FDTD Acceleration using MATLAB Parallel Computing Toolbox and GPU

Joseph E. Diener and Atef Z. Elsherbeni

Electrical Engineering and Computer Science Department
Colorado School of Mines, Golden, CO 80401, USA
jdiener@mines.edu, aelsherb@mines.edu

Abstract ─ We present a MATLAB based finite difference time domain (FDTD) method accelerated using the GPU functions in MATLAB's parallel computing toolbox (PCT). Procedures to achieve significant speedups over a CPU implementation of the same code are outlined. The use of specialized code with NVIDIA's compute unified device architecture (CUDA) programming results in impressive computational speedups. However, this requires specialized programming knowledge to efficiently implement. The MATLAB PCT can be applied directly to pre-existing MATLAB FDTD code and obtain reasonable speedups over equivalent CPU code. We demonstrate several modifications to increase the efficiency on several different NVIDIA graphics cards. Benchmarks are presented on problems of practical size (millions of cells) with a CPML terminated domain.

Index Terms ─ FDTD, GPU, MATLAB.

I. INTRODUCTION

For problems of practical size using FDTD method, domains on the order of tens of millions of cells with large number of time stepping need to be solved. This leads to correspondingly long computation times. Significant speedups in the computation time of FDTD solvers are possible by shifting the computation from the CPU to a GPU. While the most efficient solvers employ code specifically written to run on a GPU, often using CUDA kernels [1, 3-4, 6], this requires specialized programming and is non-trivial to implement. MATLAB is an easy to use high-level programming language available at many universities, currently available for many students and practicing engineers. Efficient implementations of FDTD code on MATLAB can also be an effective educational tool for electromagnetic simulations. This paper shows both the efficiency increases that the PCT allows over regular MATLAB based code, as well as some techniques that can be used to further optimize performance. Performance is benchmarked using the solver speed in millions of cells per second (MCPS), as suggested in [3]. This allows for easy comparison across different platforms and problem sizes. It is worth noting that the codes used for this paper are highly general, and no assumptions are made about uniform discretization to simplify the formulation. Thus, each updating coefficient array is unique and valid for the general FDTD formulation. Speeds of up to ~300MCPS on problems with only near-field excitations, and ~267MCPS with a total-field scattered field (TF/SF) plane-wave source are obtained. These results are slower than those presented using CUDA written codes [1], [8], which presented peak speeds of ~1600MCPS using an NVIDIA Titan-Z card. Our results are significant improvements on regular vectorized-CPU MATLAB code (~11->12 MCPS), and does not require any CUDA programming.

II. IMPLEMENTATION IN MATLAB

The second-order FDTD MATLAB implementation as given in [2] is used as a starting point for code modifications. MATLAB is most efficient with vectorized code, so the updating equations are written in vectorized form as much as possible. As a result, the base FDTD code represents an efficient and straightforward implementation on the MATLAB engine. Since GPU benchmarks are commonly done using single precision because of the superior computing ability on GPUs for single precision problems, single precision is used for the CPU as well. By default, MATLAB operates in double precision, but can be cast into single precision through the single() function. Two benchmark cases are presented, where the solver runs for sufficient time-steps to ensure an accurate representation of the throughput. The first benchmarking problem is the excitation of a dielectric sphere of relative permittivity of 4 and radius of 1mm using the field radiated by a dipole antenna, and the second problem is the excitation of the same sphere using a plane wave. The reason for the two different configurations is to show the performance using a total field FDTD formulation versus a scattered field formulation performance. The CPU MATLAB code in this paper is written in an efficient vectorized style, which will perform matrix operations using multiple processor cores for simple functions (such as *, +, -) which constitute a large portion of the FDTD work load. To examine the efficiency of the code modifications over

Submitted On: July 15, 2016
Accepted On: March 18, 2017

1054-4887 © ACES

larger domain sizes, the problem space is discretized using increasingly smaller cubic cells. The computational domain boundary is terminated using an air buffer of 8 cells on each side from the sphere, and 6 CPML cells in each direction. The speed of the code in MCPS is presented over domains measured in millions of cells (MC). The problem execution time is measured using the tic/toc functions in MATLAB over the time marching loop, thus the one-time array initializations are not included. Additionally, all benchmarking is performed in MATLAB 2016b. Initial benchmarking is done using an NVIDIA Tesla K40C card, with the CPU code running on an Intel i7-4770 @3.4GHz (8 cores).

III. GPU IMPLEMENTATION

By moving the updating equations onto the GPU, substantially improved performance is obtained compared with the CPU implementation. The most direct implementation on the GPU is simply to call each array as its GPU equivalent, such as Ex = gpuArray(Ex). This requires no further modifications to the code than the above for each field variable and updating coefficients. Figure 1 shows the throughput obtained when directly porting each array into the GPU. For the dipole source, maximum speedups of ~10x are seen, with performance peaking around 115MCPS. The throughput of the GPU code increases with increasingly large domains, so that it never achieves a steady level of constant performance. In contrast, the CPU code has an essentially flat performance of 11 to 12MCPS for domains larger than ~2MC, and only slightly slower speeds for smaller problem sizes.

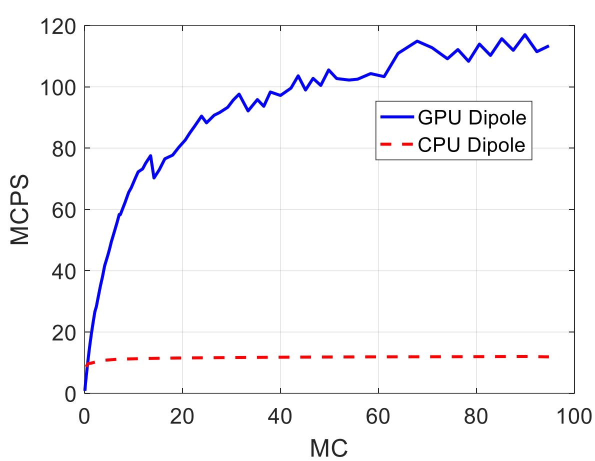

Fig. 1. NVIDIA Tesla K40C direct FDTD port compared against the CPU implementation of the code.

A. Optimize CPML with arrayfun

The updating of the fields in the CPML region on either the GPU or CPU is a significant portion of total computation time in comparison with the E and H field updating in the computational domain. In part this is due to the CPML being unable to be easily vectorized on the CPU. A variety of problems can be vectorized and show dramatic improvement in performance when using arrayfun() on the GPU [4-5, 7]. Here, we demonstrate the application of arrayfun() to the FDTD CPML absorbing boundary condition. The arrays must be of appropriate dimension to perform the scalar expansion employed in arrayfun(), which can necessitate reshaping the array in a preprocessing step. A snippet of the code listing for the arrayfun application to the updating of the Ez-field in the CPML region is shown in listing 1. Similar modifications are required for the other components in the CPML code to allow for arrayfun to be used for each field component. This modification is very efficient and results in much higher computational speeds as shown in Fig. 2. This yields peak speeds of ~157MCPS for the dipole source, a speedup of ~13x compared with respect to the CPU. Peak performance is for a problem size of ~24MC, with nearly steady performance for larger problem sizes.

```
cpml_b_ez_zn=reshape(cpml_b_ez_zn,1,1, ncpml_zn);
Psi_exz_zn=arrayfun(@times,cpml_b_ez_zn,
Psi_exz_zn)+arrayfun(@times,cpml_a_ez_zn,
Hy(:,:,1+cpmlznvec) - Hy(:,:,cpmlznvec) );

Psi_eyz_zn=arrayfun(@times,cpml_b_ez_zn,
Psi_eyz_zn)+arrayfun(@times,cpml_a_ez_zn,
Hx(:,:,1+cpmlznvec) - Hx(:,:,cpmlznvec) );

Ex(:,:,cpmlznvec+1)=Ex(:,:,cpmlznvec+1)+
CPsi_exz_zn.*Psi_exz_zn;
Ey(:,:,cpmlznvec+1)=Ey(:,:,cpmlznvec+1)+
CPsi_eyz_zn.*Psi_eyz_zn;
```

Listing 1. Modification of Ez CPML component from [2] using arrayfun on the GPU.

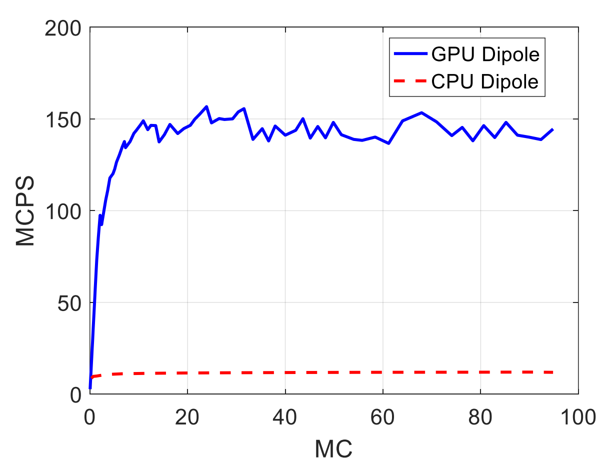

Fig. 2. Tesla K40C performance in comparison with the dipole benchmark on CPU using CPML modification.

B. Optimized E-field updating process

MATLAB tends to be fastest when using vectorized code, and is capable of performing element-wise computation very efficiently. Performance decreases when blocks of a matrix must be multiplied instead of the entire matrix. The staggered grid in FDTD creates differently sized field components, and this leads to explicit indexing in the updating equations for either the electric or magnetic field components, depending on which field terminates the computational domain. In our analysis, the electric field is used to terminate the computational domain. The electric field components are updated with the staggered magnetic field component differences, as shown in listing 2.

```
Ex(1:nx,2:ny,2:nz)=Cexe(1:nx,2:ny,2:nz).*
Ex(1:nx,2:ny,2:nz)+Cexhz(1:nx,2:ny,2:nz).*...
(Hz(1:nx,2:ny,2:nz)-Hz(1:nx,1:ny-1,2:nz)) ...
+ Cexhy(1:nx,2:ny,2:nz).*...
 (Hy(1:nx,2:ny,2:nz)-Hy(1:nx,2:ny,1:nz-1));
```

Listing 2. Ex updating equation for FDTD in MATLAB as presented in [2].

This means that portions of the electric field components and updating coefficients arrays are multiplied with portions of the magnetic field components. Quicker computation can be achieved by multiplying and writing to the entire electric field component arrays, without explicit indexing. By concatenating an array of zeros along the appropriate dimensions, the Hz array for updating Ex can be used without indexing, and becomes a simple vector operation. Listing 3 shows the equivalence of the two operations.

$$A = [Hz\ 0],$$
$$B = [0\ Hz].$$

Listing 3. Example of zero-padding arrays to accomplish indexless updating in electric field components.

Consider two new matrices, A and B. A is the 'right' zero padded matrix, and B is the 'left' zero padded matrix, where the zero padding is of appropriate size such that the resulting matrices have equal size to Ex. This removes any need to index A or B, and reproduces the staggered difference in Hz with the vector operation $A - B$. The first and last row in y will write incorrect values to Ex, but this can be handled by zeroing the updating coefficient array, $Cexhz(:,1,:) = 0$, $Cexhz\ (:,j+1,:) = 0$. Defining similar matrices $C = [Hy\ 0]$, $D = [0\ Hy]$, and zeroing the coefficient array $Cexhy(:,:,1) = 0$, and $Cexhy(:,:,end) = 0$, we can write the updating equation as: $Ex = Cexe * Ex + Cexhz * (A - B) + Cexhy * (C - D)$. This results in a speedup in MATLAB by removing the explicit subscripting that is ordinarily required. The example code listing below shows how the zero padding is performed within MATLAB:

$$A = \mathrm{cat}\big(2, \mathrm{Hz}, \mathrm{zeros}(\mathrm{nx}, 1, \mathrm{nz}+1)\big),$$
$$B = \mathrm{cat}(2, \mathrm{zeros}(\mathrm{nx}, 1, \mathrm{nz}+1), \mathrm{Hz}),$$

the updating modifications to the other electric field components are similar. The integer in the 'cat' command represents the dimension along which the zero array is appended. '1' corresponds to the x-axis, '2' corresponds to the y-axis, and '3' to the z-axis. The effect of these changes are seen in Fig. 3, where a large speedup is obtained over solely modifying the CPML. This concatenation approach yields peak speeds of ~220MCPS – an increase of ~70MCPS over just the CPML modification. This performance can be increased further by putting the various FDTD updating arrays into a single function call, with each component of the electric or magnetic field using an arrayfun() call on a sub-function that will update the field component. This is detailed in the listings given in the appendix. The CPML boundaries are similarly put into a single function that internally updates each boundary using the arrayfun() approach. The results of this optimal updating are shown in Fig. 4. This shows a maximum increase of ~90MCPS, yielding maximum performance of 310MCPS. With the optimal updating established, the analysis for the plane-wave benchmark is shown in Fig. 5. The peak performance in the plane wave is ~270MCPS, a full 40MCPS less than the dipole case. The overall shape of the two curves are very similar, with a performance loss incurred by the extra updating required in the TF/SF formulation.

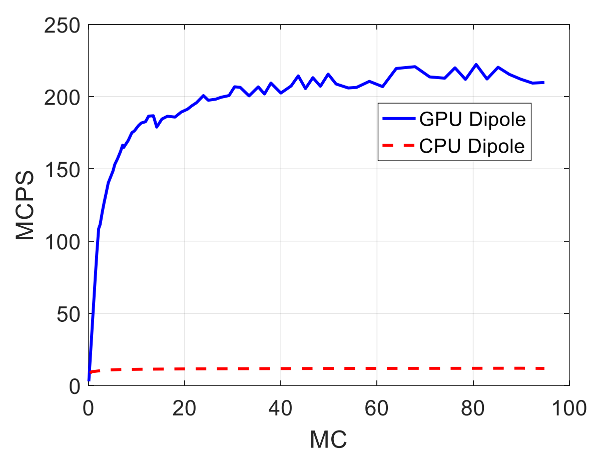

Fig. 3. NVIDIA Tesla K40C performance with concatenated E-field modification.

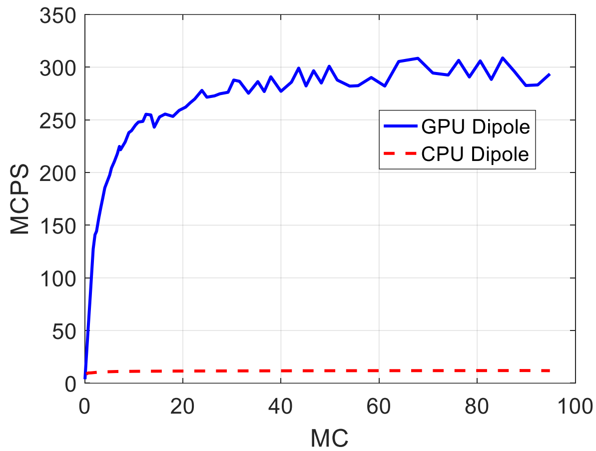

Fig. 4. NVIDIA Tesla K40C performance with optimal updating.

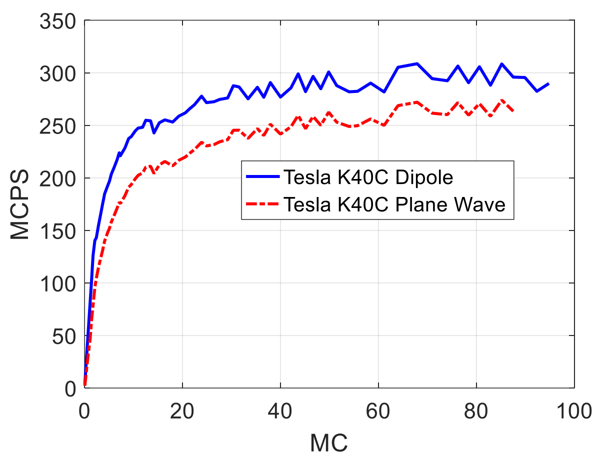

Fig. 5. Comparison of planewave benchmark and dipole benchmark on K40C.

IV. BENCHMARKING SEVERAL GPUs

With the completion of the optimization of the code, the same benchmarking analysis is performed on different NVIDIA graphics cards with each method, for both the plane wave and dipole cases. The analysis is restricted to the dipole case, as the plane wave results show essentially the same performance curves, with reduced maximum speeds. An NVIDIA GTX-780 (3GB) and NVIDIA Titan-Z (12GB) are chosen to compare the results of the developed code. While the Titan-Z nominally has 12GB of memory, it is spread across two 6GB processors on the same physical card, which are addressed separately within MATLAB. Thus, only 6GB of memory is addressable at a time in the current implementation. In Fig. 6, the comparison between the Tesla K40C, GTX-780, and Titan-Z is shown for the dipole case. Similar max speeds are obtained for each of the cards. The K40C has a maximum speed of 310MCPS, the GTX a maximum speed of 303MCPS, and the Titan-Z a maximum speed of 367MCPS. However, both the GTX and Titan-Z demonstrate a marked reduction in speed after hitting their peak performance – thus, an 'optimal' problem size is smaller than one using the K40C card. Since different versions of MATLAB can sometimes improve or even reduce the performance of different code [9], this behavior may change with different versions of MATLAB and the PCT.

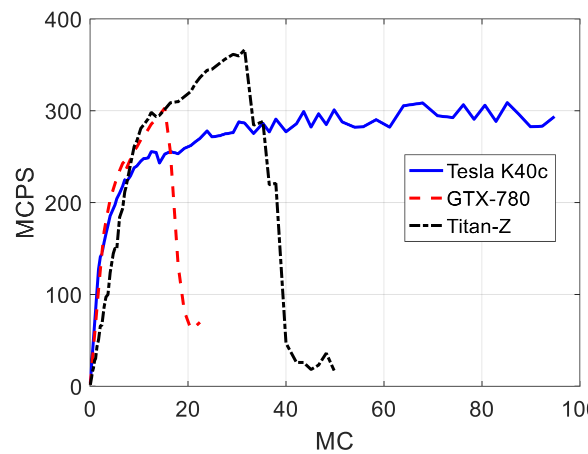

Fig 6. Comparison of optimal updating for three NVIDIA cards for dipole problem.

V. CONCLUSION

In this paper, the implementation of a FDTD solver in MATLAB using the parallel computing toolbox and its GPU computing capabilities is examined. The appendix lists in some detail functions from the best code developed in this paper. We avoid the use of specialized CUDA based programming in order to present an easy to implement code that can achieve substantial speedups in MATLAB. Code is benchmarked across several GPUs and problem types. Sizeable computational speeds on problems of practical sizes with CPML absorbing boundaries are achieved. A method of removing explicit indexing for one set of field-updating in the FDTD loop is presented that shows strong improvements on throughput that might be similarly useful in other vectorized programming languages.

APPENDIX

A more complete listing of the optimized updating code for a generic FDTD problem is given in this listing. First, the form of the electric field updating step within the main FDTD loop is shown in listing 4. The function outputs the updated electric field components, and takes as inputs the field components, updating matrices, and computational domain size.

```
[Ex, Ey, Ez] = updateEfields( Ex, Cexe, Cexhz, Cexhy,
    Ey, Ceye, Ceyhx, Ceyhz,  Ez, Ceze, Cezhx, Cezhy,
```

```
    Hx, Hy, Hz, nx, ny, nz);
```

Listing 4. Form of the electric field updating step in the time marching loop.

The function "updateEfields" contains the concatenated field updating as a separate step for each field component. Listing 5 shows this for the y-component of the field, with the equations and matrices A1 and B1 having a similar form for the other components.

```
A1 = zeros(nx+1, ny, 1, 'gpuArray');
B1 = zeros(1, ny, nz+1, 'gpuArray');

Ey = arrayfun(@updateEcomponent, Ey, Ceye, Ceyhx,
   cat(3, Hx, A1), cat(3, A1, Hx), ...
   Ceyhz, cat(1, Hz, B1), cat(1, B1, Hz));
```

Listing 5. Ey updating step within "updateEfields" function.

Finally, this calls the subfunction "updateEcomponent", which takes as input the appropriate field component, coefficient matrices, and auxiliary matrices. This is shown in listing 6. The function updates the input field with purely element wise operations.

```
function [A] = updateEcomponent(A, B, C, D, E, F, G,
H)

 A = A.*B + C.*(D - E) + F.*(G - H);
```

Listing 6. Function updateEcomponent.

A nearly identical set of functions are defined for the magnetic field components updating step. Similarly, the CPML updating step for the electric and magnetic fields are called as one function, which contains bsxfun() function calls for efficient updating. This is shown in listings 7 and 8.

```
[Hx, Hy, Hz, Psi_hyx_xn, Psi_hzx_xn, Psi_hzy_yn,
Psi_hxy_yn, Psi_hxz_zn, Psi_hyz_zn,   Psi_hyx_xp,
Psi_hzx_xp,    Psi_hzy_yp, Psi_hxy_yp,Psi_hxz_zp,
Psi_hyz_zp]  = update_magnetic_field_CPML_ABC
(Hx, Hy, Hz, Ex, Ey, Ez, cpml_b_mx_xn,
cpml_a_mx_xn, Psi_hyx_xn, Psi_hzx_xn,
CPsi_hyx_xn, CPsi_hzx_xn,  cpml_b_my_yn,
cpml_a_my_yn, Psi_hzy_yn, Psi_hxy_yn,
CPsi_hzy_yn, CPsi_hxy_yn,  cpml_b_mz_zn,
cpml_a_mz_zn, Psi_hxz_zn, Psi_hyz_zn, CPsi_hxz_zn,
CPsi_hyz_zn,  n_cpml_xn, n_cpml_yn, n_cpml_zn, ...
cpml_b_mx_xp, cpml_a_mx_xp, Psi_hyx_xp,
Psi_hzx_xp, CPsi_hyx_xp, CPsi_hzx_xp, ...
cpml_b_my_yp, cpml_a_my_yp, Psi_hzy_yp,
Psi_hxy_yp, CPsi_hzy_yp, CPsi_hxy_yp, ...
cpml_b_mz_zp, cpml_a_mz_zp, Psi_hxz_zp,
Psi_hyz_zp, CPsi_hxz_zp, CPsi_hyz_zp, ...
n_stmx, n_stmy, n_stmz, nx, ny, nz);
```

Listing 7. The function call for updating all the CPML boundaries within the domain for the magnetic field.

```
Psi_hyx_xn = bsxfun(@times, cpml_b_mx_xn,
Psi_hyx_xn)  + bsxfun(@times, cpml_a_mx_xn,
diff(Ez(1:n_cpml_xn+1, :,:) , 1, 1) );

Psi_hzx_xn =bsxfun(@times,  cpml_b_mx_xn,
Psi_hzx_xn)   + bsxfun(@times, cpml_a_mx_xn,
diff(Ey(1:n_cpml_xn+1, :,:) , 1, 1) );

Hy(1:n_cpml_xn, :,:) = Hy(1:n_cpml_xn, :,:)  +
CPsi_hyx_xn.* Psi_hyx_xn;
Hz(1:n_cpml_xn, :,:) = Hz(1:n_cpml_xn, :,:)  +
CPsi_hzx_xn.* Psi_hzx_xn;
```

Listing 8. The 'xn' boundary of CPML updating within the function call.

The bsxfun() call is used for updating the CPML matrices efficiently. A boolean check can be implemented for domains with mixed boundaries with little impact on the performance.

REFERENCES

[1] V. Demir and A. Z. Elsherbeni, "Compute unified device architecture (CUDA) based finite-difference time-domain (FDTD) implementation," *ACES Journal*, vol. 25, no. 4, pp. 303-314, April 2010.

[2] A. Z. Elsherbeni and V. Demir, *The Finite-Difference Time-Domain Method for Electromagnetics with MATLAB Simulations*. second edition, ACES Series on Computational Electromagnetics and Engineering, SciTech Publishing, an Imprint of IET, Edison, NJ, 2016.

[3] V. Demir, "A stacking scheme to improve the efficiency of finite-difference time-domain solutions on graphics processing units," *ACES Journal*, vol. 25, no. 4, pp. 323-330, April 2010.

[4] Illustrating three approaches to GPU Computing: the Mandelbrot Set, http://www.mathworks.com/help/distcomp/examples/illustrating-three-approaches-to-gpu-computing-the-mandelbrot-set.html, May, 2016.

[5] Improve Performance of Element-wise MATLAB Functions on the GPU using ARRAYFUN, http://www.mathworks.com/help/distcomp/examples/improve-performance-of-element-wise-matlab-functions-on-the-gpu-using-arrayfun.html, May, 2016.

[6] M. J. Inman, A. Z. Elsherbeni, and C. J. Reddy, "CUDA based LU decomposition solvers for CEM

applications," *ACES Journal*, vol. 25, no. 4, pp. 339-347, April 2010.

[7] Joss Knight, High-Performance MATLAB with GPU Acceleration, https://devblogs.nvidia.com/parallelforall/high-performance-matlab-gpu-acceleration, January 2017.

[8] V. Demir, A. Z. Elsherbeni, CEMS Software Package, based on [2], 2014.

[9] MATLAB Answers Forum Question, http://www.mathworks.com/matlabcentral/answers/239817-matlab-s-r2015b-new-jit-experiences-a-severe-degradation-in-speed-in-the-following-example-but-the, February 2017.

Joseph E. Diener obtained his bachelor's degree in Physics from the University of Puget Sound in 2013. Currently he is a Master's Student at the Colorado School of Mines studying Electrical Engineering. His research interests include finite difference time domain methods, antennas, microwave measurements, and phased arrays.

Atef Z. Elsherbeni received his Ph.D. degree in Electrical Engineering from Manitoba University, Winnipeg, Manitoba, Canada, in 1987. Elsherbeni was with the University of Mississippi from 1987 to 2013. He was a Finland Distinguished Professor from 2009 to 2011. In August 2013 he joined the Electrical Engineering and Computer Science Department at Colorado School of Mines where he is now the Dobelman Distinguished Chair Professor and the Head of the Electrical Engineering Department. His research interest includes the scattering and diffraction of EM waves, finite-difference time-domain analysis of antennas and microwave devices, field visualization and software development for EM education, interactions of electromagnetic waves with the human body, RFID and sensor integrated FRID systems, reflector and printed antennas and antenna arrays, and measurement of antenna characteristics and material properties. Elsherbeni is a Fellow Member of IEEE and ACES. He is the Editor-in-Chief for ACES Journal. He was the General Chair for the 2014 APS-URSI Symposium and was the President of ACES Society from 2013 to 2015.