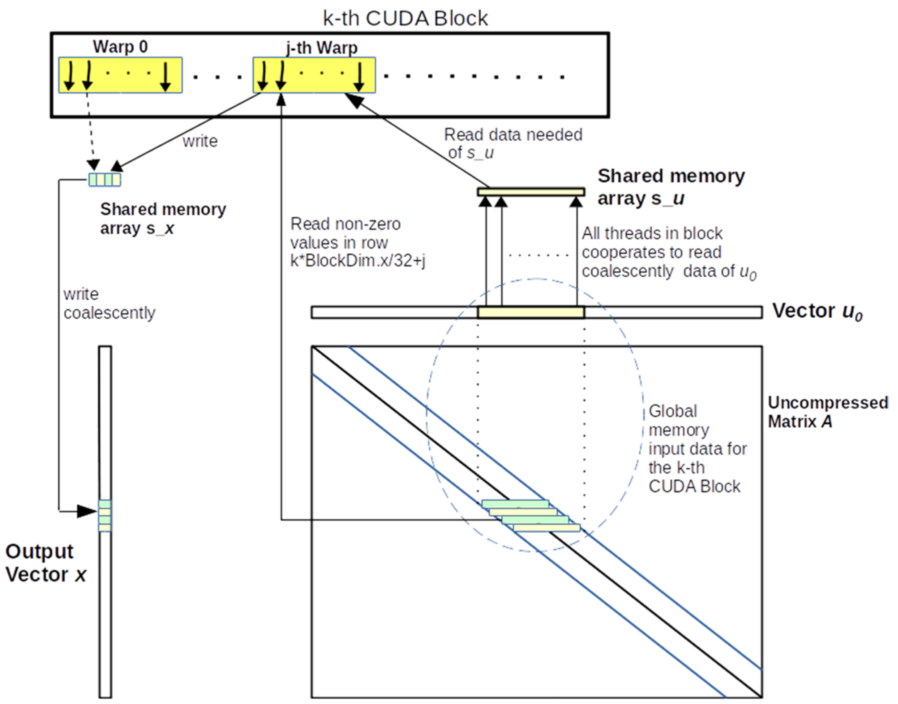

Fig. 3 Matrix–vector product: $\boldsymbol{x} = \boldsymbol{A} \cdot \boldsymbol{u_0}$. Each CUDA warp computes one element of the output vector $\boldsymbol{x}$

thread in the *j*-th warp computes one partial value and all the threads in the warp will cooperate following a reduction algorithm based on a warp shuffle operation [23], to add efficiently their previously computed values. In particular, we have used the operation `__shuffle_xor_sync` (we assume a compute capability higher or equal than 3.x) to perform the addition at warp level.

The components of the vector $\boldsymbol{x}$ which are obtained by each block are stored in a shared-memory array ***s_x*** to be written coalescently in the global memory vector $\boldsymbol{x}$.

In our implementation of SRJ, the system is preconditioned by left-multiplying by $\boldsymbol{D}^{-1}$ in such a way that the matrix of the linear system contains only values 1 on the diagonal.

We use another CUDA kernel, which also uses one-dimensional CUDA blocks, to complete the SRJ step by computing the residual vector

$$\boldsymbol{x} = \boldsymbol{b} - \boldsymbol{A}\,\boldsymbol{u_0} \tag{15}$$

and updating the next approximation to the solution

$$\boldsymbol{u_1} = \boldsymbol{u_0} + \omega_i \boldsymbol{x}. \tag{16}$$

In order to control the convergence after completing an SRJ cycle, we implement an efficient CUDA reduction algorithm based on [18, 25] to jointly perform the infinity norm of two vectors: the residual vector ($|\boldsymbol{b} - \boldsymbol{A}\,\boldsymbol{u}_{\mathbf{0}}|_\infty$) and the new approximation ($|\boldsymbol{u}_{\mathbf{0}}|_\infty$). In the reduction CUDA kernel, one half of the CUDA block processes a chunk of the residual vector and the other half processes the corresponding chunk of the other vector.

4 Implementation strategies: Diagonalization of the Schrödinger matrix

We need to compute the lowest N_{sbn} eigenvalues and relative eigenvectors of matrix $\mathcal{L}_{v,i}$ in (6). It is known that for a tridiagonal symmetric matrix like $\mathcal{L}_{v,i}$, the characteristic polynomial $p(X)$ can be computed via a recursive sequence of polynomials [36]:

$$\begin{aligned} &p_0(X) = 1 \\ &p_1(X) = \big(d_0 - X\big) \\ &p_j(X) = \big(d_{j-1} - X\big)p_{j-1}(X) - e_{j-2}^2\,p_{j-2}(X) \qquad \text{for } 2 \le j \le \mathrm{n}, \end{aligned} \tag{17}$$

such that $p(X) = p_{\mathrm{n}}(X)$. In order to seek for the zeros of this polynomial, we shall employ two strategies: either a multi-section iterative algorithm (a generalization of the bisection algorithm) or a Newton-Raphson iterative algorithm. The first one is extremely robust, can unconditionally provide selected eigenvalues, but is costly, whilst the second one is faster but needs proper seeding. Therefore, the strategy will be the following: at the first step of the time evolution we shall use the multi-section algorithm; after that, we shall switch to Newton-Raphson.

4.1 The multi-section algorithm for eigenvalues

The bisection algorithm is a well-known tool for computing eigenvalues, described, for instance, in [11, 36].

In our case, instead of using bisection, we can divide the interval into an arbitrary number of sub-intervals, which we shall call N_{multi} in the following. If we think of it in a sequential way, the algorithm is less efficient than usual bisection ($N_{\mathrm{multi}} = 2$); nevertheless, this approach could be advantageous on a GPU platform because it better exploits parallelism: we can compute concurrently the σ function

$$\sigma(\xi) := \text{ number of sign changes in } \big(p_{\mathrm{n}}(\xi), p_{\mathrm{n}-1}(\xi), \dots, p_1(\xi), p_0(\xi)\big)$$

at all the intermediate points, and hence use fewer iterations to converge to the desired accuracy. We recall that polynomials $\{p_j\}_{j=0,\dots,n}$ represent the (reversed, backward indexed) Sturm chain (17), for which the following result holds: let α a real number, then the number of zeros in the interval $]-\infty, \alpha[$ is given by $\sigma(\alpha)$. Suppose that the eigenvalues are ordered $\epsilon_0 < \epsilon_1 < \epsilon_2 < \cdots < \epsilon_{\mathrm{n}-1}$. As eigenvalue ϵ_p corresponds to the $(p+1)^{\mathrm{th}}$ zero of polynomial p, then

$$\epsilon_p < \xi \Longrightarrow \sigma(\xi) \ge p + 1 \qquad \text{and} \qquad \epsilon_p > \xi \Longrightarrow \sigma(\xi) \le p. \tag{18}$$

The situation is sketched in Fig. 4.

In order to implement the multi-section algorithm for N_{multi} sub-intervals, we shall use the following magnitudes (all indices start from zero):

- Interval $\left[Y_{\min}, Z_{\max}\right]$ is such that it contains all the eigenvalues, and $L := Z_{\max} - Y_{\min}$. This interval can be easily be obtained via Gershgorin circle theorem.
- Integer $n \in \mathbb{N} \setminus \{0\}$ indexes the iteration of the multi-section algorithm.
- Array $\epsilon^{\inf}_{v,p,i}$ of size $N_{\text{valleys}} \times N_{\text{sbn}} \times N_x$ represents a left-approximation of eigenvalue $\epsilon_{v,p,i}$, in the sense that

$$\epsilon_{v,p,i} \in \left]\epsilon^{\inf}_{v,p,i}, \epsilon^{\inf}_{v,p,i} + \frac{L}{(N_{\text{multi}})^{n+1}}\right[.$$

- $\sigma_{v,p,i,k}$ of size $N_{\text{valleys}} \times N_{\text{sbn}} \times N_x \times (N_{\text{multi}} - 1)$ represents the number of sign changes at point

$$\xi_{v,p,i,k} := \epsilon^{\inf}_{v,p,i} + (k+1)\frac{L}{(N_{\text{multi}})^{n+1}}.$$

So, the general view of the methods is:

$$\begin{array}{ll} \text{init}\left\{\begin{array}{ll} 1 & \text{Compute Gershgorin circles } \left[Y_{v,i}, Z_{v,i}\right] \text{ on the GPU} \\ 2 & \text{Compute minimum } Y_{\min} \text{ and maximum } Z_{\max} \text{ and let } L = Z_{\max} - Y_{\min} \\ 3 & \text{Inizialize } \epsilon^{\inf}_{v,p,i} = Y_{\min} \\ 4 & \text{Compute the number of iterations } n_{\text{iters}} := \left\lfloor \dfrac{\ln\left(\frac{L}{\varepsilon_{\text{tol}}}\right)}{\ln\left(N_{\text{multi}}\right)} \right\rfloor + 1 \end{array}\right. \\ \text{loop}\left\{\begin{array}{ll} 5 & \text{Loop: for } \left(n = 0; n < n_{\text{iters}}; n \leftarrow n+1\right) \\ 6 & \quad \text{Compute } \sigma_{v,p,i,k} \text{ on the GPU} \\ 7 & \quad \text{Update } \epsilon^{\inf}_{v,p,i} \text{ on the GPU} \end{array}\right. \end{array} \tag{19}$$

The last instruction inside the loop part, i.e. instruction 7 of (19), requires a reduction, as we need to compute

$$\tilde{k} := \max\left\{k \in \{-1, \ldots, N_{\text{multi}} - 2\} \text{ such that } \sigma_{v,p,i,k} \le p\right\}$$

to finally update

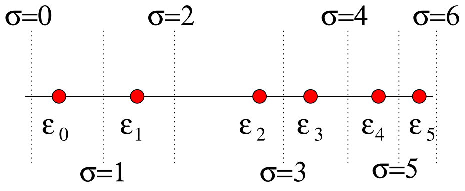

Fig. 4 Multi-section algorithm for eigenvalues. The discontinuity points of function σ identify the eigenvalues

$$\epsilon^{\inf}_{\nu,p,i} \longleftarrow \epsilon^{\inf}_{\nu,p,i} + \left(\tilde{k}+1\right)\frac{L}{(N_{\mathrm{multi}})^{n+1}}. \tag{20}$$

4.1.1 Implementation details

We use multi-section with 32 points, i.e. with $N_{\mathrm{multi}} = 33$. It is set like this so that we shall make each warp take care of updating one value of $\epsilon_{\nu,p,i}$. As $N_{\mathrm{sbn}} = 6$, it seems reasonable to use either 1 or 2 or 3 or 6 warps per block, to load only one matrix $\mathcal{L}_{\nu,i}$ per block. Blocks, therefore, will also be of size $\{32, 64, 96, 192\}$. Let N_{w} the number of warps per block, the block will be of size $32 \times N_{\mathrm{w}}$. As dimensions are ordered $i > \nu > p$, the $32 \times N_{\mathrm{w}}$ threads will take care of computing (for fixed (ν, i))

$$\underbrace{\left\{\sigma_{\nu,p,i,k}\right\}_{k=0}^{31}}_{0}, \quad \underbrace{\left\{\sigma_{\nu,p+1,i,k}\right\}_{k=0}^{31}}_{1}, \quad \dots, \quad \underbrace{\left\{\sigma_{\nu,p+N_{\mathrm{w}}-1,i,k}\right\}_{k=0}^{31}}_{N_{\mathrm{w}}-1}.$$

By using a device of Compute Capability (CC) higher or equal than 3.x, we can exploit warp shuffle functions to perform the reduction (19)-7 at warp level. In particular, we use `__shuffle_xor_sync` to compute the maximum $\tilde{k}$ of a vector $\Sigma_{\nu,p,i,\cdot}$ stored in shared memory and containing

$$\Sigma_{\nu,p,i,k} = \begin{cases} k & \text{if } \sigma_{\nu,p,i,k} \le p \\ -1 & \text{otherwise} \end{cases}$$

in such a way that we can update $\epsilon^{\inf}_{\nu,p,i}$ following (20).

In order to perform a coalescent reading from global memory of matrix $\mathcal{L}_{\nu,i}$, whose entries are used several times by each thread, we use shared memory. Matrix $\mathcal{L}$ is stored as described in Fig. 5, so that each block loads `SCHROED_MATRIX_ROW` elements, i.e. 128 doubles with our standard parameters, out of which only 125 are really useful and 3 are just used for padding with zeros.

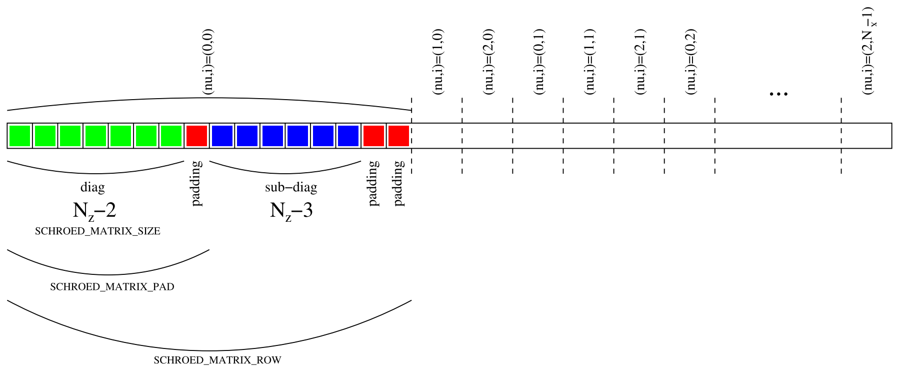

Fig. 5 Schrödinger matrices. Storage format of matrices $\mathcal{L}$

4.2 Newton-Raphson iterative method for eigenvalues

The Newton-Raphson algorithm can also be found in the classical book [36]. In our implementation the iteration is controlled by the CPU, and each call to a kernel updates the guess for the eigenvalues. We use one CUDA thread per eigenvalue. The implementation does not need any sophisticated technique; therefore, we do not give further details here.

4.3 Inverse Power Iterative Method (IPIM) for the approximation of the eigenvectors

Once the eigenvalues have been computed, it is the turn of the relative eigenvectors. In order to do that, we have used the IPIM (aka "inverse iteration" algorithm) [16] to approximate the eigenvector $\psi_{\nu,p,i}$ of $\mathcal{L}_{\nu,i}$ using the previously obtained eigenvalue $\epsilon_{\nu,p,i}$. The algorithm is described in Table 2.

Algorithm 2 IPIM for the approximation of eigenvector $\psi_{\nu,p,i}$.

```
Input: matrix $\mathcal{L}_{\nu,i}$ and the eigenvalue $\epsilon_{\nu,p,i}$
Output: $\psi_{\nu,p,i}$= eigenvector of $\mathcal{L}_{\nu,i}$ corresponding to $\epsilon_{\nu,p,i}$
1: $q^{(0)}$ = initial approximation to $\psi_{\nu,p,i}$
2: $k = 0$
3: while not convergence do
4:     Solve $(\mathcal{L}_{\nu,i} - \epsilon_{\nu,p,i} I)x^{(k)} = q^{(k-1)}$
5:     $q^{(k)} = x^{(k)}/||x^{(k)}||_2$
6:     $convergence = ||q^{(k)} - q^{(k-1)}||_\infty < tol$
7: end while
8: $\psi_{\nu,p,i} = q^{(k)}$
```

We have to approximate $N_{\text{valleys}} \times N_{\text{sbn}} \times N_x$ eigenvectors (with N_z elements) using this algorithm. We have used a different CUDA thread $\mathcal{T}_{\nu,p,i}$ to approximate the eigenvector $\psi_{\nu,p,i}$. Each thread solves locally the tridiagonal linear system using the Thomas algorithm [40]. Since each tridiagonal coefficient matrix $\mathcal{L}_{\nu,i}$ is symmetric, it is represented using two vectors with $N_z - 2$ double precision elements. Several CUDA threads work with the same coefficient matrix (threads $\mathcal{T}_{\nu,p,i}$ with $p \in \{0,\dots,5\}$, $\nu = \gamma$ and $i = \delta$ use the matrix $\mathcal{L}_{\gamma,\delta}$).

We have implemented two different CUDA kernels to implement this algorithm on GPU:

- Kernel A, where these vectors are read from global memory for each value of k in Algorithm 2, and
- Kernel B, which stores these vectors in shared memory. In this version, all the vectors which are needed for the threads in a CUDA block are loaded coalescently from global memory.

In both cases, we use one-dimensional CUDA blocks with 32 threads to avoid excessive spilling of available registers in the multiprocessor.

Table 1 shows the average runtime (measured in seconds) spent by both CUDA kernels in a time step for several values of N_z, using grid $N_x = 65$, $N_w = 300$, $N_\phi = 48$ (N_{dim} ($dim \in \{x, z, w, \phi\}$) is the number of discretization points for dimension *dim* in the grid), CFL condition 0.6, a source-drain voltage of 0.1 V and a source-gate voltage of 0.5 V. We can see that both kernels lead to very similar execution times. However, since kernel A achieves better times in all cases except for $N_z = 129$, we have opted for this version.

5 Numerical results

We have analyzed the performance and accuracy of the parallel solver, focusing on the GPU implementation of the Schrödinger-Poisson block (herein called the *iter* phase).

5.1 Description of the platform and solvers

The numerical experiments have been performed on a computing server with dual Intel Xeon Silver 4210 CPUs (in total, 20 physical cores with a base frequency of 2.2 GHz each and 40 logical processors) with 96 GB RAM, and 4 TB solid state hard drive. The system includes a NVIDIA Tesla V100 GPU (5120 cuda cores, 7 TFlops of double-precision peak performance and 32 GB DDR5 SDRAM) with CUDA Compute Capability (CC) 7.0 and an NVIDIA GeForce RTX 3090 GPU (5248 cores, 556 GFLOPS of double-precision peak performance and 24 GB GDDR6X) with CUDA CC 8.6. The operating system is Linux Debian 10.9 with GCC version 10.2.1 and the CUDA 11.2 runtime.

We have developed two implementations of the solver:

- **OpenMP solver**: This solver only exploits the cores of the CPUs in the platform by using *OpenMP* directives and functions (see [38] for additional details). In the experiments, this solver is run using 40 threads (two per physical core). To compile the OpenMP solver, we have used the GNU compiler `g++` version 10.2.1 using the switches `-fopenmp -O3 -m64 -use_fast_math`.

Table 1 Average runtimes (seconds) spent by both CUDA kernels implementing IPIM scheme for one time step

N_z	Kernel A	Kernel B (shared mem.)
33	.000266	.000272
49	.000366	.000520
65	.000503	.000557
97	.000960	.001060
129	.001520	.001470

- **CUDA solver**: This heterogeneous code performs all the relevant computing phases on one of the available GPUs (Tesla V100 or RTX 3090) under the control of a CPU thread which invokes the corresponding CUDA kernels. In the compilation with `nvcc`, we have used the switches `-O3 -m64 -use_fast_math` and the options necessary to generate PTX code and object code optimized to the particular GPU architecture.

In the OpenMP solver, we use exactly the same numerical methods as in the CUDA solver.

5.2 Experimental validation of convergence

The convergence of the Boltzmann-Schrödinger-Poisson solver has been experimentally validated by studying the results obtained with different grids at $t = 0.1$ picoseconds using a CFL condition 0.6, a source-drain voltage of 0.1 V and a source-gate voltage of 0.5 V. In order to avoid excessive complexity, two macroscopic magnitudes that capture characteristics of the solution at a time point are analyzed: the total current density $j(x)$ and the total surface density $\varrho(x)$. These magnitudes are computed as follows:

$$j(x) = 2\sum_{\nu,p}\int_{w'=0}^{+\infty}\int_{\phi'=0}^{2\pi} a_\nu^1(w',\phi')\Phi_{\nu,p}(w',\phi')\,\mathrm{d}\phi'\,\mathrm{d}w', \qquad \varrho(x) = 2\sum_{\nu,p}\varrho_{\nu,p}(x).$$

As reference solutions for these magnitudes, the numerical results obtained by the solver for a very fine grid, given by $N_x = 129$, $N_z = 129$, $N_w = 600$ and $N_\phi = 96$, are used, being N_{dim} ($dim \in \{x, z, w, \phi\}$) the number of discretization points for dimension *dim* in the grid.

For each magnitude, the reference solution is compared with respect to the numerical solutions obtained for several coarser grids. These coarser grids have fewer discretization points in all dimensions. Figure 6 shows how the numerical solutions of the quantities vary as the number of points in all grid dimensions increases. It is very evident that as grids with a higher number of points are used, the solution obtained is closer to the reference solution for both quantities.

5.3 General view

In Fig. 7, we draw the average runtime cost for one time step of both computational phases (*BTE* and *iter*) and also show the speedup obtained with both solvers (for the full simulation of one time step) with respect to the sequential version (for only one thread) of the OpenMP solver. These results have been obtained by averaging the execution time of 10 time steps using grid $N_x = 65$, $N_z = 65$, $N_w = 300$, $N_\phi = 48$, CFL condition 0.6, a source-drain voltage of 0.1 V and a source-gate voltage of 0.5 V.

For the OpenMP solver, the bottleneck is the integration of the Boltzmann Transport Equations (*BTE* phase). The port to GPU of this phase has already been

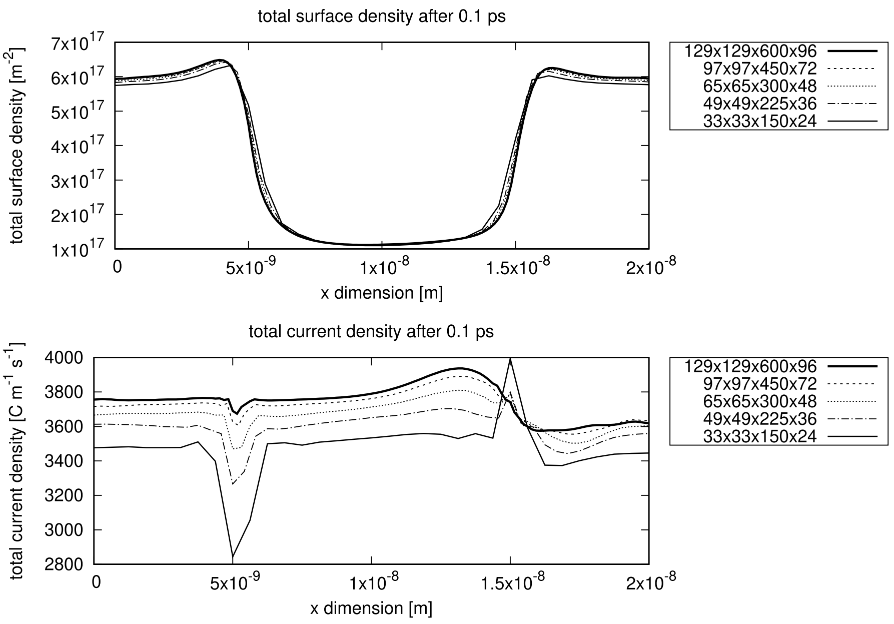

Fig. 6 Convergence to the reference solution ($129 \times 129 \times 600 \times 96$). Numerical solutions at $t = 0.1$ ps for the total surface density and the total current density obtained with several grids

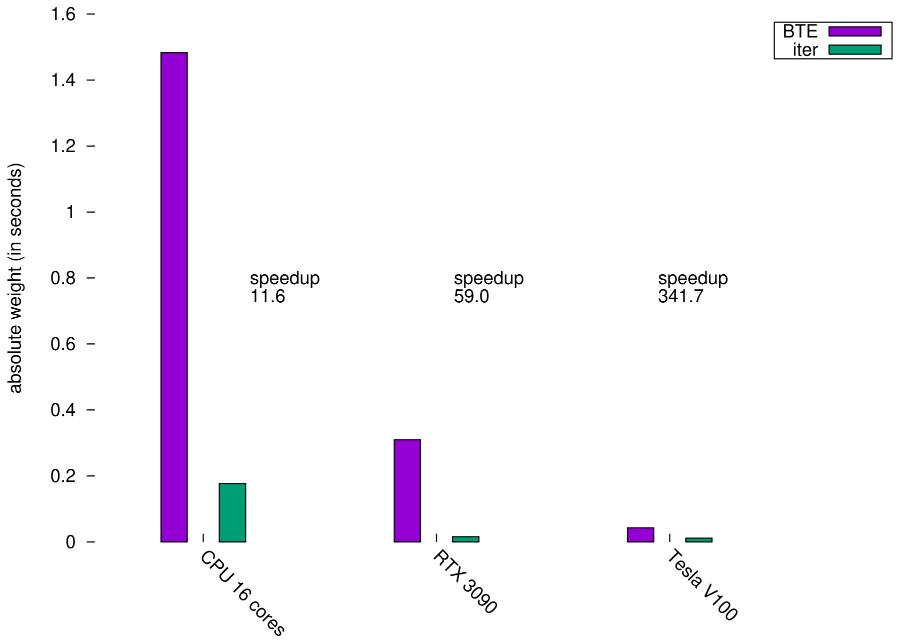

Fig. 7 Phases. Comparison of the cost of both computational phases between the OpenMP solver and the CUDA solver. The speedup is obtained for the full simulation of one time step using $N_x = 65$, $N_z = 65$, $N_w = 300$, $N_\phi = 48$, CFL condition 0.6, a source-drain voltage of 0.1 V and a source-gate voltage of 0.5 V

described in [27], where the *iter* phase was solved on the multiprocessor host platform by using OpenMP. In the following, we shall analyze the impact of the CUDA port of this phase.

Table 2 shows the speedup obtained with both solvers with respect to the sequential version in the main computing phases (*BTE* and *iter*). We can observe that the speedup obtained on Tesla V100 GPU in the *BTE* phase is significantly higher than the one obtained on RTX 3090 GPU (416.4 on Tesla V100 and 57.5 on RTX 3090). Conversely, for the *iter* phase, the CUDA solver on both GPUs achieves a closer speedup (129.6 on Tesla V100 and 93.2 on RTX 3090). We claim that this is because the *BTE* phase is much more intense in double-precision arithmetics than the *iter* phase and exhibits a higher degree of data parallelism (see section 5.4.1).

5.4 The *iter* phase

In Fig. 8, we sketch the cost of each computational section inside the *iter* phase and show the speedup obtained with respect to the sequential version of the OpenMP solver.

The dominant part in all cases is the solution of the linear system (8) (section `iter.solvelinsys`). The evaluation of the directional derivative (10) (section `iter.Frechet`) is the second costliest section in the OpenMP solver, but it is not so in the CUDA solver because it scales better than the implementations of the other sections. In the CUDA solver, the computation of the eigenstates (2) (section `iter.eigen`) also has a dominant role in the runtime. This section does not produce high runtime improvements on GPU because it does not exhibit a high arithmetic intensity and the CUDA kernels for this section spend a long time accessing global memory and short time computing with those data (see information about the kernels `cuda_tridiag_Thomas` and `cuda_eigenvalues_NR` in Table 5). Finally, the construction of the linear system (section `iter.constrlinsys`) is clearly the least expensive part in the *iter* phase.

Figure 8 also shows that the runtimes obtained on both GPUs are similar in all the sections of the *iter* phase, except for the evaluation of the directional derivative where the Tesla V100 GPU achieves considerably shorter runtimes.

In Fig. 9, we sketch the speedups achieved by the CUDA solver (on both GPUs) and the OpenMP solver with respect to a sequential version of the OpenMP solver for each of these four main sections (inside the *iter* phase). Table 3 shows the particular data sketched in Fig. 9. These data confirm that the CUDA kernel for the evaluation of the directional derivative efficiently exploits the double-precision

Table 2 Speedup obtained in the main computing phases with a typical grid ($65 \times 65 \times 300 \times 48$), CFL condition 0.6, a source-drain voltage of 0.1 V and a source-gate voltage of 0.5 V

Phase	2-cores CPU	4-cores CPU	8-cores CPU	16-cores CPU	RTX-3090	Tesla-V100
BTE	1.96	3.75	7.29	12.0	57.50	416.40
iter	1.88	3.43	5.98	8.4	93.2	129.6

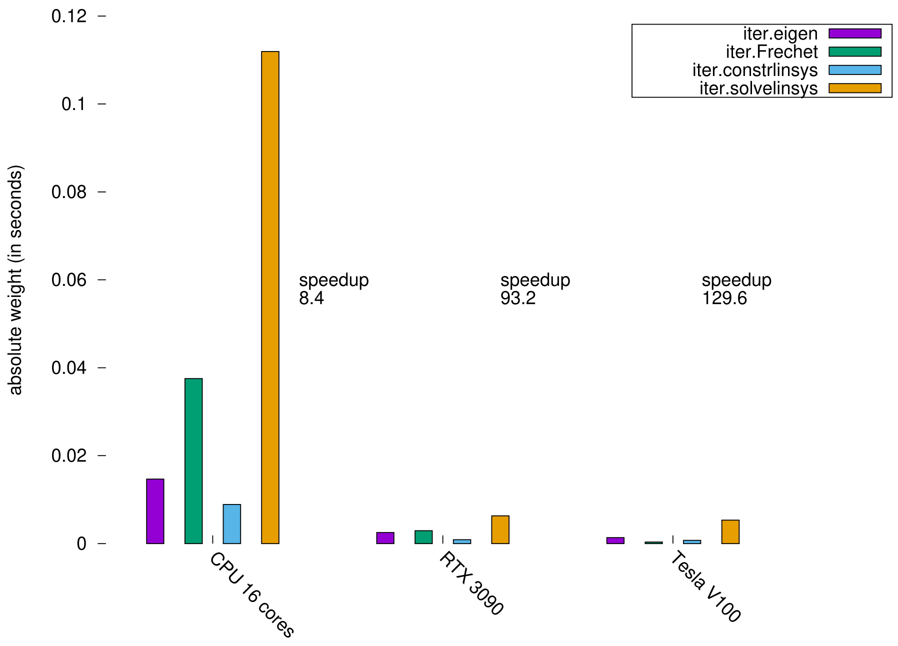

Fig. 8 Iter. Comparison of the cost of the computational phases inside the *iter* phase between OpenMP and a full GPU execution, when it is used a the grid $N_x = 65$, $N_z = 65$, $N_w = 300$, $N_\phi = 48$, CFL condition 0.6, a source-drain voltage of 0.1 V and a source-gate voltage of 0.5 V

computational power of the Tesla V100 GPU. For the other sections, the exploitation of the Tesla V100 power is not so efficient because of the much lower double-precision arithmetic intensity.

5.4.1 Behavior of the CUDA kernels

Table 4 shows the average runtime (measured in microseconds) spent by the main CUDA kernels in the *iter* phase per time step.

More into details, we are analyzing the behavior of six kernels, playing a role in four computational phases:

- the phase computing the eigenstates (eigenvalues and eigenvectors) of the Schrödinger matrices, labeled `iter.eigen`, involves the CUDA kernels
  - `cuda_eigenvalues_NR` for the computations detailed in Sect. 4.2;
  - `cuda_Tridiag_Thomas` implementing the algorithm in Table 2;
- the phase computing the directional derivative, labeled `iter.dirderiv`, involves the CUDA kernel
  - `cuda_compute_frechet` for the computation of (10);

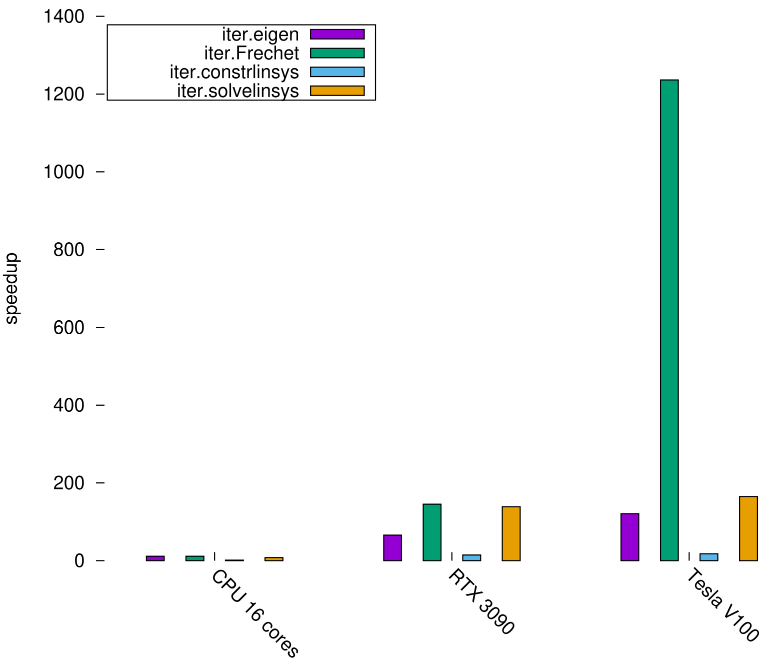

Fig. 9 Speedups. Speedup of the main sections inside the *iter* phase with respect to a sequential version of the OpenMP solver, when it is used the grid $N_x = 65$, $N_z = 65$, $N_w = 300$, $N_\phi = 48$, CFL condition 0.6, a source-drain voltage of 0.1 V and a source-gate voltage of 0.5 V

Table 3 Speedups for the main sections inside the *iter* phase with a typical grid $N_x = 65$, $N_z = 65$, $N_w = 300$, $N_\phi = 48$, CFL condition 0.6, a source-drain voltage of 0.1 V and a source-gate voltage of 0.5 V

Section	16-cores CPU	RTX-3090	Tesla-V100
`iter.eigen`	11.27	65.80	120.81
`iter.Frechet`	11.39	145.46	1236.38
`iter.solvelinsys`	7.85	138.89	164.98
`iter.constrinsys`	1.45	14.64	17.56

Table 4 Total runtimes (microseconds) spent by the main CUDA kernels for one time step, when it is used the grid $N_x = 65$, $N_z = 65$, $N_w = 300$, $N_\phi = 48$, CFL condition 0.6, a source-drain voltage of 0.1 V and a source-gate voltage of 0.5 V

CUDA kernel	Per-step RTX3090	Per-step V100
cuda_eigenvalues_NR	245.1	205.4
cuda_tridiag_Thomas	774.8	576
cuda_compute_frechet	3994.6	322.2
cuda_constrlinsys	1055.2	692.1
cuda_matvec_product	6499.4	5194.5
cuda_update_x	1376.8	1496.7

Table 5 Metrics provided by `Nsight` profiler. We have used: (1) = `gpu__compute_memory_throughput.avg.pct_of_peak_sustained_elapsed` (2) = `sm__throughput.avg.pct_of_peak_sustained_elapsed`

CUDA Kernel	Phase	Time RTX-3090	Time V100	(1) for RTX-3090	(1) for V100	(2) for RTX-3090	(2) for V100
`cuda_phonons_loss`	BTE	2503.5 ms	166.7 ms	0.53 %	8.1 %	85.4 %	92.6 %
`cuda_WENO_W`	BTE	365.4 ms	57.9 ms	10.3 %	66.5 %	86.1 %	76.7 %
`cuda_WENO_PHI`	BTE	631.2 ms	44.9 ms	2.8 %	39.9 %	86.1 %	87.4 %
`cuda_WENO_X`	BTE	315.6 ms	43.9 ms	10.9 %	73.6 %	85 %	36 %
`cuda_phonons_gain`	BTE	128.2 ms	20.5 ms	1.4 %	10 %	84.1 %	31 %
`cuda_compute_Wm1`	BTE	6.6 ms	13.8 ms	8 %	5.6 %	8 %	2.5 %
`cuda_pdftilde`	BTE	7.9 ms	8.7 ms	92 %	96.4 %	35 %	5.8 %
`cuda_set_fluxes_a2`	BTE	5.4 ms	7.0 ms	80.4 %	63.1 %	82 %	24.7 %
`cuda_set_fluxes_a3`	BTE	4.9 ms	7.0 ms	87.7 %	63.1 %	45 %	24.1 %
`cuda_perform_RK_2_3`	BTE	6.1 ms	6.6 ms	95.5 %	90.6 %	47.8 %	10 %
`cuda_perform_RK_3_3`	BTE	6.1 ms	6.5 ms	95.5 %	90.7 %	47.8 %	9.5 %
`cuda_perform_RK_1_3`	BTE	4.7 ms	4.8 ms	92.6 %	92 %	15.5 %	11.2 %
`cuda_matrix_vector_product`	iter	65 ms	51.9 ms	37.2 %	50.5 %	54.1 %	17.5 %
`cuda_update_x`	iter	13.7 ms	14.9 ms	4.2 %	4.3 %	1.8 %	0.5 %
`cuda_constr_linsys`	iter	10.5 ms	6.9 ms	41.2 %	29.1 %	3.3 %	2.9 %
`cuda_tridiag_Thomas`	iter	7.7 ms	5.7 ms	4.4 %	8.7 %	10.8 %	1.7 %
`cuda_compute_frechet`	iter	39.9 ms	3.2 ms	3.3 %	48.6 %	84.7 %	77.2 %
`cuda_eigenvalues_NR`	iter	2.4 ms	2.0 ms	1.8 %	2.6 %	17.9 %	1.6 %

Table 6 Averaged values for the metrics (1) and (2) at each phase (*BTE* and *iter*)

Phase	(1) for RTX-3090 (%)	(1) for V100 (%)	(2) for RTX-3090 (%)	(2) for V100 (%)
BTE	3.4	35.7	85	68.6
iter	22.1	36.6	51	14.1

- the phase constructing the linear system, labeled `iter.constrlinsys`, involved the CUDA kernel
  - `cuda_constrlinsys` for the computation of (11)-(12);
- the phase solving the linear system, labeled `iter.solvelinsys`, involves the CUDA kernels:
  - `cuda_matvec_product` for the computation of (14);
  - `cuda_update_x` for the computation of (15)-(16).

Additionally, a comparison of the most relevant CUDA kernels in the solver has been made, taking into account the throughtput achieved in the CUDA multiprocessors

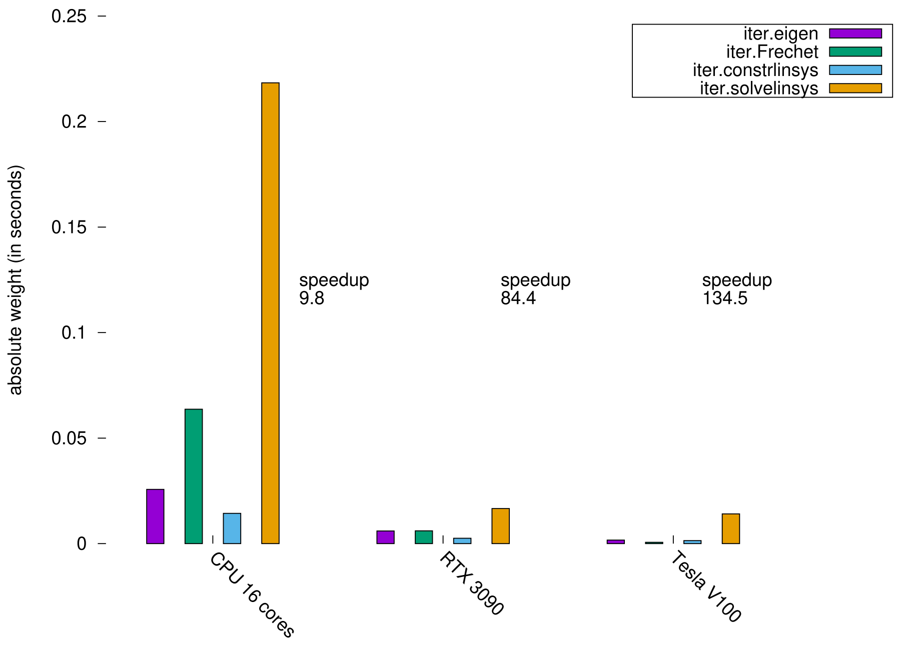

Fig. 10 Scaling with different grids. Doubling the points along x-dimension. Grid N_x = 129, N_z = 65, N_w = 300, N_Φ = 48, CFL condition 0.6, a source-drain voltage of 0.1 V and a source-gate voltage of 0.5 V

and in the memory access. For this purpose, we have used the NVIDIA Nsight Compute tools [28] to collect data about the following metrics:

1. `gpu__compute_memory_throughput.avg.pct_of_peak_sustained_elapsed`: measures the throughput of internal activity within caches and DRAM (as a percentage with respect to the peak throughput).
2. `sm__throughput.avg.pct_of_peak_sustained_elapsed`: measures the multiprocessor throughput assuming ideal load balancing across the multiprocessors of the GPUs (as a percentage with respect to the peak throughput).

Table 5 shows the values obtained for these metrics in the most relevant CUDA kernels of the phases *iter* and *BTE*. Table 6 shows the averaged values (taking into account the runtime of each CUDA kernel) of these metrics for each phase (*BTE* and *iter*) and GPU (RTX-3090 and Tesla V100). We can see that, for the Tesla V100 GPU, while the memory throughput (metric 1) is similar for both phases, the multiprocessor throughput (metric 2) is considerably higher for the BTE phase than for the iter phase. This shows that the BTE phase performs a much higher number of arithmetic operations in double precision per data read than the iter phase.

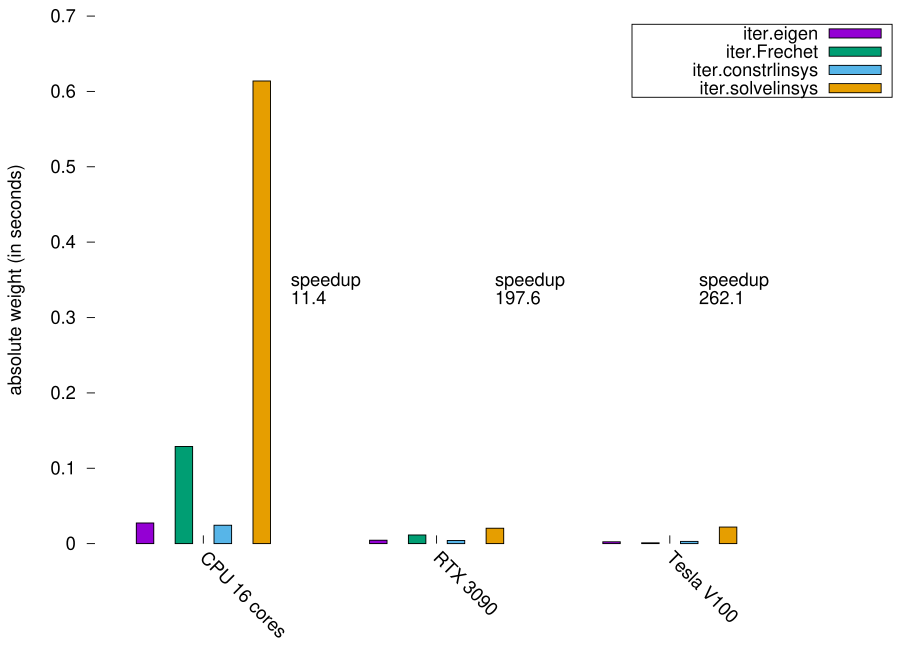

Fig. 11 Scaling with different grids. Doubling the points along z-dimension. Grid $N_x = 65$, $N_z = 129$, $N_w = 300$, $N_\phi = 48$, CFL condition 0.6, a source-drain voltage of 0.1 V and a source-gate voltage of 0.5 V

5.5 Scaling with different grids

In this subsection, we analyze how the change in the number of discretization points at a particular dimension affects the runtime performance of the solvers. The main goal is to determine the role played by the different dimensions. Obviously, there are dimensions which affect more the performance because most of the numerical schemes depend strongly on those dimensions from the point of view of the algorithmic complexity.

In Figs. 10, 11, 12 and 13, we double the points along dimensions x, z, w and ϕ and observe how this modifies the speedup in the *iter* phase. It is observed that the *iter* phase does not really depend on w and weakly depends on ϕ, and we actually see that the speedup obtained with respect to the speedup in Fig. 8 is very similar. The same applies to x, which acts as a parameter for this computational phase.

On the opposite, where we do observe a larger speedup is when we add points along the z-dimension: in Fig. 11, we remark a more significant speedup, because the GPU multiprocessors are better exploited by feeding them with a larger amount of computations. The number of discretization points in the z-dimension affects more strongly the computational cost because increasing this variable further increases the computational cost of the numerical methods related to confinement.

Table 7 shows the speedup obtained in the *iter* phase by both the OpenMP solver (using different number of threads) and the CUDA solver (on both GPUs) with

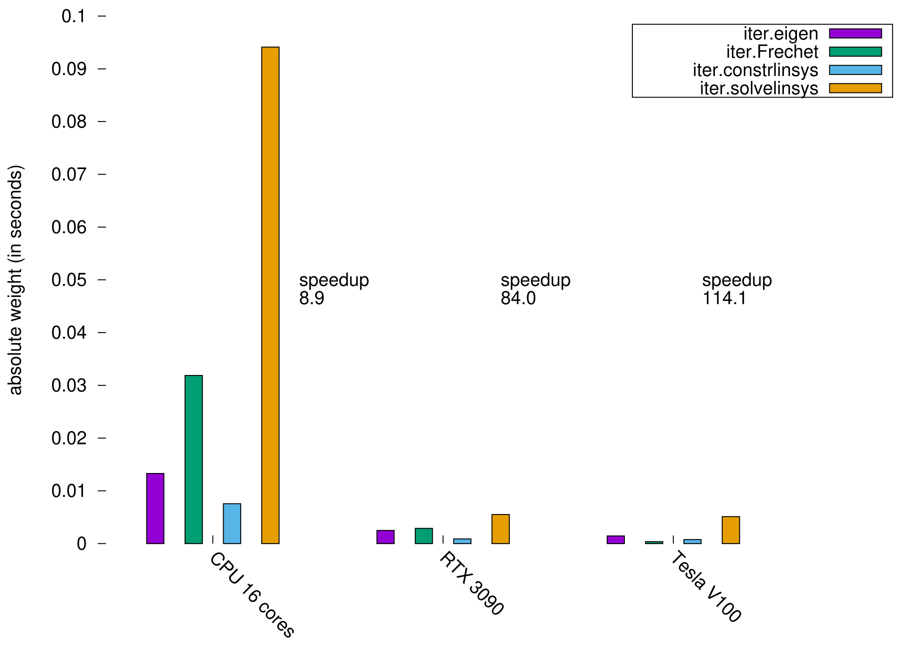

Fig. 12 Scaling with different grids. Doubling the points along w-dimension. Grid $N_x = 65$, $N_z = 65$, $N_w = 600$, $N_\phi = 48$, CFL condition 0.6, a source-drain voltage of 0.1 V and a source-gate voltage of 0.5 V

respect to a sequential version of the OpenMP solver (for only one thread) when using different grids for the simulation of one time step. In this table, we can see how the speedup increases as the number of points in the grid increases, which shows the trends of scalability for the different solvers.

6 Conclusions and perspectives

In this work, a simulator of nanoscale DG MOSFETs which solves the Boltzmann-Schrödinger-Poisson system performing all the computing phases on a NVIDIA GPU is described. Now all the computing phases of the simulator can be fully performed on GPU and show good performance, and reasonable computational times, taking into account the huge computational cost of this deterministic solver.

The port to GPU of the iterative section, solving the Schrödinger-Poisson block, has required adapting to GPU many techniques and methods such as the Scheduled Relaxation Jacobi method, the multi-section algorithm and the inverse power iteration.

This CUDA implementation of the Schrödinger-Poisson block provides satisfactory results, as it significantly reduces the execution times obtained on a modern dual processor server with 40 logical cores. As a result, we obtain one order of magnitude speedup with the full GPU version on a Tesla V100 GPU and a very close

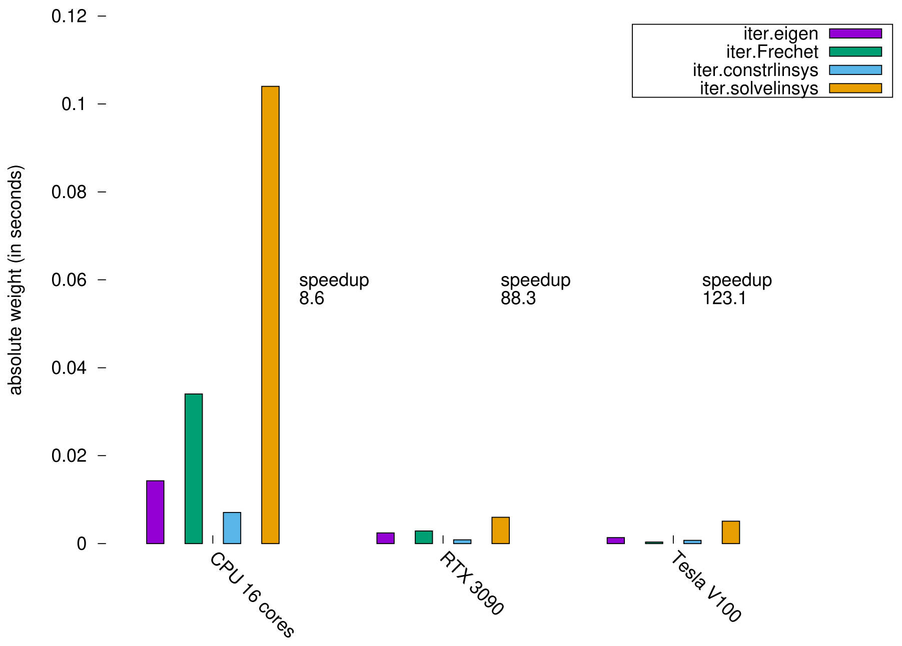

Fig. 13 Scaling with different grids. Doubling the points along ϕ-dimension. Grid $N_x = 65$, $N_z = 65$, $N_w = 300$, $N_\phi = 96$, CFL condition 0.6, a source-drain voltage of 0.1 V and a source-gate voltage of 0.5 V

Table 7 Speedup obtained in the *iter* phase with different grids

N_x	N_z	N_w	N_ϕ	2-cores CPU	4-cores CPU	8-cores CPU	16-cores CPU	RTX-3090	Tesla-V100
33	33	150	24	1.71	2.80	3.76	4.38	20.13	18.79
49	49	225	36	1.85	3.28	5.33	6.63	52.75	61.79
65	65	300	48	1.88	3.42	5.98	8.39	93.23	129.61
97	97	450	72	2.06	3.82	7.06	11.48	153.03	207.50
129	129	600	96	1.90	3.44	6.22	11.68	154.80	215.81

speedup is also obtained on a RTX 3090 GPU, which is much less powerful for double-precision computing.

Regarding the future extensions of this exploratory research, several topics can be explored. Firstly, it would be of interest to test the techniques described here in another kind of solver, in particular in a macroscopic solver, which is a goal of great interest for the semiconductor industry as it could provide significant improvement for commercial TCAD simulators. Secondly, no Monte-Carlo solver for the Boltzmann-Schrödinger-Poisson system has been ported to GPU so far, at the best of our knowledge. It would be interesting to see how the performance of such numerical methods improves. Thirdly, on a broader scale, we are working

Table 8 Values and units for several quantities

Description	Value	Unit
Confinement potential $V_c(z)$ in the SiO$_2$ region	3.15	eV
Confinement potential $V_c(z)$ in the Si region	0	eV
ε_R in the Si region	11.7	$\times\varepsilon_0\ \frac{A^2\ s^4}{Kg\ m^3}$
ε_R in the SiO$_2$ region	3.9	$\times\varepsilon_0\ \frac{A^2\ s^4}{Kg\ m^3}$
$\boldsymbol{m}_{x,v}$ and $\boldsymbol{m}_{z,v}$ in the SiO$_2$ region	0.5	$\times m_e$ Kg
N_D in source and drain regions	10^{26}	m^{-3}
N_D in the channel region	10^{18}	m^{-3}
N_D out of source, drain and channel regions	0	m^{-3}
Electron mass: m_e	$9.10938188 \times 10^{-31}$	Kg
Vacuum dielectric constant: ε_0	$8.8541878176 \times 10^{-12}$	$\frac{A^2\ s^4}{Kg\ m^3}$
ElectronVolt: eV	$1.60217653 \times 10^{-19}$	J

Table 9 Values and units for the Kane factor and the effective masses ($\boldsymbol{m}_{x,v}$ and $\boldsymbol{m}_{z,v}$) in the Si region

Valley	Kane factor $\tilde{\boldsymbol{\alpha}}_v$	$\boldsymbol{m}_{x,v}\left[\times m_e \text{ Kg}\right]$	$\boldsymbol{m}_{z,v}\left[\times m_e \text{ Kg}\right]$
$v = 1$	0.5 eV^{-1}	0.98	0.19
$v = 2$	0.5 eV^{-1}	0.19	0.19
$v = 3$	0.5 eV^{-1}	0.19	0.98

on improving the description of the MOSFET device at physical level, for example by introducing other scattering phenomena into the collisional operator, and in particular the surface roughness and the Coulomb interaction. Additionally, devices composed of different materials and heterostructures can be simulated and, when the semiconductor device must be simulated in the 3D physical space, the high number of points of the resultant mesh suggests deriving an implementation for multiple GPUs.

A Information about several quantities of the numerical scheme

In this appendix we describe the value and magnitudes of several physical constants in Equations (2) and (3). Table 8 describes the values and units for the dielectric constant ε_R, the effective masses in the SiO$_2$ region, the confinement potential and several auxiliary quantities. Table 9 describes the values and units for the Kane factor $\tilde{\boldsymbol{\alpha}}_v$ and for the effective masses in the Si region.

Acknowledgements We wish to thank A. Vidal from Universitat Politècnica de València and A. Godoy from Universidad de Granada for their valuable technical support. We also thank the Software Engineering Department of Universidad de Granada for the use of its computing server.

Author contributions All authors contributed to the study conception and design. Material preparation, data collection and analysis were performed by FV and JMM, PA worked mainly in the algorithms presented in Sect. 4. The first draft of the manuscript was written by FV and JMM. and all authors commented on previous versions of the manuscript. All authors read and approved the final manuscript.

Funding Funding for open access publishing: Universidad de Granada/CBUA. Francesco Vecil and J. M. Mantas acknowledge the project PID2020-117846GB-I00 funded by the Spanish Ministerio de Ciencia e Innovación. J. M. Mantas also acknowledges the project A-TIC-344-UGR20 funded by European Regional Development Fund.

Data availability The datasets generated during and/or analyzed during the current study are available from the corresponding author on reasonable request.

Declarations

Conflict of interest The authors have no relevant financial or non-financial interests to disclose.

Ethics approval Not applicable.

Consent to participate Not applicable.

Consent for publication Not applicable.

References

1. Abal-Kassim CA, Magoulès F (2017) Efficient implementation of Jacobi iterative method for large sparse linear systems on graphic processing units. J Supercomput 73:3411–3432. https://doi.org/10.1007/s11227-016-1701-3
2. Adsuara J, Cordero-Carrión I, Cerdá-Durán P, Aloy M (2016) Scheduled relaxation Jacobi method: improvements and applications. J Comput Phys 321:369–413. https://doi.org/10.1016/j.jcp.2016.05.053
3. Adsuara JE, Cordero-Carrión I, Cerdá-Durán P, Mewes V, Aloy MA (2017) On the equivalence between the scheduled relaxation Jacobi method and Richardson's non-stationary method. J Comput Phys 332:446–460. https://doi.org/10.1016/j.jcp.2016.12.020
4. Anderson E et al. (1999) LAPACK Users' Guide. SIAM. Third Edition
5. Balay S, Gropp WD, Curfman L, Smith BF (1997) Efficient management of parallelism in object oriented numerical software libraries. In: Arge E, Bruaset AM, Langtangen HP (eds) Modern software tools in scientific computing. Birkhäuser Press, pp 163–202
6. Balay S et al (2010) PETSc users manual. Technical Report ANL-95/11 - Revision 3.1. Argonne National Laboratory

7. Ben Abdallah N, Cáceres MJ, Carrillo JA, Vecil F (2009) A deterministic solver for a hybrid quantum-classical transport model in nanoMOSFETs. J Comput Phys 228(17):6553–6571. https://doi.org/10.1016/j.jcp.2009.06.001
8. Camiola VD, Mascali G, Romano V (2013) Simulation of a double-gate MOSFET by a non-parabolic energy-transport subband model for semiconductors based on the maximum entropy principle. Math Comput Model 58(1–2):321–343
9. Carrillo JA, Gamba IM, Majorana A, Shu CW (2003) A WENO-solver for the transients of Boltzmann-Poisson system for semiconductor devices: performance and comparisons with Monte Carlo methods. J Comput Phys 184(2):498–525. https://doi.org/10.1016/S0021-9991(02)00032-3
10. Chapman B, Jost G, van der Pas R (2008) Using OpenMP: portable shared memory parallel programming. The MIT Press, Cambridge
11. Demmel JW (1997) Applied Numerical Linear Algebra. SIAM
12. Donetti L, Sampedro C, Ruiz FG, Godoy A, Gámiz F (2018) A Multi-Subband Ensemble Monte Carlo simulations of scaled GAA MOSFETs. Solid-State Electron 143:49–55
13. El-Ayyadi A, Jüngel A (2005) Semiconductor simulations using a coupled quantum drift-diffusion Schrödinger-poisson model. SIAM J Appl Math 66(2):554–572
14. Espiñeira G, García-Loureiro AJ, Seoane N (2021) Parallel approach of Schrödinger based quantum corrections for ultrascaled semiconductor devices. J Comput Electron (In review). https://doi.org/10.21203/rs.3.rs-787168/v1
15. Frigyik BA, Srivastava S, Gupta MR (2008) An introduction to functional derivatives. UWEE Technical Report Number UWEETR-2008-0001. Department of Electrical Engineering University of Washington http://www.ee.washington.edu/techsite/papers/documents/UWEETR-2008-0001.pdf
16. Golub GH, Van Loan CF (1996) Matrix Computations. The Johns Hopkins University Press, Baltimore
17. Gummel HK (1964) A self-consistent iterative scheme for one-dimensional steady state transistor calculations. IEEE Trans Electron Devices 11(10):455–465
18. Harris M (2008) Optimizing Parallel Reduction in CUDA. NVIDIA Developer Technology. https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf
19. Jourdana C, Pietra P (2019) A quantum drift-diffusion model and its use into a hybrid strategy for strongly confined nanostructures. Kinetic Related Models 12(1):217–242
20. Clément J (2011) Mathematical modeling and numerical simulation of innovative electronic nanostructures. Université Paul Sabatier - Toulouse III; Università degli studi di Pavia
21. Kargar Z, Ruić D, Jungemann C (2015) A self-consistent solution of the Poisson, Schrödinger and Boltzmann equations for GaAs devices by a deterministic solver. Int Conf Simul Semicond Proc Devices 2015:361–364. https://doi.org/10.1109/SISPAD.2015.7292334
22. Li Y, Chao T-S, Sze SM (2015) A novel parallel approach for quantum effect simulation in semiconductor devices. Int J Model Simul 23(2):94–102. https://doi.org/10.1080/02286203.2003.11442259
23. Yuan L, Vinod G (2018) Using CUDA Warp-Level Primitives. NVIDIA Developer Blog. https://devblogs.nvidia.com/using-cuda-warp-level-primitives/
24. Sy-Shin L, Bernard P, Ahmed S (1987) A multiprocessor algorithm for the symmetric tridiagonal eigenvalue problem. SIAM J Sci Stat Comput 8(2):s155–s165. https://doi.org/10.1137/0908019
25. Luitjens J (2014) Faster parallel reductions on Kepler. NVIDIA Technical Blog. https://developer.nvidia.com/blog/faster-parallel-reductions-kepler/
26. Mascali G, Romano V (2012) A non parabolic hydrodynamical subband model for semiconductors based on the maximum entropy principle. Math Comput Model 55(3–4):1003–1020
27. Mantas JM, Vecil F (2019) Hybrid CUDA-OpenMP parallel implementation of a deterministic solver for ultra-shortDG MOSFETs. Int J High Perform Comput Appl 34(1):81–102
28. NVIDIA Developer Zone (2023) Nsight Compute. https://docs.nvidia.com/nsight-compute/index.html
29. Pietra P, Vauchelet N (2008) Modeling and simulation of the diffusive transport in a nanoscale Double-Gate MOSFET. J Comput Electron 7:52–65. https://doi.org/10.1007/s10825-008-0253-z
30. Pratapa PP, Suryanarayana P, Pask JE (2016) Anderson acceleration of the Jacobi iterative method: An efficient alternative to Krylov methods for large, sparse linear systemsT. J Comput Phys 306:43–54
31. Saad Y (2003) Iterative methods for sparse linear systems. Soc Indust Appl Math. https://doi.org/10.1137/1.9780898718003

32. Saint-Martin J, Bournel A, Monsef F, Chassat C, Dollfus P (2006) Multi sub-band Monte Carlo simulation of an ultra-thin double gate MOSFET with 2D electron gas. Semicond Sci Technol 21:29–31
33. Salas O, Lanucara P, Pietra P, Rovida S, Sacchi G (2011) Parallelization of a quantum-classical hybrid model for nanoscale semiconductor devices. Revista de Matemática: Teoría y Aplicaciones 18(2):231–248
34. Seoane N, Nagy D, Indalecio G, Espiẽira G, Kalna K, García-Loureiro A (2019) A multi-method simulation toolbox to study performance and variability of nanowire FETS. Materials 12(15):2391. https://doi.org/10.3390/ma12152391
35. Steiger S, Povolotskyi M, Park H, Kubis T (2011) NEMO5: a parallel multiscale nanoelectronics modeling tool. IEEE Trans Nanotechnol 10(6):1464–1474
36. Stoer J, Bulirsch R (1991) Introduction to numerical analysis, texts in applied mathematics. Springer, New York
37. Valín R, Sampedro R, Seoane N, Aldegunde M, García-Loureiro A, Godoy A, Gámiz F (2012) Optimisation and parallelisation of a 2D MOSFET multi-subband ensemble Monte Carlo simulator. Int J High Perform Comput Appl 27(4):483–492. https://doi.org/10.1177/1094342012464799
38. Vecil F, Mantas JM, Cáceres MJ, Sampedro C, Godoy A, Gámiz F (2014) A parallel deterministic solver for the Schrödinger-Poisson-Boltzmann system in ultra-short DG-MOSFETs: comparison with monte Carlo. Comput Math Appl 67:1703–1721. https://doi.org/10.1016/j.camwa.2014.02.021
39. Yang XIA, Mittal R (2014) Acceleration of the Jacobi iterative method by factors exceeding 100 using scheduled relaxation. J Comput Phys 274:695–708. https://doi.org/10.1016/j.jcp.2014.06.010
40. Zhang Y, Cohen J, Owens J (2010) Fast Tridiagonal Solvers on the GPU. Sigplan Notices - SIGPLAN. 45. https://doi.org/10.1145/1837853.1693472

Authors and Affiliations

Francesco Vecil[1] · José Miguel Mantas[2] · Pedro Alonso-Jordá[3]

Francesco Vecil
francesco.vecil@gmail.com

Pedro Alonso-Jordá
palonso@upv.es

[1] Laboratoire de Mathématiques Blaise Pascal, Université Clermont Auvergne, Clermont-Ferrand, France

[2] Departamento de Lenguajes y Sistemas Informáticos, Universidad de Granada, Granada, Spain

[3] Departamento de Sistemas Informáticos y Computación, Universitat Politècnica de València, Valencia, Spain