Simulation acceleration of image filtering on CMOS vision chips using many-core processors

Ginés Doménech-Asensi
Dpto. de Electrónica, Tec. de Computadoras y Proyectos
Universidad Politécnica de Cartagena
Cartagena, Spain
gines.domenech@upct.es

Tom J. Kazmierski
Dpt. of Electronics and Computers Science
University of Southampton
Southampton, UK
tjk@ecs.soton.ac.uk

***Abstract*—This paper describes an efficient numerical solution to speed up transient simulations of analog circuits on a many-core computer. The technique is based on an explicit integration method, parallelised on a multiprocessor architecture. Although the integration step is smaller than the required one by traditional simulation methods based on Newton–Raphson iterations, explicit methods do not require to compute complex calculations such us matrix factorizations, which lead to long CPU simulation times. The proposed technique has been implemented on a NVIDIA GPU and has been demonstrated simulating Gaussian filtering operations performed by a CMOS vision chip. These type of devices, which are used to perform computation on the edge, include built-in image processing functions, turning them into very complex and time consuming circuits during their design. The proposed method is faster that Ngspice for different image sizes, and for 128 x 128 pixels image size it achieves a speed up of two orders of magnitude.**

Keywords—simulation acceleration; state-space technique; GPU; CMOS vision chip

I. Introduction

Nowadays, the so called Deep Learning is monopolizing most of the research lines in the field of computer vision. However, the computational power and memory resources required for deep neural networks is notably larger than those used for classical computer vision algorithms. This represents a serious drawback for portable devices, where power resources are scarce, and so, specific low power hardware must be used [1]. As a consequence, edge computing based on low power devices represents an alternative to the cloud based computing. In this sense, current CMOS vision sensors used to perform computation on the edge are incorporating built-in image processing functions at pixel level, before the analog to digital conversion. Gaussian filtering is a basic task for early vision. It is used for reducing the noise associated to the image capture without affecting subsequent processing stages. Moreover, the utility of Gaussian filtering reaches its maximum level when the smoothing degree of the image, is under the control of the user. From a pure analog perspective, this can be achieved, using RC networks synthesised at pixel level, which can be implemented by means of MOS transistors working on the triode region whose channel resistance is controlled by their gate voltage [2]. This gives an idea of the increasing complexity of such vision chips, which consequently leads to extremely large simulation times during their design cycle. Thus, the acceleration of transient simulations required to evaluate these chips performance becomes a keystone to shorten the time to market.

Nowadays, electronic design tools use SPICE [3] type simulators to perform transient simulations of analog circuits. These family of simulators are based on the modified nodal analysis, which use implicit differentiation techniques based on Newton–Raphson iterations to solve the analog equations at each time step. This technique has largely proven to be reliable and numerically stable. However it still consumes large CPU times which easily last hours or days. Faced with this technique, explicit integration methods can offer an advantage on simulation time. Although explicit methods require significantly smaller time steps compared to implicit ones, given that their computational work load is lighter, the overall computation time is smaller compared to that of implicit methods. An example of the use of such methods is demonstrated in [4] where a mixed signal system is modelled through space state equations and simulated using an explicit integration method.

Nevertheless, new techniques are needed to speed up the simulation of increasingly complex circuits besides the introductions of alternative integration algorithms. Is in this context where parallelization becomes a keystone to increase the time performance of analog simulators. In the last years, different works have proposed the use of general purpose Graphic Processing Units (GPUs) to accelerate the simulation of analog circuits [5-7]. These platforms have become very popular since the advent of the so called Compute Unified Device Architecture (CUDA) [8], a programming model that allowed developers to use C as a high level programming language. More recently, the focus has been placed on the sparse matrix solver by LU factorization [9-12]. However, all of these proposals are still based on classical implicit methods used for SPICE-type simulators. This paper describes a numerical solution based on an explicit integration schema to model and speed up the simulation of CMOS vision chips. In particular, the paper focuses on the Gaussian filtering function, one of the more common functions used in computer vision algorithms. The integration technique comprises its parallelisation over a many-core processor, which in this case is a general purpose NVIDIA

GPU. The proposed technique uses a fourth order Adams–Bashforth formula to solve circuit formulation based on state variables. The rest of the paper is organised as follows: Section II describes the linearized space state technique and the implementation of the algorithm on a GPU. The technique is demonstrated with an example of a CMOS-C imager in Section III. Finally, conclusions are drawn up in Section IV.

II. PARALLELLIZATION OF THE LINEARIZED STATE SPACE TECHNIQUE

Let (1) describe the linearized state equation of a given system at time point t_k, k = 0, 1…:

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

being x is the vector of N state variable wave-forms, e_x a vector of excitations and J_x and E coefficient matrices. J_k is the Jacobian of the linearized model at the time point t_k. This linearized state equation can be solved in a fast explicit march-in-time integration process without Newton-Raphson iterations. However, the main drawback of an explicit integration process is that the step-size must be limited to ensure stability [13], and not only to control the accuracy of the solution. Stability control is a time consuming process given that the maximum eigenvalue λ_k of the Jacobian J_k at each step size must be computed [13]. However, in [4] an alternative stability technique is proposed, which 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, this is the technique used in this work to estimate the maximum allowed step size. Given a set of ordinary differential equations of the form:

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

its Adams–Bashforth integration scheme is described by:

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

being h the time step and β_i, $i = 0, \ldots,$ p the Adams-Bashforth coefficients [14]. The technique described in [4] proves that the stability of the integration scheme in (3) is achieved if:

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

This method provides step sizes which are expected to be smaller than the maximum allowed step sizes used in implicit methods. However, the advantage of this technique is speed, given that time-consuming matrices factorization calculations in implicit methods are avoided.

The linearized space state equation described in (1) must be computed at each time point $t_{k.}$. The procedure used to compute the explicit integration schema is shown in Algorithm 1. 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.

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);
```

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 $\dot{x}_i$.

Regarding the general purpose GPUs, the programming model CUDA, defines GPUs as computing devices with their own memory and able to run many threads in parallel. The program running on a GPU is called kernel, and this kernel can launch several threads which are grouped into thread blocks. The thread blocks are physically distributed to different streaming multiprocessors (SMs). The architecture of current GPUs allows that each thread block can contain typically a maximum of 1024 threads, being these threads grouped into warps, each one containing 32 threads. Thus, these different levels of parallelism inside a GPU, i.e. blocks, warps and threads, provide a wide range of possibilities when programming a given algorithm, which achieve different performance in terms of processing speed. So, some considerations must considered to obtain the best performance of high-performance parallel algorithms [15]. First, it must be taken into account that all threads running in a same thread block are able to access to a common shared memory, while the common memory for threads from different blocks is of type global. Given that global memory is slower that shared memory, one should make extensive use of the first one, depending on the possibilities of the algorithm. Second, all the threads inside a same warp are executed following a single-instruction-multiple-threads (SIMT) pattern. So, any divergences of instructions between threads in a same warp, forces that threads corresponding to different instructions are executed serially. This leads to a decrease in the GPU efficiency. And third, each GPU kernel is launched and managed by the CPU, being the interaction between the CPU and the GPU process which consumes a lot of computational resources. To avoid this slowdown, data transfers between CPU and GPU should be reduced to the minimum and, if possible, the whole computation should be done inside the GPU. In this last case, the CPU would be used only to launch the kernel and to collect the final results.

The implementation of the integration algorithm on a GPU proposed in this work is shown in Fig. 1.

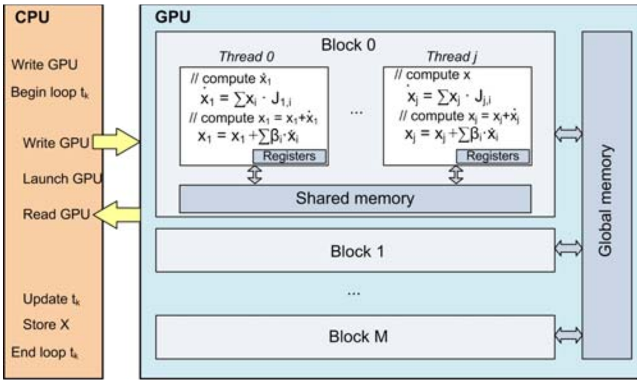

Fig. 1. Distribution of multiply and accumulation operations to compute a state variable at a given time step in multiple threads.

The figure is a simplified schema of the GPU architecture, interacting with a CPU. Each thread inside a block computes a single state variable and accesses its own registers for local variables. These on-chip registers are the fastest among the memory hierarchy but they are very limited in size. The CPU starts writing the values of the jacobian J to the GPU and then runs the integration loop. In each step the value of state variables at time t_k are copied into the GPU, then the kernel is launched and finally the values of the variables at time t_{k+1} are read from the GPU. After this, the value of the current time is updated in the CPU. The integration schema for each single state variable x is executed on a single thread. This allows all the threads in a same block to access to the same shared memory and achieve a higher efficiency. The calculation of each new $x_{j,k+1}$ requires to read the values of the $x_{i,k}$ variables, where i=1,…N. The values of $x_{j,k+1}$ are then written into memory to be used in the next integration step. So, although each thread runs in parallel, the set of variables $x_{j,k+1}$ is shared by all of them. The rest of blocks have the same architecture of that shown for block 0.

III. Example of CMOS-C networks for time-controlled Gaussian spatial filtering

Fig. 2 shows an all-MOS implementation of a piece of an RC network which performs time-controlled Gaussian spatial filtering [2]. In the circuit, each node represents a single pixel in the imager, where the value on light intensity V_{ij} is stored as the initial value in the respective capacitor C_{ij}, as it is done in current CMOS imagers. Each capacitor is connected to adjacent capacitors through a MOS device acting as a resistor and whose channel resistance can be controlled though the gate voltage. So, once an image has been captured and stored in the array of capacitors, the Gaussian filtering starts when a given voltage is applied to the MOS devices to enable the channel resistance. For the coupled interconnect model shown in Fig 2, using nodal analysis and solving for dv_{ij}/dt the space state equation is obtained as:

$$RC\frac{dV_{i,j}}{dt} = -4V_{i,j} + V_{i,j+1} + V_{i,j-1} + V_{i+1,j} + V_{i-1,j} \quad (6)$$

where R is the MOS channel resistance. Solving this equation, it is proved that the evolution of the voltage along time is a Gaussian function with $\sigma = (2t/RC)^{0.5}$, being t the time during which the MOS channels are ON [16].

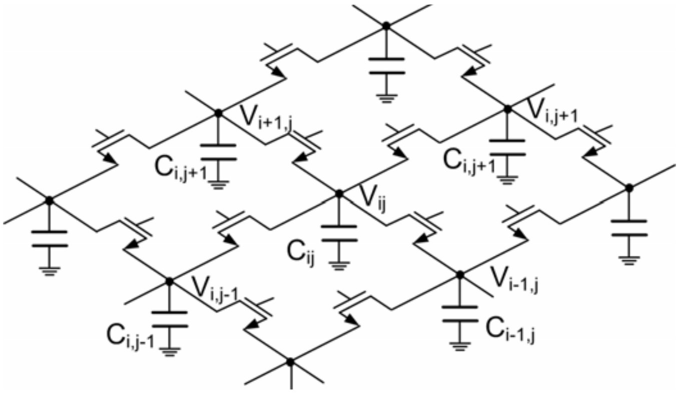

Fig. 2. Circuit representation of the MOS-C network.

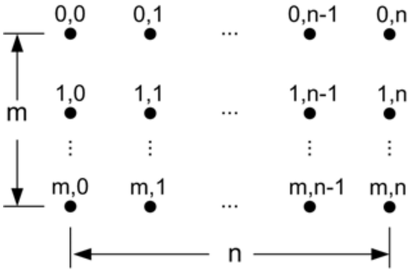

Fig. 3. Matrix representation of the m x n pixels MOS-C imager.

Pixels placed in an edge or in a corner of the image have only three or two neighbours respectively. Thus, the corresponding equations are modified accordingly. For an $m \times n$ pixels imager, described as in Fig. 3, its corresponding space state equation (2) is as follows:

$$RC\frac{d}{dt}\begin{pmatrix} v_{0,0} \\ v_{0,1} \\ \vdots \\ v_{m,n-1} \\ v_{m,n} \end{pmatrix} = \begin{pmatrix} A_1 & I & 0 & \cdots & 0 & 0 \\ I & A_2 & I & \cdots & 0 & 0 \\ 0 & I & A_2 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & A_2 & I \\ 0 & 0 & 0 & \cdots & I & A_1 \end{pmatrix}\begin{pmatrix} v_{0,0} \\ v_{0,1} \\ \vdots \\ v_{m,n-1} \\ v_{m,n} \end{pmatrix} (7)$$

being I, the identity matrix, 0 the null matrix, A_1 and A_2 all of them $m \times m$ submatrices which compose the $n \times n$ matrix A. The value of A_1 is given by:

$$A_1 = \begin{pmatrix} -2 & 1 & 0 & \cdots & 0 & 0 \\ 1 & -3 & 1 & \cdots & 0 & 0 \\ 0 & 1 & -3 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & -3 & 1 \\ 0 & 0 & 0 & \cdots & 1 & -2 \end{pmatrix} \quad (8)$$

while A_2 is:

$$A_2 = \begin{pmatrix} -3 & 1 & 0 & \cdots & 0 & 0 \\ 1 & -4 & 1 & \cdots & 0 & 0 \\ 0 & 1 & -4 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & -4 & 1 \\ 0 & 0 & 0 & \cdots & 1 & -3 \end{pmatrix} \quad (9)$$

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

Image size	***Explicit on GPU (ms)***	***Ngspice on CPU (ms)***	***Speedup***
4 x 4	0.000223	0.00154	6.918239
8 x 8	0.000327	0.00434	13.28843
16 x 16	0.000916	0.01638	17.886
32 x 32	0.009622	0.24594	25.56071
64 x 64	0.04382	3.78276	86.32497
128 x 128	0.473508	84.72056	178.9211

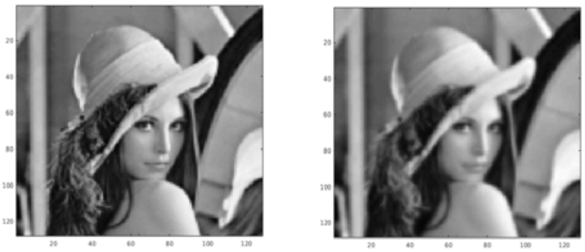

Fig. 4. Transient simulation of the MOS-C image Gaussian filter applied to a 128 x 128 pixels image for σ =8.9.

Matrix A is diagonally dominant and negative definite. Equation (5) has been obtained through nodal analysis and manual transformation. However, for circuits with increased complexity, the method described in [17] can be useful.

The simulation technique described in previous section has been applied to the system described in equations (7) to (9) for different values of square imagers. The model has been described in C++ and programmed on a NVIDIA GPU following Algorithm 1. The algorithm has been coded so that each thread computes the value of a single state variable $v_{i,j}$. Table I details the processor time required for a 1 µs transient simulation for different number of pixels. The many-core processor used has been a NVIDIA GeForce GTX 1080, 3584 Core, 1531MHz and 11 GB of RAM GPU. To evaluate the speed up of the proposed technique, a MOS-C model of the imager has been also simulated using Ngspice on an AMD Ryzen Threadripper 1950X 16-Core Processor, 2180 MHz and 64 GB of RAM.

The table shows the average simulation times for five runs for each image size and for each one of the methods, the proposed explicit one parallelised on GPU, and implicit on CPU. For all the image sizes, the explicit method is faster than the implicit one, being of two orders of magnitude for 128 x 128 pixel images. Fig. 4 shows an example of an 128x128 pixels image Gaussian filtering for σ =8.9. For values of C=250 pF and a channel resistance of 10 Ω the channel resistances have been ON during 100 ns.

IV. CONCLUSION

This paper has presented a technique to speed up the simulation of computer vision functions implemented as analog CMOS circuits in complex vision chips. The technique is based on the combination of state variables modelling of analog circuits with explicit integration schemas parallelised over a many-core computer. Although the proposed technique has been demonstrated modelling and simulating the hardware implementations of a Gaussian filtering function in a CMOS vision chip, it can be used to model and simulate any other function. Moreover, the technique can be used to model and accelerate the simulation of other types of analog circuits.

ACKNOWLEDGEMENTS

This work has been partially funded by Spanish government through project RTI2018-097088-B-C33 and by EPSRC (the UK Engineering and Physical Sciences Research Council) under grant EP/N0317681/1. The research stays at University of Southampton (UK) have 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 and by Universidad Politécnica de Cartagena - Campus de Excelencia Internacional Mare Nostrum.

REFERENCES

[1] Sze, V. et al., "Efficient Processing of Deep Neural Networks: A Tutorial and Survey". Proc. of IEEE, vol. 105, no. 12, pp. 2295-2329, Dec. 2017.

[2] J. Fernandez-Berni and R. Carmona-Galan, "All-MOS implementation of RC networks for time-controlled Gaussian spatial filtering", *Int. J. Circ. Theor. Appl.* 2012; vol. 40, pp. 859–876

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

[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] 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.

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

[7] 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.

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

[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 Dist. Systems, vol. 26, pp. 786-795, March 2015.

[10] 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.

[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*.* doi: 10.1109/TVLSI.2018.2858014.

[12] 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

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

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

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

[16] B.E. Jahne "Multiresolutional signal representation". In Chapter 4, Handbook of Computer Vision and Applications. Volume 2: Signal Processing and Pattern Recognition, Academic Press, 1999.

[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.