Efficient Algorithms for the Green's Function Formalism

Semiconductor Transport Simulations on CPUs and GPUs

Jan Jacob, Bodo Krause-Kyora, Lothar Wenzel, Qing Ruan, Darren Schmidt, Vivek Amin and Jairo Sinova

Abstract We present efficient implementations of the non-equilibrium Green's function method for numeric simulations of transport in semiconductor nanostructures. The algorithms are implemented on CPUs and GPUs using LabVIEW 2011 64-Bit together with the Multicore Analysis and Sparse Matrix Toolkit and the GPU Analysis Toolkit.

Keywords Algorithm · GPU · Green's function · Inversion · LabVIEW · Simulations

J. Jacob (✉)
Institute of Applied Physics, University of Hamburg, Jungiusstraße 11, 22359 Hamburg, Germany
e-mail: jjacob@physnet.uni-hamburg.de

B. Krause-Kyora
PHYSnet Computing Center, University of Hamburg, Jungiusstraße 9, 22359 Hamburg, Germany
e-mail: krause@physnet.uni-hamburg.de

L. Wenzel · Q. Ruan · D. Schmidt
National Instruments, 11500 N Mopac Expwy, Austin, TX 78759-3504, USA
e-mail: lothar.wenzel@ni.com

Q. Ruan
e-mail: qing.ruan@ni.com

D. Schmidt
e-mail: darren.schmidt@ni.com

V. Amin · J. Sinova
Physics Department, Texas A&M University, 4242 TAMU, College Station, TX 77843-4242, USA
e-mail: aminvp@physics.tamu.edu

J. Sinova
e-mail: sinova@physics.tamu.edu

G.-C. Yang et al. (eds.), *IAENG Transactions on Engineering Technologies*, Lecture Notes in Electrical Engineering 186, DOI: 10.1007/978-94-007-5651-9_2,

1 Introduction

The continuing circuit miniaturization leads to device dimensions where quantum effects can be detrimental to the operation of standard CMOS devices. At the same time these effects yield the potential for novel energy efficient devices, since their channels no longer have to be depleted completely for switching [2, 9]. An electric field across the channel that is generated by a gate can influence the electron's spin precession length—and thereby its spin orientation with respect to a spin-sensitive detector [25]. Besides optical generation and detection of spin-polarized currents [18] and by ferromagnetic electrodes [23] also all-semiconductor devices are considered [6, 7] to pave the way to possible spin-based computing.

While charge-based devices are well understood, the realization of their spin-based cousins is still challenging. Analytical predictions for these nano structures often cannot be directly compared with experiments due to oversimplification. Numerical simulations bridge the gap between analytical descriptions and experiments allowing a thorough understanding of the quantum effects in such semiconductor devices by accounting for donor impurities, lattice imperfections, and interactions within a sample that are inaccessible to most analytical methods. Simulations with both realistic dimensions and appropriate grid sizes provide significant computational challenges, due to the large processing and memory load.

The non-equilibrium Green's function (NEGF) method [1] is a common approach to transport simulations. Other approaches, like the transfer matrix algorithm [24], can be translated to the NEGF method exhibiting similar mathematical structure and numerical techniques with the same computational challenges. They can be broken down to eigenvalue problems as well as multiplications and inversions of large, but often sparse matrices. We have explored implementations of the Green's function method with respect to their memory footprint and scalability over multiple threads, as well as their portability to general purpose graphics processing units (GPU) [8] and now expand our work to multi-GPU and multi-node implementations.

2 Mathematical and Physics Background

We employ the Green's function method to simulate transport in mesoscopic structures [1] with dimensions smaller than the coherence length. Here the conductance is obtained via the Landauer formula from the transmission probability T describing the probability of carriers to be transmitted from the input to the output contact $\mathbf{G} = \frac{2e^2}{h}T$, where e is the elementary charge and h is Planck's constant. The transport in these systems is through different quantum-mechanical modes, each contributing $2e^2/h$ to the total conductance for a perfect transmission $T = 1$. In the case of coherent transport, where the conductor is smaller than the phase-relaxation length, the total transmission probability can be decomposed into the sum of the individual channels:

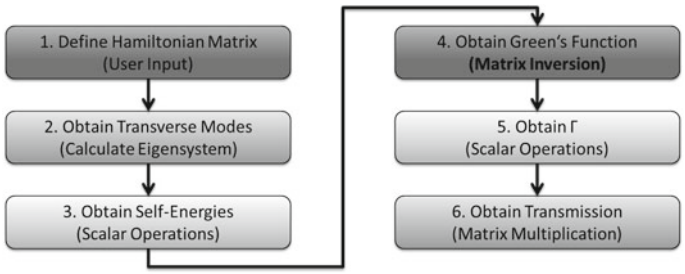

Fig. 1 Flowchart for the basic green's function algorithm

$$\mathbf{G} = \frac{2e^2}{h}\sum_{m,n} T_{mn}, \quad \text{where } T_{mn} = |s_{mn}|^2. \tag{1}$$

Green's functions provide a convenient way to obtain the *s*-matrix of the microscopic sample. Not going into the detailed relationship between Green's functions and the s-matrix, we will show how to calculate the Green's function for a particular sample and how to obtain T from the Green's function. Starting from the Schrödinger equation $H\Psi = E\Psi$ we define a Green's function, for a system with the Hamiltonian $\hat{H}(r)$

$$\left[E - \hat{H}(r)\right] G(r, r') = \delta(r - r'). \tag{2}$$

by rewriting the Schrödinger equation with an added source term. The Green's function can be interpreted as the wave function in terms of the position vector r of the source. Now the device is discretized on a grid in a tight-binding model of finite differences, such that $G(r, r') \to G_{ij}$, where i and j are denoting different lattice positions corresponding to r and r'. The differential equation becomes a matrix equation $[EI - H]G = I$, where each row or column represents a particular lattice site and each element defines the hopping between the two sites of its row and column. For a system of N_x horizontal and N_y vertical sites the matrix is of dimension $N_xN_y \times N_xN_y$. Considering realistic dimensions of some micrometer and a fine grid spacing of a few nanometer to precisely simulate all quantum effects in such nano structures this leads to huge matrices. Converting the Hamiltonian to a matrix operator also requires discretized derivative operators that are now given by

$$\left[\frac{\mathrm{d}F}{\mathrm{d}x}\right]_{x=\left(j+\frac{1}{2}\right)a} \to \frac{1}{a}\left[F_{j+1} - F_j\right] \tag{3}$$

$$\left[\frac{\mathrm{d}^2F}{\mathrm{d}x^2}\right]_{x=ja} \to \frac{1}{a^2}\left\{F_{j+1} - 2F_j + F_{j-1}\right\}. \tag{4}$$

For a one-dimensional system, where only a simple kinetic and a potential term are considered, the Hamiltonian matrix is given by

$$H = \begin{pmatrix} \cdots & -t & 0 & 0 & 0 \\ -t & U_{-1} + 2t & -t & 0 & 0 \\ 0 & -t & U_0 + 2t & -t & 0 \\ 0 & 0 & -t & U_1 + 2t & -t \\ 0 & 0 & 0 & -t & \cdots \end{pmatrix}, \tag{5}$$

where $t = \hbar^2/2ma^2$ is the hopping parameter and U_i denotes the potential at each lattice site. With appropriate labeling such a matrix can also be given for two or three dimensional systems. The Green's function can now be compute through matrix inversion.

$$G = [EI - H]^{-1}. \tag{6}$$

There are two independent solutions for G, the retarded and advanced Green's function; often an imaginary parameter is added to the energy in Eq. 6 in order to force one of the solutions. Solutions like Eq. 6 only yield information about scattering inside the sample. The leads are included by connecting them to the sample at various lattice site. Only directly neighboring sites are consider to be relevant to compute the lead's full effect on the transmission of semi-infinite, homogeneous, and reflection-less leads. Considering a sample c represented by a $N_x \times N_y$ grid and fully connected to two leads on either vertical side, labeled p and q, one can rewrite the Green's function in block matrix form as

$$G = \begin{pmatrix} G_c & G_{cp} & G_{cq} \\ G_{pc} & G_p & 0 \\ G_{qc} & 0 & G_q \end{pmatrix}. \tag{7}$$

Carriers enter or leave the sample via G_{cp}, G_{cq} or G_{pc}, G_{qc} and travel through the sample via G_c and in the leads via G_p and G_q. As there is no connection between the leads, carriers must transmit through the sample to travel between p and q. We assume this structure for G:

$$G = \begin{pmatrix} EI - H_c & \tau_p & \tau_q \\ \tau_p^\dagger & EI - H_p & 0 \\ \tau_q^\dagger & 0 & EI - H_q \end{pmatrix}^{-1}, \tag{8}$$

where the dimension of each element of the block matrix depends on the number of lattice sites corresponding to the portion they describe, and $[\tau_{p(q)}]_{ij} = t\delta_{ij}$, assuming that carriers may only enter the sample through a site adjacent to a lead. One can show, after some algebra, that

$$G_c = [EI - H_c - \Sigma]^{-1}, \quad \text{where} \tag{9}$$

$$\Sigma = \begin{pmatrix} t^2 g_p & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} + \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & t^2 g_q \end{pmatrix} = \Sigma_p + \Sigma_q \quad \text{and} \tag{10}$$

$$g_{p(q)} = \left[EI - H_{p(q)}\right]^{-1}. \tag{11}$$

The $\Sigma_{p(q)}$ are $N_x N_y \times N_x N_y$ matrices and g_p and g_q are $N_y \times N_y$ matrices. In the final step the transmission probability is then calculated by

$$T = \sum_{m,n} T_{mn} = \mathrm{Tr}\left[\Gamma_p G_c \Gamma_q G_c^{\dagger}\right], \quad \text{where } \Gamma_{p(q)} = i[\Sigma_{p(q)} - \Sigma_{p(q)}^{\dagger}] \tag{12}$$

3 Basic Implementation

The implementation of this algorithm includes six steps (see Fig. 1): First the Hamiltonian H for the system is created. As the matrix H is of size $(N_x N_y) \times (N_x N_y)$ it can cause memory issues, when implemented as a dense matrix. The second step determines the eigensystem for the transverse Hamiltonian H_y of the size $N_y \times N_y$ and can take significant computing resources for large systems. This is used in step three to define the self-energies of the leads by scalar operations. The fourth step calculates the Green's function by inversion of a $(N_x N_y) \times (N_x N_y)$ matrix representing the bottleneck of the algorithm. Step five creates the Γ matrices, requiring only scalar operations on the elements of Σ^A and Σ^R. This can be done in parallel to the inversion in step four. In the sixth step the transmission probability is calculated from Eq. 12 that includes the product of four $(N_x N_y) \times (N_x N_y)$ matrices.

4 Optimizations

The matrix H is of the size $(N_x N_y) \times (N_x N_y)$ scaling with the system size. However, the matrix's structure allows memory optimizations:

$$H_{(N_x N_y)\times(N_x N_y)} = \begin{pmatrix} \ddots & Y & 0 & 0 \\ Y & X & Y & 0 \\ 0 & Y & X & Y \\ 0 & 0 & Y & \ddots \end{pmatrix}, \quad \text{with} \tag{13}$$

$$X_{N_y \times N_y} = \begin{pmatrix} \ddots & B & 0 & 0 \\ B & A & B & 0 \\ 0 & B & A & B \\ 0 & 0 & B & \ddots \end{pmatrix} \quad \text{and } Y_{N_y \times N_y} = \text{ diag } [C], \tag{14}$$

where $A = 4t + V(x, y)$, $B = C = -t$, with the potential energy at a given site $V(x, y)$. The transverse Hamiltonian has an even simpler structure.

$$H_y = \begin{pmatrix} \ddots & B & 0 & 0 \\ B & D & B & 0 \\ 0 & B & D & B \\ 0 & 0 & B & \ddots \end{pmatrix}, \tag{15}$$

where $D = 2t$. While a sparse implementation of the matrices would help, it is most efficient to create only $N_y \times N_y$ matrix blocks, when needed in the algorithm. Solving the Eigenproblem for H_y in step two yields the modes and wave functions for the self energies and has potential for optimizations. However, benchmarks have shown that the compute time is much smaller than for step four. Therefore, we do not focus on optimizations of the Eigensolver. The inversion in Eq. 9 executed in step four is most demanding and therefore the focus of our optimizations. We denote the basic implementation as Algorithm A shown in Fig. 2. Four of the eight multiplications of $N_y \times N_y$ matrices in step five to obtain Eq. 12 can be done in parallel, which is already ensured by the parallel nature of the LabVIEW development environment. The same is true for the second set of four multiplications that need the results of the first set. As the computational load of this part is small compared to the inversion no further optimizations have been done to this part.

Optimized linear algebra functions. The Multicore Analysis and Sparse Matrix Toolkit (MASMT) [12] provides Intel's Math Kernel Library's (MKL) [5] linear algebra functions in LabVIEW, optimized execution on multi-core processors and operation on large matrices. Replacing the matrix inversion with the corresponding MKL function led to Algorithm B, that however, still is limited by memory inefficiency due to the dense matrix representation as the benchmarks in Sect. 6 show.

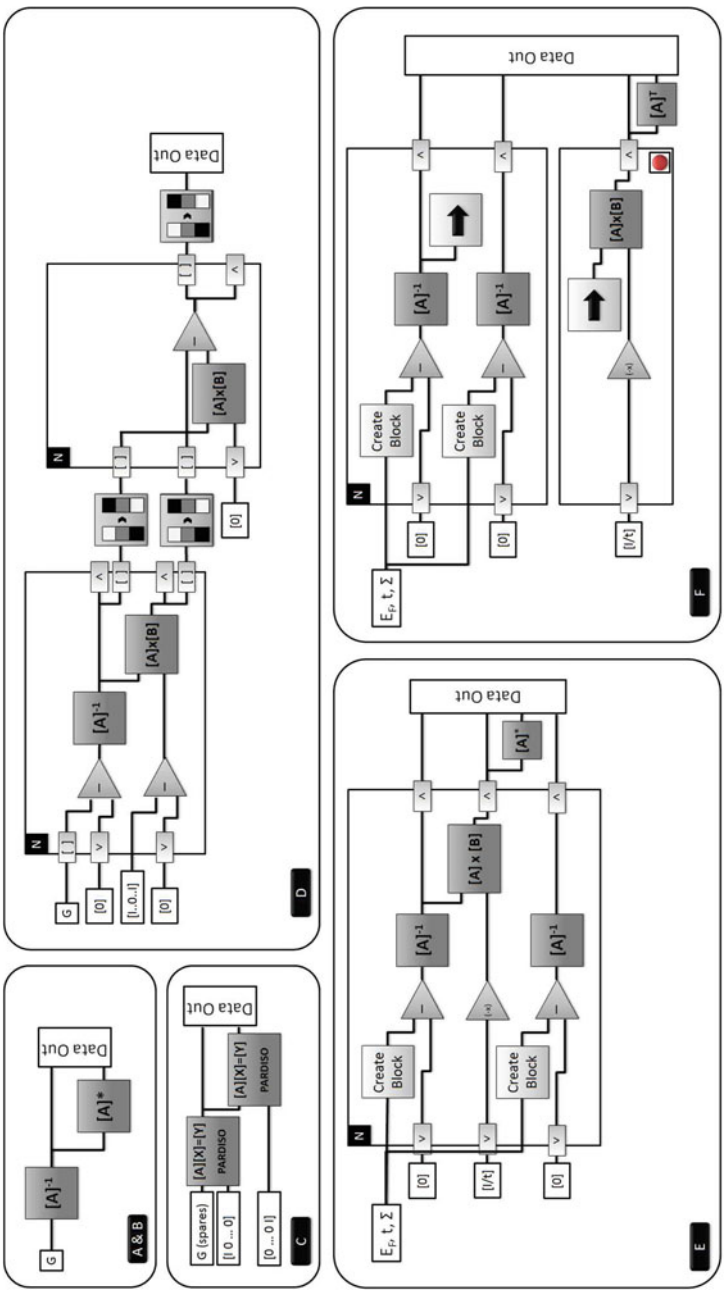

Fig. 2 Visualization of the different algorithms for the matrix inversion

Table 1 Summary of the benchmark results for the CPU-based algorithms

$N_{x,y}$	A (s)	B (s)	C (s)	D (s)	E (s)	F (s)
10	0.007	0.017	0.002	0.001	0.001	0.001
20	0.192	0.407	0.006	0.004	0.004	0.003
30	2.096	2.684	0.013	0.016	0.013	0.008
40	11.745	13.261	0.026	0.038	0.024	0.017
50	49.714	47.328	0.054	0.081	0.048	0.038
60	148.369	138.163	0.088	0.154	0.072	0.058
70	346.183	339.151	0.134	0.215	0.114	0.094
80	769.706	730.780	0.201	0.371	0.151	0.127
90	1,647.595	1,543.517	0.241	0.468	0.214	0.187
100	2,964.965	2,949.634	0.357	0.715	0.279	0.236
200	o.o.m.	o.o.m.	2.194	8.662	2.428	1.765
300	o.o.m.	o.o.m.	7.560	42.750	7.804	5.767
400	o.o.m.	o.o.m.	18.323	130.317	20.709	14.643
500	o.o.m.	o.o.m.	39.306	311.673	57.965	33.411
600	o.o.m.	o.o.m.	72.519	595.367	102.021	61.147
700	o.o.m.	o.o.m.	125.120	o.o.m.	168.005	109.006
800	o.o.m.	o.o.m.	o.o.m.	o.o.m.	263.918	191.874
900	o.o.m.	o.o.m.	o.o.m.	o.o.m.	389.083	297.420
1000	o.o.m.	o.o.m.	o.o.m.	o.o.m.	538.907	422.620

Version A is the original direct inversion algorithm. Version B uses the optimized LabVIEW high-performance computing libraries, Version C makes use of the matrices' sparsity, Version D is the first implementation of the block-tridiagonal solver, Version E is the optimized block-tridiagonal solver, and Version F 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).

Sparse matrices. Also part of MASMT is the PARDISO direct sparse linear solver [20–22] that we implemented as Algorithm C and that is faster than the dense solver, but still memory limited above $N_x = N_y = 700$ as the solver creates large intermediate data.

Block-Tridiagonal solver. Utilizing the block tri-diagonal structure of H we replaced the PARDISO solver in Algorithm D with the generalized Thomas algorithm [19]. Our block tri-diagonal linear system

$$\begin{bmatrix} A_1 & B_1 & & & 0 \\ C_1 & A_2 & B_2 & & \\ & C_2 & A_3 & \ddots & \\ & & \ddots & \ddots & B_{N_x-1} \\ 0 & & & C_{N_x-1} & A_{N_x} \end{bmatrix} \begin{bmatrix} X_1 \\ X_2 \\ X_3 \\ \vdots \\ X_{N_x} \end{bmatrix} = \begin{bmatrix} Y_1 \\ Y_2 \\ Y_3 \\ \vdots \\ Y_{N_x} \end{bmatrix}, \tag{16}$$

where A_k, B_k and C_k are all $N_y \times N_y$ blocks, can be computed in two steps. Step 1: for k from 1 to N_x

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

$N_{x,y}$	Matrixsize (elem.)	GPU pipelined BT solver (s)
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

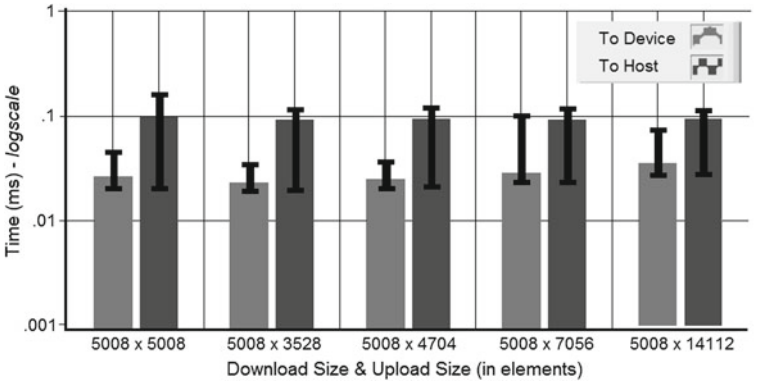

Fig. 3 Execution time for data transfers of single precision elements

$$\bar{B}_k = (A_k - C_{k-1}\bar{B}_{k-1})^{-1}B_k \tag{17}$$

$$\bar{Y}_k = (A_k - C_{k-1}\bar{B}_{k-1})^{-1}(Y_k - C_{k-1}\bar{Y}_{k-1}). \tag{18}$$

Step 2: for k from N_x to 1

$$X_k = \bar{Y}_k - \bar{B}_k X_{k+1}. \tag{19}$$

While the algorithm takes advantage of the sparse matrix structure it still requires quite large memory to store $3N_y$ $N_y \times N_y$ matrices between step 1 and step 2. This results in about 48 GB RAM for $N_x = N_y = 1000$ forcing slow storage on hard disk media. Since only the four $N_y \times N_y$ corners of the inverse matrix are relevant for the transmission, we are solving two linear systems with the right hand sides

$$\begin{bmatrix} I_{N_y} & 0 & \dots & 0 \end{bmatrix}', \text{ and } \begin{bmatrix} 0 & \dots & 0 & I_{N_y} \end{bmatrix}', \tag{20}$$

where I_{Ny} is an $N_y \times N_y$ identity matrix, and only the first and last blocks are of interest. As the last block of each system is already computed after the first step in the Thomas algorithm, we propose another method to compute the first block. Denote

$$K = \begin{bmatrix} 0 & & & I_{N_y} \\ & & I_{N_y} & \\ & \iddots & & \\ I_{N_y} & & & 0 \end{bmatrix}. \tag{21}$$

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

$$A = \begin{bmatrix} A_1 & B_1 & & & 0 \\ C_1 & A_2 & B_2 & & \\ & C_2 & A_3 & \ddots & \\ & & \ddots & \ddots & B_{N_x-1} \\ 0 & & & C_{N_x-1} & A_{N_x} \end{bmatrix}, \text{ then} \tag{22}$$

$$KAK = \begin{bmatrix} A_{N_x} & B_{N_x-1} & & & 0 \\ C_{N_x-1} & A_{N_x-1} & B_{N_x-2} & & \\ & C_{N_x-2} & A_{N_x-2} & \ddots & \\ & & \ddots & \ddots & B_1 \\ 0 & & & C_1 & A_1 \end{bmatrix}. \tag{23}$$

Since $(KAK)^{-1} = KA^{-1}K$, the upper left(right) corner of A^{-1} is equal to the lower right(left) corner of $(KAK)^{-1}$ and the first step of Thomas algorithm with KAK gives the upper left(right) corner of A^{-1}. This new Algorithm E saves memory because the second step is omitted by the cost of an extra matrix inversion. As this additional inversion can be computed in parallel a significant performance gain can be seen in Sect. 6 and the memory efficiency allows very large systems. For further performance gain on parallel architectures we pipelined sequential linear

algebra operations in Algorithm F and adjusted each group of operations to roughly the same complexity ensuring a constant high core utilization during the full inversion algorithm while at the same time keeping the memory usage below 3 GB for systems of up to $N_x = N_y = 1000$.

5 Implementation on GPUs

The optimized pipelined block-tridiagonal solver is still the most demanding part of the code. To further improve the performance we employed GPUs as they yield a high performance potential for the matrix multiplications and inversions of the algorithm. The rest of the code is executed exclusively on the host as it lacks computational complexity. For the implementation on the GPUs we used a prototype of the LabVIEW GPU Analysis Toolkit (LVGPU) [11]. The small memory footprint of the solver allows us to download the entire problem to the GPU and invoke the solver on the device, retrieving only the final result and thereby minimizing communication between the host and the GPU. Also multiple independent simulation steps (i.e. as part of a sweep of e.g. potential or Fermi Energy) could be executed if multiple GPUs are available. To control GPU devices from LabVIEW, LVGPU includes multiple interfaces for calling CUDA-based functions for execution on NVIDIA GPUs: LVCUDA encompasses functions based on the CUDA APIs for selecting and managing NVIDIA GPUs safely from the LabVIEW environment. LVCUBLAS and LVCUFFT functions call into CUBLAS and CUFFT available from NVIDIA's website [13, 14]. All GPU computations performed using LVGPU execute in parallel as if they were native LabVIEW operations. The current block-tridiagonal solver on the GPU uses LabVIEW and multi-core CPUs to orchestrate parallel execution of multiple LVGPU functions within an iteration of the solver. Results for this solver are consistent with performance gains achieved from LVGPU to solve other demanding problems. For example, an NVIDIA Tesla C2070 computes large FFTs (e.g. 512 K elements) in a few milliseconds including data transfers to and from the host. This meets real-time constraints for real-world problems such as quench detection of superconducting magnets. Performing an FFT on this extreme signal size overwhelms the core and cache size of present day multi-core CPUs. Experiments on multiple GPU devices were equally successful. The same GPU solver producing the benchmarks in Table 2 was deployed to two GPUs simultaneously. For a 1 K system size, the increase in execution time was 250 ms—adding only a fraction of the single GPU target time of 136.685 s. Distribution across multiple GPU devices is trivial when using LVGPU and potentially important to evolving algorithm variations such as Algorithm F from Fig. 2 as described in Sect. 7. To ensure that distribution to multiple GPU devices is feasible at multiple implementation levels, the jitter in data transfers to and from a GPU device was benchmarked. In Fig. 3, the mean execution time for I/O to (light bar) and from (dark bar) an NVIDIA

Tesla C1060 is shown. Error bars for each reflect the variation in transfer time recorded over several hundred iterations. Jitter on the order of 100 µs affords implementations of the current solver which deploy GPU kernels to multiple devices at virtually any level.

6 Benchmarks

We ran code implementing the direct inversion of the Green's function matrix (Algorithm A) and the different optimizations (Algorithm B–F) on an IBM i-dataplex dx360 M3 [3] with two Intel Xeon X5650 six-core processors, running at 2.67 GHz, 48 GB RAM, and two NVIDIA Tesla M2050 GPUs with 3 GB RAM [17]. All code was written in LabVIEW 2011 using functionality from MASMT and LVGPU. Internally, MASMT called Intel's Math Kernel Library (MKL) v10.3 for execution on the CPU's multiple cores. LVGPU invokes routines from NVIDIA's CUDA Toolkit v4.0 [15]. Benchmarks were performed with LabVIEW 2011 64-Bit running under Windows 2008 Server Enterprise, with the NVIDIA Tesla GPU in TCC mode. Results from the CPU-based implementations are shown in Table 1. Results for the code in version F which executed primarily on GPUs are shown in Table 2. The results include just the execution of the inversion in Algorithm 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 we summarize the results and show the number of system slices along the x-direction that can be simulated per hour on a single node or GPU in Fig. 4. 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.

7 Multiple GPU Implementation

For larger simulations and to further speed up the algorithm we present a scheme for multi-GPU distribution. The limited GPU-memory size can be circumvented by distributing the matrices over two or more GPUs. This in particular is convenient, when multiple operations on independent matrices can be executed in parallel as in our Algorithm F. Since CUDA-Version 4.0 multiple GPUs can share the same memory-environment allowing GPU-Memory addressing in a single namespace. Algorithm F consists of three threads where the third thread depends

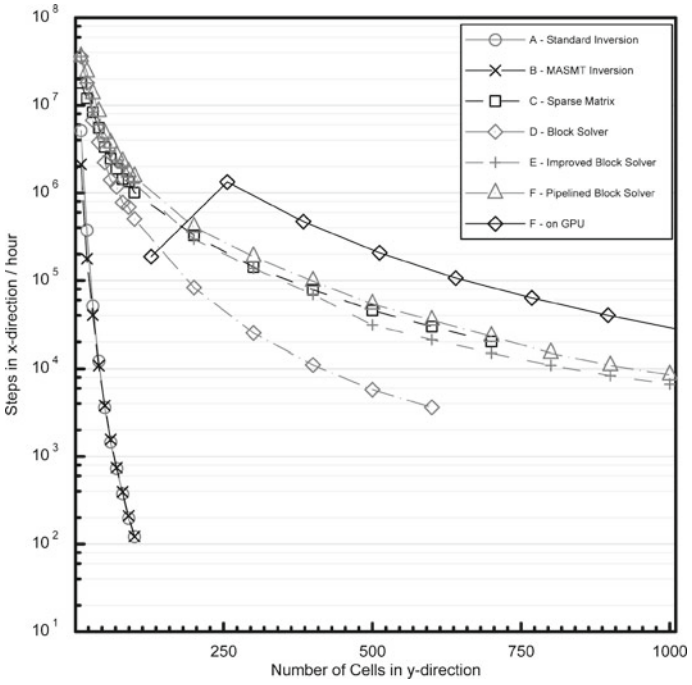

Fig. 4 Benchmark results in terms of simulation steps in *x*-direction per hour in dependence of the system size in *y*-direction

on previous results as shown in Fig. 5. One has to design a synchronized pipeline of these different threads to fully utilize the total available computing power.

While the third GPU is calculating the matrix multiplication, the first and second GPU can already compute their next step. A deployment on just two GPUs would leads to an asymmetric load: While both GPUs would start calculating the two inversions in parallel the GPU1 would have to calculate the multiplication in the end while the GPU2 would be idle. However, as all necessary results for the next set of inversions are already present, the GPU2 could take over the thread of inversion plus multiplication of the second step. As soon as the first multiplication is done the GPU1 starts calculating the other inversion of the second step. As this one does not require the final multiplication both GPUs would be in sync again at the end of the second step and the procedure would start over for the next step. As the CPU is not computing while the GPUs do, one can leverage the CPU-Power to compute the initial part of the next problem within a sequence (e.g. when varying a gate voltage, etc.) or cope with the communication. As the employed servers are equipped with two GPUs, the next logical step is to distribute the problem to several nodes. In this case multiple scenarios can be considered: First each of the three tasks in Fig. 5 could be run on a dedicated node with a hybrid CPU/multi-GPU implementation with algorithm as developed for example in the MAGMA

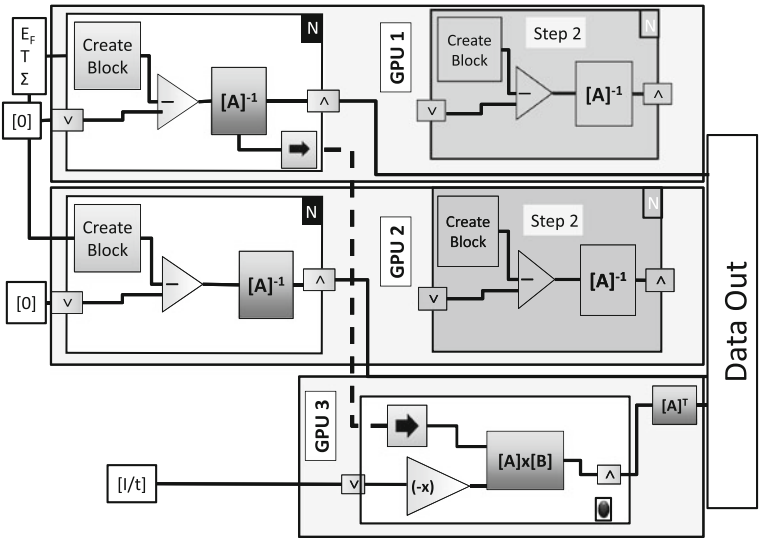

Fig. 5 Visualization of the multi-GPU setup

toolkit [10]. Second several steps of a parameter sweep can be distributed to several nodes as such a sweep is embarrassingly parallel. Third nodes could be specialized to distinct tasks: One node will create the Hamiltonian and solve the Eigenproblem and afterwards move the data to another set of nodes calculating the Green's function including the distribution of this subproblem to multiple nodes as described before. While the parallel execution of multiple simulations within a sweep is extremely easy, it is often desirable to receive results from a single step as fast as possible, giving the other two approaches additional priority. However, these two approaches will only be useful if a high-bandwidth-connection system like infiniband [4] is provided between the nodes as the information passed between the nodes would be matrices of size $(N_x N_y) \times (N_x N_y)$ in the first and still matrices of the size $N_x \times N_x$ in the third approach. To speed-up the transfer of data between GPUs on different nodes remote direct memory access RDMA, which will be available in the upcoming CUDA toolkit 5.0 release [16], will be very beneficial.

8 Conclusion

Simulations of transport in semiconductor nanostructures relying on the Green's function algorithm require—in their direct implementation—the inversion of gigantic matrices if realistic device sizes and sufficient small grid spacings are applied, making such a solution impractical. By exploiting the underlying sparse structure of the matrices we have presented an optimized implementation that

avoids the direct inversion by employing a block-diagonal solver to reduce the memory load from $(N_xN_y) \times (N_xN_y)$ matrices to $N_y \times N_y$ matrices. We enhanced the parallelism of the algorithm and balanced the computational load using pipelining to maximize the performance yield. The small memory footprint allows to implement the whole algorithm on a NVIDIA Tesla M2050 GPU 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 on CPUs with another performance gain by a factor of three on the GPUs*. Taking into account the further parallelization approaches outlined in Sect. 7 that allow multi-GPU and multi-node implementations further significant performance gain can be achieved. 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 h. If further spread over multiple GPUs in multiple nodes, complete simulations can be obtained within very short time spans allowing almost just-in-time direct comparison of numeric results with experiments. Having demonstrated that it is feasible to perform such simulations in reasonable times, we are now exploring codes for three-dimensional structures with multiple bands contributing to the transport. While adding bands increases the matrix size to $(N_xN_yN_s) \times (N_xN_yN_s)$, where N_s is the number of bands, the addition of a third dimension increases the matrix size to $(N_yN_z) \times (N_yN_z)$ for each "slice" of the system letting the matrices again grow very fast. Therefore we will intensify the research on multi-GPU and multi-node implementations as well as on further optimizations of the algorithm.

Acknowledgments This work was supported by the Deutsche Forschungsgemeinschaft via GK 1286 and Me916/11-1, the 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. Datta S (1999) Electronic transport in mesoscopic systems. Cambridge University Press, Cambridge
2. Datta S, Das B (1990) Appl Phys Lett 56(7):665
3. IBM. idataplex dx360 M3 datasheet. http://www-03.ibm.com/systems/x/hardware/idataplex/dx360m3/index.html
4. Infiniband TA. Introduction to Infiniband. http://members.infinibandta.org/kwspub/Intro_to_IB_for_End_Users.pdf
5. Intel. Intel Math Kernel Library. http://software.intel.com/en-us/articles/intel-mkl/
6. Jacob J, Lehmann H, Merkt U, Mehl S, Hankiewicz E (2011) DC-biased InAs spin-filter cascades. J Appl Phys 112:013706
7. Jacob J, Meier G, Peters S, Matsuyama T, Merkt U, Cummings AW, Akis R, Ferry DK (2009) Generation of highly spin-polarized currents in cascaded InAs spin filters. J Appl Phys 105:093714

8. Jacob J, Wenzel L, Schmidt D, Ruan Q, Amin V, Sinova J (2012) Numerical transport simulations in semiconductor nanostructures on CPUs and GPUs. Lecture notes in engineering and computer science: proceedings of the international multiconference of engineers and computer scientists 2012, IMECS
9. Koo HC, Kwon JH, Eom J, Chang J, Han SH, Johnson M (2009) Control of spin precession in a spin-injected field effect transistor. Science 325(5947):1515–1518
10. MAGMA. Magma—matrix algebra on gpu and multicore architectures. http://icl.cs.utk.edu/magma/
11. National instruments (2012) LabVIEW GPU Analysis Toolkit. beta version
12. National instruments (2012) LabVIEW multicore analysis and sparse matrix toolkit. https://decibel.ni.com/content/docs/DOC-12086
13. NVIDIA. CUDA BLAS implementation description. http://developer.nvidia.com/cuBLAS
14. NVIDIA. CUDA version 4.0 datasheet. http://developer.nvidia.com/cuFFT
15. NVIDIA. CUDA version 4.0 datasheet. http://developer.nvidia.com/cuda-toolkit-40
16. NVIDIA. CUDA version 5 RDMA feature. http://developer.nvidia.com/gpudirect
17. NVIDIA (2011) Tesla M2050 GPGPU datasheet
18. Oestreich M, Bender M, Hubner J, Hägele D, Rühle WW, Hartmann Th, Klar PJ, Heimbrodt W, Lampalzer M, Volz K, Stolz W (2002) Spin injection, spin transport and spin coherence. Semicond Sci Technol 17(4):285–297
19. Press WH, Teukolsky SA, Vetterling WT, Flannery BP (1999) Numerical recipes in C, vol 123. Cambridge University Press, Cambridge, p 50
20. Schenk O, Bollhoefer M, Roemer R (2008) SIAM Rev 50:91–112
21. Schenk O, Waechter A, Hagemann M (2007) J Comput Optim Appl 36(2–3):321–341
22. Schenk O, Gärtner K (2004) Journal of Future Generation Computer Systems 20(3):475–487
23. Schmidt G, Ferrand D, Molenkamp LW, Filip AT, van Wees BJ (2000) Fundamental obstacle for electrical spin injection from a ferromagnetic metal into a diffusive semiconductor. Phys Rev B 62(8):R4790–R4793
24. Usuki T et al (1994) Phys Rev B 50:7615–7625
25. Wunderlich J, Park B-G, Irvine AC, Zarbo LP, Rozkotov E, Nemec P, Novak V, Sinova J, Jungwirth T (2010) Science 330(6012):1801–1804