Using MATLAB's Parallel Processing Toolbox for Multi-CPU and Multi-GPU Accelerated FDTD Simulations

Alec J. Weiss [1], Atef Z. Elsherbeni [1], Veysel Demir [2], and Mohammed F. Hadi [1]

[1] Department of Electrical Engineering
Colorado School of Mines, Golden, Colorado 80401, United States of America
aweiss@mines.edu, aelsherb@mines.edu, mhadi@mines.edu

[2] Department of Electrical Engineering
Northern Illinois University, DeKalb, Illinois 60115, United States of America
vdemir@niu.edu

Abstract — MATLAB is a good testbed for prototyping new FDTD techniques as it provides quick programming, debugging and visualization capabilities compared to lower level languages such as C or FORTRAN. However, the major disadvantage of using MATLAB is its slow execution. For faster simulations, one should use other programming languages like Fortran or C with CUDA when utilizing graphics processing units. Development of simulation codes using these other programming languages is not as easy as when using MATLAB. Thus the main objective of this paper is to investigate ways to increase the throughput of a fully functional finite difference time domain method coded in MATLAB to be able to simulate practical problems with visualization capabilities in reasonable time. We present simple ways to improve the efficiency of MATLAB simulations using the parallel toolbox along with the multi-core central processing units (CPUs) or the multiple graphics processing units (GPUs). Native and simple MATLAB constructs with no external dependencies or libraries and no expensive or complicated hardware acceleration units are used in the present development.

Index Terms — FDTD, MATLAB, multi-cores, multi-GPUs, parallel computing.

I. INTRODUCTION

The finite difference time domain (FDTD) method provides wide bandwidth simulations using time domain techniques to provide accurate, full wave electromagnetic simulations. The equations for the updating of field components can be solved through the use of finite difference approximations of Maxwell's equations. This method is inherently parallel as the update of field components of each cell within the simulation grid relies on field values all of which have been calculated at previous time and are within the vicinity of the cell. In their final form, the ease of parallelization becomes apparent. As an example, consider the updating equation for the x component the electric field:

$$\begin{aligned} E_x^{n+1} = {} & C_{exe}(i,j,k) \times E_x^n(i,j,k) \\ & + C_{exhz}(i,j,k) \times \left(H_z^{n+\frac{1}{2}}(i,j,k) - H_z^{n+\frac{1}{2}}(i,j-1,k) \right) \\ & + C_{exhy}(i,j,k) \times \left(H_y^{n+\frac{1}{2}}(i,j,k) - H_y^{n+\frac{1}{2}}(i,j,k-1) \right), \\ & + C_{exj}(i,j,k) \times J_{ix}^{n+\frac{1}{2}}(i,j,k), \end{aligned} \quad (1)$$

which is given from [1]. In this equation, we update e-fields using h-fields, currents, the previous value of the e-field, and a set of coefficients denoted with the letter 'C'. This equation demonstrates how each component is only updated based on previously calculated values, making it a prime candidate for relatively easy parallelization.

While most previous work focuses on parallelization of FDTD method in lower level programming languages using various software and hardware acceleration techniques such as in [2]–[6] (including extensive research on GPU acceleration), only a small number have researched parallelization using simpler languages such as MATLAB [7], [8]. This paper explores the use of MATLAB's parallel computing toolbox to extend the FDTD computation onto multiple cores of a central processing unit (CPU). The increased throughput of the simulation time observed on various systems using parallelization across multiple CPU cores is investigated. The research is then extended to look at the effectiveness of utilizing multiple graphics processing units (GPUs) for the same MATLAB based FDTD solver.

II. PARALLEL CPU COMPUTATION USING MATLAB

The MATLAB environment has two types of parallelization that can be utilized. These two types

Submitted On: April 29, 2019
Accepted On: May 31, 2019

are built-in parallelization, also known as implicit parallelization, and parallelism using MATLAB workers, also known as explicit parallelization [9], [10].

A. Implicit parallelization with MATLAB

Implicit parallelization in MATLAB takes place without any extra work required from the programmer. When performing operations such as elementwise multiplication, as done in many FDTD codes, MATLAB automatically will parallelize this to use multiple cores on a CPU. This will occur any time MATLAB finds a parallelizable operation of ample size that it thinks could be sped up by use of multiple cores. This provides speed increases for large and computationally dense problems. Unfortunately, this method gives the user no control over the parallelization of the code. It also is limited to a non-distributed machine and is unable to take advantage of capabilities of the GPU hardware. These locations where implicit parallelization is limited are places where explicit parallelization can be used better speed increases.

B. Explicit parallelization with MATLAB

Contrary to MATLAB's implicit parallelization, explicit parallelization does require extra work from the programmer to tell the system how to parallelize the code. This is done by first launching worker threads using the *parpool()* command. The number of workers to launch can be specified by putting a number as the first argument to this function. Once a parallel pool has been started in MATLAB, the single program multiple data (SPMD) keyword can be used to directly address each of the workers in the parallel pool. Once this keyword has been used, each of the workers can be individually addressed and communicate with one another using constructs similar to the message passing interface (MPI). We can access the index of each of the cores using the *labindex* variable and get the total number of cores in the pool with the *numlabs* variable.

III. EXPLICIT PARALLELIZATION OF MATLAB BASED FDTD SOLVER FOR CPUs

The code created and tested in this paper is an edited version of the code generated from [1]. Because of this, the FDTD solver is written purely in MATLAB and has many capabilities such as convolutional perfectly matched layer (CPML) boundary conditions, the ability to set material properties, and the capability to add lumped elements, sources, and test ports for full S-parameter simulations. Each of these capabilities is present in the final code providing a simulation environment that can be utilized for real world experiments. However at the current stage, it does not provide the near to far field capability.

Before parallelization could begin, some minor changes were required to be performed on the original code. This included changing all scripts (MATLAB m files) that will be called within the SPMD environment to be changed to functions. This is something that is required by MATLAB for transparency but also makes profiling and debugging in the SPMD environment easier and more informative. All functions used within the time marching loop along with some for data distribution and gathering fit into this scenario.

A. Distribution of domain

For the parallelization of the FDTD algorithm, the FDTD computational domain was split evenly amongst each of the cores. To minimize communication time and programming difficulty, the domain is only split along the z-axis. Distribution across only the z-axis is optimal because it reduces the number of block communications to a maximum of 2 whereas distribution in x-, y-, and z-axes would require up to 6 per core. The z-axis was chosen to decrease the data transfer time. Like all programming languages, MATLAB matrices are stored linearly in memory. In MATLAB arrays are accessed consecutively in memory first in x, then in y, and then in the z direction. This means that data in an xy plane is all accessed consecutively from memory therefore communicating data in an xy plane across the z direction is more efficient than communicating in other directions. A visualization of the subdomains can be seen in Fig. 1.

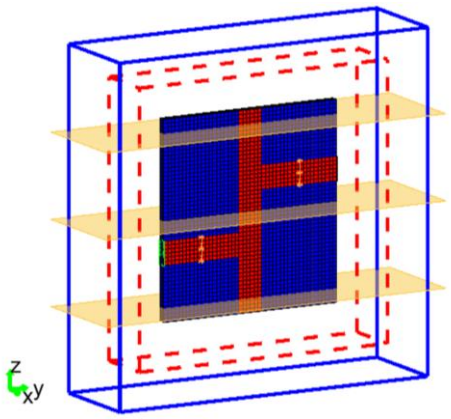

Fig. 1. Distribution of FDTD domain for a microstrip filter problem. The yellow planes represent the boundaries across which the domain is split.

The distribution of these subdomains in MATLAB is performed by first generating a list of z indices that each core must update. The problem domains are split up into subdomains evenly in the z direction. Each core will then calculate the E- and H-fields for one of these subdomains. These arrays of indices are then prepended or appended to provide memory for the transfer regions which will be discussed in a later section. Using these indices, the field components Ex, Ey, Ez, Hx, Hy, Hz and all of their updating coefficient arrays are distributed across multiple cores within our SPMD environment.

To ensure that the smallest amount of data is transferred across each of these boundaries, the original simulation space of this problem from [1] is rotated such that the largest dimension of the domain is along the z-axis. This rotation ensures that the boundaries along which the domain is split are the smallest possible. This is important because data must be communicated across these boundaries.

B. Distribution of sources, samples, and lumped elements

With the domain split across cores, the sources, sampling locations, and lumped elements in the simulation must also be distributed.

The lumped elements are simply contained as coefficients in the FDTD as formulated in [1]. This means that they are precalculated before the simulation time marching begins and are distributed with the rest of the domain. Because of this no further work is required to distribute them past the distribution of the domain coefficients as described in the previous section.

The sources and sampling locations are not precalculated as part of the updating coefficients. The sources are added as currents described in our updating equations (e.g., $J_{ix}^{n+\frac{1}{2}}$ for E_x^n). The sampled values are calculated from averages of our field components at given indices within the grid. Two steps must be taken to distribute the sources and samples. These steps are first to flag the core or cores on which the source or sample lies. This is done by comparing the z location of the source or sample to the z_{idx} values on each of the cores. If at least part of the source or sample resides on that core a flag is set. Each core which contains part of this source or sample then has the currents that it must update transferred to it. Once we have the required updating data on each of the cores containing parts of the sources or samples, each of the indices of these sources and samples are mapped to the local indices of the local matrices holding the field values on each core. This mapping allows these sources and samples to be updated in the same manner they typically are on a single core system.

Once the end of the time marching loop is reached, one final step must be taken to gather the sampled domain parameters back to the host CPU from which post-processing steps can be performed in the same way as a non-multicore implementation. For each sample, this is performed by looping through each core, checking if they have the flag set of containing that specific sample, and if they have copied the sampled data back to a single core. Once the data has been gathered from each core that contained part of the sample, the total sample can be reconstructed. After gathering and reconstruction, these samples can be postprocessed in the same way as data that was created from a single core implementation.

C. Transfer of domain boundaries

As previously mentioned, the domain is split evenly among cores along the z-axis. For continuity of the domain, field values must be transferred across these boundaries. For this implementation an approach like that implemented in [11] was taken. In this communication scheme a single xy-plane slice of the H-fields would be transferred up to the core containing the next section of the domain, and a single xy-plane of the E-fields would be transferred down providing full continuity across the boundary.

Because our domain is only split along the z-axis, the transfers are only required for our Ex, Ey, Hx, and Hy components. A figure of this transfer can be seen in Fig. 2. This highlights the H fields that must be transferred up and the E fields that must be transferred down. Taking a look specifically at one of the field updates we can see why this transfer is needed. In order to update the field Ex(i,j,k+1) we must first calculate the finite difference approximation from the fields Hz(i,j,k+1), Hz(i,j-1,k+1), Hy(i,j,k+1), and Hy(i,j,k). Looking at the figure, it can be seen that all of the required fields are available on core 2 except Hy(i,j,k) which must be transferred from core 1. This same concept applies when updating our Ey, Hx, and Hy fields.

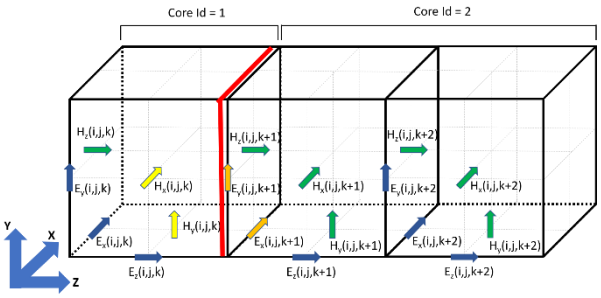

Fig. 2. Figure showing the transfer of E- and H-fields across a Z-axis split boundary. The red line denotes where the domain is split. H-field components that need to be transferred are in yellow and E-fields requiring transfer are in orange.

Each of these transfers in MATLAB can be performed using the labSend() and labReceive() commands. The syntax of these commands is very similar to mpiSend() and mpiRecv() commands when using MPI. Each of these transferred xy planes are held in memory buffers appended to the beginning and end of the e-field and h-field data on each core. It is also important to note that the field arrays were distributed such that the H-fields transferred up and E-fields transferred down were stored on the receiving core at the beginning and end of the field arrays respectively. This means that the original single core updating equations could be utilized with

only minor edits to work with the new multi-CPU code.

IV. EXTENSION TO MULTI-GPU

Up to this point, we have discussed running FDTD simulation on multiple CPUs. Luckily MATLAB provides a simple interface to extend this multi-CPU code to multiple GPUs.

A. Utilizing multiple GPUs in MATLAB

Running code on a single GPU in MATLAB simply requires declaring a variable (scalar value or matrix) within the *gpuArray()* command. Once that has been completed, all arithmetic operations and even most built-in MATLAB commands performed on the data will be done on the GPU. For example declaring two variables *a=gpuArray(magic(100));* and *b=gpuArray(magic(100));* and performing elementwise multiplication *c=a.*b* will accelerate the elementwise multiplication using the GPU. This same process can then be extended to multiple GPUs by declaring *gpuArray()* variables within an SPMD loop. When a parallel pool is created with *parpool()* MATLAB automatically maps each GPU device to a worker in the parallel pool (if there is enough hardware available). From here all GPU commands run within a SPMD loop from a parallel pool with labindex *n* will be run on the corresponding GPU *n*.

B. Extending FDTD simulation to multiple GPUs

Because the multi-CPU code was built to run on any number of cores *n*, the extension to using any given number of GPUs was simple. First a parallel pool is created with the number of workers equal to the number of GPUs we want to run on. If we were to say run on 4 GPUs, this would be *parpool(4)*. Once the parallel pool has started, each of the 4 parallel workers will automatically be mapped to a unique GPU device. This unique mapping gives us direct control over each of the GPUs. A figure to describe the mapping from the FDTD grid to the GPUs and then to the threads can be seen in Fig. 3.

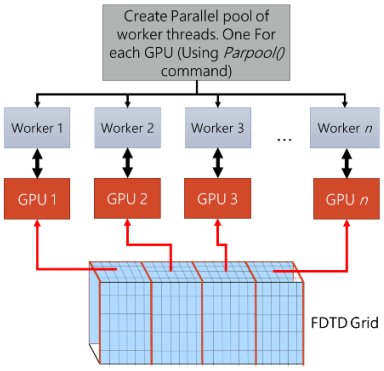

Fig. 3. Figure showing the mapping of our FDTD grid to *n* GPUs, and the mapping of those *n* GPUs to their corresponding parallel pool workers

With each of our workers mapped to a GPU, each of the field components and their updating coefficients, any arrays for updating sources, and any sampling arrays that typically reside on a CPU core are transferred to the GPU using a command such as *Hx=gpuArray(Hx);* because the typical CPU arrays are simply overwritten by our GPU arrays, no change in the code for updating or sampling is required. Once the simulation has finished, the code to gather the data also can stay the same. This is a result of the fact that MATLABs *gather()* command is overloaded to work both with distributed data on the CPU and on the GPU.

V. SIMULATION RESULTS

With the multicore code completed, correct operation was ensured by directly comparing voltage, current, and S-Parameter results of a microstrip filter problem seen in Fig. 1 (as first described on pages 171-177 in [1]) from the multi-CPU and multi-GPU codes to a verified single CPU code. This problem proved the simulation capability and correct operation of the new multi-CPU code. The comparison of the multi-GPU/CPU results can be seen in Fig. 4.

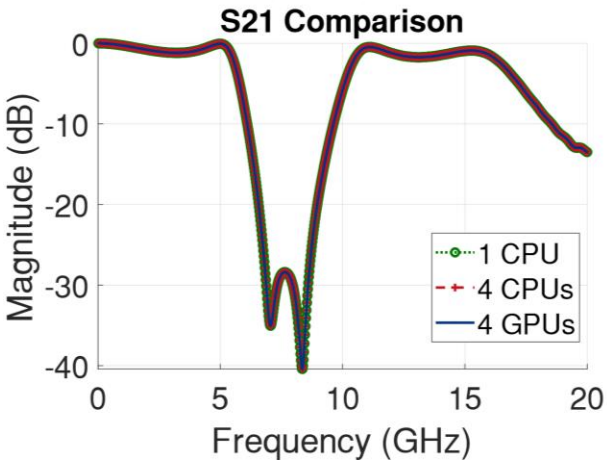

Fig. 4. Comparison of filter results obtained using a single CPU, a multi-CPU, and a multi-GPU FDTD implementation. This simulation was performed with about 1.4 million cells and 3000 time steps.

For the throughput testing of the codes, the air gap between the filter and the CPML region was incrementally increased evenly in all dimensions. This was done to prevent from impartially favoring grids with very large z dimensions, but with very small x and y dimensions. Increasing just the z dimension would provide the optimal throughput because it would provide the highest possible compute to data transfer ratio. Simulations with extremely large z dimensions are very uncommon and unrealistic for most simulation scenarios so they were avoided in the results shown here.

A. Multi-CPU results

The throughput of the multi-CPU code was timed on various computers with a varying number of cores. Tests were also performed to look at increases in throughput when moving from a single physical processor to 2 processors on the same motherboard, and when moving from physical cores to logical (hyperthreaded) cores. This timing was also performed using the non-multicore code and compared. The results from each of these computers can be seen in Figs. 5 through 7.

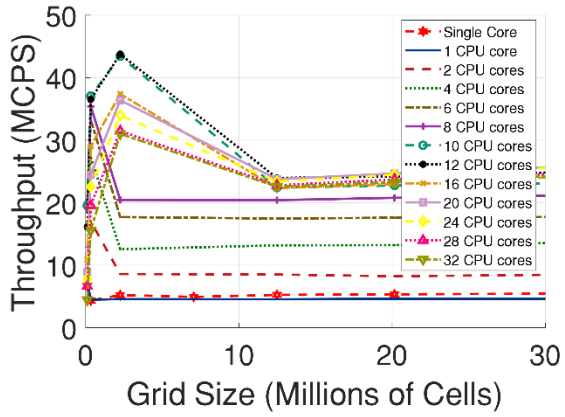

Fig. 5. Throughput in million cells per second (MCPS) vs grid size with various core counts when running an FDTD filter simulation. These simulations were performed on two Intel Xeon E5-2680 CPUs with 20MB of cache and 256GB of RAM. Each processor has 8 physical cores for a total of 16 physical cores and 32 virtual cores.

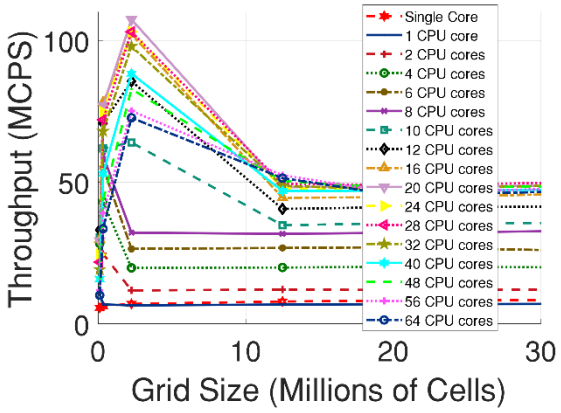

Fig. 6. Throughput in million cells per second (MCPS) vs grid size with various core counts when running an FDTD filter simulation. These simulations were performed on an Intel AMD Ryzen 2990WX and 128GB of RAM. This system has 32 physical cores on a single processor with hyperthreading for a total of 64 logical cores.

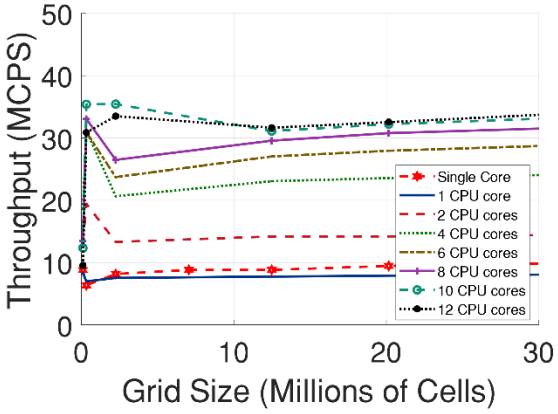

Fig. 7. Throughput in million cells per second (MCPS) vs grid size with various core counts when running an FDTD filter simulation. These simulations were performed on an Intel i7-5280 with 15MB of cache and 32GB of RAM. This system has 6 physical cores on a single processor with hyperthreading for a total of 12 logical cores.

When comparing the throughput of a single core without using SPMD (i.e., the original non-multicore code), it can be noticed that on every machine the non-multicore code outperformed the multicore code when using a single core. Again, this is due to the implicit parallelization MATLAB performs. Once the multicore code begins utilizing more than a single core, the benefits of explicit parallelization in MATLAB quickly surpass the throughput of implicit parallelization. The results of the speedup can be seen in Table 1 when the domain size is 20 million cells.

It should also be noticed that on each system there exists a region of higher throughputs when the grid size is sufficiently small. The size of the grid that this bump exists at is relative to the size of the system cache. This is shown clearly on the systems with multiple processors. The length of the bump can clearly be seen to extend when moving from utilizing a single processor to multiple processors (e.g., in Fig. 6 when moving from 8 cores to greater than 8 cores).

The final important trend to be observed is how the throughput is affected when a system moves from using only physical cores to using both physical and logical cores. Once the requested core count extends past the number of physical cores the system begins using logical hyperthreaded cores. Because this requires a lot of resource sharing within a core, in most cases we see little or no speedup by adding logical cores. The only case where this does not hold true is for the system used in Fig. 4 where substantial gains can be seen by utilizing both physical and logical cores.

Table 1: Speedups of a multicore implementation for various processors with SPMD over a standard single core MATLAB implementation

No. of Cores (Physical / Logical)	i7-5280 (6/12)	Xeon E5-2680 (16/32)	Ryzen 2990WX (32/64)
1	0.85	0.86	0.83
2	1.52	1.59	1.44
4	2.54	2.54	2.38
6	3.03	3.32	3.11
8	3.33	3.96	3.91
12	3.57	4.66	4.94
16	N/A	4.51	5.46
32	N/A	4.54	5.83
48	N/A	N/A	5.79
64	N/A	N/A	5.57

B. Multi-GPU results

The same code was then run on two computers with two different GPUs configurations. The first run was on a system with 2 discrete RTX 2080 GPUs and the second was on a system with 4 Titan-Z GPUs with each PCIe slot containing 2 GPUs. These results can be seen in Fig. 8 and Fig. 9.

As expected, GPU results are an order of magnitude faster than their CPU counterpart. The topic of single GPU acceleration of FDTD with MATLAB was described in depth in [7]. It can be seen from the current multi-GPU results that for low cell counts, the addition of multiple GPUs provides little to no speedup regardless of the system or how many GPUs are added. It is worth noting though that while it does not provide an increased throughput, it also does not seem to slow the simulation down. This similar throughput is caused by a low computation to data transfer time. It is worth noting also that compared to the CPU implementation, the data transfer between GPUs is very slow in MATLAB because data transfer between GPUs is done through the host. As we move to larger grid sizes, we can see that the larger number of GPUs provides a great increase in throughput over lower numbers of GPUs. Because of this, using multiple GPUs in almost all scenarios would be beneficial.

Each GPU throughput typically dropped a relatively drastic amount as grid sizes were increased. The exact cause of this while using MATLAB is unknown but it can be seen that this effect can be moved to much larger grid sizes by increasing the number of GPUs. This behavior was not observed while performing FDTD simulations using C/CUDA development [2]. When the FDTD formulation in [1] is programmed using C/CUDA for a single GPU, the performance on the RTX2080 GPU is on the order of 4 billion cells per second as demonstrated using the CEMS package [12].

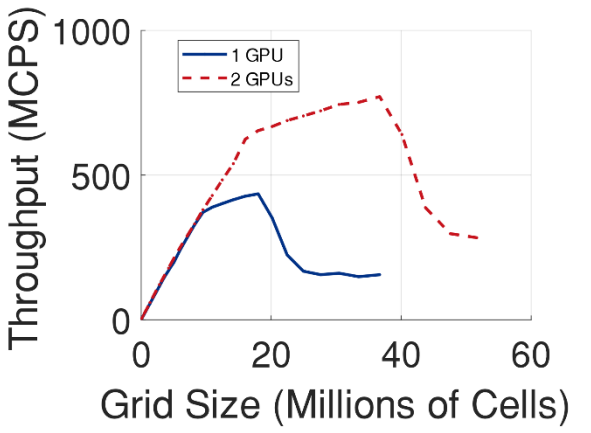

Fig. 8. Throughput in million cells per second (MCPS) vs. grid size for a FDTD filter problem on 2 RTX-2080 GPUs. Each of the 2 GPUs has 8GB of memory.

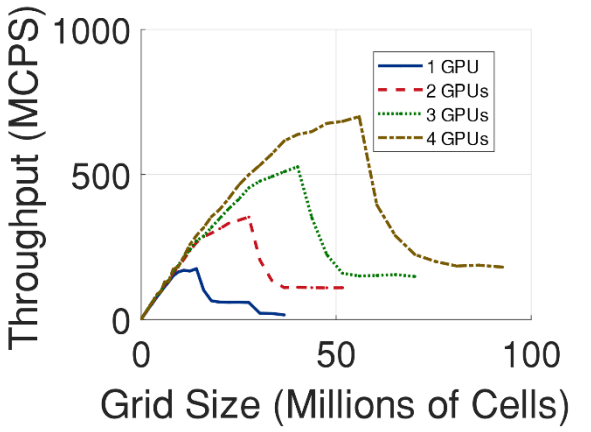

Fig. 9. Throughput in million cells per second (MCPS) vs. grid size for a FDTD filter problem on 4 Titan-Z GPUs. Each of the 4 GPUs has 6GB of memory.

VI. CONCLUSION

This paper described some of the methods used for developing a multi-CPU and multi-GPU FDTD simulation code using explicit MATLAB parallelization. After running on multiple computers, it was shown that utilizing explicit parallelization on CPUs always provided speed increases in the FDTD code over using MATLAB's built-in implicit parallelization. In some cases, speedups of 5x were attained by utilizing explicit parallelization. It was also shown that while parallelizing to physical cores on a system provided speed increases, extending this into hyperthreaded cores provided little to no benefit and in some cases can even decrease throughput. With the multi-GPU implementation we saw similar or increased performance of that on a single GPU.

It is expected that the implementation of a similar solver in a lower level language such as C or FORTRAN would drastically increase the throughput and efficiency

of moving to multiple devices. The use of exclusively MATLAB here though shows that relatively easy edits made to an FDTD solver in a higher level language can provide reasonable speedups over a single GPU or CPU implementation.

ACKNOWLEDGEMENT

This work was partially supported by a gift from Futurewei Technologies, Inc. to ARC Research Group at Colorado School of Mines.

REFERENCES

[1] A. Z. Elsherbeni and V. Demir, *The Finite-Difference Time-Domain Method for Electromagnetics with MATLAB® Simulations*. 2nd edition, Edison, NJ: Scitech Publishing, 2015.

[2] M. J. Inman, A. Z. Elsherbeni, J. G. Maloney, and B. N. Baker, "GPU based FDTD solver with CPML boundaries," in *2007 IEEE Antennas and Propagation Society International Symposium*, pp. 5255-5258, 2007.

[3] K. Hayakawa and R. Yamano, "Multi-core FPGA execution for electromagnetic simulation by FDTD," in *2015 2nd International Conference on Information Science and Control Engineering*, pp. 829-833, 2015.

[4] X.-M. Guo, Q.-X. Guo, W. Zhao, and W. Yu, "Parallel FDTD simulation using NUMA acceleration technique," *Progress In Electromagnetics Research Letters*, vol. 28, pp. 1-8, 2012.

[5] T. Nagaoka and S. Watanabe, "Multi-GPU accelerated three-dimensional FDTD method for electromagnetic simulation," in *2011 Annual International Conference of the IEEE Engineering in Medicine and Biology Society*, pp. 401-404, 2011.

[6] M. R. Zunoubi, J. Payne, and M. Knight, "FDTD multi-GPU implementation of Maxwell's equations in dispersive media," presented at the *Optical Interactions with Tissue and Cells XXII*, vol. 7897, p. 78971S, 2011.

[7] J. E. Diener and A. Z. Elsherbeni, "FDTD acceleration using MATLAB parallel computing toolbox and GPU," vol. 32, p. 6, 2017.

[8] W. Shao and W. McCollough, "Multiple-GPU-based frequency-dependent finite-difference time domain formulation using MATLAB parallel computing toolbox," *Progress In Electromagnetics Research M*, vol. 60, pp. 93-100, 2017.

[9] "MATLAB multicore," [Online]. Available: https://www.mathworks.com/discovery/matlab-multicore.html. [Accessed: 10-Jan-2019].

[10] "Parallel MATLAB: Multiple processors and multiple cores," [Online]. Available: https://www.mathworks.com/company/newsletters/articles/parallel-matlab-multiple-processors-and-multiple-cores.html. [Accessed: 10-Jan-2019].

[11] P. F. Baumeister, T. Hater, J. Kraus, D. Pleiter, and P. Wahl, "A performance model for GPU-accelerated FDTD applications," in *2015 IEEE 22nd International Conference on High Performance Computing (HiPC)*, pp. 185-193, 2015.

[12] V. Demir and A. Elsherbeni, *Computational Electromagnetics Simulator (CEMS) Software Package*. 2010.