

Article

GPU Accelerated Nonlinear Electronic Circuits Solver for Transient Simulation of Systems with Large Number of Components

David Černý * and Josef Dobeš

Department of Radioelectronics, Czech Technical University in Prague, Technická 2,
166 27 Prague, Czech Republic; dobes@fel.cvut.cz

* Correspondence: cernyd1@fel.cvut.cz

Received: 3 September 2020; Accepted: 30 October 2020; Published: 2 November 2020



Abstract: GPU cards have been used for scientific calculations for many years. Despite their ever-increasing performance, there are cases where they may still have problems. This article addresses possible performance and memory issues and their solutions that may occur during GPU calculations of iterative algorithms. Specifically, the article focuses on the optimization of transient simulation of extra-large highly nonlinear time-dependent circuits in SPICE-like electronic circuit simulator core enhanced with NVIDIA/CUDA (Compute Unified Device Architecture) interface and iterative Krylov Subspace methods with emphasis on improved accuracy. The article presents procedures for solving problems that may occur during this integration and negatively affect either the simulation speed or the accuracy of the calculation. Finally, a comparison of the implementation of an iterative calculation procedure with the use of GPU cards, calculation by the direct method and calculation on the CPU only is presented.

Keywords: GPU; CUDA; SPICE; transient simulation; BicgStab; ILU

1. Introduction

This article proposes a new implementation of computational core of SPICE-like (Simulation Program with Integrated Circuit Emphasis) simulation program of electrical circuits with CUDA/GPU [1] parallelized BicgStab (Biconjugate Gradient Stabilized) method [2].

Several articles have recently been published CUDA parallelization [3–6]. This article focuses on simulation of electric circuits whose published articles can be divided into several groups: individual problems that can be encountered when solving simulations of electrical circuits, such as the parallel solution of sparse matrices [7–9], accelerated algorithms for calculations of preconditioning of matrices [10–13], or general acceleration of simulation of electric circuits [14,15]. There is even a special parallelized version of the SPICE simulator, CUSpice. CUSpice can also simulate giant circuits accelerated on the GPU. However, this article also focuses on extending the standard simulation procedure with an iterative BicgStab method [16]. In contrast to these works, this paper publishes a new simulation procedure for simulation of the circuits whose mathematical representation leads to the solution of huge nonlinear time-dependent systems. Systems with more than a million elements are marked as huge in the article.

The simulation of electrical circuits in SPICE-like simulators heavily relies on solving one or more linear systems. There are certain exceptions, but in general, the decomposition of a given electric circuit will always lead to some kind equation system represented in computer memory as a matrix system. This is caused by the fact that the circuit is loaded to memory through Modified Nodal Analysis (MNA) [17]. The main aspect of this is that for each unknown (e.g., voltage nodes) it

increases the dimension of the assembled matrix [18]. It is quite easy to realize that for huge circuits with a large number of nodes, the matrix will be both huge but also practically almost empty [19]. Therefore, when simulating electrical circuits, we will almost always have something to do with the solution of a linear system of equations defined by a huge sparse matrix. An example of a CMOS Shifter circuit assembled by an MNA is shown in Figure 1. It is a visualization of a sparse matrix for one time data when solving a transient analysis of the CMOS amplifier circuit. The axes show the dimension of the matrix while the black points are non-zero values. It can be noted that although the matrix has about 16,000 rows and columns, there will be significantly fewer non-zero values, and it is, therefore, necessary to take this into account when implementing algorithms.

To address similar results from published literature we should mention the article published directly on the web of Nvidia [20]. Among other things, you can find there a comparison of the performance of the calculation of the G3 circuit with the dimension 1 M, and the ASIC circuit with the dimension 321 k. The results indicate that it is possible to achieve up to double the acceleration in calculations on the GPU versus the CPU. However, they note that performance depends on the sparsity pattern of the coefficient matrix. This can also be seen in the relatively small difference in the performance of the computation difference of ASIC circuit.

On the contrary, close to the topic is this article [21], which presents a comparison of computational performance between the iterative GPU method and the iterative CPU method. When to monitor the GPU calculation acceleration again. Unfortunately, the authors only deal with nuts up to 4 k in size.

A very similar article, but GPU-oriented parallelization Jacob's iterative algorithm is [10]. The authors again present several modifications of the algorithm for accelerating the GPU calculation method. The article does not provide a comparison of the sizes of different matrices. Unfortunately, the algorithms cannot be applied to matrices created by an electrical circuit simulator. Methods such as GMRES or BicgStab mentioned in the article are much more suitable for this.

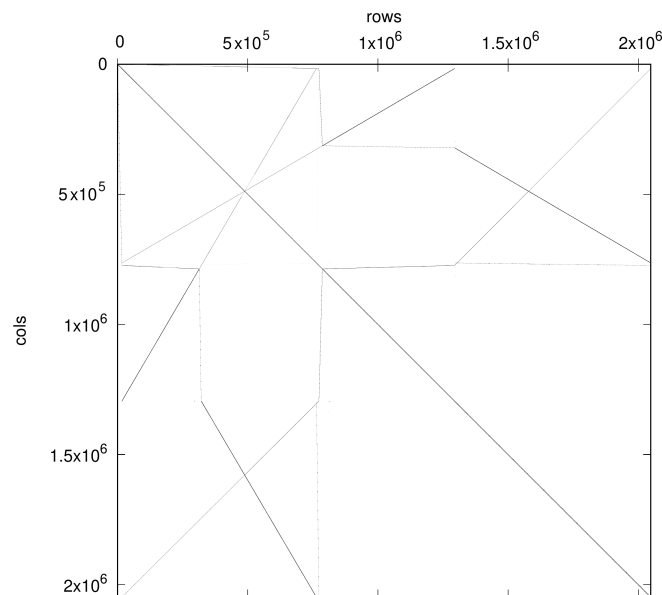


Figure 1. The visualization of the assembled sparse matrix of the SRAM Circuit. After pivoting and factorization.

2. Methodology

2.1. Simulation

In this article, for the sake of simplicity, we are going to address aspects of transient type simulation only [22]. It can be generalized as consecutive computation of circuit node voltages on given circuit components during a specific time interval that computes and visualize events

occurring in a time interval between two steady states. In the steady-state, the energy of the system does not change (or changes periodically), in the transient state, the energy of the system changes. The occurrence of the phenomenon is conditioned by changes in energy in the storage elements of the circuit (capacitors, coils). It should be noted that the transient simulation, from the perspective available simulation types in SPICE-like programs, is not exotic, its core functions are shared across other simulations. On the contrary, it could be said that in terms of the number of algorithms used, transient analysis ranks among the most complex simulation processes. (More on this topic can be found in NGSpice documentation [23]). Even though these are simulations of nonlinear time-dependent processes, all these systems will sooner or later simplify the simulation of a linear problem. It is, therefore, quite understandable that operations related to the solution of linear systems must be solved as optimally as possible. The vast majority of the simulators implements LU factorization methods of sparse matrices with emphasis on pivoting and realignment according to various criteria, such as Markowitz's criterion [24]. As will be shown later in the article (Table 1), factorization, and reordering are very demanding operations. Many publications can be found in the literature dealing with methods that provide various methods to solve this correctly and effectively [25,26], but even without that, the reader can imagine the time required for such an operation from Figure 1, that displays the resulting matrix after the factorization of the SRAM circuit.

Table 1. Time required for pivoting, reordering a factorization for different sizes of sparse matrix.

Matrix Dimension	nnz	Factorization Time (s)
2,048,129	10,581,374	2220.86
1,024,129	5,297,272	540.19
512,129	2,679,498	71.79
128,129	661,994	8.27
64,129	338,004	1.12
12,629	65,248	0.098

2.2. Parallelization

In the past (before 2010), scientific simulations on a graphics card using double precision was not entirely possible or with great difficulties [27]. (Initial version of CUDA libraries was released in 2007). This is practically no longer the case today, and the new graphics cards HW includes double-precision computation units and can easily compute in double precision. It is only necessary to realize that those operations will be proportionally slower in performance than in single precision [15,28,29]. The graphic card used in simulations in this article, NVIDIA GeForce RTX 2080 Ti has an estimated performance by the manufacturer in double-precision (64-bit) floating to 420.2 GFLOPS but for half-precision FP16 16-bit (Half Precision) it is incredible 26.90 TFLOPS [30].

2.3. Accuracy

A very important aspect of the circuit simulation is its accuracy. Especially when simulating electrical events, the input numbers of the circuit equations and the result can differ by several dozen of orders. An additional change to the computational process that we made was the implementation of a supporting iterative solution of a linear system using the BicgStab method [2].

The aim was to enable the simulation core in the individual steps of the simulation to obtain a more accurate result even in the case of an ill-conditioned system. It turns out that although most of the equations produced by simulations of electrical circuits can usually be calculated by the LU factorization methods, there are cases where the use of BicgStab methods is justified. The first one is when we want to achieve results with defined accuracy. Computation with floating numbers will always be affected by some degree of errors and therefore final precision will be limited by the precision of floating number representation. Additionally, certain operations with those numbers also affect the precision of the result and in the worst-case scenario can accumulate errors and decrease the

precision of the result even more. The algorithm introduced in Algorithm 1 as will be shown later, despite its name, would not be stable enough to be used in the core of the simulation program of electrical circuits. Fortunately, the BicgStab method can be supported by a preconditioner. The matrices that the numerical computational unit of the simulator has to deal with during the calculation can cause a divergence of the solution. The preconditioner helps to transform a linear system into a form that is more suitable for a numerical iterative solver. A description of the properties and derivation of the algorithm is defined in the cited publications. For clarity, \mathbf{x}_0 represents initial guess, A and \mathbf{b} defines computed linear system. The vector \mathbf{r}'_0 is chosen such that $(\mathbf{r}'_0, \mathbf{r}_0) \neq 0$ and ϵ_0 stands for stopping criterion. The Incomplete LU (ILU) preconditioner [31] has already proved to be the most successful [32]. It is very easy to implement it and it can easily work with sparse matrices. The implementation of the algorithm in Matlab source code follows in Listing 1.

Algorithm 1 Biconjugate Gradient Stabilized Method Algorithm [2].

```

 $\mathbf{r}_0 = \mathbf{b} - A\mathbf{x}_0$ 
 $\mathbf{p}_0 = \mathbf{r}_0$ 
for  $j = 0, 1, \dots$  do
     $\alpha_j = (\mathbf{r}_j, \mathbf{r}'_0) / ((A\mathbf{p}_j), \mathbf{r}'_0)$ 
     $\mathbf{s}_j = \mathbf{r}_j - \alpha_j A\mathbf{p}_j$ 
     $\omega_j = ((A\mathbf{s}_j), \mathbf{s}_j) / ((A\mathbf{s}_j), (A\mathbf{s}_j))$ 
     $\mathbf{x}_{j+1} = \mathbf{x}_j + \alpha_j \mathbf{p}_j + \omega_j \mathbf{s}_j$ 
     $\mathbf{r}_{j+1} = \mathbf{s}_j - \omega_j A\mathbf{s}_j$ 
    if  $\|\mathbf{r}_{j+1}\| < \epsilon_0$  then
        break
    end if
     $\beta_j = (\alpha_j / \omega_j) \times (\mathbf{r}_{j+1}, \mathbf{r}'_0) / (\mathbf{r}_j, \mathbf{r}'_0)$ 
     $\mathbf{p}_{j+1} = \mathbf{r}_{j+1} + \beta_j (\mathbf{p}_j - \omega_j A\mathbf{p}_j)$ 
end for
 $\mathbf{x} = \mathbf{x}_{j+1}$ 

```

Listing 1. ILU in Matlab.

```

function [M]= ILU(A)
n = length(A);
M = A;
for k = 1:n-1
    for i = k+1:n
        if A(i,k)==0
            continue
        end
        M(i,k) = M(i,k) / M(k,k);
    for j = k+1:n
        if A(i,j)==0
            continue
        end
        M(i,j) = M(i,j) - M(i,k) * M(k,j);
    end
end

```

2.4. Setup

All calculations were carried out by using a desktop computer with an Intel Core i9-9900K (16 threads, 8 cores 3.60 GHz), 31.3 GB of RAM memory and a NVIDIA GeForce RTX 2080 Ti graphics card with 11 GB of memory configure.

For compiling the executables, the CUDA compilation tools, release 10.2, V10.2.89 were used.

3. Implementation

In the case of calculations on GPU with CUDA, it should be taken into attention that the graphic card is a support device for calculating of graphic operations. Compared to the CPU, it can perform a massive number of parallel computations. In CUDA the code of the application is divided into two parts, the one is running on CPU (HOST) and the second is running on GPU (DEVICE). It can be seen in Figure 2. Not so much obvious problem is that it is necessary to first properly allocate the memory on the graphic card device. Then copy all data from HOST to GPU device, subsequently perform computation, and at the end copy results back to the HOST. This can be problematic, especially for iterative algorithms, where it can cause excessive copying of data between the CPU and the GPU back and forth for each iteration. Conceptually, it is necessary to design the process of calculation in such a way that as much of it as possible takes place directly on the graphics card, otherwise, the advantage given by the parallelization of the calculation will be lost with each subsequent iteration of the algorithm and reallocation of data. To take full advantage of the simulation calculation potential of the graphics card, we have modified the transient simulation process to minimize data copying between CPU graphics card devices. The biggest problem was the implementation of adaptive memory allocation. It is necessary to rearrange the rows and columns of the matrix on the simulation calculation so that it can be calculated by LU factorization. This process can be very time consuming, but it can be very successfully parallelized. Therefore, already at this moment, it is advantageous to load the matrix into the memory of the device. During the simulation, as the simulation progresses over time, the assembled matrix may change so much that it will need to be rearranged. For this case, we implemented a heuristic algorithm that allocates a percentage more memory than it needs at the beginning of the process. This is precisely because if new fill-ins [33] were created during the simulation and thus the total size of non-zero data would increase. In case the space is not enough, the algorithm copies the matrix back to the HOST and reallocates the space to the GPU device again.

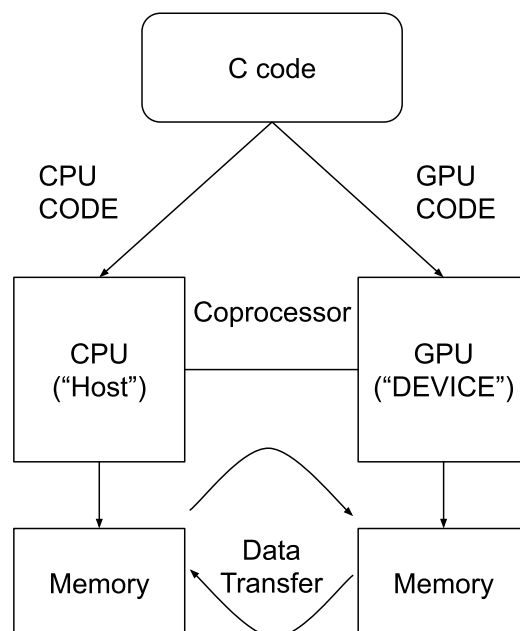


Figure 2. Count of LU operations vs matrix dimension.

Algorithm 2 shows a modified transient analysis process using CUDA parallel calculation of LU factorization on a graphics card. The algorithm adjusts the behavior during the entire transient analysis to reduce possible copying between the GPU and the card. An increase in the matrix due to changes during the simulation is solved as the sum of estimating the number of possible fill-ins and approximating the possible increase in the size of the simulation matrix due to changes in the circuit.

Algorithm 2 Transient analysis process using CUDA parallel calculation of LU factorization on a graphics card.

```

MNA (HOST)
Initial Matrix Composition (HOST)
Device Memory and Data Copy from HOST to DEVICE
Pivoting, Reordering (DEVICE)
for Timeline do
    Linear system (LU Factorization) (DEVICE)
    repeat {Nonlinear system (Newton-Raphson) (DEVICE) }
        Jacobian matrix (LU Factorization) (DEVICE)
        Next estimate (vector-matrix product) (DEVICE)
        Residual (vector norm) (DEVICE)
        Copy result from DEVICE to HOST
    until Stopping criteria
    if not (Convergence) then
        return Convergence problem
    end if
    Adaptive Memory Reallocation (DEVICE)
    Optional Pivoting and Reordering (DEVICE)
end for
Free DEVICE memory

```

The algorithm for a space allocation on graphic card was derived as the worst-case scenario that can happen during the computation of transient simulation, where time-dependent devices were not present in the circuit at the beginning of the simulation. The estimated space to be allocated on graphic card N_{alloc} we defined as max value of two c_D that characterizes the number of capacitance and l_D that characterizes the number of inductance devices in the circuit. F_{nnz} is the number of non-zero elements after factorization and A_{nnz} is the original number of non-zero elements before any computation.

$$N_{alloc} = 2F_{nnz} - A_{nnz} + \max(\sum(l_D), \sum(c_D)) \quad (1)$$

The equation is based on the assumption that the number of non-zero values in the matrix does not increase during the solution of the whole simulation over twice the difference between the values after and before factorization, the number of elements before factorization and the protection number given more of the number of inductive elements or capacitive elements in the circuit. This is an estimate and in case it would not be enough, it is necessary to reallocate the field.

BicgStab with ILU preconditioner was rewritten in C++ language and then added into the computational core of the NgSpice simulator. BLAS [34] and LAPACK libraries were used for CPU calculations, Eigen libraries were used for GPU calculations, as well as original implementations directly from the Nvidia documentation. The complete algorithm then performed all the calculations for each time step of the transient simulation separately and stored the results in a table.

The algorithms were implemented in the kernel of the open-source SPICE simulator NgSpice [23]. They were implemented not as a replacement of the standard computing process, but as an extension of it. Modified core of NgSpice performs standard factorization method using LU factorization (denoted as SPICE) and then resulting solution is passed via the CUDA interface to various implementation of parallelized BiS method on the graphics card.

4. Results

Iterative algorithms, if they converge, end the iteration after a certain number of steps, or at a specific moment when a certain degree of accuracy is achieved. When used for solving systems of linear equations, they will be normally slower than direct methods. Only in special cases, iterative algorithms will be able to find the result faster. For example, when methods such as GMRES and BicgStab

converge to a result without the use of preconditioners. In such situations, matrices do not need to be pivoted or rebalanced in advance, unlike when their calculation would be performed using the direct method. In the case of sparse matrices, factorization, pivoting and rearrangement are some of the most time-consuming operations. In this case, the iterative algorithms will overtake the direct method. This can be seen by comparing the data in Tables 1 and 2, which shows the times required to fully pivot the matrix so that it can be solved using LU factorization. For matrix dimensions above 64 k, the times rise steeply. Firstly, because of the huge amount of data that needs to be kept in memory, but also because of the number of operations that need to be performed before the matrix, and therefore the linear system, can be solved.

From a practical point of view and easy scalability computation of Transient Analysis of a series of SRAM circuits was chosen as a simulation problem. Subsequently, transient simulation was performed on each of them where at least thousand time steps were computed. This can be seen in Figure 3. The average value was then calculated from the values for each dimension of constructed circuit matrix.

Table 2 is divided into two parts, the first part contains the final accuracy of the result, with which the algorithm continued to work. Any values above 10^{-8} can be taken as unfulfilled accuracy condition, meaning that solution could not be found and the algorithm diverged. The simulated memory circuit was computed for sizes from 128 B to 2 KB.

Each table then compares the iterative method implemented for the CPU (CPU BiS + ILU, CPU BiS) and then parallelized to the GPU (GPU BiS + ILU, GPU BiS). The last SPICE LUF column, which shows the calculation of the direct LUF factorization, serves as a reference. The nnz column indicates the number of non-zero elements in the matrix and Bis denotes BicgStab method.

The results were obtained by calculating the whole simulation of the transient analysis. As an example one simulation run is shown in Figure 3. This is a comparison of the speed of calculation of individual points for the CPU BiS ILU and CUDA BiS ILU methods.

A comparison of the individual methods is shown in Table 2 (the values were averaged and the deviation determined) and also in Figure 4. What needs to be noted is also how this implementation changed the procedure for calculating Transient Analysis. According to predefined properties, the simulation program goes through the simulation and calculates the solution for the given moment by calculating the linearized system. In our case, the result was calculated by all values in each step. The algorithm then determined which of the algorithms reached the most accurate calculation and used the result. From all the time measurements that the calculation made during the whole simulation, the average was then calculated.

It is quite clear from the results that the BicgStab method supplemented with ILU preconditioner is more stable in terms of accuracy than not preconditioned version but also than the direct method, Table 2. It might not be so obvious that from a certain size of the matrix, the parallel calculations on the graphics card are slower than on the processor. This adds to the difficulty of processing such a large amount of data at once and the relatively limited size of the memory on the graphics card.

Table 3 shows the CUDA geometry used during circuit simulation. The values in brackets represent the size of each dimension (x, y, z) of CUDA grid and thread block. The number of threads that can run parallel on a CUDA device is simply the number of Streaming Multiprocessors (SM) multiplied by the maximum number of threads each SM can support. The device we used has 68 SM with a maximum of 2048 resident threads per SM. The minimal size of wrap (smallest unit that can be scheduled) is 32. Therefore, the size of the thread block should be always multiple of 32. We set the block size to 128 based on our observations during testing on the GPU device, although fine-tuning of the core operations should definitely be part of the further work. (Another work that deals with the thread-block size can be found in [35] e.g.).

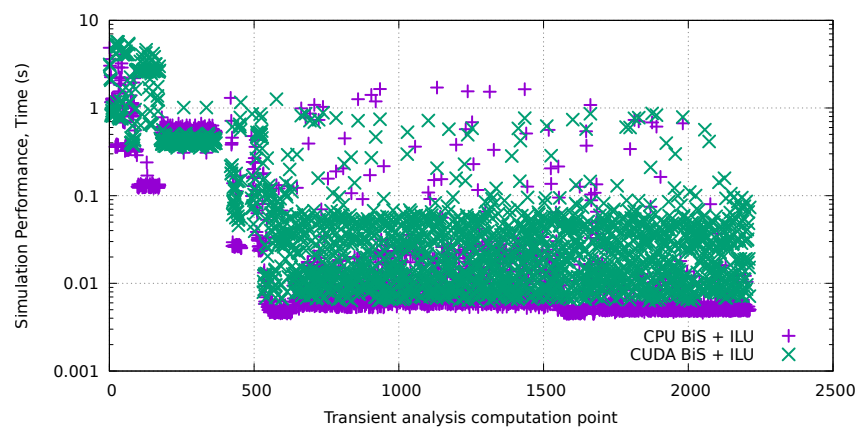
Table 2. Comparison of performance and reliability of the result of individual algorithms for different sizes of simulation matrix.

Simulation Accuracy						
Dimension	nnz	CPU BiS + ILU	CPU LUF	CUDA BiS + ILU	CUDA BiS	SPICE LUF
12,629	65,248	4.68×10^{-15}	3.04×10^{-10}	2.48×10^{-11}	1.49×10^{-19}	7.24×10^{-10}
64,129	331,124	2.92×10^{-15}	2.69×10^{-11}	4.16×10^{-9}	1.16×10^{-14}	3.98×10^{-12}
128,129	661,994	2.02×10^{-16}	1.40×10^{-22}	4.25×10^{-9}	3.24×10^{-15}	3.48×10^{-12}
512,129	2,707,540	4.08×10^{-10}	7.26×10^{-17}	2.48×10^{-11}	3.54×10^{-17}	2.98×10^{-11}
2,048,129	10,581,374	4.54×10^{-12}	1.16×10^{-20}	6.57×10^{-8}	1.25×10^{-15}	4.42×10^{-14}

Simulation Performance, Time (s)						
Dimension	nnz	CPU BiS + ILU	CPU LUF	CUDA BiS + ILU	CUDA BiS	SPICE LUF
12,629	65,248	1.115227	0.083508	0.004725	0.0161	0.004726
64,129	331,124	1.145234	0.083578	0.07299	0.176499	0.022568
128,129	661,994	1.152585	0.117093	0.026133	0.168729	0.025677
512,129	2,707,540	1.293445	0.867127	1.187724	1.245928	0.195807
2,048,129	10,581,374	2.462626	3.14092	4.873112	5.117584	0.352128

Table 3. CUDA geometry.

Matrix Dimension	Grid Size	Block Size
5,295,952	(128,005, 1, 1)	(128, 1, 1)
2,647,976	(64,005, 1, 1)	(128, 1, 1)
1,323,988	(32,005, 1, 1)	(128, 1, 1)
661,994	(16,005, 1, 1)	(128, 1, 1)
338,004	(8005, 1, 1)	(128, 1, 1)
130,496	(4005, 1, 1)	(128, 1, 1)

**Figure 3.** Transient simulation timeline.

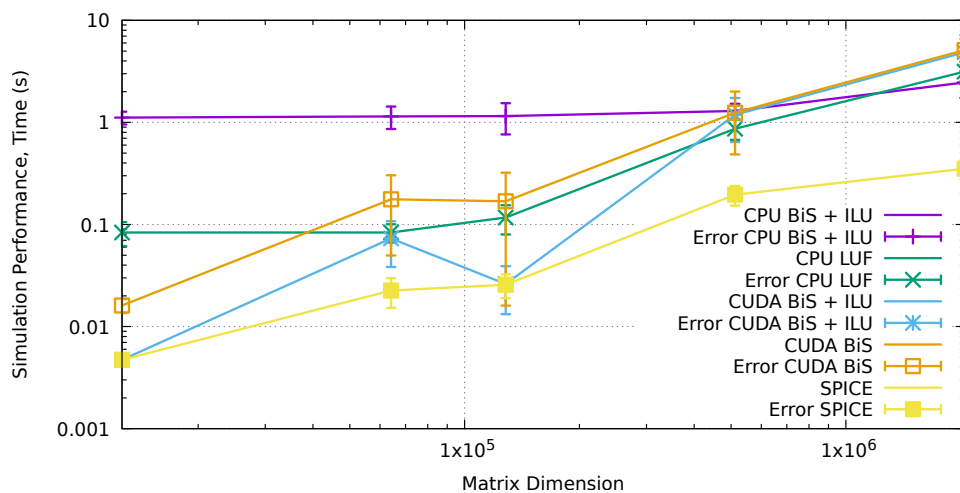


Figure 4. Simulation Performance, Time (s).

5. Conclusions

In this paper, a novel possible implementation of the core of the simulation program SPICE parallelizing the calculation of Transient Analysis with an emphasis on increasing the accuracy and reliability of the calculation was presented. The computational mathematical core of the SPICE program was extended with parallelization using CUDA/GPU libraries. In the case of the direct LU factorization method, it is a quite straightforward method to speed up the simulation calculation as shown in the article [36]. However, it brings new problems to the solution, which we have pointed out in this article. Especially the need to copy data to and from the graphics card, which can be very disadvantageous specifically in the calculation of Transient Analysis but also in any other analysis that requires linearization of the circuit using iterative algorithms.

The main goal was to replace the direct LU factorization with an iterative algorithm. The motivation for this replacement was the accuracy of the calculation of the individual steps of the simulation. In order for this to be possible, it was necessary to structure the calculation process of the transient analysis so that certain parts of the calculation were done on the graphics card, see Algorithm 2. Specifically, the best results were achieved with usage of parallelized BicgStab method supplemented with an ILU preconditioner. In the tests, we have shown that this method can in some cases achieve the same performance as the non-parallelized direct method. This corresponded to about 10 times the unparallelized BicgStab method. However, it turned out that with the increasing matrix and thus the memory requirement, this difference decreased to the size of the matrix when the iterative method calculated on the CPU was faster than on the GPU, see Table 2. Indirectly, this proves that a graphics card, although a very powerful device designed for calculating graphics operations, is not entirely suitable in all situations. This supports today's development, when solutions that can be found in practice are more inclined to processor arrays, fast and wide buses and fast SSDs than increasing the performance of graphics cards.

Implemented GPU accelerated algorithms were optimized only from the point of view of the form of data as they are arranged on the memory during GPU simulation. More advanced GPU optimizations techniques such as the use of CUDA constant and shared memory are planned as part of the further work.

The most important result of the performed tests was the demonstration of the possibility of an iterative BicgStab method with an ILU preconditioner to obtain a simulation result with defined accuracy. Using this method we achieved better precision result than with direct LU factorization. The accuracy of the calculations of individual results is very important especially in the moments when the iterative algorithm linearizes the nonlinear time-dependent equations of the circuit. It depends on the requirements for solving the problem. If speed is crucial, it will certainly

be more appropriate to use parallel direct LU factorization. If accuracy is preferred, it seems better to use iterative methods.

Author Contributions: CPU BiS ILU and CUDA BiS ILU methods, determining CUDA geometries, implementing calculations on GPU with CUDA, preparing and running test simulations, D.Č.; algorithms for transient analysis of nonlinear large-scale circuits, fast and slow modes of sparse-matrix LU factorization, advanced pivoting methods, J.D. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by the Czech Science Foundation under the grant no. GA20-26849S.

Acknowledgments: This paper has been supported by the Czech Science Foundation under the grant no. GA20-26849S.

Conflicts of Interest: The authors declare no conflict of interest. The funder had no role in the design of the study; in the collection, analyses, or interpretation of data; in the writing of the manuscript, or in the decision to publish the results.

References

1. Lippuner, J. *NVIDIA CUDA*; Technical Report; Los Alamos National Laboratory (LANL): Los Alamos, NM, USA, 2019.
2. Van der Vorst, H.A. Bi-CGSTAB: A Fast and Smoothly Converging Variant of Bi-CG for the Solution of Nonsymmetric Linear Systems. *SIAM J. Sci. Stat. Comput.* **1992**, *13*, 631–644. [[CrossRef](#)]
3. Garg, A.; Gupta, D.; Sahadev, P.P.; Saxena, S. Comprehensive analysis of the uses of GPU and CUDA in soft-computing techniques. In Proceedings of the 2019 6th International Conference on Signal Processing and Integrated Networks (SPIN), Noida, India, 7–8 March 2019; pp. 584–589.
4. Myasishchev, A.; Lienkov, S.; Dzhulii, V.; Muliar, I. Using GPU NVIDIA for Linear Algebra Problems. In *Collection of scientific works of the Military Institute of Kyiv National Taras Shevchenko University*; Taras Shevchenko National University of Kyiv: Kyiv, Ukraine, 2019.
5. Tsai, Y.M.; Cojean, T.; Anzt, H. Sparse linear algebra on AMD and NVIDIA GPUS—the race is on. In Proceedings of the International Conference on High Performance Computing, Frankfurt am Main, Germany, 22–25 June 2020; pp. 309–327.
6. Yang, C. Hierarchical Roofline Analysis: How to Collect Data using Performance Tools on Intel CPUs and NVIDIA GPUs. *arXiv* **2020**, arXiv:2009.02449.
7. Li, H.; Ge Li, K.; An, J.; Ge Li, K. An Online and Scalable Model for Generalized Sparse Non-negative Matrix Factorization in Industrial Applications on Multi-GPU. *IEEE Trans. Ind. Informat.* **2019**, *1*. [[CrossRef](#)]
8. Lee, J.; Kang, S.; Yu, Y.; Jo, Y.; Kim, S.; Park, Y. Optimization of GPU-based Sparse Matrix Multiplication for Large Sparse Networks. In Proceedings of the 2020 IEEE 36th International Conference on Data Engineering (ICDE), Dallas, TX, USA, 20–24 April 2020; pp. 925–936.
9. Dufrechou, E.; Ezzatti, P. Solving Sparse Triangular Linear Systems in Modern GPUs: A Synchronization-Free Algorithm. In Proceedings of the 2018 26th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP), Cambridge, UK, 21–23 March 2018; pp. 196–203.
10. Aslam, M.; Riaz, O.; Mumtaz, S.; Asif, A.D. Performance Comparison of GPU-Based Jacobi Solvers Using CUDA Provided Synchronization Methods. *IEEE Access* **2020**, *8*, 31792–31812. [[CrossRef](#)]
11. Dziekonski, A.; Fotyga, G.; Mrozowski, M. Preconditioners With Low Memory Requirements for Higher-Order Finite-Element Method Applied to Solving Maxwell’s Equations on Multicore CPUs and GPUs. *IEEE Access* **2018**, *6*, 53072–53079. [[CrossRef](#)]
12. Thuerck, D.; Naumov, M.; Garland, M.; Goesele, M. A Block-Oriented, Parallel and Collective Approach to Sparse Indefinite Preconditioning on GPUs. In Proceedings of the 2018 IEEE/ACM 8th Workshop on Irregular Applications: Architectures and Algorithms (IA3), Dallas, TX, USA, 12 November 2018; pp. 1–10.
13. He, G.; Yin, R.; Gao, J. An efficient sparse approximate inverse preconditioning algorithm on GPU. *Concurr. Comput. Pract. Exp.* **2020**, *32*, e5598. [[CrossRef](#)]
14. Lee, W.; Achar, R.; Nakhla, M.S. Dynamic GPU Parallel Sparse LU Factorization for Fast Circuit Simulation. *IEEE Trans. Very Large Scale Integr. Syst.* **2018**, *26*, 2518–2529. [[CrossRef](#)]

15. Santen, V.; Amrouch, H.; Henkel, J. Reliability Estimations of Large Circuits in Massively-Parallel GPU-SPICE. In Proceedings of the 2018 IEEE 24th International Symposium on On-Line Testing And Robust System Design (IOLTS), Platja d'Aro, Spain, 2–4 July 2018; pp. 143–146, [CrossRef]
16. Lannutti, F.; Menichelli, F.; Olivieri, M. CUSPICE: The revolutionary NGSPICE on CUDA Platforms. In Proceedings of the 12th MOS-AK ESSDERC/ESSCIRC Workshop, Venice Lido, Italy, 26 September 2014.
17. Ho, C.; Ruehli, A.E.; Brennan, P.A. The Modified Nodal Approach to Network Analysis. *IEEE Trans. Circuits Syst.* **1975**, *22*, 504–509.
18. Černý, D.; Dobeš, J. Common LISP as Simulation Program (CLASP) of Electronic Circuits. *Radioengineering* **2011**, *20*, 880–889.
19. Cerny, D.; Dobes, J. Adaptive sparse matrix indexing technique for simulation of electronic circuits based on λ -calculus. In Proceedings of the 2015 European Conference on Circuit Theory and Design (ECCTD), Trondheim, Norway, 24–26 August 2015; pp. 1–4.
20. Corporation, N. Incomplete-LU and Cholesky Preconditioned Iterative Methods Using cuSPARSE and cuBLAS. Available online: <https://docs.nvidia.com/cuda/incomplete-lu-cholesky/index.html> (accessed on 15 October 2020).
21. De Paula, L.; Soares, A. Parallel Implementation of the BiCGStab(2) Method in GPU Using CUDA and Matlab for Solution of Linear Systems. *J. Commun. Comput.* **2015**, *11*, 339–346. [CrossRef]
22. Gubian, P.; Zanella, M. Stability properties of integration methods in SPICE transient analysis. In Proceedings of the IEEE International Symposium on Circuits and Systems, Singapore, 11–14 June 1991.
23. Vogt, H.; Hendrix, M.; Nenzi, P.; Warning, D. Ngspice Users Manual Version 33. Available online: <http://ngspice.sourceforge.net/> (accessed on 18 October 2020).
24. Dobes, J. A modified Markowitz criterion for the fast modes of the LU factorization. In Proceedings of the 48th Midwest Symposium on Circuits and Systems, Covington, KY, USA, 7–10 August 2005; Volume 2, pp. 955–959.
25. Grigori, L.; Cosnard, M.; Ng, E. On the row merge tree for sparse LU factorization with partial pivoting. *BIT Numer. Math.* **2006**, *47*, 45–76. [CrossRef]
26. Bateman, D.; Adler, A. Sparse Matrix Implementation in Octave. *arXiv* **2006**, arXiv:cs/0604006.
27. Gulati, K.; Croix, J.; Khatri, S.; Shastri, R. Fast circuit simulation on graphics processing units. In Proceedings of the 2009 Asia and South Pacific Design Automation Conference, Yokohama, Japan, 19–22 January 2009; pp. 403–408. [CrossRef]
28. Jagtap, S.; Rao, Y. GPU accelerated circuit analysis using machine learning-based parallel computing model. *SN Appl. Sci.* **2020**, *2*, 883, [CrossRef]
29. Lei, C.U.; Man, K.; Zhang, N.; Wu, Y. GPU-Accelerated Non-Linear Analog and Mixed-Signal Circuit Transient Simulation. In Proceedings of the International MultiConference of Engineers and Computer Scientists 2012 (IMECS 2012), Hong Kong, China, 14–16 March 2012, Volume 2; pp. 1151–1152.
30. Lee, K. Nvidia GeForce RTX 2080 Ti Review. Available online: <https://www.techradar.com/reviews/nvidia-geforce-rtx-2080-ti-review> (accessed on 1 January 2020).
31. Zhao, Z.; Zhang, Q.; Tan, G.; Xu, J.M. A new preconditioner for CGS iteration in solving large sparse nonsymmetric linear equations in semiconductor device simulation. *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.* **1991**, *10*, 1432–1440. [CrossRef]
32. Cerny, D.; Dobes, J. Composing Scalable Solver for Simulation of Electronic Circuits in SPICE. In Proceedings of the 2018 International Conference on Intelligent and Innovative Computing Applications (ICONIC), Plaine Magnien, Mauritius, 6–7 December 2018; pp. 1–5, [CrossRef]
33. Dobes, J.; Cerny, D.; Biolek, D. Efficient procedure for solving circuit algebraic-differential equations with modified sparse LU factorization improving fill-in suppression. In Proceedings of the 20th European Conference on Circuit Theory and Design (ECCTD), Linköping, Sweden, 29–31 August 2011; pp. 689–692, [CrossRef]
34. Blackford, L.S.; Petit, A.; Pozo, R.; Remington, K.; Whaley, R.C.; Demmel, J.; Dongarra, J.; Duff, I.; Hammarling, S.; Henry, G.; et al. An updated set of basic linear algebra subprograms (BLAS). *ACM Trans. Math. Softw.* **2002**, *28*, 135–151.

35. Langdon, W.B. A many threaded CUDA interpreter for genetic programming. In *European Conference on Genetic Programming*; Springer: Berlin/Heidelberg, Germany, 2010; pp. 146–158.
36. Chen, X.; Ren, L.; Wang, Y.; Yang, H. GPU-Accelerated Sparse LU Factorization for Circuit Simulation with Performance Modeling. *IEEE Trans. Parallel Distrib. Syst.* **2015**, *26*, 786–795, [[CrossRef](#)]

Publisher’s Note: MDPI stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.



© 2020 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<http://creativecommons.org/licenses/by/4.0/>).