Proceedings of the International MultiConference of Engineers and Computer Scientists 2012 Vol II,
IMECS 2012, March 14 - 16, 2012, Hong Kong

Numerical Transport Simulations in Semiconductor Nanostructures on CPUs and GPUs

Jan Jacob, Lothar Wenzel, Darren Schmidt, Qing Ruan, Vivek Amin, and Jairo Sinova

***Abstract*—We present numerical simulations of charge transport through semiconductor nanostructures performed within the Green's function formalism. The commonly used algorithm to compute the conductance of nanostructures in this formalism has been adapted for parallel execution on both multicore computers and general-purpose graphics processing units (GPU). Additionally, memory utilization has been optimized so that larger systems may be simulated, enabling realistic device sizes.**

***Index Terms*—numerical simulation, Green's function, GPU, multicore, algorithm.**

I. Introduction

THE continuing miniaturization of circuits comes not without side effects; as the dimensions of such devices decrease, quantum effects become relevant to their operation. While these effects can be detrimental to standard CMOS devices, they also pave the way for new devices that have the potential for performance improvements. Spin-based transistors, utilizing magnetic source and drain contacts along with a spin-orbit coupled semiconductor channel, provide an important example. Such devices would require less energy for switching processes, since their channels no longer have to be depleted completely [1]. By using gates, one can influence an electron's spin precession length and change its spin orientation with respect to the magnetization of the drain electrode [2]. Such controllable phenomena could lead to a new spin-based information processing paradigm.

Although the physics of charge-based devices is well understood, physicists and engineers still face challenges regarding the experimental realization of their spin-based cousins. While analytical predictions regarding these nanoscale devices can be made, direct comparison with experiments is challenging and often impossible due to the oversimplification of the analytical model. However, numerical transport simulations of nano-structured semiconductor devices provide a very important bridge between analytical descriptions and experimental results. Such simulations can account for donor impurities, lattice imperfections, and interactions within a sample that are ordinarily inaccessible to most analytical computations, providing researchers with more accurate predictions to compare with experiments. Unfortunately, it is extremely challenging to simulate these devices with both realistic dimensions and appropriately minute grid sizes, due to the 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 complementary numerical techniques. In this work we explore optimized implementations of the Green's function method and their scalability over multiple threads, as well as their portability to general purpose graphics processing units (GPU).

Manuscript received December 30, 2011; revised December 30, 2011. 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 InAs Heterostructures", 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, NSF-DMR-1105512, NHARP

J. Jacob is with the Institute of Applied Physics, University of Hamburg, Germany, e-mail: jjacob@physnet.uni-hamburg.de

L. Wenzel, D. Schmidt, and Q. Ruan are with National Instruments, Austin, TX, USA e-mail: lothar.wenzel@ni.com, darren.schmidt@ni.com, qing.ruan@ni.com

V. Amin and J. Sinova are with the Department of Physics and Astronomy, Texas A& M University, College Station, TX, USA, e-mail: aminvp@physics.tamu.edu, sinova@physics.tamu.edu

II. Mathematical and physics background

We briefly introduce the Green's function method to simulate transport in mesoscopic structures [3] whose dimensions are smaller than the coherence length. Here the conductance must be calculated by the Landauer formula, where the transmission probability T describes the transmission probability of carriers through the sample from one contact to the other. Overall the Landauer formula is given by

$$\mathbf{G} = \frac{2e^2}{h} T \tag{1}$$

where e is the elementary charge and h is Planck's constant. In these systems there exist different quantum-mechanical modes in which carriers can propagate through. We define the matrix elements s_{mn} as the probability amplitudes for a carrier entering the sample in the m-th mode of one contact to leave 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. Thus one may write

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

where,

$$T_{mn} = |s_{mn}|^2 . \tag{3}$$

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. We begin by defining a Green's function, for a system governed by some Hamiltonian $\hat{H}(r)$

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

Essentially we have rewritten the Schrödinger equation with an added source term. From this perspective, one can imagine

ISBN: 978-988-19251-9-0
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

the Green's function to simply be the wave function given in terms of the position vector r that describes 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

$$G(r, r') \rightarrow G_{ij}, \tag{5}$$

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

$$[EI - H]\, G = I, \tag{6}$$

where each row or column in the above matrices stands for a particular lattice site within the entire sample. Therefore the matrix for a two-dimensional system of N_x horizontal sites and N_y vertical sites is of dimension $N_xN_y \times N_xN_y$. Since we have converted the Hamiltonian from a differential operator to a matrix operator, we must introduce discretized derivative operators, given by

$$\left[\frac{\mathrm{d}F}{\mathrm{d}x}\right]_{x=\left(j+\frac{1}{2}\right)a} \rightarrow \frac{1}{a}\left[F_{j+1} - F_j\right] \tag{7}$$

$$\left[\frac{\mathrm{d}^2F}{\mathrm{d}x^2}\right]_{x=ja} \rightarrow \frac{1}{a^2}\left\{F_{j+1} - 2F_j + F_{j-1}\right\}. \tag{8}$$

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

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

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. Written in this way, one can compute G through matrix inversion.

$$G = \left[EI - H\right]^{-1}. \tag{10}$$

It should be noted that there exist two independent solutions 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, the concept has to be expanded to include the leads. One normally proceeds by assuming that the sample is connected to the leads at various lattice sites, and that only the directly neighboring sites within the lead itself are relevant to compute the lead's full effect on transmission. If the leads are semi-infinite, homogeneous, and reflectionless, one can show that this is 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. Each lead shall be connected fully to either vertical side of the sample, giving N_y neighboring points in each lead. One can then rewrite the Green's function in block matrix form as

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

All carriers enter or leave the sample via G_{cp} or G_{cq} and propagate throughout the sample via G_c. Propagation within the leads themselves is included via G_p or G_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:

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

Note that each element in the above block matrix has different dimensions, depending on the number of lattice sites corresponding to the portion they describe. By assuming that a carrier may only enter the sample through a site horizontally adjacent to the lead, one may write

$$[\tau_{p(q)}]_{ij} = t\delta_{ij}. \tag{13}$$

and solve for G_c. One can show, after some algebra, that

$$G_c = \left[EI - H_c - \Sigma\right]^{-1}, \tag{14}$$

where

$$\Sigma = \begin{pmatrix} t^2 g_p & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} + \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & t^2 g_q \end{pmatrix} = \Sigma_p + \Sigma_q \tag{15}$$

and

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

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 describes the Green's function in terms of the hopping parameter t, the Fermi energy E, the conductor's Hamiltonian H_c, and the lead's Hamiltonians $H_{p(q)}$. The transmission probability is then calculated by

$$T = \sum_{m,n} T_{mn} = \mathrm{Tr}\left[\Gamma_p G_c \Gamma_q G_c^\dagger\right], \tag{17}$$

where

$$\Gamma_{p(q)} = i[\Sigma_{p(q)} - \Sigma_{p(q)}^\dagger]. \tag{18}$$

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. They are used in step three to define the self-energies of the leads. With these information the Green's function can be calculated in the fourth step. Step five creates the Γ matrices to calculate in the sixth step the transmission probability.
The first step contains the user input processing and does not require significant computational resources. However, the matrix H is of the size $(N_xN_y) \times (N_xN_y)$, and gets extremely big for large systems, causing memory issues

ISBN: 978-988-19251-9-0
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

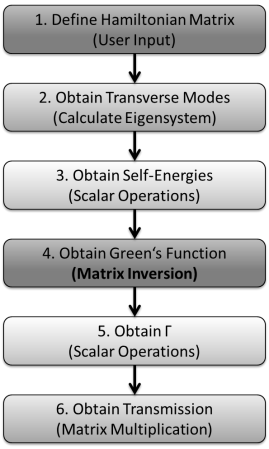

Fig. 1. Flowchart for the basic Green's function algorithm

when implemented as a dense matrix. The Eigenvalue problem in the second step of the size $N_y \times N_y$ can take a significant share of computing resources for extremely large systems. The self energies Σ^A and Σ^R are created by simple scalar operations in step three. The fourth step represents the bottleneck of this basic implementation due to the inversion of a $(N_xN_y) \times (N_xN_y)$ matrix in Eqn. 14, which will be addressed extensively below. Creating the Γ matrices in step five only involves scalar operations on the elements of Σ^A and Σ^R and can be done in parallel to the more demanding step four. The final step six obtains the transmission and reflection coefficients from the traces given by Eqn. 17 and includes the product of four matrices of $(N_xN_y) \times (N_xN_y)$.

IV. Optimizations

A. Analysis of the basic implementation

a) Problem 1: Memory efficiency of the matrix H: The matrix H is of the size $(N_xN_y) \times (N_xN_y)$. As it grows fast with system size this becomes the limiting memory factor. However, the structure of the matrix allows optimization of the memory performance:

$$H_{N_xN_y \times N_xN_y} = \begin{pmatrix} \ddots & Y & 0 & 0 \\ Y & X & Y & 0 \\ 0 & Y & X & Y \\ 0 & 0 & Y & \ddots \end{pmatrix}, \text{ with} \quad (19)$$

$$X_{N_y \times N_y} = \begin{pmatrix} \ddots & B & 0 & 0 \\ B & A & B & 0 \\ 0 & B & A & B \\ 0 & 0 & B & \ddots \end{pmatrix}, \quad (20)$$

$$Y_{N_y \times N_y} = \text{diag}\,[C] \quad (21)$$

where $A = 4t + V(x,y)$, $B = C = -t$, with the 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 an even simpler structure:

$$H_y = \begin{pmatrix} \ddots & B & 0 & 0 \\ B & D & B & 0 \\ 0 & B & D & B \\ 0 & 0 & B & \ddots \end{pmatrix}, \quad (22)$$

where $D = 2t$. However, even a sparse representation of the 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.

b) Problem 2: The Eigenvalue solver: The Eigenvalues of H_y 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 Eigenproblem by solving it analytically as it 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.

c) Problem 3: Matrix inversion: As most of the computation time is spent on the matrix inversion to determine the Green's function in Eqn. 14 our main focus was to optimize this part. There are several ways to avoid the direct inversion of the full matrix G. One approach is to use blockwise inversion. While this improves the performance over a direct inversion and can be applied later in optimized codes to improve the performance of the remaining inversions of smaller sub-matrices, it is more advantageous to first utilize the special structure of the matrix. The optimizations are described in detail in Sec. IV-C through IV-F.

d) Problem 4: The final matrix multiplication: For Eqn. 17 eight multiplications of $N_y \times N_y$ matrices are necessary. 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, but can then 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 Performance Analysis Library (HPAL) [5]. HPAL exposes linear algebra functions from Intel's Math Kernel Library (MKL)[6] from within LabVIEW. These functions are optimized for execution on multi-core processors and designed to work when the input matrices are extremely large.

We replace the functions for matrix multiplication and matrix inversion. The benchmark results in Section VI show that we are still limited by the inefficient use of memory by using 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 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.

ISBN: 978-988-19251-9-0
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

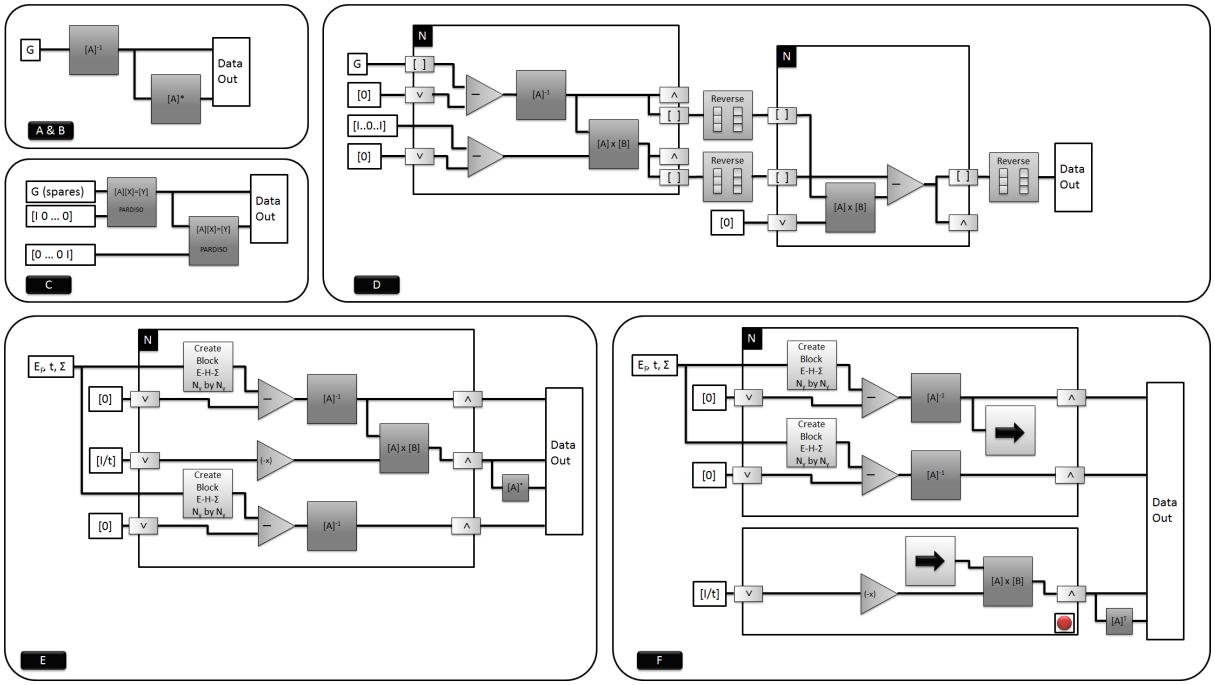

Fig. 2. Visualization of the different algorithms for the matrix inversion

Assuming that the block tri-diagonal linear system is

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

where A_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

$$\begin{aligned} \bar{B}_k &= (A_k - C_{k-1}\bar{B}_{k-1})^{-1}B_k \\ \bar{Y}_k &= (A_k - C_{k-1}\bar{B}_{k-1})^{-1}(Y_k - C_{k-1}\bar{Y}_{k-1}). \end{aligned}$$

Step 2: for k from N_x to 1

$$X_k = \bar{Y}_k - \bar{B}_k X_{k+1}.$$

This algorithm takes advantage of the sparsity of the matrix to achieve significant speedup, but still requires relatively large memory. There are $3N_y$ $N_y \times N_y$ 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

$$\begin{bmatrix} I_{N_y} & 0 & \dots & 0 \end{bmatrix}',$$

and

$$\begin{bmatrix} 0 & \dots & 0 & I_{N_y} \end{bmatrix}',$$

where I_{Ny} 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

$$K = \begin{bmatrix} 0 & & & I_{N_y} \\ & & I_{N_y} & \\ & \iddots & & \\ I_{N_y} & & & 0 \end{bmatrix}.$$

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

$$A = \begin{bmatrix} A_1 & B_1 & & & 0 \\ C_1 & A_2 & B_2 & & \\ & C_2 & A_3 & \ddots & \\ & & \ddots & \ddots & B_{N_{x-1}} \\ 0 & & & C_{N_{x-1}} & A_{N_x} \end{bmatrix},$$

then

$$KAK = \begin{bmatrix} A_{N_x} & B_{N_x} & & & 0 \\ C_{N_x} & A_{N_{x-1}} & B_{N_{x-1}} & & \\ & C_{Nx-1} & A_{N_{x-2}} & \ddots & \\ & & \ddots & \ddots & B_2 \\ 0 & & & C_2 & A_1 \end{bmatrix}.$$

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.

ISBN: 978-988-19251-9-0
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

F. Fifth optimization: Pipelined Block-Tridiagonal solver

To further improve the performance by making use of parallel architectures we pipelined sequential linear algebra calculations. By adjusting each group of operations to have roughly the same complexity, we ensured a constant high level of utilization on all available cores during the full inversion algorithm. At the same time the memory usage stays below 3 GB for system of up to $N_x = N_y = 1000$.

V. Implementation on GPUs

As the optimized pipelined block-tridiagonal solver still represents the most demanding part of our code, we further improved its performance by employing GPUs. As almost exclusively matrix multiplications and inversions have to be performed and several operations are done in parallel, GPUs yield a high performance potential. The other parts of the code focus on pre- and post-processing steps which lack computational complexity and are executed exclusively on the host processor cores. We used a prototype of the LabVIEW GPU Analysis Toolkit for the implementation on GPUs [10]. The small memory footprint of the optimized block-tridiagonal solver allows us to download the entire problem to the GPU and invoke the solver on the GPU device, retrieving only the final results. This minimizes communication between host and GPU during the most critical processing time. The efficient memory structure also allows the host to execute multiple independent simulation steps (i.e. as part of a sweep of e.g. potential or Fermi Energy) in parallel 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 [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 for execution on the Tesla GPUs. The benchmarks were performed with a 64-bit version of LabVIEW 2011 running under Windows 2008 Server Enterprise Edition, with the NVIDIA Tesla GPU set to TCC mode. 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 postprocessing 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 data to and from the GPUs. To visualize the performance of the different implementations we summarized the results we show the number of system slices along the x-direction that can be simulated per hour on a single node or a single GPU

TABLE II
Benchmark results for the GPU implementation of the pipelined and optimized block-tridiagonal matrix inversion solver

Systemsize ($N_x = N_y$) (sites)	Matrixsize ($N_x \cdot N_y$) (elements)	GPU Pipelined BT-Solver (seconds)
128	16,384	2.463
256	65,536	0.691
384	147,456	2.936
512	262,144	8.887
640	409,600	21.255
768	589,824	43.610
896	802,816	80.244
1024	1,048,576	136.685
1280	1,638,400	332.707
1536	2,359,296	688.338
1792	3,211,264	1,272.800
2048	4,194,304	2,170.260
2560	6,553,600	5,290.440
3072	9,437,184	10,964.600
3584	12,845,056	20,297.700
4096	16,777,216	34,616.500
5120	26,214,400	84,462.700

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.

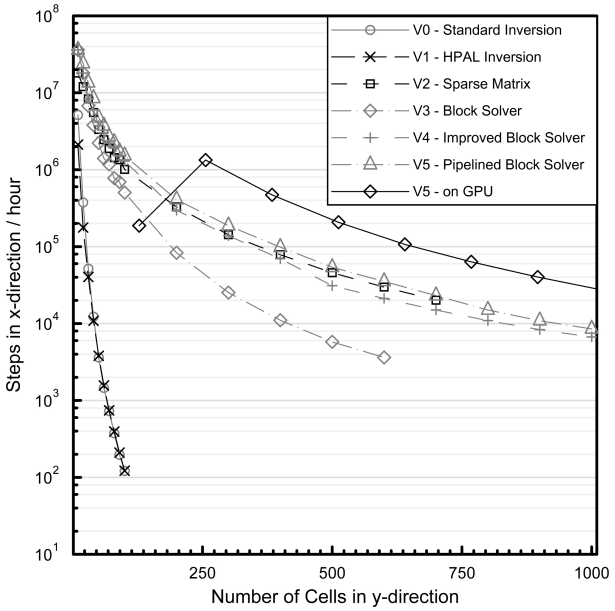

Fig. 3. Benchmark results in terms of simulation steps in x-direction per hour in dependence of the system size in y-direction

VII. Conclusion and Outlook

Transport simulations in semiconductor nanostructures rely on the Green's function algorithm. Direct implementations of 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

ISBN: 978-988-19251-9-0
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

TABLE I
SUMMARY OF THE BENCHMARK RESULTS FOR THE CPU-BASED ALGORITHMS. VERSION 0 IS THE ORIGINAL DIRECT INVERSION ALGORITHM. VERSION 1 USES THE OPTIMIZED LABVIEW HIGH-PERFORMANCE COMPUTING 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 (O.O.M. STANDS FOR OUT OF MEMORY – THIS BENCHMARK COULD NOT BE PERFORMED ON THE TEST MACHINE), AND VERSION 5 IS THE OPTIMIZED BLOCK-TRIDIAGONAL SOLVER WITH PIPELINING FOR IMPROVED THREAD UTILIZATION.

Systemsize $(N_x = N_y)$ (sites)	Matrixsize $(N_x \cdot N_y)$ (elements)	Version 0 direct inversion (seconds)	Version 1 HPAL library (seconds)	Version 2 sparse matrices (seconds)	Version 3 BT-solver A (seconds)	Version 4 BT-solver B (seconds)	Version 5 pipelining (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	1,600	11.745	13.261	0.026	0.038	0.024	0.017
50	2,500	49.714	47.328	0.054	0.081	0.048	0.038
60	3,600	148.369	138.163	0.088	0.154	0.072	0.058
70	4,900	346.183	339.151	0.134	0.215	0.114	0.094
80	6,400	769.706	730.780	0.201	0.371	0.151	0.127
90	8,100	1,647.595	1,543.517	0.241	0.468	0.214	0.187
100	10,000	2,964.965	2,949.634	0.357	0.715	0.279	0.236
200	40,000	o.o.m.	o.o.m.	2.194	8.662	2.428	1.765
300	90,000	o.o.m.	o.o.m.	7.560	42.750	7.804	5.767
400	160,000	o.o.m.	o.o.m.	18.323	130.317	20.709	14.643
500	250,000	o.o.m.	o.o.m.	39.306	311.673	57.965	33.411
600	360,000	o.o.m.	o.o.m.	72.519	595.367	102.021	61.147
700	490,000	o.o.m.	o.o.m.	125.120	o.o.m.	168.005	109.006
800	640,000	o.o.m.	o.o.m.	o.o.m.	o.o.m.	263.918	191.874
900	810,000	o.o.m.	o.o.m.	o.o.m.	o.o.m.	389.083	297.420
1000	1,000,000	o.o.m.	o.o.m.	o.o.m.	o.o.m.	538.907	422.620

problem size coupled with the required dense matrix computations make such a solution impractical. Our optimized implementations avoid the massive matrix sizes by exploiting the underlying sparse structure using a block-diagonal solver to reduce memory load from $(N_x N_y) \times (N_x N_y)$ 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 $1\ \mu\mathrm{m}$ by $1\ \mu\mathrm{m}$ and a grid spacing of $1\ \mathrm{nm}$ takes a total time of approximately 19 hours. Given the large system size, the fine grid and sweep resolution as well as 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, the performance of the presented algorithm allows precise simulations of transport in nanostructures with high precision and 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 $(N_x N_y N_s) \times (N_x N_y N_s)$, where N_s 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 $(N_y N_z) \times (N_y N_z)$. 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.

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.

ISBN: 978-988-19251-9-0
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)