UACEE International Journal of Computer Science and its Applications - Volume 2: Issue 1 [ISSN 2250 - 3765]

Efficient Algorithm for the Non-Equlibriums Green's Function Method for NumericalTransport-Simulations in Nanostructures on CPUs and GPUs

Jan Jacob
Institute of Applied Physics
University of Hamburg
Hamburg, Germany
jjacob@physnet.uni-hamburg.de

Lothar Wenzel, Darren Schmidt, and Qing Ruan
National Instruments
Austin, TX, USA

Vivek Amin and Jairo Sinova
Department of Physics and Astronomy
Texas A&M University
College Station, TX

***Abstract*—We present an efficient algorithm for numerical simulations of charge transportthrough semiconductor nanostructures performed withinthe Green's function formalism. The commonly used algorithmto compute the conductance of nanostructures in this formalismhas been adapted for parallel execution on both multicore computersand general-purpose graphics processing units (GPU) in a memory-efficient way to allow simulations of devices with realistic dimensions.**

***Keywords*— numerical simulation, semiconductor, Green's function, GPU, multicore, algorithm**

I. INTRODUCTION

The continuing miniaturization of circuits comes notwithout side effects; as the dimensions of such devicesdecrease, quantum effects become increasingly relevant to their operation.While these effects can be detrimental to standard CMOSdevices, they also pave the way for new kinds of devices that havethe potential for significant performance improvements by employing the spin degree of freedom of the electrons as an additional or alternative information carrier. Spin-based transistors,utilizing magnetic source and drain contacts connected to a spin-orbit coupled semiconductor channel, provide animportant example. Such devices would require less energyfor switching processes, since their channels no longer haveto be depleted completely [1]. With rather small gate voltages, one canchange the electric field across the channel and thereby the spin-orbit coupling strength. This changestheelectron'sspin-precession length and thereby its spin orientation with respect to the magnetization of thedrain electrode [2].Controllable phenomena such as this could leadto a new paradigm in information processing, whereby the spin of an electron becomes a relevant degree of freedom.

Although the physics of charge-based devices is quite well understood,physicists and engineers still face challenges regardingthe experimental realization of their spin-based cousins.While analytical predictions regarding these nanoscaledevicescan be made, it is challenging and often impossible to compare such predictions with experimental results due to the oversimplificationof the analytical model. However, numerical simulations of the transport properties of nano-structured semiconductor devicesprovide a very important bridge between analytical descriptionsand experimental results. In such simulations donor impurities, lattice imperfections, and interactions within a sample that are ordinarily inaccessible to most analytical computations can be taken into account, providing researchers with far more accurate predictions to compare with experiments. Unfortunately, while these devices are rather small for fabrication they are still huge for numerical simulations if one wants to simulate these devices with both realistic dimensions and appropriately minute grid sizes, resulting in large computational and memory load.

One of the common approaches to simulate transport in semiconductor nanostructures is the non-equilibrium Green's function (NEGF) method [3]. Other popular approaches, such as the stabilized transfer matrix algorithm [4], can be translated in terms of the NEGF method; it exhibits a similar mathematical structure and thus employs similar numerical techniques. In this work we explore optimized implementations of the Green's function method for application to parallel architectures of multi-core CPUs and their scalability over multiple threads, as well as their portability to general purpose graphics processing units (GPU).

II. MATHEMATICAL AND PHYSICS BACKGROUND

We briefly introduce the Green's function method simulatetransport in mesoscopic structures [3]. In conductors whose dimensionsare small enough such that interface resistances and the number of transverse modes an electron can occupy are delimiting factors, one must use the Landauer formula in order to calculate the samples' conductance. Essential to this formula is the transmission probability T, describing the total probability of carriers to transmit through

SEEK DIGITAL LIBRARY

the sample from one contact to the other. Overall the Landauer formula is given by

$$\mathbf{G} = \frac{2e^2}{h} T \quad (1)$$

where e is the elementary charge and h is Planck's constant. It is natural in these systems that different quantum-mechanical modes exist in which carriers can propagate through, not unlikely different channels or lanes in a highway. We define the matrix elements s_{mn} as the quantum-mechanical probability amplitudes of a carrier that begins in the m-th mode of one contact, scatters through the sample, and leaves in the n-th mode of the other contact.

If the conductor is much smaller than the phase-relaxation length, then transport is coherent and the total transmission probability can be decomposed into the sum of these amplitudes squared (i.e. with no interference). Thus one may write

$$\mathbf{G} = \frac{2e^2}{h} \sum_{m,n} T_{mn}, \quad (2)$$

where,

$$T_{mn} = |s_{mn}|^2 \quad (3)$$

Thus by determining the s-matrix of the microscopic sample, one can compute the conductance using the Landauer formula. Green's functions provide a convenient way to do this; however, the relationship between Green's functions and the s-matrix is beyond the scope of this work. Instead, after demonstrating how to calculate the Green's function for a particular sample, we will merely state the formula for T. It is useful, however, to keep in mind that physical meaning of the Landauer formula.

We begin by defining a Green's function G (not to be confused with the conductance $\mathbf{G}$), for a system governed by some Hamiltonian $\hat{H}(r)$

$$\left[E - \hat{H}(r)\right] G(r, r') = \delta(r - r'). \quad (4)$$

Essentially we have rewritten the Schrödinger equation with an added source term. From this perspective, one can imagine the Green's function to simply be the wavefunction given in terms of the position vector r where r' is a parameter describing the location of the source.

To calculate the Green's function the above concept is applied to a tight-binding model by the method of finite differences, such that

$$G(r, r') \to G_{ij} \quad (5)$$

where i and j are indices denoting different lattice positions corresponding (in the continuum limit) to r and r'. As a result the aforementioned differential equation becomes a matrix equation

$$[EI - H]G = I \quad (6)$$

It should be made clear that, regardless of the dimensionality of the system, each row or column in the above matrices stands for a particular lattice site within the entire sample. For example, a two-dimensional system containing N_x horizontal sites and N_y vertical sites is described by matrices of dimension $(N_xN_y) \times (N_xN_y)$. Therefore, in the case of two (or three) dimensional systems, one must keep track of an appropriate labeling system to denote which row or column corresponds to which lattice site.

Since we have converted the Hamiltonian from a differential operator to a matrix operator, we must introduce discretized derivative operators, given by

$$\left[\frac{dF}{dx}\right]_{x=\left(j+\frac{1}{2}\right)a} \to \frac{1}{a}\left[F_{j+1} - F_j\right] \quad (7)$$

$$\left[\frac{d^2F}{dx^2}\right]_{x=ja} \to \frac{1}{2a^2}\left[F_{j+1} - 2F_j + F_{j-1}\right] \quad (8)$$

As an example of a Hamiltonian matrix for a one-dimensional system, with a simple kinetic and potential term, one can consider,

$$H = \begin{pmatrix} \cdots & -t & 0 & 0 & 0 \\ -t & U_{-1} + 2t & -t & 0 & 0 \\ 0 & -t & U_0 + 2t & -t & 0 \\ 0 & 0 & -t & U_1 + 2t & -t \\ 0 & 0 & 0 & -t & \cdots \end{pmatrix} \quad (9)$$

where $t = \hbar^2/2ma^2$ is the so-called hopping parameter and U_i denotes the potential at each lattice site. Such a matrix can be rewritten for two or three-dimensional systems given an appropriate labeling system, as mentioned before. Written in this way, one can compute G through matrix inversion.

$$G = [EI - H]^{-1} \quad (10)$$

It should be noted that there exist two independent solutions (corresponding to different boundary conditions) for G, normally referred to as the retarded and advanced Green's functions; often an imaginary parameter is added to the energy in Eqn. 10 in order to force the solution to be one or the other. For our present purposes we shall omit this imaginary factor.

As solutions like Eqn. 10 for Hamiltonians such as Eqn. 9 only provide information about scattering within a sample, and say nothing of the sample's connection to external leads, which are paramount in calculating conductance in these regime and are always present in experiments. One normally proceeds by assuming that the sample is connected to the leads at various lattice sites, and that only the directly neighboring lattice sites within the lead itself are relevant to compute the lead's full effect on transmission. If the leads are semi-infinite, homogenous, and reflection-less, one can show that this is not an approximation, but an exact statement. We shall consider the case in which there are two leads, labeled p and q and the sample is denoted as c. Our sample shall be represented by a $N_x \times N_y$ grid, where the x-direction runs horizontal and the y-direction runs vertical. We assume that each lead is connected

fully to either vertical side of the sample, such that there are $N_y$neighboring points in each lead. One can then rewrite the Green's function in block matrix form as

$$G = \begin{pmatrix} G_c & G_{cp} & G_{cq} \\ G_{pc} & G_p & 0 \\ G_{qc} & 0 & G_q \end{pmatrix} \quad (11)$$

All carriers enter or leave the sample via G_{cp} or $G_{cq}$and propagate throughout the sample via G_c. We also include the possibility for carriers to propagate within the leads themselves via $G_p$orG_q. One can see through inspection of the block matrix that there is no direct connection between differing leads; carriers must transmit through the sample to travel between p and q. We assume the following structure for G:

$$G = \begin{pmatrix} EI - H_c & \tau_p & \tau_q \\ \tau_p^\dagger & EI - H_p & 0 \\ \tau_q^\dagger & 0 & EI - H_q \end{pmatrix}^{-1} \quad (12)$$

Note that each element in the above block matrix hasdifferent dimensions, depending on the number of latticesites corresponding to the portion they describe. We make one more assumption, namely that a carrier present in a lead may only enter the sample through a horizontally adjacent site. Then one may write

$$\left|\tau_{p(q)}\right|_{ij} = t\delta_{ij} \quad (13)$$

We now have to solve for G_c. One can show, after some algebra, that

$$G_c = [EI - H_c - \Sigma]^{-1} \quad (14)$$

where

$$\Sigma = \begin{pmatrix} t^2 g_p & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} + \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & t^2 g_q \end{pmatrix} = \Sigma_\mathrm{p} + \Sigma_\mathrm{q} \quad (15)$$

and

$$g_{p(q)} = \left[EI - H_{p(q)}\right]^{-1} \quad (16)$$

Note that the$\Sigma_{p(q)}$are $(N_xN_y) \times (N_xN_y)$matrices, while $g_p$and$g_q$are$N_y \times N_y$matrices. Equation 14 successfully describes the Green's function for a sample with two leads in terms of the hopping parameter t, the Fermi energy E, the conductor's Hamiltonian H_c, and the lead's Hamiltonians $H_{p(q)}$. All that is left, using this information, is to calculate the transmission probability. This is then calculated by

$$T = \sum_{m,n} T_{mn} = Tr\left[\Gamma_\mathrm{p} \mathrm{G_c} \Gamma_\mathrm{q} \mathrm{G_c^\dagger}\right] \quad (17)$$

where

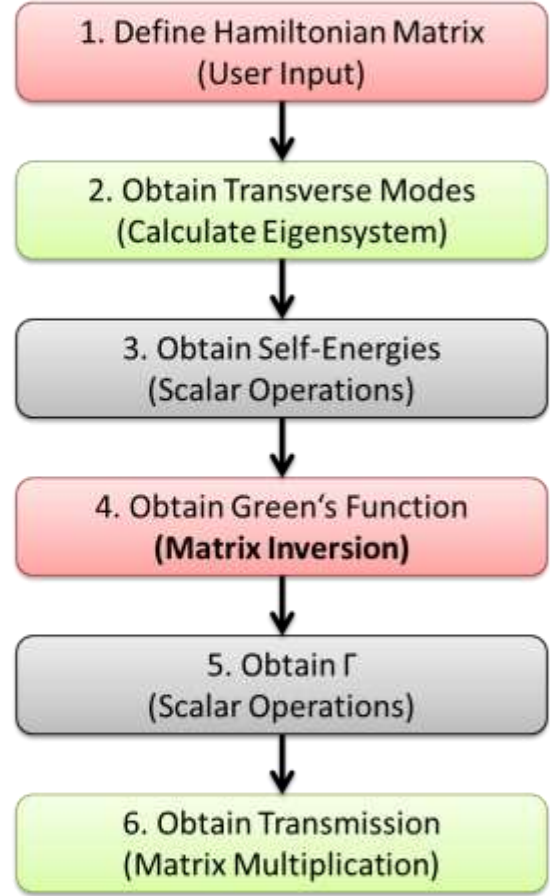

Fig.1 Flowchart for the basic Green's function algorithm. The grey steps (3.) and (5.) do not need much computational resources and do not yield high optimization potential. The yellow steps (2.) and (6.) have potential of optimization. However, their computational or memory load is that small compared to the main bottlenecks that they were not the main focus of this work. The red tasks (1.) and (4.) yield the highest potential for optimization by enhancing the memory efficiency in (1.) and developing a new algorithm for (4.).

$$\Gamma_{\mathrm{p(q)}} = \mathrm{i}\left|\Sigma_{\mathrm{p(q)}} - \Sigma_{\mathrm{p(q)}}^\dagger\right| \quad (18)$$

III. Basic Implementation

The straightforward implementation of this algorithm includes the following steps (see Fig. 1): First the potential landscape of the sample, the Hamiltonian H for the system as well as the transverse Hamiltonian H_y describing the hopping within one transversal slice are defined. In the second step the eigenvalues and vectors for H_y are determined. In step three they are then used to define the self-energies $\Sigma^A(\Sigma^R)$of the leads. In the fourth step the Green's function can be calculated with these information. In step five the Γ matrices are created. In step sixthe transmission probability is calculated from the Green's functions and the Γ matrices.

The first step contains the user input processing and does not require significant computational resources. However, the

SEEK DIGITAL LIBRARY

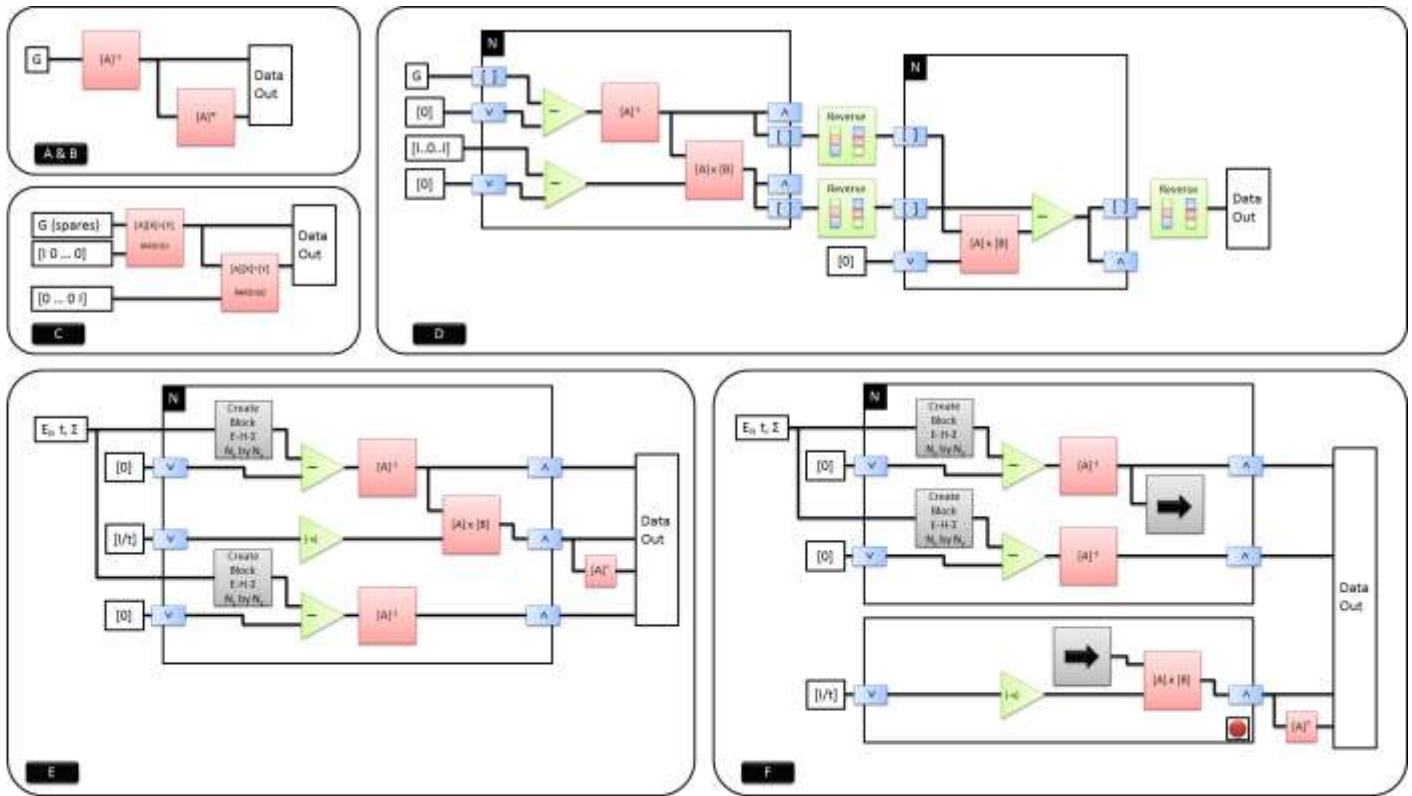

Fig.2 Visualization of the different algorithms for the matrix inversion. A&B in the upper left corner represents the basic algorithm with a direct matrix inversion to obtain the Green's function as used in Version 0 and 1. C on the left corresponds to the spares matrix PARDISO solver algorithm in Version 2. D on the upper right shows the first implementation of the block-tridiagonal solver in Version 3. E in the lower right shows the improve block-tridiagonal solver that eliminates the need to store intermediate data. F in the lower right finally shows the pipelined version of this code that makes the most efficient use of the available computational resources and has been implemented as Version 5 on the CPU and the GPU system.

matrix of the Hamiltonian$H$ is of the size $(N_xN_y) \times (N_xN_y)$, and therefore immediately gets extremely big for increasingsystemsize, causing memory issues when implemented as a dense matrix. This would limit the applicability of the concept to systems of realistic dimensions already before the calculation itself starts.

The Eigenvalue problem in the second step can take a significant share of computing resources for extremely large systems. It computes the Eigenvalues and Eigenvectors of a $N_y \times N_y$ matrix.

The calculation of the self energies $\Sigma^A$and Σ^Rin step three involves only simple scalar operations executed on the elements of the previously defined matrices.

The fourth step represents the computational bottleneck of the NEGF method and in this basic implementation heavily limits the system size as a $(N_xN_y) \times (N_xN_y)$ matrix has to be inverted in Eqn. 14. This is the most resource-demanding part of the methodandwill be addressed extensively by our optimizations below.

Creating the matrices Γ_p and $\Gamma_q$in step five only involves scalar operations on the elements of the self energies$\Sigma^A$and Σ^R. As this calculation is independent of the inversion in step four, it can be done in parallel. It is also of much lower resource demand as the inversion step.

The final step six obtains the transmission and reflection coefficients from the traces of several matrix products as described by Eqn. 17. The four matrices that have to be multiplied for each trace are of $(N_xN_y) \times (N_xN_y)$ size.

IV. OPTIMIZATIONS

A. Analysis of the basic implementation

1) Problem 1: Memory efficiency of the matrix H: TheHamiltonianmatrix H is of the size$(N_xN_y) \times (N_xN_y)$. As it grows fastwith system size this becomes the limiting memory factor already for very small systems.However, the matrix allows optimization ofthe memory performance by making use of its sparsity:

$$H = \begin{pmatrix} A & B & 0 & \cdots & C & 0 & 0 \\ B & A & B & \cdots & 0 & C & 0 \\ 0 & B & A & \cdots & 0 & 0 & C \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ C & 0 & 0 & \cdots & A & B & 0 \\ 0 & C & 0 & \cdots & B & A & B \\ 0 & 0 & C & \cdots & 0 & B & A \end{pmatrix} \quad (19)$$

where $A = 4tI + V(x, y)$, $B = C = -tI$, with the hopping parameter $t = 1/2a^2$ and the potential energy at a given site $V(x, y)$. Using the extreme sparsity of this matrix significantly enhances the memory performance.The same is true for the transverse Hamiltonian, that has aneven simpler structure:

$$H = \begin{pmatrix} \ddots & B & 0 & 0 \\ B & D & B & 0 \\ 0 & B & D & B \\ 0 & 0 & B & \ddots \end{pmatrix} \quad (20)$$

where$D = 2tI$. However, even a sparse representation ofthe matrix still grows dramatically with the system size. Therefore it is most efficient to create only $N_y \times N_y$ matrix blocks directly, when they are needed in the algorithm.

2) Problem 2: The Eigenvalue solver:

To obtain the modes and wave vectors of the system, we have to solve the Eigenvalues problem for the transverse Hamiltonian H_y. The Eigenvalues represent the modes and the Eigenvector matrix yields the wave functions for the self energy matrix. While there is probably potential to optimize the algorithm for the Eigenvalue solver, it is even more advantageous to completely remove the numerical Eigenvalue problem by solving it analytically. The analytical solution to the Eigenvalue problem is a simple one-dimensional quantum well problem and the solution is known. So we do not focus on any numerical optimizations of this part and instead suggest to replace it with its analytic solution.

3) Problem 3: Matrix Inversion:

As most of the computation time is spent on the matrix inversion to determinethe Green's function in Eqn. 14 our main focus was tooptimize this part of the code. There are several ways to avoid the directinversion of the full matrix to obtain$G$. One approach is to use blockwise inversion. This improves the performance over a direct inversion and can be applied recursively. However, it is more advantageous to apply the blockwise inversion only later in optimized codesto improve the performance of remaining inversions ofsmaller sub-matrices, and to first utilizethe special structure of the matrix to enhance the performance and parallelism. The optimizations aredescribed in detail in Sec. IV-C through IV-F.

4) Problem 4: The final matrix multiplication:

To obtain the transmission and reflection coefficients for the two leads of our system eight multiplications of $N_y \times N_y$matrices are necessaryaccording to Eqn. 17. The structure of the algorithm allows to execute four of them in parallel. The other four need the results of the first set of multiplications. After receiving these results they can also be executed in parallel. By making use of high-performance matrix multiplication functions this task reaches a high level of parallel execution (see below). As the computational load of this part is small compared to the inversion no further optimizations have been done to this part.

B. First Optimization: optimized linear algebra functions

National Instruments developed a LabVIEW High PerformanceAnalysis Library (HPAL) [5]. HPAL exposes linearalgebra functions from Intel's Math Kernel Library (MKL)[6]from within LabVIEW. These functions are optimized forexecution on multi-core processors and designed to workwhen the input matrices are extremely large.We replace the functions for matrix multiplication andmatrix inversion. The benchmark results in Section VI showthat we are still limited by the inefficient use of memory byusing dense matrices.

C. Second optimization: sparse matrices

To take advantage of the sparsity of the matrices, we employed the sparse matrix functions in the HPAL library replacing the inversion by the PARDISO direct sparse linear solver [7], [8]. The benchmarks show that the PARDISO solver is faster than a dense solver, but still has memory issues above $N_x = N_y = 700$ as the PARDISO solver generates a lot of intermediate data.

D. Third optimization: Block-Tridiagonal solver

The matrix representing the Hamiltonian$H$ has a block tri-diagonal structure, suggesting to use the generalized Thomas algorithm [9] as the replacement of the PARDISO solver in the previous section. Assuming that the block tri-diagonal linear system is

$$\begin{bmatrix} A_1 & B_1 & & & 0 \\ C_1 & A_2 & B_2 & & \\ & C_2 & A_3 & \ddots & \\ & & \ddots & \ddots & B_{N_x-1} \\ 0 & & & C_{N_x-1} & A_{N_x} \end{bmatrix} \begin{bmatrix} X_1 \\ X_2 \\ X_3 \\ \vdots \\ X_{N_x} \end{bmatrix} = \begin{bmatrix} Y_1 \\ Y_2 \\ Y_3 \\ \vdots \\ Y_{N_x} \end{bmatrix} \quad (21)$$

whereA_k, B_k,and$C_k$are all $N_y \times N_y$blocks. The solution can be computed by the following two steps.

Step 1: for k from 1 to N_x

$$B_k = (A_k - C_{k-1}B_{k-1})^{-1}B_k \quad (22)$$

$$\hat{Y}_k = (A_k - C_{k-1}B_{k-1})^{-1}(Y_k - C_{k-1}\hat{Y}_{k-1}) \quad (23)$$

Step 2: for k from $N_x$to 1

$$X_k = \hat{Y}_k - B_k X_{k+1} \quad (24)$$

This algorithm takes advantage of the sparsity of the matrix to achieve significant speedup, but still requires relatively large memory. There are $3N_yN_y \times N_y$complex matrices generated between step 1 and step 2. For $N_x = N_y = 1000$, storing these matrices would need 48 GB RAM and forcing slow data exchange with hard disk media.

E. Fourth optimization: Improved Block-Tridiagonal solver

Since we only need the four $N_y \times N_y$corners of the inverse matrix, we are solving two linear systems with the right hand sides

$$[I_{N_y} \quad 0 \quad \ldots \quad 0]' \quad (25)$$

and

$$[0 \quad \dots \quad 0 \quad I_{N_y}]' \qquad (26)$$

Systemsize (N_x=N_y)	Matrixsize (N_x•N_y)	Version 0 direct inversion	Version 1 HPAL library	Version 2 Sparse matrices	Version 3 BT-solver A	Version 4 BT-solver B	Version 5 Pipelining
(sites)	(elements)	(seconds)	(seconds)	(seconds)	(seconds)	(seconds)	(seconds)
10	100	0.007	0.017	0.002	0.001	0.001	0.001
20	400	0.192	0.407	0.006	0.004	0.004	0.003
30	900	2.096	2.684	0.013	0.016	0.013	0.008
40	1600	11.745	13.261	0.026	0.038	0.024	0.017
50	2500	49.714	47.328	0.054	0.081	0.048	0.038
60	3600	148.369	138.163	0.088	0.154	0.072	0.058
70	4900	346.183	339.151	0.134	0.215	0.114	0.094
80	6400	769.706	730.780	0.201	0.371	0.151	0.127
90	8100	1647.595	1543.517	0,241	0.468	0.214	0.187
100	10000	2964.965	2949.634	0.357	0.715	0.279	0.236
200	40000	o.o.m.	o.o.m.	2.194	8.662	2.248	1.765
300	90000	o.o.m.	o.o.m.	7.560	42.750	7.804	5.767
400	160000	o.o.m.	o.o.m.	18.323	130.317	20.709	14.643
500	250000	o.o.m.	o.o.m.	39.306	311.673	57.965	33.411
600	360000	o.o.m.	o.o.m.	72.519	595.367	102.021	61.147
700	490000	o.o.m.	o.o.m.	125.120	o.o.m.	168.005	109.006
800	640000	o.o.m.	o.o.m.	o.o.m.	o.o.m.	263.918	191.874
900	810000	o.o.m.	o.o.m.	o.o.m.	o.o.m.	389.083	297.420
1000	1000000	o.o.m.	o.o.m.	o.o.m.	o.o.m.	538.907	422.620

Table 1 Summary of the benchmark results for the CPU-based algorithms. Version 0 is the original inversion algorithm. Version 1 uses the optimized LabVIEW High-Performance Conuting libraries. Version 2 makes use of the matrices' sparsity. Version 3 is the first implementation of the block-tridiagonal solver. Version 4 is the optimized block-tridiagonal solver. Version 5 is the optimized block-tridiagonal solver with pipelining for improved thread utilization.
(o.o.m. stands for out of memory – this benchmark could not be performed on the test machine)

where I_{N_y} is an $N_y \times N_y$ identity matrix. We are only interested in the first and last blocks in the solutions. The last block of each linear system is already computed after the first step in the Thomas Algorithm. Thus, we propose another method to compute the first block.

Denote

$$K = \begin{bmatrix} 0 & & I_{N_y} \\ & I_{N_y} & \\ \ddots & & \\ I_{N_y} & & 0 \end{bmatrix} \qquad (27)$$

where K satisfies $K^T = K$ and $K^2 = I$. Furthermore, if

$$A = \begin{bmatrix} A_1 & B_1 & & & 0 \\ C_1 & A_2 & B_2 & & \\ & C_2 & A_3 & \ddots & \\ & & \ddots & \ddots & B_{N_x-1} \\ 0 & & & C_{N_x-1} & A_{N_x} \end{bmatrix} \qquad (28)$$

then

$$KAK = \begin{bmatrix} A_{N_x} & B_{N_x} & & & 0 \\ C_{N_x} & A_{N_x-1} & B_{N_x-1} & & \\ & C_{N_x-1} & A_{N_x-2} & \ddots & \\ & & \ddots & \ddots & B_2 \\ 0 & & & C_2 & A_1 \end{bmatrix} \qquad (29)$$

Since $(KAK)^{-1} = KA^{-1}K$, the upper left(right) corner of A^{-1} is equal to the lower right(left) corner of $(KAK)^{-1}$. The first step of Thomas algorithm with KAK would thus give the upper left(right) corner of A^{-1}. The new algorithm we propose saves memory because it does not go through the second step. Although the algorithm introduces an extra matrix inversion by going through the first step twice with A and KAK separately, the extra calculation could be compensated by parallelization on multi-core machines. The benchmark results in Section VI show that this algorithm is much faster and can handle very large grid sizes.

F. Fifth optimization: Pipelined Block-Tridiagonal solver

To further improve the performance by making use ofparallel architectures we pipelined sequential linear algebracalculations. By adjusting each group of operations to haveroughly the same complexity, we ensured a constant highlevel of utilization on all available cores during the fullinversion algorithm. As can be seen in Fig. 2E the algorithm of Sec. IV-E consists of two inversions and one matrix

SEEK
DIGITAL LIBRARY

multiplication per iterations. The two inversions are independent of each other and can be executed in parallel. However the additional multiplication relies on the results of one of the inversions. On the other hand the result of this multiplication is not needed to calculate either of the two inversions of the next iteration. This means that during the execution of the two inversions a high degree of parallel execution is already reached. But during the execution of the multiplication the second branch of the code that only contains the inversion is idle and waits for the unrelated result of the serial multiplication of the other code branch.

We eliminated this remaining bottleneck by employing the technique of pipelining. The result of the upper inversion is fed to a queue as can be seen in Fig. 2F. Now both branches of the code in the for-loop only contain one matrix inversion and – as the two matrices to be inverted are of the same size and complexity – have the roughly the same execution time. The queued matrices are fed to the separate loop in Fig. 2F where the multiplication is executed. This task is now executed in parallel to the for-loop. While the for-loop processes the inversions for step *n*, the separate while loop executes the

Table 2 Benchmark results for the GPU implementation of the pipelined and optimized block-tridiagonal matrix inversion solver.

multiplication for the *n*-1. This ensures much better utilization of the available parallel computing resources on modern multi-core platforms. At the same time the memory usagestays below 3 GB for system of up to $N_x = N_y = 1000$.

V. IMPLEMENTATION ON GPUS

As the optimized pipelined block-tridiagonal solver stillrepresents the most demanding part of our code, we furtherimproved its performance by employing GPUs. As almostexclusively matrix multiplications and inversions have tobe performed and several operations are done in parallel,GPUs yield a high performance potential. The other partsof the code focus on pre- and post-processing steps whichlack computational complexity. Therefore, these processes are executed exclusivelyon the host processor cores. We used a prototype of theLabVIEW GPU Analysis Toolkit for the implementation onGPUs [10]. The small memory footprint of the optimizedblock-tridiagonal solver allows us to download the data for the entireproblem to the GPU and invoke the solver on the GPUdevice, retrieving only the final results.

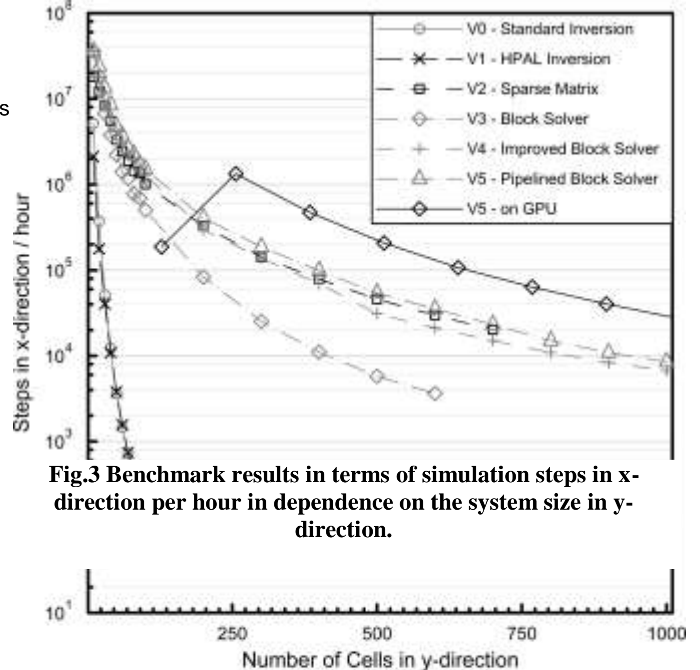

Fig.3 Benchmark results in terms of simulation steps in x-direction per hour in dependence on the system size in y-direction.

This minimizes communicationbetween host and GPU during the most criticalprocessing time. The efficient memory structure also allowsthe host to execute multiple independent simulation steps(i.e. as part of a sweep of e.g. potential or Fermi Energy) inparallel if more than one GPU is available.

VI. BENCHMARKS

We ran code implementing the direct inversion of the Green's function matrix (Version 0) and the different optimizations of Sec. IV-B through IV-F (Version 1 through 5) on an IBM idataplex dx360 M3 workstation [11] with two Intel Xeon X5650 six-core processors, running at 2.67 GHz, 48 GB random access memory, and two NVIDIA Tesla M2050 GPUs with 3 GB random access memory and 448 CUDA computing cores each [12]. All code was written in LabVIEW 2011 using functionality provided by the High Performance Analysis Library (HPAL) and the GPU Analysis Toolkit. Internally, HPAL called Intel's Math Kernel Library (MKL) v10.3 for execution on the CPU's multiple cores. The GPU Analysis Toolkit invoked routines from NVIDIA's CUDA Toolkit v4.0 and CUBLAS libraries to execute code on the Tesla GPUs. The benchmarks were performed with a 64-bit version of LabVIEW 2011 running under Windows 2008 Server Enterprise Edition. The driver for the NVIDIA Tesla GPUs was set to TCC mode to allow remote access to the machines via the Windows Remote Desktop Client.

Systemsize (N_x=N_y)	**Matrixsize (N_x•N_y)**	**GPU Pipelined BT-Solver**
(sites)	(elements)	(seconds)
128	16384	2.463
256	65536	0.691
384	147456	2.936
512	262144	8.887
640	409600	21.255
768	589824	43.610
896	802816	80.244
1024	1048576	136.685
1280	1638400	332.707
1536	2359296	688.338
1792	3211264	1272.800
2048	4194304	2170.260
2560	6553600	5290.440
3072	9437184	10964.600
3584	12845056	20297.700
4096	16777216	34616.500
5120	26214400	84462.700

Results from the CPU-based implementations are shown in TABLE I. Results for the code in version 5 which executed primarily on NVIDIA's Tesla M2050 GPUs are shown in TABLE II. The results include just the execution of the inversion algorithm described in Section IV-F. The initialization and post-processing are not taken into account as they represent just a fraction of the computation time. However, the presented benchmarks include the time for transferring the initial data to the GPUs and to retrieve the final results from them.

To visualize the performance of the different implementations we summarized the results and show the number of system slices along the *x*-directionthat can be simulated per hour on a single node or a singleGPU in Fig. 3. These timings are dependent on the number of system sites in the *y* direction. This number gives a good description of the performance related to the system size and shows the applicability of our CPU and GPU-based implementations to systems with realistic dimensions.

While the information is given for a two-dimensional system, where the transversal slice is one-dimensional, the same holds for three dimensional systems, where the number of sites is the product of height and width of the system.

VII. CONCLUSION AND OUTLOOK

Transport simulations in semiconductor nanostructures rely on the Green's function algorithm. Direct implementationsof this algorithm designed to obtain accurate results for a realistic device size using a sufficiently small grid spacing yield gigantic matrices which then need to be inverted. The problemsize coupled with the required dense matrix computations make such a solution already impractical for relatively small systems.

Our optimizedimplementations avoid the massive matrix sizes by exploiting the underlying sparse structure using a block-diagonal solver to reduce memory load from $(N_xN_y) \times (N_xN_y)$ matrices to $N_y \times N_y$ matrices. By employing pipelining we further enhanced the parallelism of the algorithm and balanced the computational load between parallel threads on different cores or devices maximizing performance. The efficient use of memory allows implementing the whole matrix inversion algorithm on a NVIDIA Tesla M2050 GPU. The calculation is done without transferring data between the host and the GPU during the calculation.

With the above summarized techniques we were able to increase the system size by a factor of 100 compared to the primitive algorithm and even beyond (which is then beyond the scope of the intended simulations). At the same time we were able speed up the calculation of the transmission function on the host computer by a factor of 12,500 demonstrating the high efficiency of our algorithm. The implementation of the inversion algorithm on the GPUs yields a further performance gain by a factor of three. Taking into account the fact that a second simulation step can be executed in parallel on the second NVIDIA Tesla M2050 GPU the performance enhancement per IBM idataplex dx360 M3 computing node by the GPU implementation is a total factor of six.

The simulation of the transport in dependence on one varied parameter (e.g. gate voltage) with 1000 steps for a device of 1 µm by 1 µm and a grid spacing of 1 nm takes a total time of approximately 19 hours. Given the large system size and the fine grid together with a high resolution for the swept parameters we reach a very high performance with our algorithm. The option of further parallelization of the simulation by distributing different steps of the sweep not only over the two GPUs of one node but also over several nodes, allows even higher performance of the presented algorithm for extreme precise simulations of transport in nanostructures with realistic dimensions in very fast computing times.

Having demonstrated the feasibility of these simulations in general, we are now expanding the code to three-dimensional structures and multiple bands for electron and hole transport. The addition of multiple bands increases the size of the matrices to $(N_xN_yN_s) \times (N_xN_yN_s)$, where Ns is the number of bands taken into account. The more demanding step is the implementation of three-dimensional systems, where each "slice" of the system is no longer represented by a matrix of $N_y \times N_y$ elements, but by a matrix of $(N_yN_z) \times (N_yN_z)$. It can easily be seen that the matrix size immediately reaches extreme dimensions bringing new challenges to the forefront. Therefore we will explore additional techniques to combine the resources of multiple GPUs within one computing node as well as to combine multiple nodes to calculate the transport properties of complex three-dimensional nanostructures.

ACKNOWLEDGMENT

This work was supported in part by the Deutsche Forschungsgemeinschaft via the Graduiertenkolleg 1286 "Functional Metal-Semiconductor Hybrid Systems"and Project Me916/11-1 "Spin-Filter Cascades in InAsHeterostructures",the Free and Hanseatic City of Hamburg via the Center of Excellence"Nanospintronics", the Office of Naval Research via ONR-N00014110780,and the National Science Foundation by NSF-MRSEC DMR-0820414, NSFDMR-1105512, NHARP

REFERENCES

[1] S. Datta and B. Das, Appl. Phys. Lett., vol. 56, no. 7, p. 665, 1990.

[2] J. Wunderlich, B.-G. Park, A. C. Irvine, L. P. Zrbo, E. Rozkotov, P. Nemec, V. Novk, J. Sinova, and T. Jungwirth, Science, vol. 330, no. 6012, pp. 1801–1804, 2010.

[3] S. Datta, Electronic Transport in Mesoscopic Systems. Cambridge University Press, 1999.

[4] T. Usuki et al., Phys. Rev. B, vol. 50, pp. 7615–7625, 1994.

[5] LabVIEW 2010 High Performance Analysis Library. National Instruments. [Online]. Available: https://decibel.ni.com/content/docs/DOC-12086

[6] Intel Math Kernel Library. Intel. [Online]. Available: http://software.intel.com/en-us/articles/intel-mkl/

[7] O. Schenk, A. Waechter, and M. Hagemann, Journal of Computational Optimization and Applications, vol. 36, no. 2-3, pp. 321–341, 2007.

[8] O. Schenk, M. Bollhoefer, and R. Roemer, SIAM Review, vol. 50, pp. 91–112, 2008.

[9] W. Press, S. Teukolsky, W. Vetterling, and B. Flannery, Numerical Recipes in C. Cambridge University Press, 1999, vol. 123, p. 50.

[10] LabVIEW GPU Analysis Toolkit. National Instruments. [Online]. Available: www.ni.com

[11] idataplex dx360 M3 Datasheet. IBM. [Online]. Available: http://www-03.ibm.com/systems/x/hardware/idataplex/dx360m3/index.html

[12] Tesla M2050 GPGPU Datasheet. NVIDIA. http://www.nvidia.com/docs/IO/105880/DS-Tesla-M-Class-Aug11.pdf.

SEEK
DIGITAL LIBRARY