An efficient numerical solution technique for VLSI interconnect equations on many-core processors

Ginés Doménech-Asensi[1], Tom J. Kazmierski[2]

[1] Dpto. de Electrónica, Tec. de Computadoras y Proyectos, Universidad Politécnica de Cartagena, Cartagena, Spain, gines.domenech@upct.es

[2] Dpt. of Electronics and Computers Science, University of Southampton, Southampton, UK, tjk@ecs.soton.ac.uk

***Abstract*—This paper presents a technique to accelerate transient simulations of analog circuits using an explicit integration method parallelised on a many-core computer. Usual methods used by SPICE-type simulators are based on Newton–Raphson iterations, which are reliable and numerically stable, but require long CPU processing times. However, although the integration time step in explicit methods is smaller than that used in implicit methods, this technique avoids the calculation of time-consuming computations due to the Jacobian matrix inversion. The proposed method uses an explicit integration scheme based on the fourth order Adams–Bashforth formula. The algorithm has been parallelised on a NVIDIA general purpose GPU using the CUDA programming model. As a case study, the RC ladder model of a VLSI interconnect is simulated on a general purpose graphic processing unit and the achieved performance is then evaluated against that of a multiprocessor CPU. The results show that the proposed technique achieves a speedup of one order of magnitude in comparison with implicit integration techniques executed on a CPU.**

Keywords—simulation acceleration; state-space technique; GPU; VLSI interconnect

I. INTRODUCTION

Nowadays, one of the most limiting factors in the overall performance of integrated circuits is the interconnect. The continuous evolution of technology nodes implies an increase of the wire resistance, at a faster speed than the technology scale factor and the capacitance [1]. Delays due to RC parasitics can easily void the improvements reached by new device and circuit architectures. Thus, a careful analysis of such elements behaviour in the early stages of a system design is a key factor to achieve the desired performance.

However, the analysis of interconnect usually requires extensive transient simulations which imply long computation times. Simulators like SPICE [2], which are based on the modified nodal analysis and implicit integration schemas, use implicit differentiation techniques based on Newton–Raphson iterations to solve the analog equations at each time step. Although Newton–Raphson iterations are reliable and numerically stable, they lead to long CPU times, often hours or even days. Likewise, explicit integration methods require significantly smaller time steps compared to implicit methods, but given that their computational work load is lighter, the overall computation time is smaller compared to that of implicit methods. In this context, the use of state-space equations combined with explicit integration methods to simulate analog circuits [3] or mixed systems [4] has proved to be a suitable technique to speed up transient simulations.

Despite the introductions of alternative integration algorithms, further techniques are needed to speed up the simulation of increasingly complex circuits. In the last decade, general purpose Graphics Processing Units (GPUs) have become very useful platforms for general purpose computing tasks, taking advantage of their parallel architectures to speed up different types of scientific computations. The use of such platforms has become even more popular since the advent of the so called Compute Unified Device Architecture (CUDA) [5] in 2006, a programming model that allowed developers to use C as a high level programming language. Different works have studied the use of GPUs to accelerate the simulation of analog circuits [6], [7], [8], and more recently, the focus has been placed on the sparse matrix solver by LU factorization [9-11]. These techniques attain different ranges of average speedups compared with PARDISO [12], a state-of-the-art parallel sparse LU solver, and commercial KLU [13], a software package specifically designed for solving sequences of unsymmetric sparse linear systems. However, all of these proposals are still based on classical implicit methods used for SPICE-type simulators.

In this paper, a numerical solution based on an explicit integration schema, parallelizable over a many-core processors is proposed. The processor used is a general purpose NVIDIA GPU programmed using the CUDA programming model. The proposed technique uses a fourth order Adams–Bashforth formula to solve circuit formulation based on state variables. The method is demonstrated through transient simulations of VLSI interconnects and the processing time is compared to that achieved by a CPU executing an implicit integration technique.

The rest of the paper is organized as follows: Section II describes the linearized space state technique. In section III, the implementation of the algorithm on a GPU is described. The technique is demonstrated with an example in Section IV. Finally, conclusions are drawn up in Section V.

II. LINEARIZED STATE SPACE TECHNIQUE

Consider the following linearized state equation of a given system at time point t_k, $k = 0, 1\ldots$:

$$\dot{x}(t_k) = J_k x(t_k) + E e_x(t_k) \qquad (1)$$

where x is the vector of N state variable wave-forms, e_x is a vector of excitations and J_x and E are coefficient matrices, being J_k the Jacobian of the linearized model at the time point t_k. Since the system is passive, the eigenvalues of J_k have negative real parts [14]. Although state-of-the-art circuit simulators use implicit, rather than explicit integration, to assure numerical stability, the linearized state described in (1) can be solved in a fast explicit march-in-time integration process without Newton-Raphson iterations. In an explicit integration process the step-size must be limited not only to control accuracy of the solution but, most important, to ensure stability [14]. However, stability control is difficult because it needs estimates of the maximum eigenvalue λ_k of the Jacobian J_k at each step size and this is typically a time-consuming process [14]. To avoid this problem, in this work the stability technique described in [4] is used. This technique takes advantage of the passivity of the system and uses a fast method for estimating the maximum allowed step size directly from the Jacobian entries. Thus, given a set of ordinary differential equations of the form:

$$\dot{x}(t) = A \cdot x(t) \qquad (2)$$

The Adams–Bashforth integration scheme is described by:

$$x_{k+1} = (I + h\beta_0 A)x_k + hA\sum_{i=1}^{p} \beta_i x_{k-i}; k = 1, \ldots \qquad (3)$$

where h is the time step and β_i, $i = 0, \ldots, p$ are the Adams-Bashforth coefficients [15]. According to [4], the stability of the integration scheme in (3) is achieved if:

$$\left|1 - \beta_{max} h \left|a_{r,r}\right|\right| \leq 1; r = 1, \ldots, N \qquad (4)$$

Step sizes obtained from using this technique are expected to be smaller than the maximum allowed step sizes used in implicit methods, which are calculated from the exact values of the Jacobian's eigenvalues. However, the advantage of this technique is speed, given that time-consuming eigenvalue calculations are avoided.

III. SIMULATION ON A MANY-CORE COMPUTER

The linearized space state equation described in (1) must be computed at each time point t_k. Algorithm 1 describes the procedure used to compute the explicit integration schema. The value of each individual variable $\dot{x}_i$ at time point t_k is obtained working a sequence of multiply and accumulate operations, which can be carried out in parallel for each variable $\dot{x}_i$. At the end of each time point t_k, the values of x_{k+1} are worked out and the process is repeated for the new t_{k+1}. This means that each state variable can be computed at each time point independently of the rest of the state variables, and the algorithm can take advantage of a parallel implementation to speed up large transient simulations of analog circuits. So, for a given problem with N state variables, the algorithm can run on N parallel processing units, each one of them working out the value of a single variable $\dot{x}_i$.

Among the different architectures of many-core processors, general purpose GPUs have been progressively introduced in the field of electronic design automation. Moreover, the development of different software tools has extended their use in recent years. CUDA is a programming model which defines GPUs as computing devices having their own memory and running many threads in parallel [5]. The program running on a GPU is referred to as a kernel. The threads launched by the kernel are grouped into thread blocks, which are distributed to different streaming multiprocessors (SMs). Inside every block, threads are grouped into warps, each one containing 32 threads On current GPUs, a thread block may contain up to 1024 threads. Therefore, there are different levels of parallelism inside a GPU: blocks, warps and threads, which lead to many possibilities when programming a same algorithm. However, in order to achieve the best performance of high-performance parallel algorithms, some considerations must be taken into account [16]. All threads in a same block can access a common shared memory, while threads from different blocks can access a global memory. Since shared memory is faster than global memory, it is desirable to make extensive use of the first one, according to the possibilities of the algorithm. On the other hand, threads inside a same warp execute in single-instruction-multiple-threads (SIMT) pattern. When there are divergences of instructions inside a warp, threads corresponding to different instructions are executed serially and so the efficiency decreases. Finally, GPU kernels are launched and managed by the CPU. The interaction between the CPU and the GPU consumes a great portion of the overall computational resources. In this sense, data movements between CPU and GPU should be minimized and, whenever is possible, all the computation should be done inside the GPU, being the only task of the CPU launching the kernel and collecting the final results.

Fig. 1 shows the implementation of the integration algorithm on a GPU proposed in this work. The figure is a simplified schema of the GPU architecture. Threads are grouped into thread blocks, which also contain a shared memory, while all the blocks have access to a common global memory.

Algorithm 1: integration scheme

t=0
do // Loops for simulation time
  i=0;
  do // Loops for rows in J
    $\dot{x}_{i,k} = E_j \cdot e_{xk}$
    j=0;
    do // Loops for columns in J
      $\dot{x}_{i,k} = \dot{x}_{i,k} + x_{i,j,k} J_{i,j}$
      j++;
    while (*j*<*N*)
    $x_{i,k+1} = x_{i,k} + h \sum_{l=1}^{p} \beta_l x_{i,k-l}; k = 1, \ldots$
    i++;
  while (i<N)
  k++; // Updates step and time
  t=*t*+*h*;
while (t<simulation time);

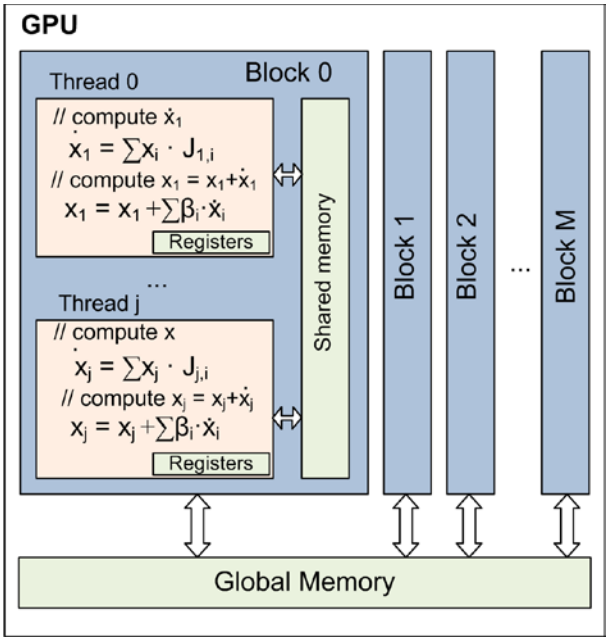


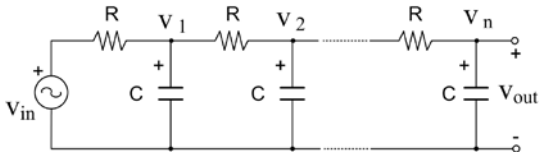
Fig. 1. Distribution of multiply and accumulation operations to compute a state variable at a given time step in multiple threads.

Moreover, each thread can access its own registers for local variables, which are on-chip and are the fastest among the memories in GPU. However, they are very limited in size. The rest of blocks have the same architecture of that shown for block 0. The integration schema for each single state variable *x* is executed on a single thread. This allows all the threads access to the same shared memory and achieve a higher efficiency. This distribution allows the processing of up to 1024 variables in (2) in a same thread block. Larger matrices require additional thread blocks, which decreases the memory bandwidth as the slower global memory needs to be accessed more frequently. The calculation of each new $x_{j,k+1}$ requires to read the values of the $x_{i,k}$ variables, where $i=1,\ldots N$. The values of $x_{j,k+1}$ are then written into memory to be used in the next integration step. This means that, although each thread runs in parallel, the set of variables $x_{j,k+1}$ is shared by all of them. Moreover, to increase the instructions throughput, there are not divergences between the threads. This has been achieved avoiding the use of conditional sentences in the kernel code and decomposing the computations into single multiply and accumulate operations as shown in Algorithm 1.

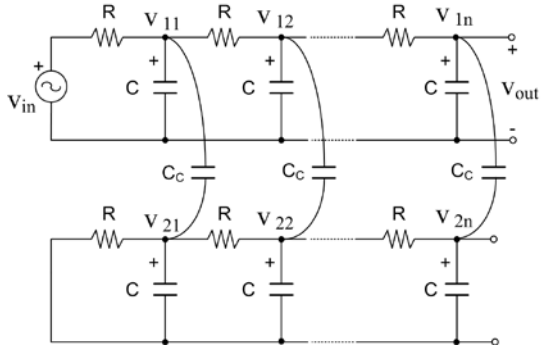
IV. EXAMPLE OF VLSI INTERCONNECTS

The linearized state-space formulation described in Section II was applied to the RC models of an isolated and a coupled VLSI interconnect shown in Fig. 2. The circuits are composed of *n* RC stages, being the first one excited by a voltage source. The space state equation (1) applied to the isolated model shown in Fig 2.a is as follows:

$$RC\frac{d}{dt}\begin{pmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{pmatrix} = \begin{pmatrix} -2 & 1 & 0 & \dots & 0 \\ 1 & -2 & 1 & \dots & 0 \\ 0 & 1 & \ddots & & \vdots \\ \vdots & \vdots & & -2 & 1 \\ 0 & 0 & \dots & 1 & -1 \end{pmatrix}\begin{pmatrix} v_1 \\ v_2 \\ v_n \end{pmatrix} + \begin{pmatrix} v_{in} \\ 0 \\ \vdots \\ 0 \end{pmatrix} \quad (5)$$



a)



b)

Fig. 2. Circuit representation of n-stage RC models of a) a isolated interconnect, and b) a coupled interconnect.

where the jacobian *J* is negative definite and diagonally dominant. The equation has been obtained through nodal analysis and manual transformation, although for more complex circuits, the method detailed in [17] may be useful. For the coupled interconnect model shown in Fig 2.b, the space state equation is given by:

$$\frac{RC(C+2C_C)}{C+C_C}\cdot\frac{d}{dt}\begin{pmatrix} v_{11} \\ \vdots \\ v_{1n} \\ v_{21} \\ \vdots \\ v_{2n} \end{pmatrix} = \begin{pmatrix} J_{11} & J_{12} \\ J_{21} & J_{22} \end{pmatrix}\begin{pmatrix} v_{11} \\ \vdots \\ v_{1n} \\ v_{21} \\ \vdots \\ v_{2n} \end{pmatrix} + \begin{pmatrix} v_{in} \\ \vdots \\ 0 \\ \frac{C_C}{C+C_C}v_{in} \\ \vdots \\ 0 \end{pmatrix} \quad (6)$$

where the submatrices $J_{11} = J_{22}$ are the Jacobian matrix defined in (5) and $J_{12} = J_{21}$ are given by:

$$J_{12} = J_{21} = \frac{C_C}{C+C_C}\cdot J_{11} \quad (7)$$

Following the method described in previous section, two different sets of tests have been carried out for an isolated and for a coupled transmission lines. In the first test, equation (5) has been programmed on a NVIDIA GPU following Algorithm 1, for different number of RC stages. In the second test, equations (6) and (7) have been programmed on the same GPU. In both cases, the algorithm has been coded so that each thread computes the value of a single state variable *x*. The state variables, which are read and written at each time step, are stored in the shared memory. However, given that the shared memory is limited to 48 K, the coefficients of *J* and *E* are stored in the global memory. Although this is a slower memory, it is only accessed to read these coefficients. The values of the passive elements of the circuit have been fixed to R = 10Ω and C = 250 pF. The coefficients of the fourth order Adams Bashforth formula are $\beta_0 = 55$, $\beta_1 = -59$, $\beta_2 = 37$ and $\beta_3 = 9$. For these values the time step has been fixed to 100 ps.

The performance of the algorithm has been tested through transient simulations of 1μs each. Table I details the processor time required for a μs transient simulation for different number of RC stages, for the isolated and for the coupled transmission line. The many core processor used has been a NVIDIA GeForce GTX 1080, 3584 Core, 1531MHz and 11 GB of RAM GPU. . The results are compared with those achieved by an Intel Xeon CPU E5-2680 v4 @2.40 GHz processor, with two physical processor chips, 14 cores and 56 processors, 35M Cache and 64 GB of RAM.

Each table shows the average values for five runs in each configuration. For the isolated interconnect model, the CPU processing time increases exponentially with the number of stages while the GPU processing time remains approximately around 11ms for a number of RC stages up to 100. Then, the GPU time increases proportionally to the complexity of the circuit from 200 to 1000 RC nodes, due to the time-consuming eigenvalue calculations required by the implicit integration method. Thus, the speedup achieved by the GPU over the CPU remains above one order of magnitude and is increased with the complexity of the circuit (Fig. 3). The behaviour of the GPU processing time can be better understood analysing its memory bandwidth. Table I shows how the memory bandwidth increases with the complexity of the circuit, although the increase rate slows down continuously as the circuit complexity is increased, when bigger number of threads are competing for the access to the same memory banks. This behaviour of the memory bandwidth implies that the processing time is linearly increased with the size of the circuit.

TABLE I. CPU AND GPU TIMES FOR A TRANSIENT SIMULATION OF 1 μS

Num. RC stgs	**Isolated interconnect**			**Coupled interconnects**		
	Average processing time		***GPU BW (GB/s)***	**Average processing time**		***GPU BW (GB/s)***
	CPU (s)	***GPU (s)***		***CPU (s)***	***GPU (s)***	
10	0.226	0.009	3.46	0.232	0.023	4.50
20	0.318	0.010	6.64	0.353	0.024	8.48
30	0.316	0.010	9.77	0.374	0.240	11.34
40	0.379	0.011	11.99	0.387	0.027	13.74
50	0.369	0.011	15.06	0.413	0.031	16.93
60	0.374	0.011	17.93	0.157	0.031	19.83
70	0.400	0.011	20.80	0.404	0.033	22.24
80	0.398	0.011	23.96	0.498	0.032	25.69
90	0.395	0.011	27.04	0.396	0.033	28.73
100	0.423	0.011	30.29	0.448	0.033	32.03
200	0.472	0.015	47.95	0.670	0.046	46.81
300	0.559	0.022	53.48	1.178	0.060	54.44
400	0.710	0.026	64.62	1.643	0.067	66.98
500	0.960	0.030	74.30	2.583	0.079	72.77
600	1.201	0.037	75.50	3.561	0.092	76.19
700	1.528	0.044	78.10	4.534	0.108	77.38
800	1.842	0.048	85.54	5.821	0.120	81.10
900	2.317	0.055	88.28	7.379	0.137	81.75
1000	2.870	0.058	96.03	9.364	0.156	81.09

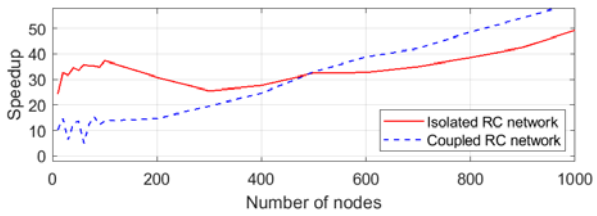
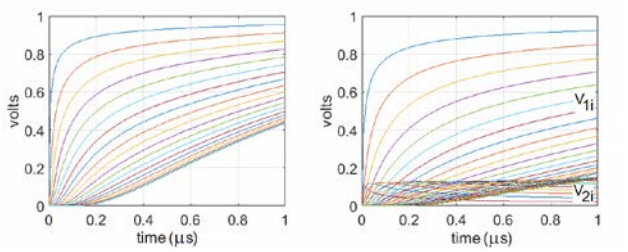


Fig. 3. Speedup of the GPU over the CPU for isolated and coupled networks.



a) b)

Fig. 4. Transient simulation of voltages at each node of the a) isolated RC network and b) coupled RC network.

Results of the coupled interconnect model are slightly different. Both the CPU and the GPU processing times are larger since the circuit complexity is increased, remaining the GPU one order of magnitude faster than the CPU (Fig. 3). The results of the GPU processing time are in concordance with the parallelism hierarchy of the GPU. So, the GPU processing time is linearly increased, while the CPU processing time increases exponentially. Regarding the memory bandwidth, an asymptotic behaviour can be observed, as for the isolated RC example, reaching the maximum values between 800 and 1000 stages. Fig. 4 plots the voltage at each node of the RC model of the transmission lines for a 1µs transient simulations and a step input voltage of 1v.

V. Conclusion

The continuous evolution of technology nodes, which implies more RC delays, added to the increasing frequencies in data and signal processing require the use of new techniques to accelerate the transient simulations performed during a circuit design cycle. This paper has shown an implementation of an explicit integration method on a many-core platform aimed to speed up the transient simulations of analog circuits. The parallelized integration technique has been tested on a VLSI interconnect RC model with different levels of complexity.

The results obtained show that the GPU implementation of the explicit integration schema is at least one order of magnitude faster than a classic implicit differentiation techniques based on Newton–Raphson iterations executed on a multiprocessor CPU. The results also show that memory bandwidth plays a key role in the efficiency of parallelisation techniques. For this implementation, as complexity increases and the number of threads reaches a certain number, the GPU processing time is proportionally increased. However, this growth is slower than that obtained by multiprocessor CPUs, which supports the validity of the method proposed in this paper.

VI. Acknowledgements

This work has been partially funded by Spanish government through projects RTI2018-097088-B-C33 and TEC2015-66878-C3-2-R (MINECO/FEDER, UE), by EPSRC (the UK Engineering and Physical Sciences Research Council) under grant EP/N0317681/1 and by Universidad Politécnica de Cartagena - Campus de Excelencia Internacional Mare Nostrum. The research stay at University of Southampton (UK) has been supported by Ministerio de Educación, Cultura y Deporte within the “Programa Estatal de Promoción del Talento y su Empleabilidad en I+D+i, Subprograma Estatal de Movilidad, del Plan Estatal de I+D+I” under grant PRX18/00565.

References

[1] T. Huynh-Bao et al., "Statistical Timing Analysis Considering Device and Interconnect Variability for BEOL Requirements in the 5-nm Node and Beyond," IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 25, pp. 1669-1680, May 2017.

[2] L. W. Nagel, “SPICE 2: A computer program to stimulate semiconductor circuits,” Ph.D. dissertation, University of California, Berkeley, California, US, 1975.

[3] K. C. A. Lam and M. Zwolinski. "Circuit simulation using state space equations." Proceedings of the 2013 9th Conference on Ph.D. Research in Microelectronics and Electronics (PRIME). Villach. 2013. pp. 177-180.

[4] T. J. Kazmierski. L. Wang. B. M. Al-Hashimi and G. V. Merrett. "An Explicit Linearized State-Space Technique for Accelerated Simulation of Electromagnetic Vibration Energy Harvesters." IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems. vol. 31, pp. 522-531. April 2012.

[5] NVIDIA. CUDA C Programming Guide Version 7.0. Accessed: Mar. 5, 2018. [Online]. Available: http://docs.nvidia.com/cuda/cudac-programming-guide/

[6] K. Gulati, J. F. Croix, S. P. Khatri and R. Shastry, "Fast circuit simulation on graphics processing units," Asia and South Pacific Design Automation Conference, Yokohama, 2009, pp. 403-408.

[7] R. E. Poore, "GPU-accelerated time-domain circuit simulation" IEEE Custom Integrated Circuits Conference, Rome, 2009, pp. 629-632.

[8] L. Han and Z. Feng, "TinySPICE Plus: Scaling up statistical SPICE simulations on GPU leveraging shared-memory based sparse matrix solution techniques," IEEE/ACM International Conference on Computer-Aided Design (ICCAD), Austin, TX, 2016, pp. 1-6.

[9] X. Chen, L. Ren, Y. Wang and H. Yang, "GPU-Accelerated Sparse LU Factorization for Circuit Simulation with Performance Modeling" IEEE Trans. on Parallel and Distributed Systems, vol. 26, pp. 786-795, March 2015.

[10] K. He, S. X. -. Tan, H. Wang and G. Shi, "GPU-Accelerated Parallel Sparse LU Factorization Method for Fast Circuit Analysis," in *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 24, pp. 1140-1150, March 2016.

[11] W. Lee, R. Achar and M. S. Nakhla, "Dynamic GPU Parallel Sparse LU Factorization for Fast Circuit Simulation," IEEE Transactions on Very Large Scale Integration Systems, vol. 26, pp. 2518-2529, Nov. 2018.

[12] O. Schenk and K. Gartner, “Solving unsymmetric sparse systems of linear equations with PARDISO,” Future Generat. Comput. Syst., vol. 20, no. 3, pp. 475–487, Apr. 2004.

[13] T. A. Davis and E. P. Natarajan, “Algorithm 907: KLU, a direct sparse solver for circuit simulation problems,” ACM Trans. Math. Softw., vol. 37, no. 3, Sep. 2010, Art. no. 36.

[14] L. O. Chua and P. Y. Lin. Computer-Aided Analysis of Electronic Circuits: Algorithms and Computational Techniques. Englewood Cliffs. NJ: Prentice-Hall. 1975.

[15] D. Zwillinger, Handbook of Differential Equations, 2nd ed. San Diego, CA: Academic, 1989.

[16] CUDA C best practices guide, October 2018,. [Online]. Available: http://docs.nvidia.com/cuda/pdf/CUDA_C_Best_Practices_Guide.pdf

[17] Y. Kang and J. Lacy. “Conversion of mna equations to state variable form for nonlinear dynamical circuits.” Electronics Letters. vol. 28, pp. 1240–1241, 1992.