Electrostatic discharge simulation using a GPU-accelerated DGTD solver targeting modern graphics processors

Matteo Cicuttin[1], Anthony Royer[1], Peter Binde[2] and Christophe Geuzaine[1]
[1] University of Liège, Montefiore Institute B28 B-4000 Belgium, matteo.cicuttin@uliege.be
[2] Dr. Binde Ingenieure Design & Engineering

The presence of graphics processors (GPUs) in supercomputers constantly increased in the last decade. FDTD and DGTD (Time Domain Discontinuous Galerkin) are traditionally employed on GPUs for their scalability, however the limitations of past hardware required particular care in the implementation in order to obtain good performance. In this work, we discuss an implementation of DGTD for the Maxwell's equations on modern GPUs and we assess its performance on the simulation of an electrostatic discharge.

***Index Terms*—Electrostatic discharge, Discontinuous Galerkin, GPU**

I. Introduction

Recent supercomputer designs increasingly rely on GPU accelerators, and this trend is unlikely to change in the foreseeable future. Because of their impressive computing power and low cost per GFlop, GPUs have always been attractive for the scientific computing community. Their usage in computational electromagnetics (CEM) can be traced back to [1], targeting FDTD, and [2] targeting Discontinuous Galerkin (DG). The latter method is especially attractive on GPU, as it combines the advantages of FEM (handling of unstructured meshes) and FDTD (time-explicit iteration and easy parallelization). Early GPUs however, required careful implementation in order to not incur into major performance penalities, as studied in [2].

Recently, renewed interest in CEM on GPU is found for example in [3] for the Discrete Geometric Approach method and in [4], [5] for FDTD. In this work we revisit the implementation strategy of [2], to determine to which extent the improvements found in recent hardware (CUDA compute capability $\geqslant$ 3.5) would allow an algorithmic simplification without compromising efficiency.

II. Problem setting

Let $t \in \mathbb{R}^+$ be the time and $\boldsymbol{x} \in \mathbb{R}^3$ be the position vector. The first-order formulation of the time-domain Maxwell equations reads

$$\epsilon \frac{\partial \boldsymbol{e}(\boldsymbol{x},t)}{\partial t} = \nabla \times \boldsymbol{h}(\boldsymbol{x},t) - \sigma \boldsymbol{e}(\boldsymbol{x},t) - \boldsymbol{j}_s(\boldsymbol{x},t), \quad (1)$$

$$\mu \frac{\partial \boldsymbol{h}(\boldsymbol{x},t)}{\partial t} = -\nabla \times \boldsymbol{e}(\boldsymbol{x},t), \quad (2)$$

where $\boldsymbol{e}$ and $\boldsymbol{h}$ are the electric and magnetic fields, μ, ϵ and σ are the magnetic permeability, the electric permittivity and the conductivity respectively, and $\boldsymbol{j}_s$ is the source current density.

Spatial discretization of equations (1) and (2) is obtained using a DG approach [2], [6], [7], whereas for temporal integration either a fourth-order Runge-Kutta or a second-order leapfrog scheme [7] can be used. Our implementation builds on GMSH [8] and supports arbitrary polynomial order. Differently from [2] it runs in double precision, supports curved elements and integration can be performed using full Gauss quadratures, giving some flexibility in the handling of materials. In this work however, only the quadrature-free linear tetrahedral element part of the implementation is discussed. The DGTD/RK4 formulation is classical [6], whereas the semi-discrete DGTD/Leapfrog formulation yields the staggered-in-time equations

$$\boldsymbol{E}^{n+1} = \boldsymbol{E}^n + \Delta t\, \mathcal{L}_E^h(\boldsymbol{H}^{n+1/2}, \boldsymbol{E}^n), \quad (3)$$

$$\boldsymbol{H}^{n+3/2} = \boldsymbol{H}^{n+1/2} + \Delta t\, \mathcal{L}_H^h(\boldsymbol{E}^{n+1}, \boldsymbol{H}^{n+1/2}), \quad (4)$$

where Δt is the timestep size, n is the timestep number and $\mathcal{L}_E^h, \mathcal{L}_H^h$ are the standard DG operators which include element contributions, inter-element numerical fluxes and sources; for space reasons we refer the reader to [7, Sec. V-A.1] for their precise definition, including the discussion of sources and boundary conditions.

III. Discontinuous Galerkin on modern GPUs

The DG discretization of (1) and (2) results, because of integration by parts, in a sum of volumetric contributions and interface contributions known as numerical fluxes [7]. In turn, an application of either $\mathcal{L}_E^h$ or $\mathcal{L}_H^h$ results in two separate compute paths, as detailed in Figure 1. At the discrete level,

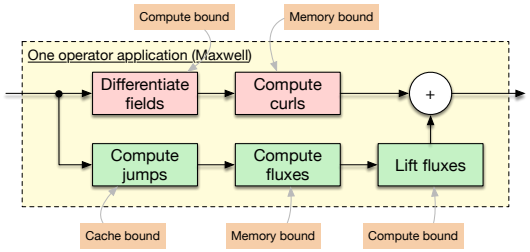

Fig. 1. Computations involved in one application of the DG operators $\mathcal{L}_E^h$ or $\mathcal{L}_H^h$. Red boxes (upper path) represent volumetric operations, whereas green boxes (lower path) represent boundary operations.

Manuscript received June 20, 2042; revised July 20, 2042; accepted September 20, 2042 Date of publication November 20, 2042; date of current version February 20, 2042. Corresponding author: M. Cicuttin (email: matteo.cicuttin@uliege.be). Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org. Digital Object Identifier 10.1109/TMAG.2042.0000000.

$\mathcal{L}_E^h$ and $\mathcal{L}_H^h$ yield a differentiation matrix, which operates on volumetric quantities, and a lifting matrix, which operates on numerical fluxes between elements.

The volumetric path, marked in red in Figure 1, is entirely element-local. For each element, the DoFs of the six fields (x, y, z components of $\boldsymbol{e}$ and $\boldsymbol{h}$) are multiplied by the reference element differentiation matrices; the result is subsequently brought back to the physical element via the Jacobians. A vector subtraction then yields the curls. Differentiation matrices, which are stored already pre-multiplied by the inverse mass matrix, are placed in the texture cache, whereas Jacobians are fetched directly from the global GPU memory. As locality and arithmetic intensity on this path are ideal, full GPU utilization is easily achieved. Differently from [2], our approach does not make use of shared memory, so we are not doing any microblocking [2] or padding. In turn, this simplifies the code considerably without noticeable performance impact.

The flux path, marked in green in Figure 1, begins with the computation of inter-element jumps, which are subsequently stored into six auxiliary arrays. This operation completely lacks locality, as the DOFs of two adjacent elements could be located very far apart in memory. It also has very low arithmetic intensity as it consists of just a subtraction. At least in principle then, this step could quickly become the bottleneck of the algorithm. For this reason in [2] this part has been the subject of a careful analysis, which led to a clever element-reordering algorithm. We observed however that, on recent hardware, architectural improvements of the memory hierarchy made this non-locality not problematic, especially at high polynomial orders: in our implementation, this stage of the algorithm achieves full memory bandwidth utilization. The subsequent step of computing fluxes from jumps has perfect locality and is limited only by the available memory bandwidth. Finally, flux lifting is a plain matrix-vector multiplication applied element-wise on the flux DoFs; we store the reference-element lifting matrix premultiplied by the mass matrix in the texture cache. In some situations lifting can not reach ideal performance as data reuse is not optimal; we are currently studying improved implementation strategies.

The results of the two paths are finally added and time integration is performed. This phase has ideal memory access patterns and is limited only by the available bandwidth.

The two paths are computed one after the other; we tried different scheduling strategies (especially the usage of multiple streams on GPU) without observing significant differences.

Volumetric field sources are subsequently applied by adding the appropriate terms at the end of the volumetric path, whereas interface sources (e.g. plane-wave condition [9]) are applied by modifying the flux terms computed in the second step of the flux path [7]. In the GPU version of the solver, we exploit asynchronous CPU/GPU computation to evaluate on the CPU the sources for timestep $n+1$ at the same time the GPU is computing timestep n. The source contributions are subsequently uploaded to the GPU with an asynchronous copy involving only the strictly needed DOFs.

A goal of our implementation is to be compatible with both NVidia and AMD hardware, however we do not consider the AMD toolchain sufficiently mature yet. Therefore, we chose to do our implementation in native CUDA and rely on the Hipify tool to generate AMD code on the fly at compile-time. Despite the automatic translation, the code resulting from our approach has very good performance on AMD hardware (Table V). Performance portability frameworks like OpenACC and Kokkos were also considered; we determined however that, at the time of writing, they do not provide sufficient control over the GPU hardware to efficiently implement DG.

IV. Solver validation

Our solver consists in a CPU and a GPU implementation of the DG method, and is available at the address https://gitlab.onelab.info/gmsh/dg/. The code is written in C++17 and targets distributed memory HPC machines. The CPU code is parallellized with MPI; METIS is used for mesh partitioning. The GPU code is based on CUDA and currently limited to single GPU. Current GPUs come with fairly large amounts of memory ($\geqslant$ 32GB) and real world models are easily handled: the DG/RK4 solver (worst case) allows for roughly 6.27 million DOFs per GByte of available GPU memory, or 250 million DOFs on a NVidia A100 GPU. In order to tackle very large scale problems however, multi-GPU support is currently being added.

A. Convergence analysis and cost estimation

In order to perform a basic validation of the solver, convergence rates of the error $||\boldsymbol{e}-\boldsymbol{e}_h||_{L^2(\Omega)}$ between the analytical solution $\boldsymbol{e}$ and the numerical solution $\boldsymbol{e}_h$ were measured on the $[0,1]^3$ resonant cavity model problem using three successively refined meshes with element size $h \in \{0.2, 0.1, 0.05\}$. In addition, we compared the accuracy of the DG approximation against a FEM solution of the second-order formulation

$$\nabla \times \frac{1}{\mu} \nabla \times \boldsymbol{e}(\boldsymbol{x},t) + \sigma \frac{\partial \boldsymbol{e}(\boldsymbol{x},t)}{\partial t} + \epsilon \frac{\partial^2 \boldsymbol{e}(\boldsymbol{x},t)}{\partial t^2} = -\frac{\partial \boldsymbol{j}_s(\boldsymbol{x},t)}{\partial t}.$$

This formulation is solved with our FEM code [11] on the same meshes used for the DG computation. While multiple FEM strategies are possible, in order to have numerical properties similar to DG, we chose to do temporal integration via a Newmark $\beta = 1/2, \gamma = 1/4$ scheme which, like the Leapfrog scheme, is non-dissipative. Despite the more relaxed timestep requirements of FEM, we chose $\Delta t = 10^{-11}s$ for both methods, which ensures at the same time that (i) DG is stable and (ii) the error in FEM is dominated by spatial discretization.

In a comparison between DG/RK4 and FEM on the same mesh, DG provides a slightly more accurate solution (Tables I and II). DG/Leapfrog however, especially at order 2, exhibits a loss of convergence due to the insufficient accuracy of the time integration (Tables I and III). In order to restore the correct convergence order and an accuracy comparable with FEM, the DG timestep had to be lowered by one order of magnitude ($\Delta t = 10^{-12}s$), implying a 10 times increase in the number of iterations (and thus computational cost) needed to arrive at the same final time.

The FEM formulation linear system is solved with MUMPS, therefore the total computational time t_{FEM} of a FEM run of

n iterations is split between an one-time matrix factorization (T_f) and a backsubstitution at each time iteration (T_b), yielding $t_{FEM} = T_b \cdot n + T_f$. On the other hand, DG spends all the computational time in the iterations (T_d), therefore $t_{DG} = T_d \cdot n$. The two expressions cross at $n = \frac{T_f}{T_d - T_b}$, where a negative result indicates that the DG solver is asymptotically faster than FEM in the considered setting. For DG/RK4 this is always the case (Tables I and II), whereas for DG/Leapfrog, $10 \cdot T_d < T_b$ in all cases except the smaller one (Table III).

TABLE I
FEM ERRORS AND TIMES FOR ORDERS 1 AND 2 ($\Delta t = 10^{-11}$).

	FEM order 1			FEM order 2		
h	Error	T_f	T_b	Error	T_f	T_b
.2	9.37e-3	8.5e-3	4.2e-4	1.21e-3	5.7e-2	3.9e-3
.1	2.85e-3	1.3e-1	7.3e-3	2.09e-4	1.1e0	4.9e-2
.05	7.45e-4	3.5e0	1.1e-1	2.33e-5	3.7e1	6.5e-1

TABLE II
DG/RK4 ERRORS AND TIMES FOR ORDERS 1 AND 2 ($\Delta t = 10^{-11}$).

	DG/RK4 order 1		DG/RK4 order 2	
h	Error	T_d	Error	T_d
.2	2.85e-3	4.10e-4	5.90e-4	1.04e-3
.1	1.07e-3	1.20e-3	8.04e-5	5.83e-3
.05	3.88e-4	3.34e-2	1.19e-5	7.92e-2

TABLE III
DG/LEAPFROG ERRORS AND TIMES FOR ORDERS 1 AND 2. ERRORS ARE REPORTED FOR $\Delta t = 10^{-11}$ AND FOR $\Delta t = 10^{-12}$ (SEE TEXT).

	DG/Leapfrog order 1			DG/Leapfrog order 2		
h	10^{-11}	10^{-12}	T_d	10^{-11}	10^{-12}	T_d
.2	2.85e-3	2.74e-3	1.3e-4	7.52e-4	5.76e-4	2.8e-4
.1	1.31e-3	1.02e-3	3.9e-4	4.93e-4	9.14e-5	1.3e-3
.05	7.63e-4	3.81e-4	9.8e-3	3.62e-3	4.61e-5	2.1e-2

The choice of a direct solver is dictated by the fact that, as it is well known, the second-order formulation yields a linear system which is problematic for iterative solvers. The resource usage of direct solvers however severely limits the size of the problem (Table IV) if sophisticated strategies like domain decomposition [12] are not used.

TABLE IV
MEMORY CONSUMPTION OF THE TWO SOLVERS. FOR DG WE SHOW THE DG/RK4 DATA, WHICH REPRESENTS THE WORST CASE.

	FEM order 1		FEM order 2	
h	DOFs	Mem	DOFs	Mem
.2	1.1k	46M	5.3k	94M
.1	8.7k	166M	39k	805M
.05	76k	1.48G	324k	9.23G
	DG order 1		DG order 2	
.2	17k	34M	43k	34M
.1	112k	62M	281k	105M
.05	872k	414M	2.18M	759M

B. CPU and GPU performance validation

Our hardware includes NVidia K20X and V100 GPUs; AMD MI100 GPU; Xeon Gold 6126, Ryzen 5 3600X and EPYC 7542 CPUs. The GPUs are HPC-grade and capable of double precision. At the time of writing, the MI100 is the AMD's latest model and is the performance-equivalent of the latest NVidia hardware, the A100. In turn, the A100 is one generation ahead of the V100. Finally, the K20x is used in some of our development machines.

TABLE V
PERFORMANCE IN DOFS/S OF THE SOLVER ON DIFFERENT GPUS.

Order	K20X	V100	MI100
1	3.43e8	1.24e9	1.50e9
2	4.01e8	1.44e9	1.83e9
3	3.80e8	1.38e9	1.84e9

The variety of CPUs/GPUs we used allowed to validate the consistency of our approach across multiple vendors and generations of hardware. Using the GPU diagnostic tools we determined that, at order 3, the V100 draws about 140W (9.86e6 DOFs/s per watt), compared to 260W for the MI100 (7.08e6 DoFs/s per watt). The metric "DoFs/s" does not include auxiliary DoFs therefore, for DG, DoFs $= 6 \cdot \#T \cdot N^k$, where 6 is for the $\{x, y, z\}$ components of $\boldsymbol{e}$ and $\boldsymbol{h}$, $\#T$ is the number of mesh elements and N^k is the local basis size.

V. SIMULATION OF AN ESD DISCHARGE

As real-world test case we simulated a standard ESD test on the device depicted in Figure 2 using the technique described in [10]; we refer the reader to that work for the details. The

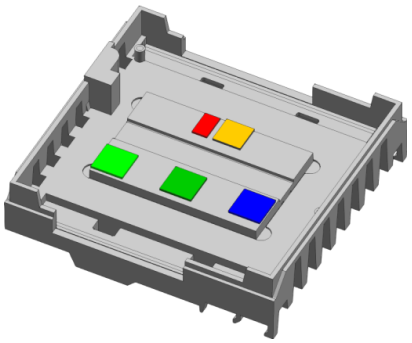

Fig. 2. An discharge is applied to the top-right corner of an heatsink (gray) containing a PCB (not shown) with some integrated circuits (colored boxes).

simulated device consists of an aluminium box ($\sigma = 36.9e6$) containing a PCB (not shown) with some integrated circuits (Figure 2). After establishing the appropriate initial electrostatic field [10] between the device and the tip of the ESD gun (aluminum cylinder at the top-right corner of the device, Figure 3) a normalized CENELEC discharge was applied [10] (Figure 5). The device is enclosed in an air sphere which, in turn, is terminated with a Silver-Müller condition. The domain was discretized with two meshes, a low resolution one and a high resolution one. The first yields a problem of 1.8M DoFs, whereas the second yields 12.96M DoFs. Order 1 DG was used, as the element size is constrained by geometry details.

The strong scaling analysis of the CPU DG code on a single node, dual socket machine based on the AMD EPYC 7542 is reported in Figure 4. Data indicates that scaling is almost ideal up to 16 processors. Beyond, the 16 memory channels

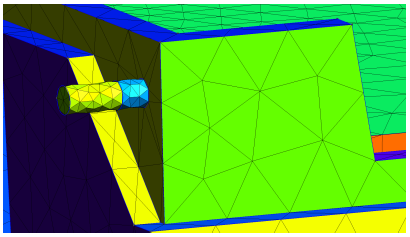

Fig. 3. The yellow cylinder models the ESD gun tip, whereas the blue cylinder models a 2mm air gap between the tip and the device.

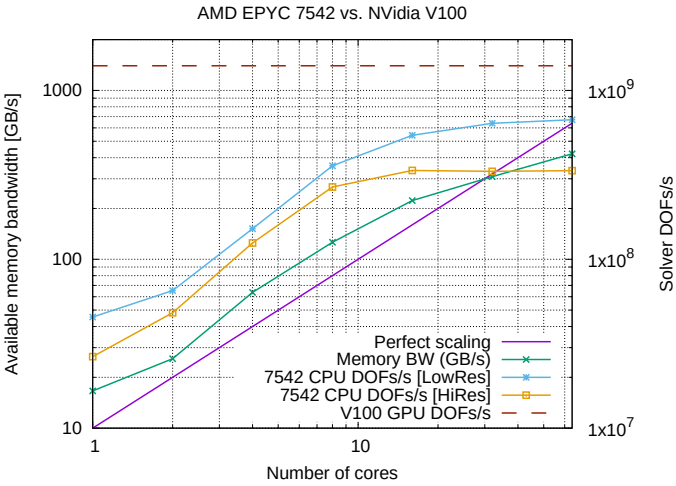

Fig. 4. CPU code scales linearly up to 16 processes. Above 16 processes, the available bandwidth limits scalability. GPU code is single-core and performance is given as reference.

available on the machine are saturated and the additional observed speedup is limited: we remark the matching between the scaling curves and the available memory bandwidth curve. Because of the intrinsic limits of direct solvers, FEM scales worse than DG. On the high resolution mesh we measured 91k DoFs/s on 1 CPU, 186k on 4 CPUs and 202k on 8 CPUs.

The GPU solver always runs with one process and speedups relative to the CPU solver are reported in Table VI. A

TABLE VI
DG PERFORMANCE ON THE TWO ESD TEST CASES AND GPU SPEEDUP

	Low resolution			High resolution		
CPUs	DOFs/s	T_d	GPU	DOFs/s	T_d	GPU
1	4.49e7	3.99e-2	31.2x	2.65e7	0.487	52.8x
2	6.51e7	2.76e-2	21.5x	4.82e7	0.269	29.1x
4	1.94e8	9.26e-3	7.2x	1.25e8	0.103	11.2x
8	3.66e8	4.91e-3	3.8x	2.68e8	0.048	5.2x
16	6.74e8	2.66e-3	2.1x	2.46e8	0.052	5.8x

2.1x speedup in the worst case may seem limited, however economic considerations must be taken into account: the price and the power consumption of a large high-end machine like the one we used in our tests surpass considerabily those of a GPU accelerator card. GPUs therefore represent a concrete computational resource for midrange technical workstations.

VI. CONCLUSIONS AND FUTURE PERSPECTIVES

We provided an account about our experience of implementing DG methods on GPUs. The main outcome is a confirmation that DG on recent GPU hardware can outperform the most high-end CPUs by a considerable margin, especially if economics are taken into account. In addition, compared to past GPUs, recent improvements allow to attain high levels of performance with a reduced algorithm design effort. Our future work will be focused in developing and benchmarking the multi-GPU DG method.

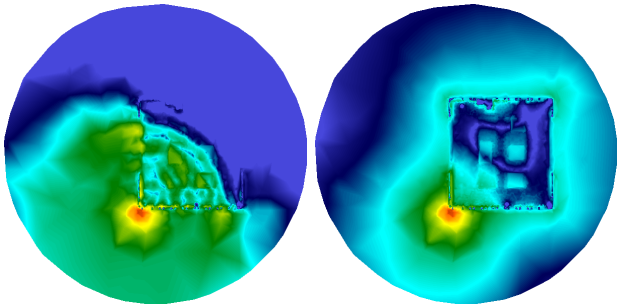

Fig. 5. Electric field magnitude during the rising of the discharge pulse (left), and after approximately 60ns (right).

VII. ACKNOWLEDGEMENT

This work was funded in part by the PRACE EU H2020 project (grant agreement 823767) and by the Wallon Region (M&SScot Network project, Mecatech call 26).

REFERENCES

[1] Takada, N., Shimobaba, T., Masuda, N., Ito, T., "High-speed FDTD simulation algorithm for GPU with compute unified device architecture". *2009 IEEE Antennas and Propagation Society International Symposium*

[2] A. Klöckner, T. Warburton, J. Bridge, J. S. Hesthaven, "Nodal discontinuous Galerkin methods on Graphics processors," *J. Comp. Phys. 228, 2009*.

[3] M. Cicuttin, L. Codecasa, B. Kapidani, R. Specogna, F. Trevisan, "GPU accelerated time domain DGA method for wave propagation problems on tetrahedral grids,". *IEEE Trans. Magn., Vol. 54(3), March 2018*

[4] C. Warren, A. Giannopoulos, A. Gray, I. Giannakis, A. Patterson, L. Wetter, A. Hamrah, "A CUDA-based GPU engine for gprMax: Open source FDTD electromagnetic simulation software", *Comput. Phys. Commun. Vol. 237, 2019*

[5] R. Calatayud, E. Navarro-Modesto, E. A. Navarro-Camba, N. T. Sangary, "Nvidia CUDA parallel processing of large FDTD meshes in a desktop computer: FDTD - matlab on GPU", *EATIS '20: Proc. of the 10th Euro-American Conf. on Telematics and Information Systems, Nov. 2020*

[6] J. S. Hesthaven, T. Warburton, "Nodal Discontinuous Galerkin Methods", Springer-Verlag New York, 2008

[7] L. Diaz Angulo, J. Alvarez, M. Fernández Pantoja, S. Gonzales Garcia, "Discontinuous Galerkin Time Domain Methods in Computational Electrodynamics: State of the Art", *Forum for Electromagnetic Research Methods and Application Technologies (FERMAT)*. Aug. 2015

[8] C. Geuzaine, J.-F. Remacle. "Gmsh: a three-dimensional finite element mesh generator with built-in pre- and post-processing facilities." *Int. J. Numer. Methods Eng. 79(11), pp. 1309-1331, 2009*

[9] S. Chialina, M. Cicuttin, L. Codecasa, R. Specogna, F. Trevisan, "Plane wave excitation for frequency domain electromagnetic problems by means of impedance boundary condition" *IEEE Trans. Magn., Vol. 51(3), 2015*

[10] M. Angeli, E. Cardelli, "Numerical modeling of electromagnetic fields generated by electrostatic discharges", *IEEE Trans. Magn., Vol. 33(2), March 1997*

[11] A. Royer, E. Béchet, C. Geuzaine, "GmshFEM: An Efficient Finite Element Library Based On Gmsh." *14th World Congress on Computational Mechanics (WCCM), ECCOMAS Congress, 2020*

[12] B. Thierry, A. Vion, S. Tournier, M. El Bouajaji, D. Colignon, N. Marsic, X. Antoine, C. Geuzaine, "GetDDM: An open framework for testing optimized Schwarz methods for time-harmonic wave problems", *Comput. Phys. Commun., Vol. 203, 2016*