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

Francesco Vecil, José Miguel Mantas, Pedro Alonso-Jordá

Noname manuscript No.
(will be inserted by the editor)

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

Francesco Vecil · José Miguel Mantas · Pedro Alonso

Received: date / Accepted: date

Abstract The germ work [1] 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 phases, 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 left to the CPU for its complexity (though parallelized using OpenMP). 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 Relaxed 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 parallel cyclic reduction to approximate the wave vectors. We want to stress that this tool 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

1 Introduction

This paper comes as a completion of the work described in [1], 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 palliate the computational weight of such a high-dimensional model. NanoscaleDG MOSFETs are a key element in modern integrated circuits, and their modeling and simulation aims at contributing to their downscaling following Moore's law.
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

Francesco Vecil
Laboratoire de Mathématiques Blaise Pascal, Université Clermont Auvergne, France
E-mail: francesco.vecil@gmail.com

José Miguel Mantas
Lenguajes y Sistemas Informáticos, Universidad de Granada, Spain
E-mail: jmmantas@ugr.es

Pedro Alonso
Departamento de Sistemas Informáticos y Computación, Universitat Politècnica de València, Spain
E-mail: palonso@upv.es

its dimensionless form as (we address the reader to [1] for the particular about these equations):

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

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

$$-\nabla\cdot(\varepsilon_{\mathrm{R}}\,\nabla V) = -\left(\sum_{\nu,p}\varrho_{\nu,p}\cdot|\psi_{\nu,p}|^2 - N_D\right). \tag{3}$$

The numerical solver described in [1] fully ports onto GPU the transport part represented by the Boltzmann Transport Equations (BTEs) (1), while the goal of the present paper is to describe how we fully port onto GPU the computation of the eigenstates through the Schrödinger-Poisson block (2)-(3). We fulfill this objective to achieve a twofold improvement:

- to exploit the highest computational power of modern GPUs to accellerate 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 [1] 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 Figure 1):

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 Relaxed Jacobi (SRJ) scheme [24,25]: 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 multisection algorithm [30] 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 implementing a Cyclic Parallel Reduction (CPR) strategy [31,32,33].

The parallel implementation of the self-consistent 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 [23]. In this way, an OpenMP implementation of a numerical solver for a drift-diffusion-Schrödinger-Poisson model is described in [5] and a 2D multi-subband ensemble Monte Carlo simulator of 2D MOSFET devices which solve the Poisson-Schrödinger block is described in [7]. 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 [10] includes a self-consistent 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 [12] using MPI and the PETSC library [13,14] in [11]. In [6], a parallel implementation to simulate a metal-oxide-semiconductor (MOS) device, where a set of self-consistent 1D Schrödinger-Poisson equations are solved, is described. 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.
In [8], a resolution scheme of 2D Schrödinger equation-based corrections compatible with an existing parallel drift-diffusion model is implemented using MPI to simulate 3D semiconductor devices in the simulation framework VENDES [9].

This 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 platforms, can be adapted to macroscopic models, that are in general preferred in industrial simulations because of their lower computational cost, like drif-diffusion solvers [15,17,5,16], Monte Carlo solvers [19,7,18], solvers based on the maximum-entropy-principle energy transport model [21,22] and Spherical Harmonics Expansion (SHE) solvers [20].
The paper is organized as follows: in Section 2 we summarize the model and the equations on which we focus; in Section 3 we describe the solvers and the strategy implemented to achieve a solution of the Poisson-like equation on GPU; in Section 4 we describe the solvers and the procedure employed to compute the eigenstates on GPU; in Section 5 we show the numerical results we have obtained by using powerful modern GPUs; in Section 6 we draw some conclusions and sketch the future work in this promising auspisious 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 $\{\epsilon_{\nu,p}\}$, the wave vectors $\{\psi_{\nu,p}\}$ and the electrostatic potential V [1,2,4], such as it is shown in Figure 1. In this figure, $\nu \in \{0,1,2\}$ denotes the valley, $p \in \{0,\ldots,N_{\text{sbn}}-1\}$ denotes the subband (we consider $N_{\text{sbn}}=6$), $i=0,\ldots,N_x-1$ 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, $j=0,\ldots,N_z-1$ 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 [3] used for time integration.
From now on, we refer to the energy levels $\{\epsilon_{\nu,p}\}$ as the eigenvalues (of the Schrödinger matrix) and the wave vectors $\{\psi_{\nu,p}\}$ as the eigenvectors.

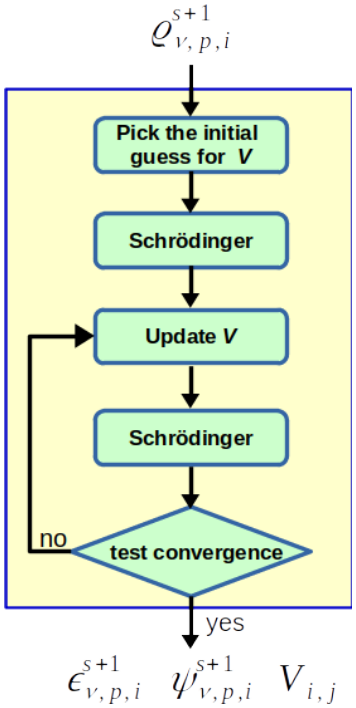

Fig. 1 Structure of the solver for the Schrödinger-Poisson block

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 and corresponding eigenvectors 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 $\psi_{\nu,p,i,j}$ 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

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

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

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

Obviously, stage $k+1$ is a refinement of the previous stage k. The derivative is meant in a directional sense. Details of the computations can be found in [4].
The scheme is sketched in Figure 1: 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 (4). 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 $\mathcal{L}$:

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

We wish to compute the first N_{sbn} eigenvalues and relative eigenvectors, called *energy levels* and *wave functions*.
In order to do this, we take into account the uniform grid described in [1] 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:

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

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

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

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 theDG MOSFET device, depend on the material:

$$m_{z,\nu,i,j} = \begin{cases} 0.5 & \text{if } (i,j) \text{ is in the } SiO_2 \text{ region} \\ 0.19 & \text{if } \nu < 2 \text{ and } (i,j) \text{ is in the Si region} \\ 0.98 & \text{if } \nu = 2 \text{ and } (i,j) \text{ is in the Si region} \end{cases}$$

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

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

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

2.2 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 [4].) This scheme boils down to the linear system on $V^{(k+1)}$

$$L^{(k)}\, V^{(k+1)} = R^{(k)}, \tag{7}$$

where

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

$$R^{(k)} = -N^{(k)}(x,z) + \int \mathcal{A}^{(k)}(x,z,\zeta)\, V^{(k)}(x,\zeta)\, \mathrm{d}\zeta,$$

being $\mathcal{A}^{(k)}(x,z,\zeta) := \mathcal{A}[V^{(k)}](x,z,\zeta)$ basically the directional derivative of the density $N^{(k)} := N[V^{(k)}]$. The Laplacian in the linear system (7) 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 finite differences

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

The integral is discretized by means of trapezoid rule

$$\left(\int \mathcal{A}^{(k)}(x,z,\zeta)\, V^{(k+1)}(x,\zeta)\, \mathrm{d}\zeta\right)_{i,j} = \frac{\Delta z}{2}\left[\sum_{j'=0}^{N_z-2} \mathcal{A}_{i,j,j'}^{(k)}\, V_{i,j'}^{(k+1)} + \sum_{j'=1}^{N_z-1} \mathcal{A}_{i,j,j'}^{(k)}\, V_{i,j'}^{(k+1)}\right], \tag{8}$$

where

$$\mathcal{A}_{i,j,j'}^{(k)} = 2\sum_{\nu,p}\sum_{p'\neq p} \frac{\varrho_{\nu,p,i}^{s+1} - \varrho_{\nu,p',i}^{s+1}}{\epsilon_{\nu,p',i}^{(k)} - \epsilon_{\nu,p,i}^{(k)}} \times \psi_{\nu,p,i,j'}^{(k)}\, \psi_{\nu,p',i,j'}^{(k)}\, \psi_{\nu,p',i,j}^{(k)}\, \psi_{\nu,p,i,j}^{(k)} \tag{9}$$

For the right hand side $R^{(k)}$, the integral is computed in a similar way to (8), and the density is simply

$$N_{i,j}^{(k)} = 2\sum_{\nu,p}\sum_{p'\neq p} \varrho_{\nu,p,i}^{s+1} \left|\psi_{\nu,p,i,j}^{(k)}\right|^2. \tag{10}$$

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

3 Highly-parallel methods for the linear system

The matrix $L^{(k)}$ representing the linear system (7) is of order $N_x \times N_z$, and contains N_x square blocks of size N_z on the diagonal. To solve efficiently this matrix, we have implemented on GPU a Scheduled Relaxed Jacobi (SRJ) method which is a promising extension of the Jacobi method for linear systems which results from discretizing Poisson-like PDEs [24,25].

3.1 The Scheduled Relaxed Jacobi (SRJ) method

The Jacobi method for the solution of the linear system provides poor convergence rate but is extremely parallel, as each value of the vector solution can be updated totally independently from all the other values of the vector solution.
Suppose

$$\boldsymbol{A} = \underbrace{\boldsymbol{L}}_{\text{lower triang.}} + \underbrace{\boldsymbol{D}}_{\text{diagonal}} + \underbrace{\boldsymbol{U}}_{\text{upper triang.}} .$$

is a square matrix of order N and $\boldsymbol{b}$ a vector of size N.
In [29], a vector form of the Jacobi algorithm is presented. In this vector form, the classical Jacobi step is rewritten to use the matrix-vector product operation in order to approximate the solution vector $\boldsymbol{u}$:

$$\begin{aligned} \eta^{(n)} &= D^{-1}(b - A\boldsymbol{u}^n) \\ \boldsymbol{u}^{(n+1)} &= \boldsymbol{u}^n + \eta^{(n)}, n = 1, 2, ... \end{aligned}$$

where $\eta^{(n)}$ denotes the error vector at iteration n.
A significant acceleration of the Jacobi algorithm can be obtained by applying the SRJ method. The SRJ method extends the classical Jacobi method by introducing an overrelaxation factor $\omega > 0$ which it is tuned using a number P of different levels to obtain a considerable reduction of the number of iterations. In the SRJ method one relaxed Jacobi step with parameter ω_n has the following form:

$$\begin{aligned} \eta^{(n)} &= \omega_n D^{-1}(b - A\boldsymbol{u}^n) \\ \boldsymbol{u}^{(n+1)} &= \boldsymbol{u}^n + \eta^{(n)} \end{aligned} \tag{11}$$

In SRJ, we complete several cycles until obtaining the convergence. At each cycle, we perform M relaxed Jacobi steps (11) where

$$M = \sum_{n=1}^{P} q_n,$$

being q_n the number of times we apply the parameter ω_n.
Therefore, a SRJ cycle consists in defining sequences of relaxed Jacobi steps:

$$\boldsymbol{\mathcal{L}} := \underbrace{\boldsymbol{\mathcal{L}}_{\omega_P} \circ \ldots \circ \boldsymbol{\mathcal{L}}_{\omega_P} \circ \boldsymbol{\mathcal{L}}_{\omega_P}}_{q_P \text{ times}} \circ \cdots \circ \underbrace{\boldsymbol{\mathcal{L}}_{\omega_2} \circ \ldots \circ \boldsymbol{\mathcal{L}}_{\omega_2} \circ \boldsymbol{\mathcal{L}}_{\omega_2}}_{q_2 \text{ times}} \circ \underbrace{\boldsymbol{\mathcal{L}}_{\omega_1} \circ \ldots \circ \boldsymbol{\mathcal{L}}_{\omega_1} \circ \boldsymbol{\mathcal{L}}_{\omega_1}}_{q_1 \text{ times}} \tag{12}$$

and updating the guess for the solution of system $\boldsymbol{A} \cdot \boldsymbol{u} = \boldsymbol{b}$ using these:

$$\boldsymbol{u}^{(\ell+1)} = \boldsymbol{\mathcal{L}}\boldsymbol{u}^{(\ell)},$$

where

$$\boldsymbol{\mathcal{L}}_\omega \boldsymbol{u} := \boldsymbol{u} + \omega D^{-1}(b - A\boldsymbol{u})$$

In our experiments, we have obtained good results with $P = 7$ cycles with $M = 93$, using the following relaxation parameters:

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

In order to avoid overflow in our numerical experiments, the schedule of a SRJ cycle does not follow the sequence (12) but the over-relaxation Jacobi steps (with $\omega_i > 1$) are evenly spaced over the SRJ cycle (see [24] for more details).
Moreover, the system is preconditioned by left-multiplying by

$$\boldsymbol{P} := \mathrm{diag}\left(\frac{1}{a_{0,0}}, \frac{1}{a_{1,1}}, \ldots, \frac{1}{a_{N-1,N-1}}\right)$$

in such a way that the matrix of the linear system contains only values 1 on the diagonal.

3.2 Implementation details

To implement each SRJ step (11) in CUDA, we have derived a version of the sparse matrix-vector product which takes into account the narrow banded structure of the matrix A.

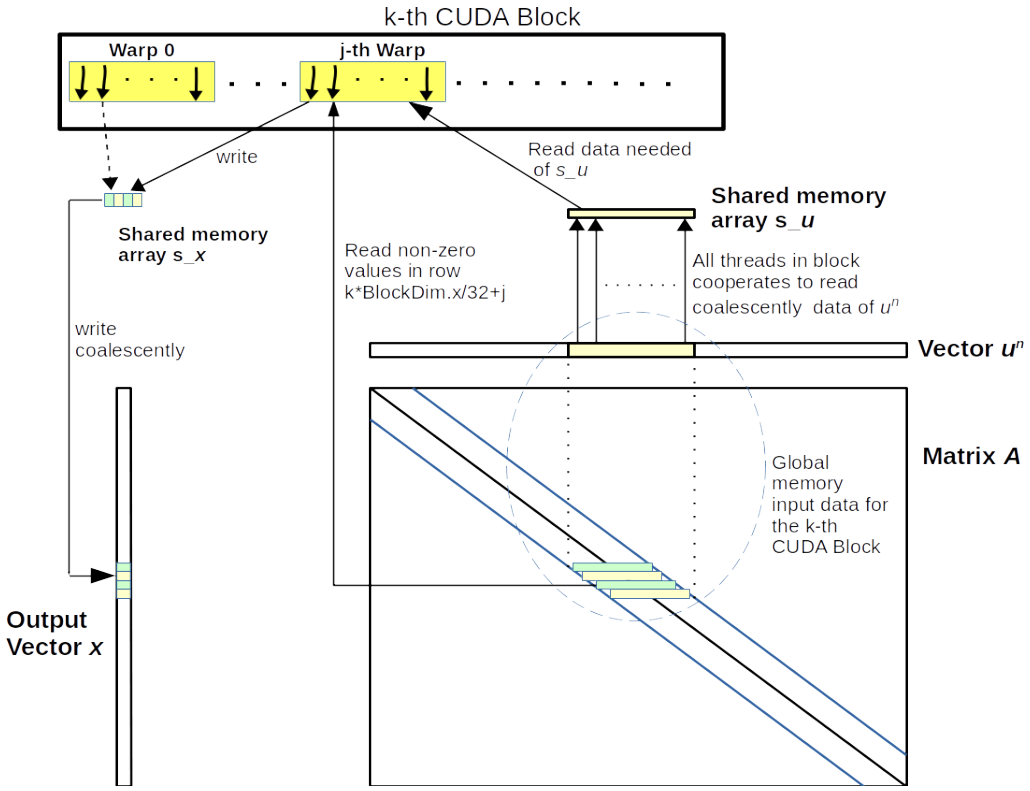

Fig. 2 Matriz-vector product when each CUDA warp computes one element of the output vector

We have developed a kernel CUDA to obtain the vector $x = A\boldsymbol{u}^n$ in (11), where the computation of the i-th element of the vector x (by performing the dot product of the i-th row of the sparse matrix A by the vector $\boldsymbol{u}^n$) is computed by a different CUDA warp (see Figure 2). We store the matrix A in the global memory of the device using the Compressed Sparse Row (CSR) storage format [35]. We use one-dimensional CUDA blocks where each CUDA block computes $\frac{B}{32}$ elements of x, being B the number of threads in a CUDA Block. Initially, all the warps in a CUDA block cooperate to read, in a coalescent way, the required values of $\boldsymbol{u}^n$ 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 A and the affected values s_u in order to compute x_t. For this, each 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 [34], to add efficienty 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 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 x.

4 Implementation strategies: Diagonalization of the Schrödinger matrix

We need to compute the lowest N_{sbn} eigenvalues and relative eigenvectors of matrix $\mathcal{L}_{\nu,i}$ in (6). It is known that for a tridiagonal symmetric matrix like $\mathcal{L}_{\nu,i}$, the characteristic polynomial $p(X)$ can be computed via a recursive sequence of polynomials [27]:

$$\begin{aligned} p_0(X) &= 1 \\ p_1(X) &= (d_0 - X) \\ p_j(X) &= (d_{j-1} - X)\, p_{j-1}(X) - e_{j-2}^2\, p_{j-2}(X) \qquad \text{for } 2 \le j \le \mathrm{n}, \end{aligned} \tag{13}$$

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 bisection algorithm for eigenvalues

The bisection algorithm is a well-known tool for computing eigenvalues, described, for instance, in [27, 28]. We report it here for the sake of clarifying the notation in the following.
The sequence (13) is a (generalized, backward indexed) Sturm chain, therefore the following result holds: let α a real number, let

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

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

$$\epsilon_p < \xi \Longrightarrow \sigma(\xi) \ge p+1 \qquad \text{and} \qquad \epsilon_p > \xi \Longrightarrow \sigma(\xi) \le p \tag{14}$$

Suppose now that we wish to approximate, at a given precision $\varepsilon_{\mathrm{tol}}$, the eigenvalue ϵ_p. In order to do that, the classical way is to use a bisection algorithm exploiting property (14). Let $[\alpha_0, \beta_0]$ an interval containing eigenvalue ϵ_p. Therefore, thanks to (14), $\sigma(\beta_0) \ge p+1$ and $\sigma(\alpha_0) < p+1$. Let $\alpha = \alpha_0$ and $\beta = \beta_0$. Iteration

$$\begin{array}{l} \text{while } (\text{some measurement} > \varepsilon_{\mathrm{tol}}) \\ \quad \text{let } \epsilon_p \longleftarrow \left(\dfrac{\alpha+\beta}{2}\right) \\ \quad \text{if } \sigma(\epsilon_p) < p+1 \\ \qquad \text{then } (\alpha, \beta) \longleftarrow (\epsilon_p, \beta) \\ \qquad \text{else } (\alpha, \beta) \longleftarrow (\alpha, \epsilon_p) \end{array}$$

produces a sequence of shrinking intervals which enclose the eigenvalue ϵ_p. When the iteration exits the loop, eigenvalue ϵ_p is then approximated at "$\varepsilon_{\mathrm{tol}}$ accuracy" (in some sense).
Still, we have to know how to pick a good initial interval $[\alpha_0, \beta_0]$ and a stop condition for the bisection method. If we apply Gershgorin circle theorem to the tridiagonal matrix $\mathcal{L}_{\nu,i}$ (6), then we know that every eigenvalue lies in at least one of the Gershgorin intervals $[d_j - R_j, d_j + R_j]$, being

$$R_j = \begin{cases} |e_0| & \text{if } j = 0 \\ |e_j| + |e_{j-1}| & \text{if } 0 < j < N_z - 3 \\ |e_{N_z-4}| & \text{if } j = N_z - 3 \end{cases}. \tag{15}$$

If we take

$$[A,B] := \left[\min_{0\le j<N_z-2} \{d_j - R_j\}, \max_{0\le j<N_z-2} \{d_j + R_j\} \right], \tag{16}$$

then we are sure that all the eigenvalues fall into interval $[A,B]$, which can thus be used as initial interval for the algorithm. After the first steps, as the energy levels evolve in a continuous way, we can exploit the already-computed values to initialize the algorithm, hence reducing the number of iterations needed. As a stop condition we shall use an absolute one

$$\beta - \alpha < \varepsilon_{\text{tol}}, \qquad \varepsilon_{\text{tol}} = 10^{-12}.$$

4.2 The multi-section iteration for eigenvalues

Instead of using bisection, i.e. dividing the interval into two parts at each iteration, 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 ($N_{\text{multi}} = 2$) because it computes the σ function more times; nevertheless, this approach could be advantageous on a GPU platform because it better exploits parallelism: we can compute concurrently the σ function at all these points, and hence use fewer iterations to converge to the desired accuracy.
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 $[Y_{\min}, Z_{\max}]$ is such that it contains all the eigenvalues, and $L := Z_{\max} - Y_{\min}$.
- Integer $n \in \mathbb{N} \setminus \{0\}$ indexes the iteration of the multi-section algorithm.
- Array $\epsilon^{\inf}_{\nu,p,i}$ of size $N_{\text{valleys}} \times N_{\text{sbn}} \times N_x$ represents a left-approximation of eigenvalue $\epsilon_{\nu,p,i}$, in the sense that
$$\epsilon_{\nu,p,i} \in \left] \epsilon^{\inf}_{\nu,p,i}, \epsilon^{\inf}_{\nu,p,i} + \frac{L}{(N_{\text{multi}})^{n+1}} \right[.$$
- $\sigma_{\nu,p,i,k}$ 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
$$\xi_{\nu,p,i,k} := \epsilon^{\inf}_{\nu,p,i} + (k+1)\frac{L}{(N_{\text{multi}})^{n+1}}.$$

So, the general view of the methods is:

$$\text{init} \begin{cases} 1 & \text{Compute Gershgorin circles } [Y_{\nu,i}, Z_{\nu,i}] \text{ on the GPU} \\ 2 & \text{Compute minimum } Y_{\min} \text{ and maximum } Z_{\max} \text{ and let } L = Z_{\max} - Y_{\min} \\ 3 & \text{Inizialize } \epsilon^{\inf}_{\nu,p,i} = Y_{\min} \\ 4 & \text{Compute the number of iterations } n_{\text{iters}} := \left\lfloor \dfrac{\ln\left(\frac{L}{\varepsilon_{\text{tol}}}\right)}{\ln\left(N_{\text{multi}}\right)} \right\rfloor + 1 \end{cases}$$

$$\text{loop} \begin{cases} 5 & \text{Loop: for } (n = 0; n < n_{\text{iters}}; n \leftarrow n+1) \\ 6 & \quad \text{Compute } \sigma_{\nu,p,i,k} \text{ on the GPU} \\ 7 & \quad \text{Update } \epsilon^{\inf}_{\nu,p,i} \text{ on the GPU} \end{cases} \tag{17}$$

The last instruction inside the loop part, i.e. instruction 7 of (17), requires a reduction, as we need to compute

$$\tilde{k} := \max\{k \in \{-1, \dots, N_{\text{multi}} - 2\} \text{ such that } \sigma_{\nu,p,i,k} \le p\}$$

to finally update

$$\epsilon^{\inf}_{\nu,p,i} \longleftarrow \epsilon^{\inf}_{\nu,p,i} + \left(\tilde{k}+1\right)\frac{L}{(N_{\text{multi}})^{n+1}}. \tag{18}$$

4.2.1 Implementation details.

We use multi-section with 32 points, i.e. with $N_{\text{multi}} = 33$. It is set like this so that we shall make each warp take care of updating one value of $\epsilon_{\nu,p,i}$. As $N_{\text{sbn}} = 6$, 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 $32 \times N_{\text{w}}$. As dimensions are ordered $i > \nu > p$, the $32 \times N_{\text{w}}$ threads will take care of computing (for fixed (ν, i))

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

By using a device of compute capability higher or equal than 3.x, we can exploit warp shuffle functions to perform the reduction (17)-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

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

in such a way that we can update $\epsilon^{\inf}_{\nu,p,i}$ following (18). The shared memory for $\Sigma_{\nu,p,i,\cdot}$ is declared as `volatile` in order to prevent the compiler from any optimization on $\Sigma_{\nu,p,i,\cdot}$, which is modified, when the reduction is performed, by the other threads in a way that the compiler cannot predict. Hence, without the `volatile` keyword, the compiler could assume that some values remain constant while they do not, thus resulting in miscalculations.
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 Figure 3, 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.

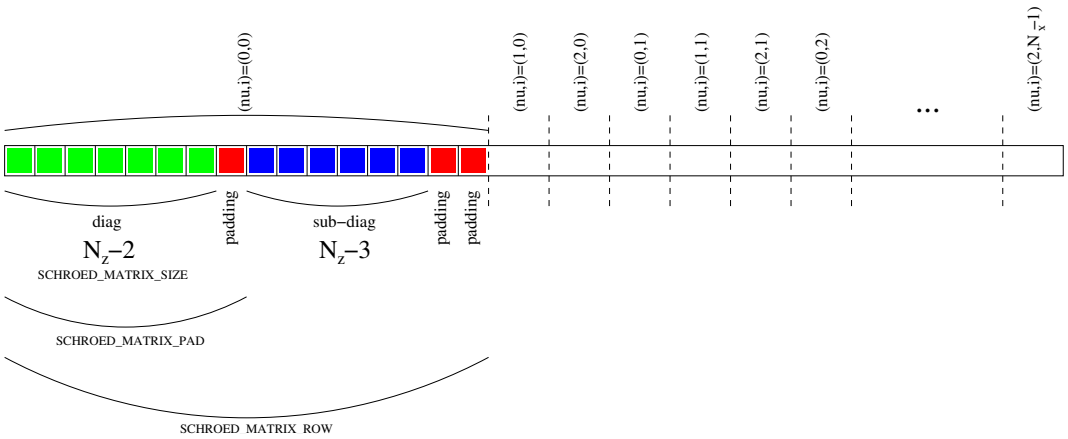

Fig. 3 Schrödinger matrices. Storage format of matrices $\mathcal{L}$.

4.3 Newton-Raphson iterative method for eigenvalues

The Newton algorithm can also be found in the classical book [27]. 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 thread per eigenvalue. The implementation does not need any sophisticated technique, therefore we do not give further details here.

4.4 Eigenvectors

Once the eigenvalues have been computed, it is the turn of the relative eigenvectors. In order to do that, we have to provide one of the two (up to the sign) normalized solutions to $\mathcal{L}_{\nu,i}\,\psi_{\nu,p,i,\cdot} = \epsilon_{\nu,p,i}\,\psi_{\nu,p,i,\cdot}$, i.e.

$$\begin{pmatrix} \tilde{d}_0 & e_0 & & & & & & & \\ e_0 & \tilde{d}_1 & e_1 & & & & & & \\ & e_1 & \tilde{d}_2 & e_2 & & & & & \\ & & \ddots & \ddots & \ddots & & & & \\ & & & e_{j-2} & \tilde{d}_{j-1} & e_{j-1} & & & \\ & & & & e_{j-1} & \tilde{d}_j & e_j & & \\ & & & & & e_j & \tilde{d}_{j+1} & e_{j+1} & \\ & & & & & & \ddots & \ddots & \ddots \\ & & & & & & & e_{\mathrm{n}-2} & \tilde{d}_{\mathrm{n}-1} \end{pmatrix} \begin{pmatrix} \psi_0 \\ \psi_1 \\ \psi_2 \\ \psi_3 \\ \vdots \\ \psi_{j-1} \\ \psi_j \\ \psi_{j+1} \\ \vdots \\ \psi_{\mathrm{n}-2} \\ \psi_{\mathrm{n}-1} \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ \vdots \\ 0 \\ 0 \\ 0 \\ \vdots \\ 0 \\ 0 \end{pmatrix}. \tag{19}$$

where we have set $\tilde{d}_i = d_i - \epsilon_{\nu,p,i}$ for the sake of lighter notations, and skipped other indices that only act as parameters.
In order to compute one of the two normalized eigenvectors solution to the problem, we set up the Inverse Power Iterative Method (IPIM). Namely, if we start from a guess $\boldsymbol{\psi}^{(0)}$ and we iterate

$$\begin{aligned} &\text{solve } (\mathcal{L}_{\nu,i} - \tilde{\epsilon}_{\nu,p,i} I)\,\boldsymbol{\psi}^{(k+1)} = \boldsymbol{\psi}^{(k)} \qquad \text{for } k \geq 0 \\ &\boldsymbol{\psi}^{(k+1)} \longleftarrow \frac{\boldsymbol{\psi}^{(k+1)}}{\left\|\boldsymbol{\psi}^{(k+1)}\right\|} \end{aligned} \tag{20}$$

then the sequence $\boldsymbol{\psi}^{(k)}$ will converge to an eigenvector relative to the eigenvalue closest to $\tilde{\epsilon}_{\nu,p,i}$. Although the system (19) is analytically singular, numerically it is not, due to the limited precision by which real numbers are represented, so the choice $\tilde{\epsilon}_{\nu,p,i} = \epsilon_{\nu,p,i}$ in the IPIM is legitimate. Actually, in a slightly counter-intuitive way, as pointed-out in [36], the fact of being close to a singular matrix in this case turns out to be an advantage and accelerates convergence.
Hence, in order to implement the IPIM (20), we need:

- a method for the solution of the linear system;
- a reduction method for the normalization of the solution;
- a reduction method to control the iterations.

The choice has been made of using one block for each eigenvalue, with a block size multiple of 32 in such a way that it is larger than $N_z - 2$. This way, we can exploit the Parallel Cyclic Reduction method (described in the following section) for the solution of linear system, and warp shuffle functions for the reductions.

4.4.1 Parallel Cyclic Reduction

We use the Parallel Cyclic Reduction (PCR) algorithm to solve the symmetric tridiagonal linear system [31,32,33].
The kernel uses one block to solve each linear system where each block consists in as many warps as needed to cover the $N_z - 2$ points (so for instance, 2 warps, i.e. 64 CUDA threads, are used for our standard meshes where $N_z - 2 = 63$).
In the PCR algorithm, to solve a tridiagonal system with n equations, we perform $\log n$ forward reduction steps. At the k-th step ($k = 0, ..., \log n - 1$), the equation j ($j = 0, 1, ..., n - 1$) is modified as a linear combination of the equations $(j + 2^k)$ and $(j - 2^k)$.
For each CUDA block, the inputs are the vectors $(\tilde{d}_0, \tilde{d}_1, \ldots, \tilde{d}_{\mathrm{n}-1})$ and $(e_1, e_2, \ldots, e_{\mathrm{n}-2})$ with the previously introduced modification (imposing $\psi_{j'} = 1$ for a particular $j' \in \{0, \ldots, \mathrm{n} - 1\}$) , and the output is $(\psi_0, \psi_1, \ldots, \psi_{\mathrm{n}-1})$. The algorithm which describe the computation of a CUDA thread in a kernel is shown in Table 1. In this algorithm, the j^{th} CUDA thread (at each CUDA block) manages locally the coefficients of j^{th} equation of the system to compute ψ_j.

Input: $(\tilde{d}_0, \tilde{d}_1, \ldots, \tilde{d}_{\mathrm{n}-1})$ and $(e_1, e_2, \ldots, e_{\mathrm{n}-2})$
Output: $(\psi_0, \psi_1, \ldots, \psi_{\mathrm{n}-1})$

Declare shared memory arrays a, b, c and y with size n
if $j < n$ **then**
    $b[j] = \tilde{d}_j$
    $c[j] = e_j$
    $y[j] = 0$
end if
Synchronize threads in CUDA Block
if $j < n - 1$ **then**
    $a[j + 1] = c_j$
end if
if $(j = 0)$ **then** $\{\ a[0] = 0;\ c[n - 1] = 0\ y[j'] = 1\ \}$
Synchronize threads in CUDA Block
for $k = 1, \ldots, \lceil \log n \rceil$ **do**
    $j^+ = j + 2^k;\ j^- = j - 2^k$
    if $j^- \geq 0$ **then**
        $\alpha = -a[j] \cdot b[j^-]^{-1}$
        $a' = \alpha \cdot a[j^-]; \quad b' = \alpha \cdot c[j^-]; \quad y' = \alpha \cdot y[j^-]$
    end if
    if $j^+ < n$ **then**
        $\beta = -c[j] \cdot b[j^+]^{-1}$
        $a' = \beta \cdot c[j^+]; \quad b' = \beta \cdot a[j^+]; \quad y' = \beta \cdot y[j^+]$
    end if
    Synchronize threads in CUDA Block
    if $j < n$ **then**
        $a[j] = a'; \quad b[j] = b'; \quad y[j] = y'$
    end if‘
end for
if $j < n$ **then**
    $\psi_j = y'/b'$
end if

Table 1 Algorithm to solve a tridiagonal system using a CUDA block

5 Numerical results

We have analyzed the performance of the parallel solver, centering in 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 (20 cores total) with 96 GB RAM, and 4TB 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) and a NVIDIA GeForce RTX 3090 GPU (5248 cores, 556 GFLOPS of double precision peak performance and 24 GB GDDR6X). The operating system is Linux Debian 10.9 with GCC version 10.2.0 and the CUDA 10.2 runtime.
We have developed two implementations of the solver:

- **OpenMP solver**: This solver only exploit the cores of the CPUs in the platform by using *OpenMP* directives and functions (see [2] for additional details). In the experiments, this solver is run using 16 cores of the server.
- **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.

5.2 General view

In Figure 4 the absolute and relative cost of each computational phase is drawn. For a CPU-parallel code using OpenMP, we see that the bottleneck is represented by the integration of the Boltzmann Transport Equations (*BTE* phase). When this part is performed on the GPU, a significant execution time reduction is obtained. The new results are represented by the CUDA solver, in which a significant reduction is obtained also for the *iter* phase.
In Figure 5 we sketch the absolute and relative cost of each computational phase inside *iter* phase. The dominant parts are the solution of the linear system (7) and the computation of the eigenstates (2).

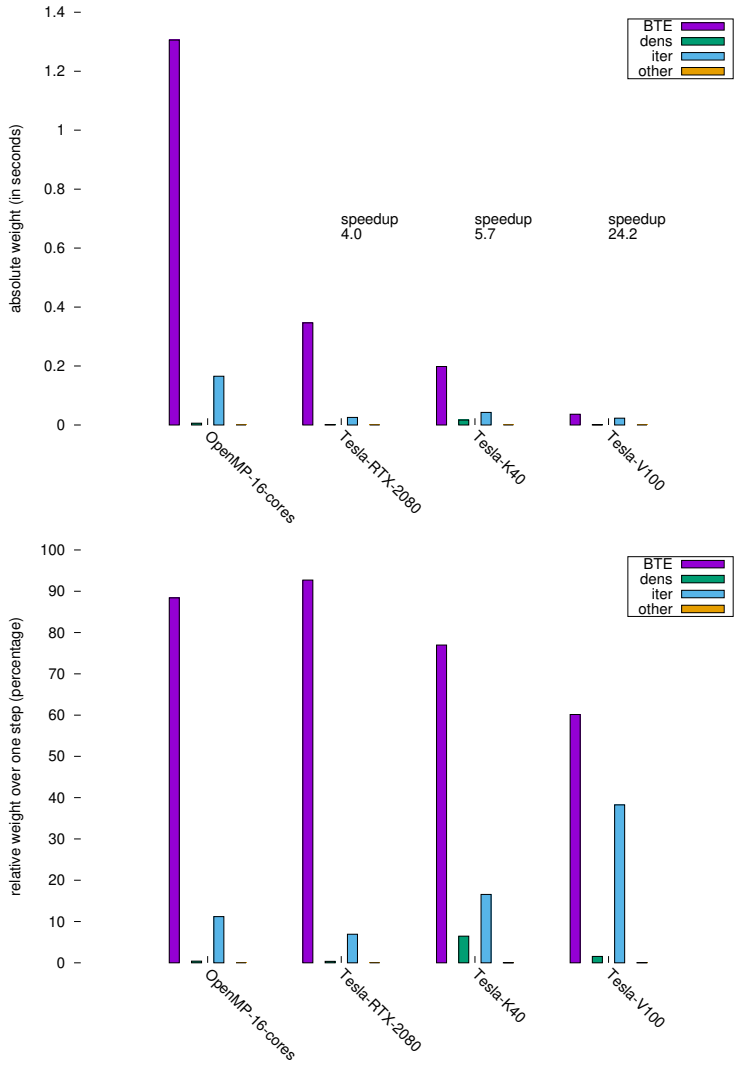

Fig. 4 Phases. The absolute and relative cost of the three computational phases for a GPU execution.

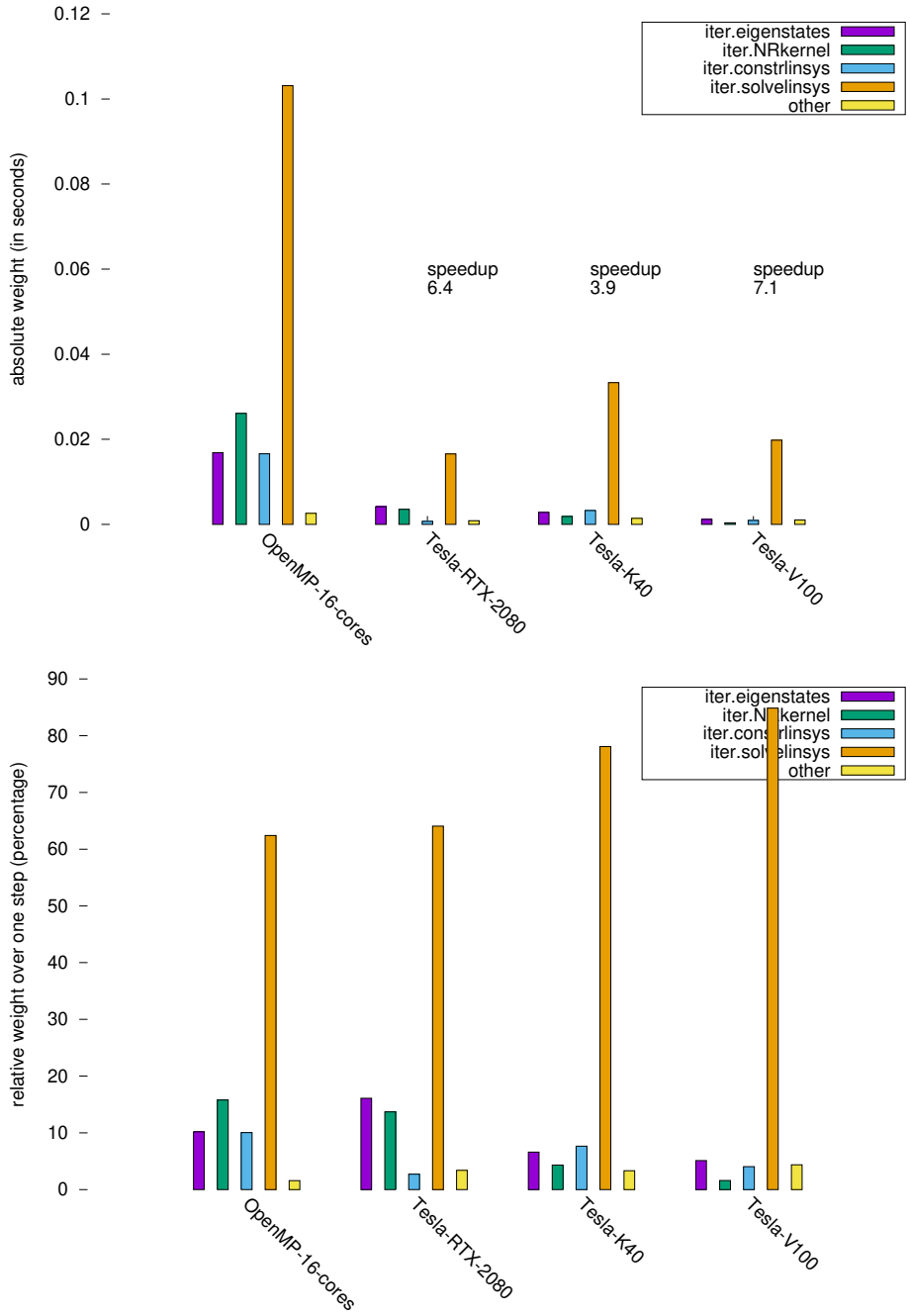

Fig. 5 Iter. The relative cost inside the *iter* phase for a GPU execution.

5.3 The ITER block

In the following, we shall analyze the computational sections inside the ITER block, namely:

- the computation of the eigenstates (eigenvalues and eigenvectors) of the Schrödinger matrices;
- the computation of the Newton-Raphson kernel for the iterative method;
- the construction of the linear system to update the guess on the potential V;
- the solution of the aforementioned linear system,

as they appear in Figure 5.
As a global picture, in Figure 6 we show the performances of the main kernels involved, for different GPUs.

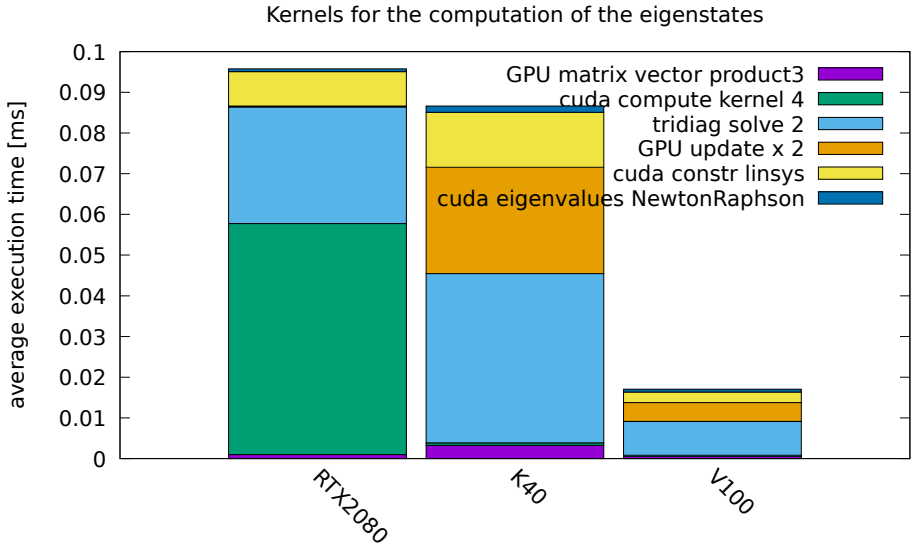

Fig. 6 The ITER computational block. Performances of the main kernels.

5.3.1 The computation of the eigenstates

First of all, we sketch in Figure 7 the speedups we have obtained for the routine diagonalizing the Schrödinger matrix.
The computation of the eigenstates relies on three sub-blocks (construction of the Schrödinger matrices; eigenvalues; eigenvectors) and six kernels in total, but only either five or three are used: at the first time step, we use a costlier but more robust method, employing five kernels.

	at step 1	at step > 1
1	`cuda_init_d_A`	`cuda_init_d_A`

2	`cuda_gershgorin`
	`cuda_initialize_eps`
	`cuda_eigenvalues_multisection`
		`cuda_eigenvalues_NewtonRaphson`

3	`tridiag_solve_2`	`tridiag_solve_2`

In Figure 8 we show how these kernels scale on different machines.

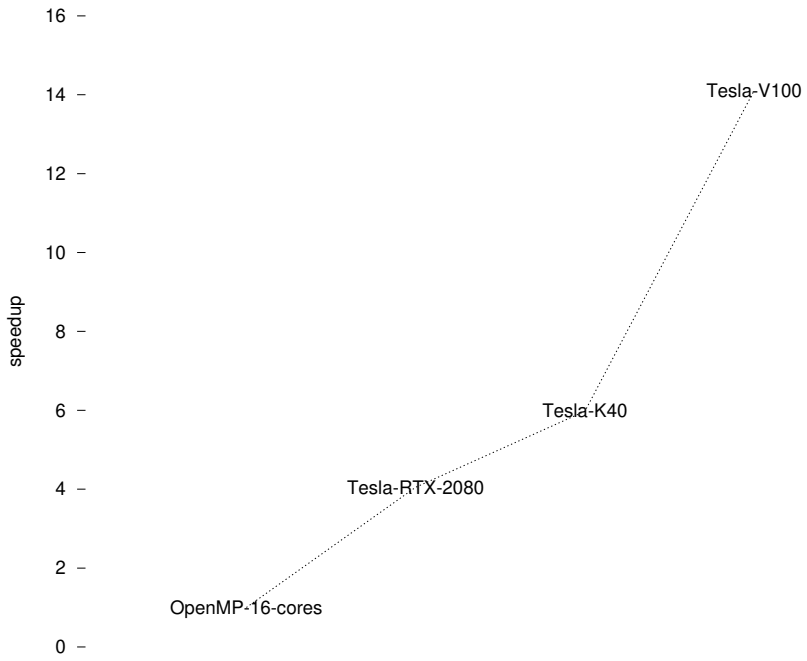

Fig. 7 Eigenstates. Speedup of the routine for the diagonalization of the Schrödinger matrix, evaluated over 10 time steps.

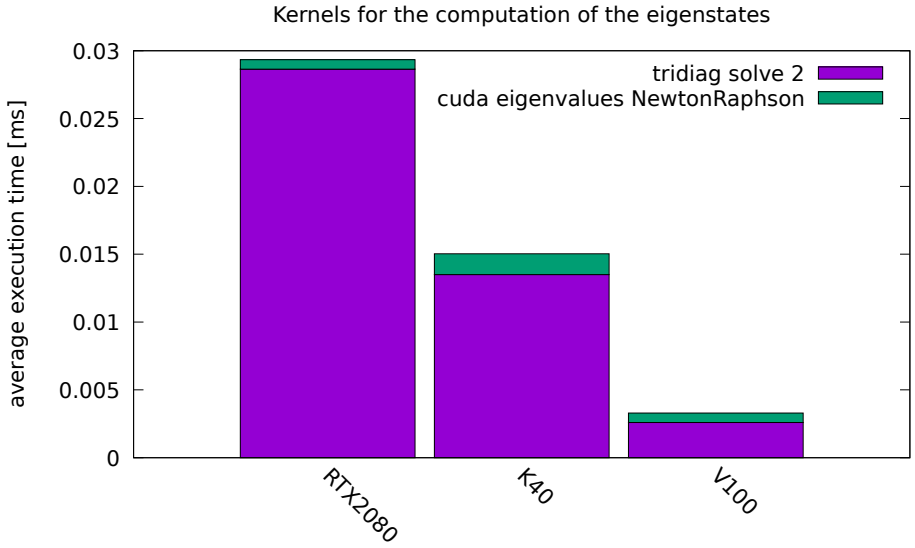

Fig. 8 Eigenstates. ...

5.3.2 The Poisson-like equation

In Figure 9 we sketch the performances of the routine for the solution of the linear system (7).

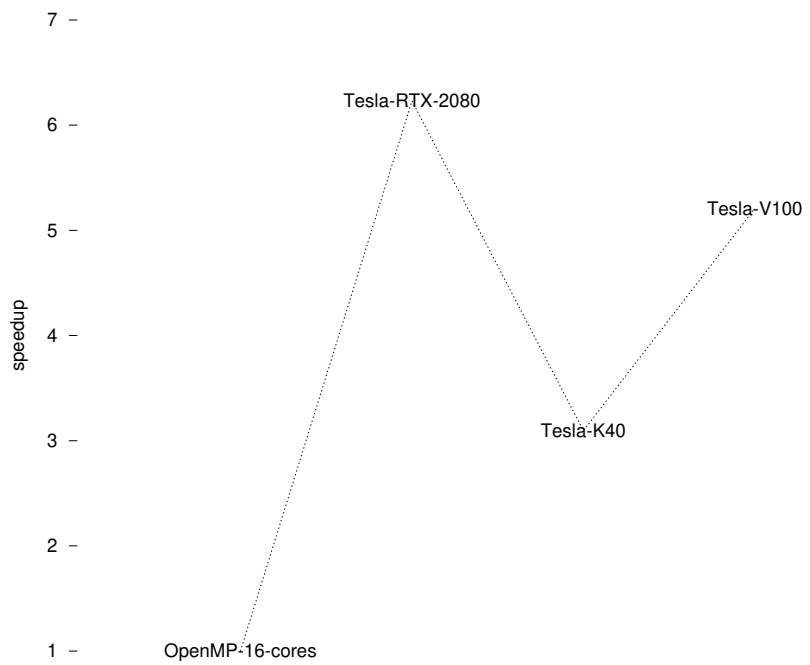

Fig. 9 Solution of the linear system. Average execution time and speedup of the routine for solution of the linear system (7).

This computational block relies on three kernels:

- `cuda_constr_linsys` builds the linear system;
- `GPU_matrix_vector_product3` intervenes in the SRJ iterative method for the solution;
- `GPU_update_x_2` intervenes in the SRJ iterative method for the solution.

In Figure 10 we show the performances of the kernels involved, for different GPUs.

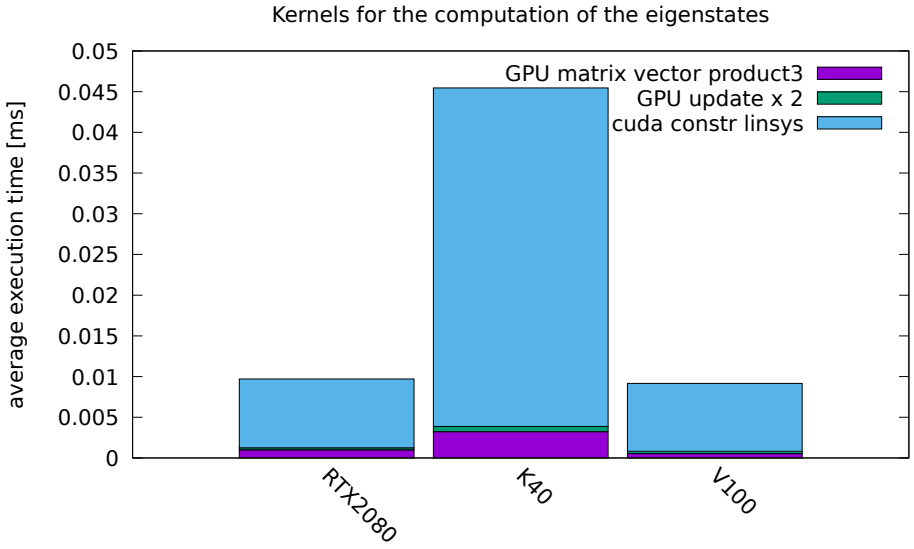

Fig. 10 Solution of the linear system....

5.3.3 The Newton-Raphson kernel

In Figure 11 we sketch the performances of the routine for the diagonalization of the Schrödinger matrix.

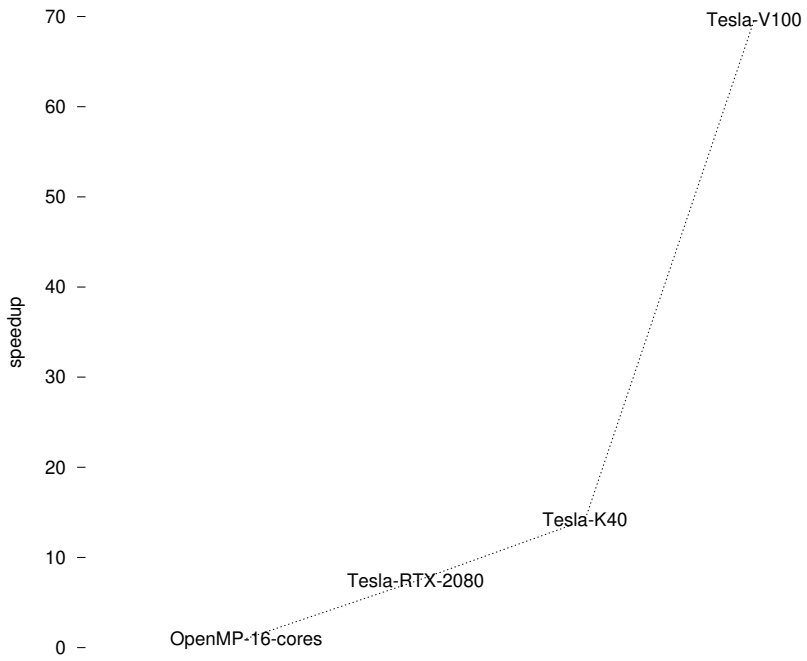

Fig. 11 Computation of the Newton-Raphson kernel. Average execution time and speedup of the routine for the computation of the Newton-Raphson kernel (9).

In Figure 10 we show the performances of the kernels involved, for different GPUs.

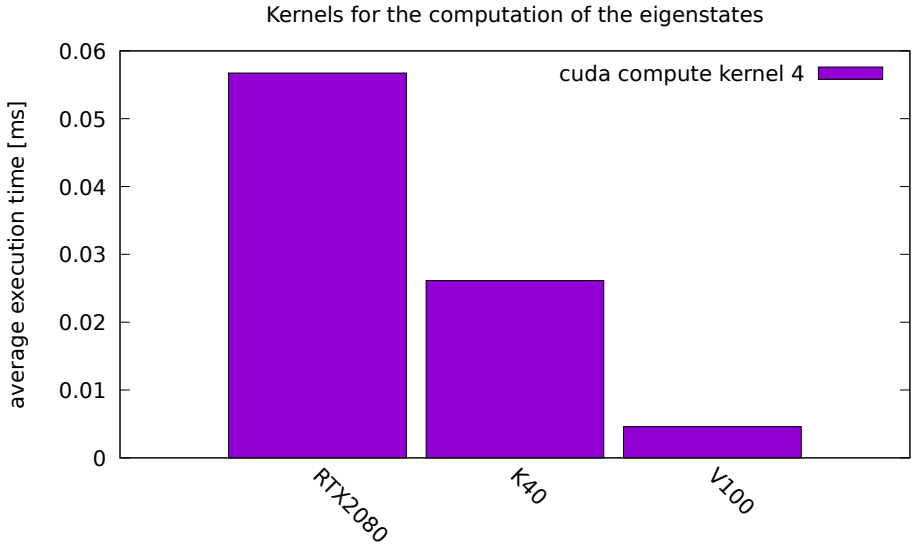

Fig. 12 Computation of the Newton-Raphson kernel....

5.3.4 Performances of the kernels

In Table 2 we report the performances of the main kernels involved in the computations of the ITER block.
The RTX-2080 graphic card does not allow for profiling, therefore the data are not reported.

QUE MAS HAY QUE PONER AQUI??

card	kernel	L1 hit	L2 hit	L2 read throughput	gld_efficiency	gld_requested_throughput	FLOPefficiency	double_precision_fu_utilization
V100	cuda_eigenvalues_NewtonRaphson	73.89%	100.00%	24.087GB/s	25.79%	22.621GB/s	0.91%	Low (1)
K40	cuda_eigenvalues_NewtonRaphson	93.20%	97.02%	13.270GB/s	6.50%	12.336GB/s	2.38%	Low (1)
V100	tridiag_solve_2	7.41%	99.63%	43.769GB/s	84.21%	40.778GB/s	40.23%	High (7)
K40	tridiag_solve_2	46.21%	94.75%	7.8179GB/s	57.15%	8.2019GB/s	43.77%	High (8)
V100	cuda_constr_linsys	83.18%	72.28%	116.11GB/s	25.71%	90.410GB/s	0.30%	Low (1)
K40	cuda_constr_linsys	64.87%	69.48%	112.91GB/s	6.34%	18.757GB/s	0.30%	Low (1)
V100	GPU_update_x_2	32.67%	96.10%	20.388GB/s	99.93%	25.619GB/s	0.04%	Low (1)
V100	cuda_compute_kernel_4	60.23%	99.39%	561.87GB/s	62.10%	891.43GB/s	43.01%	High (9)
K40	cuda_compute_kernel_4	94.68%	93.32%	34.889GB/s	24.60%	160.96GB/s	38.80%	High (8)

Table 2 Measures of the efficiency of kernels involved in updating of the eigenstates. blablabla

6 Conclusions and perspectives

In this work, a simulator of nanoscale DG MOSFETs which solves autoconsistently the Boltzmann-Schrödinger-Poisson system performing all the computing phases on a CUDA-Enabled GPU is described. The port to GPU of the iterative section solving the Schrödinger-Poisson block provides satisfactory results, as it significantly reduces the computational times of the execution on CPU. Now all the computing phases of the simulator can be fully performed on GPU and show good performances, and reasonable computational times, taking into account the huge computational cost of this deterministic solver.
Regarding the future extensions of this exploratory research, several topics can be explored. First of all, 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.
Second, no Monte-Carlo solver has been ported to GPU, at the best of our knowledge. It would be interesting to see how the performances of such numerical methods improve.
Finally, on a broader scale, we are working 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.

Acknowledgement

Francesco Vecil and J. M. Mantas acknowledge the project MTM2017-85067-P funded by the Spanish Ministerio de Economía y Competitividad (MINECO) and the European Regional Development Fund (ERDF/FEDER).
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 Departament of Universidad de Granada for the use of its computing server.

References

1. Mantas JM and Vecil F (2019) *Hybrid CUDA-OpenMP parallel implementation of a deterministic solver for ultra-shortDG MOSFETs:*, International Journal of High Performance Computing Applications 34 (1) 81–102.
2. Vecil F, Mantas JM, Cáceres MJ, Sampedro C, Godoy A and Gámiz F (2014) *A parallel deterministic solver for the Schrödinger-Poisson-Boltzmann system in ultra-short DG-MOSFETs: Comparison with Monte Carlo*, Computers and Mathematics with Applications 67 1703–1721.
3. 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*, Journal of Computational Physics 184 (2) 498–525. doi:10.1016/S0021-9991(02)00032-3.
4. Ben Abdallah N, Cáceres MJ, Carrillo JA, Vecil F (2009) *A deterministic solver for a hybrid quantum-classical transport model in nanoMOSFETs*, Journal of Computational Physics 228 (17) 6553–6571. doi:10.1016/j.jcp.2009.06.001.
5. 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.
6. Li Y, Chao T-S, Sze SM(2015) *A Novel Parallel Approach for Quantum Effect Simulation in Semiconductor Devices*, International Journal of Modelling and Simulation 23 (2) 94–102. 10.1080/02286203.2003.11442259.
7. Valín R, Sampedro R, Seoane N, Aldegunde M, García-Loureiro A, Godoy A, G/'amiz F(2012) *Optimisation and parallelisation of a 2D MOSFET multi-subband ensemble Monte Carlo simulator*. The International Journal of High Performance Computing Applications 27(4), 483–492. 10.1177/1094342012464799.
8. Espiñeira G, García-Loureiro AJ, Seoane N(2021) *Parallel Approach of Schrödinger Based Quantum Corrections for Ultrascaled Semiconductor Devices*. Journal of Computational Electronics. (In review). 10.21203/rs.3.rs-787168/v1.
9. 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. 10.3390/ma12152391.
10. Steiger S, Povolotskyi M, Park H, Kubis T(2011) *NEMO5: A Parallel Multiscale Nanoelectronics Modeling Tool*, IEEE Transactions on Nanotechnology 10 (6), 1464–1474.
11. Jourdana Clément (2011) *Mathematical modeling and numerical simulation of innovative electronic nanostructures*, Ph D. Université Paul Sabatier - Toulouse III; Università degli studi di Pavia.
12. Gummel HK(1964) *A self-consistent iterative scheme for one-dimensional steady state transistor calcu- lations*, IEEE Trans. Electron Devices, 11(10):455–465.
13. Balay S, Gropp WD, Curfman L, Smith BF (1997) *Efficient management of parallelism in object oriented numerical software libraries*, In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, Modern Software Tools in Scientific Computing, pages 163–202. Birkhäuser Press.
14. Balay S et- al. (2010). *PETSc users manual*, Technical Report ANL-95/11 - Revision 3.1, Argonne National Laboratory.
15. El-Ayyadi A, Jüngel A(2005) *Semiconductor Simulations Using a Coupled Quantum Drift-Diffusion Schrödinger-Poisson Model*, SIAM Journal on Applied Mathematics 66, No. 2, 554–572.
16. Jourdana C, Pietra P(2019) *A Quantum Drift-Diffusion model and its use into a hybrid strategy for strongly confined nanostructures*, Kinetic and Related Models 12 (1) 217–242.
17. Pietra P, Vauchelet N(2008) *Modeling and simulation of the diffusive transport in a nanoscale Double-Gate MOSFET*, Journal of Computational Electronics 7, 52–65. doi:10.1007/s10825-008-0253-z.
18. Donetti L, Sampedro C, Ruiz F.G., Godoy A., Gamiz F.(2018) *A Multi-Subband Ensemble Monte Carlo simulations of scaled GAA MOSFETs*, Solid-State Electronics 143, 49–55.
19. 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*, Semiconductor Science and Technology 21, 29–31.
20. 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*. 2015 International Conference on Simulation of Semiconductor Processes and Devices (SISPAD). 361–364. 10.1109/SISPAD.2015.7292334.
21. Mascali G, Romano V(2012), *A non parabolic hydrodynamical subband model for semiconductors based on the maximum entropy principle*, Mathematical and Computer Modelling 55 (3-4) 1003–1020.
22. 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*, Mathematical and Computer Modelling 58 (1-2) 321–343.
23. Chapman B, Jost G, van der Pas R (2008) *Using OpenMP: Portable Shared Memory Parallel Programming* The MIT Press.
24. Yang XIA, Mittal R (2014) *Acceleration of the Jacobi iterative method by factors exceeding 100 using scheduled relaxation*, Journal of Computational Physics 274 695–708. doi:10.1016/j.jcp.2014.06.010.
25. Adsuara J, Cordero-Carrión I, Cerdá-Durán P, Aloy M (2016) *Scheduled Relaxation Jacobi method: Improvements and applications*, Journal of Computational Physics 321. 369–413. doi:10.1016/j.jcp.2016.05.053.
26. 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*, Journal of Computational Physics 332 446–460. doi:10.1016/j.jcp.2016.12.020.
27. Stoer J, Bulirsch R (1991) *Introduction to Numerical Analysis*, Texts in Applied Mathematics, Springer-Verlag New York. doi:10.1007/978-0-387-21738-3.
28. Demmel JW (1997) *Applied Numerical Linear Algebra*, SIAM.
29. Abal-Kassim CA, Magoulès F(2017) *Efficient implementation of Jacobi iterative method for large sparse linear systems on graphic processing units*, The Journal of Supercomputing 73 3411–3432. 10.1007/s11227-016-1701-3.
30. Lo Sy-Shin, Philippe Bernard, Sameh Ahmed(1987) *A Multiprocessor Algorithm for the Symmetric Tridiagonal Eigenvalue Problem*, SIAM Journal on Scientific and Statistical Computing 8 (2) s155–s165. 10.1137/0908019.
31. Macintosh HJ, Warne DJ, Kelson NA, Banks JE, Farrell TW (2016) *Implementation of parallel tridiagonal solvers for a heterogeneous computing environment*, The ANZIAM Journal 56 C446–C462. .
32. Zhang Y, Cohen J, Owens J. (2010) *Fast Tridiagonal Solvers on the GPU*, Sigplan Notices - SIGPLAN. 45. 10.1145/1837853.1693472.

33. Kim Hee-Seok, Wu S, Chang Li-Wen, Hwu Wen-mei (2011) *A Scalable Tridiagonal Solver for GPUs*. Proceedings of the International Conference on Parallel Processing. 444–453. 10.1109/ICPP.2011.41.
34. Yuan Li, Grover Vinod (2018) *Using CUDA Warp-Level Primitives*. NVIDIA Devewloper Blog. .
35. Buluç A, Fineman JT, Frigo M, Gilbert J R, Leiserson CE (2009) *Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks*. ACM Symp. on Parallelism in Algorithms and Architectures.
36. William Ford (2015) *Numerical Linear Algebra with Applications, Chapter 18 - The Algebraic Eigenvalue Problem*, Academic Press (Boston). 379–438. 10.1016/B978-0-12-394435-1.00018-1.