Dispersive Möbius Transform Finite Element Time Domain Method on Graphics Processing Units

David Abraham[1], and Dennis D. Giannacopoulos[1], *Senior Member, IEEE*

[1]Department of Electrical & Computer Engineering, McGill University, Montréal, QC H3A 0E9 Canada
david.abraham@mail.mcgill.ca, dennis.giannacopoulos@mcgill.ca

We demonstrate the novel use of graphics processing units (GPUs) in accelerating dispersive finite element time domain (FETD) methods based upon the Möbius (bilinear) z-transform technique. By utilizing the immense computational potential of modern GPUs via NVIDIA's Compute Unified Device Architecture (CUDA) language, we are able to diminish the gap between dispersive FETD methods and their non-dispersive counterparts, facilitating the study of a wider range of physical phenomena. Our analyses indicate that the amount of performance gain achieved is directly related not only to the number of variables, but also to the amount of dispersive material present in the problem, with very large majoritarily dispersive problems seeing the most improvement.

***Index Terms*—Dispersive media, Finite element time domain method, Graphics processing unit, Parallel processing.**

I. Introduction

Over the past several years, three principle extensions to the finite element time domain (FETD) method have been proposed in order to accommodate the effects of dispersive media within electromagnetics simulations: recursive convolution, auxiliary differential equation, and Möbius z-transform [1]-[2]. While these methods have all succeeded in incorporating dispersive effects, the Möbius z-transform technique is, in general, more efficient and more versatile. However, owing to their additional complexity, dispersive FETD methods have remained naturally slower than their non-dispersive counterparts.

In this paper we seek to narrow the performance gap between dispersive and traditional FETD computations by investigating the use of Graphics Processing Units (GPUs) and their massively parallel architectures. While many aspects of the standard FETD algorithm have already seen GPU implementations [3], little work has been done in addressing the additional complexity inherited by FETD methods due to the presence of dispersion. Our innovation, then, is to isolate and accelerate the additional overhead imposed specifically by the dispersive elements. As such, the results obtained herein can be easily coupled to existing GPU implementations of FETD methods, for even greater computational efficiency.

In doing so, we hope to render dispersive computations marginally more expensive than their traditional counterparts, allowing for a more accurate characterization of physical phenomena, without the debilitating overhead.

II. The Möbius (bilinear) transform method

The bilinear transform method builds upon the standard FETD formulation for the second order vector wave equation, discretized via the Newmark-β scheme [4]. The inclusion of dispersion necessitates the introduction of convolutions between the fields and the material parameters within the FETD equations. However, we can transform the material's dispersive model to the z-domain by using a bilinear transform of the form:

$$j\omega = s \to \frac{2}{\Delta t}\frac{1-z^{-1}}{1+z^{-1}}\,. \tag{1}$$

In which Δt is the discrete time step used in our Newmark-β scheme. Inserting the transformed version of the dispersive model into the standard FETD equations and making use of the convolution and time shifting properties of the z-transform allows us to obtain a set of auxiliary variables and update equations for the permittivity as follows:

$$\{\mathcal{L}_\varepsilon\}^n = c_0[\mathcal{M}]\{E\}^n + \{W_1\}^{n-1} \tag{2}$$

$$\{W_\alpha\}^n = c_\alpha\{E\}^n - d_\alpha\{\mathcal{L}_\varepsilon\}^n + \{W_{\alpha+1}\}^{n-1}\ ;\ \alpha = 1,\cdots,p-1$$

$$\{W_\alpha\}^n = c_\alpha\{E\}^n - d_\alpha\{\mathcal{L}_\varepsilon\}^n\ ;\ \alpha = p$$

Where c_α and d_α are constants associated with the medium's electrically dispersive model, $[\mathcal{M}]$ is the mass matrix, $\{E\}$ is the electric field strength vector, p is the order of dispersion and lastly, $\{\mathcal{L}_\varepsilon\}$ and $\{W_\alpha\}$ are the auxiliary variables in question. Similar equations are likewise defined for the magnetic flux density and permeability. See [2] for a full treatment.

By including a combination of these auxiliary variables on the right-hand-side of the FETD update equation, the dispersion can be accurately modelled. As such, the overhead inherent to modelling the dispersion is tantamount to applying the updates (2) to the auxiliary variables in each time step of the solution process. It is these operations we now seek to parallelize.

III. Parallelization Strategy

Given that the update equations in (2) are composed entirely of matrix multiplications, vector scaling and vector addition, they are ideally suited to parallelization, since they contain many independent calculations. The Compute Unified Device Architecture (CUDA) language introduced by NVIDIA operates on the Single Instruction Multiple Thread (SIMT)

principle and is therefore aptly equipped for handling these types of operations, as each thread running on the GPU handles the computation of one element of the auxiliary vector, via identical operations on different data [5].

However, seeing as how the GPU and the host device do not share the same physical memory, it is necessary to transfer data between the two, resulting in additional overhead. From (2), it is clear that in order to obtain $\{\mathcal{L}_\varepsilon\}^{n+1}$ and $\{W_\alpha\}^{n+1}$, we require knowledge of $\{E\}^{n+1}$ upon each iteration. The mass matrix and constants will equally need to be transferred, but only once, before iterations begin, since they are invariant quantities. The algorithm within a single time step would then have the form as seen in Fig. 1, in which $\{Aux\}$ is the combination of auxiliary variables required to augment the FETD equations.

```
(…)                 // Compute {E}^{n+1}
cudaMemcpy          // Transfer {E}^{n+1} from host to GPU
<<< Kernel >>>      // Update auxiliary variables
{
    id = thread #;
    L_ε^{n+1}(id) = c_0 M(id,:){E}^{n+1} + W_1^n(id);
    (…)
}
cudaMemcpy          // Transfer {Aux}^{n+1} from GPU to host
```

Fig.1. Pseudocode to update the auxiliary variables on the GPU.

IV. Results

We now present preliminary results gathered using the above approach. The problem under consideration is a parallel plate waveguide with dimensions 20 cm by 4 cm, excited in the TEM mode, with first order absorbing boundary conditions at each end. A 4 cm doubly dispersive 4th order dielectric slab is present, such that the total amount of dispersive material varies between 25% and 90% of the total volume. All computations have been performed on a notebook computer running Windows 7 Home Premium edition, equipped with an Intel i7 Q740 CPU clocked at 1.73 GHz, with 4 GB RAM. The GPU is the main display card, an NVIDIA GeForce 310M with 16 CUDA cores.

In order to first put into perspective the significance of the dispersive overhead, we present Table 1 in which the percent of total execution time spent updating the auxiliary variables is reported as a function of the amount of dispersive material present and problem size, for 6000 time steps.

TABLE I
DISPERSIVE OVERHEAD AS A PERCENT OF TOTAL COMPUTATION TIME

Number of Variables	Proportion of Dispersive Material			
	25 %	50 %	75 %	90 %
23840	13.0 %	22.2 %	28.5 %	31.8 %
95680	10.5 %	17.7 %	23.2 %	25.6 %
383360	9.1 %	21.7 %	27.6 %	30.9 %

All code has been compiled using Visual Studio 2013 Ultimate edition and the NVIDIA NSIGHT plugin. Each code was run 10 times and the execution times averaged. Fig. 2 demonstrates the speedup achieved in updating the auxiliary variables during time stepping, including memory transfer overheads of $\{E\}$ and $\{Aux\}$, for 3 different sized problems.

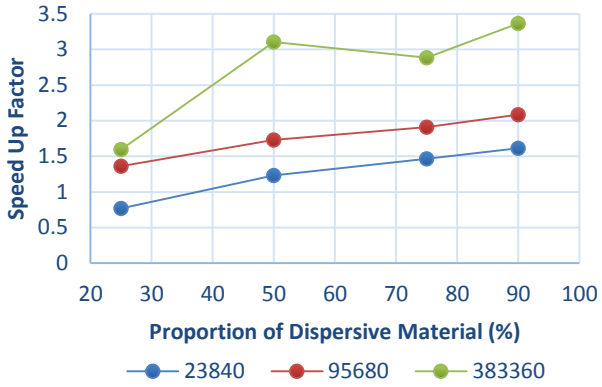

Fig. 2. Performance gain as a function of problem size and dispersiveness.

It is clear that for small problems containing little dispersive material, the GPU is not an efficient option due to the memory transfer overhead and may actually perform worse. However, as the amount of dispersive material increases, the number of operations required also increases, while memory overhead remains constant, improving efficiency. Lastly, as the number of variables grows the GPU has more data to work with, resulting in a greater use of the GPU's resources and better performance.

V. Conclusion

In conclusion, it has been demonstrated that GPUs have an excellent potential to diminish the performance gap which exists between dispersive and non-dispersive simulations. Given that good results were obtained with a relatively low power GPU (modern GPUs can contain many thousand CUDA cores), we are confident that very large improvements could be achieved on more modern hardware. Lastly, if this method were to be incorporated into an existing GPU FETD method, the performance would increase yet again, as memory transfer overheads could be removed from each iteration.

References

[1] F. L. Teixeira, "Time-Domain Finite-Difference and Finite-Element Methods for Maxwell Equations in Complex Media," *IEEE Trans. Antennas Propag.,* vol. 56, no. 8, pp. 2150-2166, Aug. 2008.

[2] A. Akbarzadeh-Sharbaf, D. D. Giannacopoulos, "A Stable and Efficient Direct Time Integration of the Vector Wave Equation in the Finite-Element Time-Domain Method for Dispersive Media," *IEEE Trans. Antennas Propag.,* vol. 63, no. 1, pp. 314-321, Jan. 2015.

[3] H.-T. Meng, B.-L. Nie, S. Wong, C. Macon, J.-M. Jin, "GPU accelerated finite-element computation for electromagnetic analysis," *IEEE Antennas Propagat. Mag.,* vol. 56, no. 2, pp. 39-62, Apr. 2014.

[4] J. Jin, "Finite Element Analysis in the Time Domain," in *The Finite Element Method in Electromagnetics,* 2nd ed. New York: Wiley-IEEE, 2002, ch. 12.

[5] N. Bell, M. Garland, "Efficient Sparse Matrix-Vector Multiplication on CUDA," NVIDIA Corp., Santa Clara, CA, Tech. Rep. NVR-2008-004, Dec. 2008.