http://dx.doi.org/10.5755/j01.eie.24.4.21475 ELEKTRONIKA IR ELEKTROTECHNIKA, ISSN 1392-1215, VOL. 24, NO. 4, 2018

Optimizing FDTD Memory Bandwidth by Using Block Float-Point Arithmetic

Stefan Pijetlovic[1,2], Milos Subotic[1,2], Nebojsa Pjevalica[1]
[1]*Computing and Control Engineering Department, Faculty of Technical Sciences, University of Novi Sad, Trg Dositeja Obradovica 6, 21000 Novi Sad, Serbia*
[2]*RT-RK Institute for Computer Based Systems, Narodnog fronta 23a, 21000 Novi Sad, Serbia*
stefan.pijetlovic@rt-rk.com

***Abstract*—Finite-difference time-domain is a numerical method used for modelling of computational electrodynamics. The method is resource intensive, especially regarding memory usage. Multiple memory accesses are required per single computation so memory bandwidth acts as a bottleneck limiting the overall performance. Existing solutions use either fixed-point or floating-point arithmetic, depending on the complexity of the target platform, to model the data. Floating-point requires less memory access but the computation is more intensive due to the normalisation. Fixed-point is the opposite – simple computation but with more memory access for the same precision. The novelty of this paper is in the block floating-point realization which is the middle ground between the two. The approach is less compute intensive than the floating-point solutions while using less memory than the fixed-point realization. This makes the solution an alternative for bit-exact platforms, such as field-programmable gate arrays. The results are compared to both floating-point and fixed-point implementations and the memory bandwidth and other resources needed for targeted platform are calculated.**

***Index Terms*—FDTD; FPGA; Block-floating point; Memory management.**

I. INTRODUCTION

Finite-difference time-domain (FDTD) is a powerful algorithm for the modelling of the electromagnetic field. It provides a direct time-domain solution of Maxwell's curl equations in differential form by discretizing both the physical region and time interval by using a uniform grid called the Yee grid, after its inventor [1]. The steps in the algorithm are the following: first all fields are initialized to 0. Afterwards, the sources of electromagnetic fields are introduced to the system. In the main loop of the algorithm the electric field is calculated based on the magnetic field. When the whole grid is updated, the magnetic field is calculated based on the electric field which has changed previously. Any objects inside the grid reflect or absorb the waves in some quantity. This process goes through time until we reach the desired iteration number. A special absorbing layer is added on the edge of the grid. This layer is called the absorbing boundary conditions (ABC) and is used to truncate the outgoing waves which would otherwise bounce off the edges of the grid, to obtain more realistic results [2], [3]. Some benefits of the FDTD algorithm are that it is good for large scale simulations because it scales almost linearly, works well for broadband and transient simulations and naturally handles nonlinear behaviour. Some drawbacks of the method are the not so efficient representation of curved surfaces due to grid representation and inefficient representation of highly resonant devices which require much longer simulations [4]. The areas of application of the FDTD include antenna design [5]–[7], optoelectronics [8] mine detection [9], microwave tomography [10], electromagnetic compatibility [11] and more.

Even though the method was introduced in the year 1966 [1], it was not widely used until the last decade due to limited computing resources which could not provide enough computation power to solve all the equations in time [12]. As a memory intensive problem, it has sparked an interest to many authors in the past in the field of optimization. To improve the performance, the algorithm has been successfully implemented on multi-core architectures [13], with additional improvements being made by using graphics processing units (GPUs) as accelerators [14], [15] and field-programmable gate arrays (FPGAs) [16], [17]. Most recently, the trends among authors is using Open Computing Language (OpenCL) in order to achieve a more flexible and portable solution based on the FPGA [18], [19].

One of the issues some of these solutions face is that they are based on the PC architecture which is compute oriented - it can do multiple computing operations in the time frame of one memory access, resulting in a lot of idle time for the computing resources when memory intensive problems are involved. The FDTD algorithm's memory access to number of operations ratio makes PC architectures suboptimal. To cope with these issues, the authors proposed a solution based on multiple FPGAs working in conjunction to achieve maximal possible bandwidth [20]. The main feature of the solution was its lower cost over-time due to low power consumption and cheaper components when compared to existing solutions. However, the performance of existing

Manuscript received 13 December, 2017; accepted 30 May, 2018.
This work was partially supported by the Ministry of Education, Science and Technological Development of the Republic of Serbia, under grant number: TR32029.

solutions could not be matched due to the far superior bandwidth of the double data rate type five synchronous graphics random-access memory (GDDR5 SDRAM) used in GPUs when compared to the double data rate type three random access memory (DDR3 SDRAM) modules the authors did the analysis for. This is mostly due to the fact that the GPUs have become a commodity because of their widespread usage in PCs, providing a better price to performance ratio. The idea that followed was to artificially improve the memory bandwidth without using more powerful FPGAs which would increase the cost and go against the main philosophy of the solution, which is a cheaper and more efficient solution. Therefore, the approach the authors took was to optimize the memory usage which came down to data compression. The goal was to use fewer bits to represent the data while maintaining precision within a specified threshold. This is the area where block-floating point arithmetic comes into play.

Block-floating point (BFP) arithmetic is a way of emulating floating-point arithmetic by using fixed-point. Unlike the standard floating-point, where every element has its own exponent and fraction, in BFP an entire block of numbers has a mutual exponent. The elements within the block are represented by their respective mantissas which are scaled according to the mutual exponent which is calculated based on the highest value.

When measuring the performance of a particular number representation, several characteristics are important. One of them is the dynamic range which is the ratio between the smallest and largest number that can be represented. The second characteristic is precision which defines the resolution or the measure of detail. When numbers are concerned, precision defines how close two consequent numbers are - the smaller this gap is, the better the precision. For the same word length, floating-point numbers have a better dynamic range whereas fixed-point representation has greater precision. The third characteristic worth mentioning is complexity. Fixed-point arithmetic is much simpler when compared to floating-point because it does not involve normalization. Adding two fixed-point numbers is no different from adding two integers. Floating-point arithmetic on the other hand requires shifting of the mantissa based on the exponent before adding two values. This results in floating-point arithmetic being more complex, requiring more transistors to do the computations which further increases the cost. So choosing the best representation for the data is an optimisation problem on its own - how to find the optimal ratio of dynamic range and precision while keeping the costs at the lowest value.

The benefit of using BFP is that it allows more aggressive scaling with a single block exponent while at the same time retaining a greater dynamic range in the output compared to typical fixed-point. Also, when compared to standard single precision floating-point, it requires fewer normalization operations because a single exponent is mutual for more elements. The important factor here is the size of the problem and the size of each block. The bigger the block and the problem, the more memory bandwidth is “saved” but with a certain lack in precision.

BFP provides a good trade off between complexity and dynamic range, making it an efficient number representation format in some cases [21]. It is most commonly used in algorithms which are suited for fixed-point processors such as DSPs but require or benefit from additional dynamic range in certain areas. One such application is the Fast Fourier Transform (FFT) and other algorithms based on it [22]–[24]. Other areas include neural networks [25], matrix multiplication [26], digital filters [27], [28], communication systems [29] and more.

II. SOLUTION

In order to use the BFP several adjustments to the existing algorithm had to be made. The first step was the profiling of the algorithm where all the variables and matrices were monitored to pinpoint the most extreme values. Afterwards scaling factors have been introduced so that all the values would fit into the range [-1, 1]. This was required in order to avoid any overflows when switching from single precision 32bit floating-point to fixed-point. The results showed that by making this change, no significant errors were introduced. Despite the satisfying results, the overall performance increase was too small to make a change so the next step was to introduce the data compression.

Experiments showed that the minimal number of bits which could be used to represent the data elements was 16. However, there were two main issues with this type of solution. The first is the significant loss of precision due to insufficient dynamic range. The second problem was that even though the results could fit into 16 bits, the intermediate values used during the computation could not which caused overflows. A more extreme scaling factor was introduced but it deteriorated the precision even further. In the end a 24bit fixed point solution was chosen as a starting point. This reduced memory usage by 25 % with a decrease in precision of around 5 % which was acceptable.

The next step was to see if a custom data representation could yield better results and so block floating-point was introduced. Due to the nature of travelling electromagnetic waves (especially the large spatial sampling rate), the Yee grid could be divided into smaller blocks called tiles with similar values within the tile as shown in Fig. 1. This made it possible to determine the mutual exponent for the tile which then allowed a more aggressive scaling.

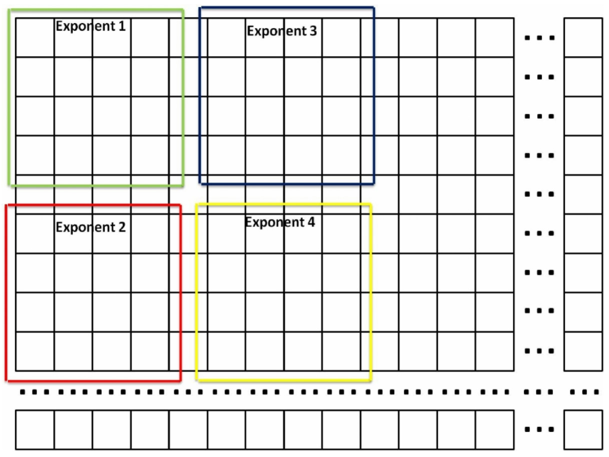

Fig. 1. Block floating-point layout.

The next task was to figure out how big should the tiles be without losing on precision by scaling them all in regard to the element with the most extreme value. If the division of the grid is too fine (having many tiles), the number of operations needed to find an exponent for each of the tiles will make the algorithm last longer and use up more memory. The most extreme case is when the tiles have the dimension of 1 × 1 which creates the equivalent of a grid with all floating-points. On the other hand, if the tiles are too large, some elements will not be represented precisely and the loss of precision will accumulate over time giving worse results as the simulation runs further. The optimal solution lies somewhere in between the extremes.

The most challenging aspect of switching to BFP was tile management. A new square tile is formed from elements from two matrices – one that holds the 16 bit fixed-point values which represent the mantissas and the other smaller matrix that contains the exponents as 8bit integers for each tile. If we examine the case where the tile size is 4 × 4, the fixed-point matrix has the dimension of 64 × 64, whereas the exponent matrix has 16 × 16 cells. First, the 4 × 4 tile of the fixed-point matrix is read and all values are expanded to 24 bit fixed-point and scaled by the exponent which corresponds to that tile. This process was named unpacking. Once the tile is unpacked, it is used in all the calculations and at the end a reverse process happens called packing. During this process a new exponent is calculated based on the elements in the tile and the values are written down in the final form which again is 16 bit. A logarithm with the base 2 is applied to the maximal value within the tile and then a ceiling function determines the value of the exponent. Afterwards, this value is stored in the matrix which holds the exponents. The fixed-point matrix is then scaled with the newly calculated exponent and written to the fixed-point matrix. A simplified algorithm is given bellow.

```
...
for tj in 1:num_of_tiles_y_axis
   for ti in 1: num_of_tiles_x_axis
      # unpacking
      for jj in 1:tile_size
         for ii in 1:tile_size
            j = ((tj - 1) << tile_shift_x) + jj
            i = ((ti - 1) << tile_shift _y) + ii
            X_tile[ii,jj] = from_block(X_bf[i,j], block_exp)
         end
      end

      do_the_calculations()     #calculations

      #packing
      for jj in 1: tile_size
         for ii in 1: tile_size
            j = ((tj - 1) << tile_shift _x) + jj
            i = ((ti - 1) << tile_shift _y) + ii
            X_bf[i,j] = to_block(X_tile[ii,jj], block_exp)
         end
      end
   end
end
...
```

The ti, tj, ii, and jj are counters used to manoeuvre around the grid. The values in the outer for loops (ti and tj) move through all the tiles, whereas the values in the inner loops (jj and ii) move within a single tile. The i and j variables are used to index the entire fixed-point grid. The X_tile is the temporary matrix used for calculations, and the X_bf is the matrix which is correspondent to the tile which is currently being worked on. Block_exp is the exponent which corresponds to the current tile. Some equations used in the algorithm require values which are located in the tiles not updated yet. To cope with this issue, the algorithm was divided into two phases and special data structures had to be introduced. During the first phase all tiles which would be needed for the second phase are calculated and written back. Special boundary cases which occur on the edge of each tile are handled by the so called bands which represent only a one-dimensional array that has the same length as the tile side as shown in Fig. 2.

These additional structures use up extra bandwidth and produce some overhead and so does the additional phase. However, they are necessary in order to reduce the internal memory needed for the proper functioning of the algorithm. Both are included in the final performance overview. For the tiles on the border of the grid there is another special boundary case. Because their neighbouring tiles are in fact outside of the grid, this is modelled by assuming that they have all zeros, and these zeros are used when updating the values of these elements. Once the entire grid has been updated, the last step is to measure the values in certain parts of the grid to be used for visualisation.

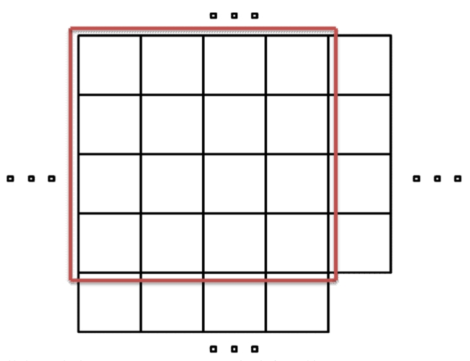
Fig. 2. Additional data structures needed for tile management.

The principle of locality also comes into play. The next tile to be processed is always close in the memory so spatial locality is satisfied. This kind of predictable behaviour enables performance optimizations for techniques such as pre-fetching, pipelining and burst memory reading. Temporal locality condition is not achieved because the data will not be reused in a time frame short enough so the applicability of caching is very limited and was thus discarded.

III. RESULTS

The Yee grid was modelled by a 64 × 64 matrix with the simulation running for 150 iterations. To find the optimal solution, the tile size varied from 2 × 2 to 32 × 32. The exponent was always an 8bit integer while the number of bits used to represent the fraction ranged from 10 to 14, with the remaining 6 to 2 bits being used for the integer part of the number. In the end, 14 bits were used to represent the fraction. So a total of 24 bits were used represent the data to have the best comparison with the 24 bit purely fixed-point solution. In order to measure the memory bandwidth, the number of bits per matrix was used as defined in (1)

$$Sum = Size \times 16 + Num \times 8. \qquad (1)$$

where *Sum* is the total number of bits, *Size* is the size of the tile (for example 4 × 4) and it is multiplied by 16 because those are fixed-point values. *Num* is the number of tiles which are needed to cover the entire Yee grid and it is multiplied by 8 because each exponent is represented by a single 8 bit integer. The differences between the sizes of the matrices are indicative of the difference between the actual bandwidth needed.

The other metric used to measure the performance along with the memory usage was precision. It was calculated by using the mean absolute relative percent difference of the Ez field (the value of the electric field oriented towards the z axis). First the error itself is defined as follows

$$D(x,y) = \begin{cases} 0, & x = 0 \wedge y = 0, \\ \dfrac{2(x-y)}{|x|+|y|}, & Otherwise. \end{cases} \qquad (2)$$

The value is 0 if both x and y are 0 to avoid division by 0, otherwise it is defined as shown above. The precision is calculated by going through an array of matrices each containing the value of the Ez field for every iteration. In the end, this sum is divided by the number of iterations (iters in the formula) and the size of the matrices (Nx and Ny being the dimensions) to form the median absolute value of the error which was explained in (2). L1 and L2 are the values from the initial floating-point and final block floating-point version of the solution

$$Err = \frac{\sum_{i=1}^{i=iters} \sum_{x=1}^{x=Nx} \sum_{y=1}^{y=Ny} |D(L1[i][x][y], L2[i][x][y]|}{iters \times Nx \times Ny}. \qquad (3)$$

This type of relative error (3) takes into account how big the difference in the values measured is when compared to the absolute value. When multiplied by 100, it becomes a relative percentage error. For instance, if the difference between the observed values is 0.1 where one value is 0.5 and the other 0.6, that is considered as a bigger error than if the difference in the value is 10 and the observed values are 550 and 560. The reasoning behind this choice it that the authors believe it provides a better perspective of the performance of the solution. The performance expressed in precision and memory bandwidth for several tile sizes is given in Table I.

The initial 32bit floating-point solution is considered to be the benchmark. Its relative error is 0 because it does not differ from itself and it requires the biggest amount of memory regardless of the size of the grid and number of iterations, so for illustrating purposes it needs 100 % of the memory bandwidth. The 1st improved solution which used 24 bit fixed-point arithmetic uses up 25 % less bandwidth because each element in the Yee grid is represented by 24 bits instead of 32. In other words 75 % of the benchmark bandwidth is required for the solution to work. Using the equation for bandwidth (1) we have calculated which amount of the initial bandwidth used for the benchmark is needed for the block floating-point solutions. The table shows only around 50 % of the memory is needed when compared to the initial solution. This is almost the same result as when using 16 bit fixed-point solution but with an error 2-6 times smaller, depending on the size of the tile. This shows that the BFP can be used in situations where memory usage needs to be as small as possible but without sacrificing the precision too much.

TABLE I. TILE SIZE, NUMBER OF TILES IN THE GRIDAND PERFORMANCE.

Tile size	**Number of tiles**	**Relative error [%]**	**Memory bandwidth [%]**
2 × 2	1024	3.59	56.25
4 × 4	256	3.79	51.56
8 × 8	64	5.84	50.39
16 × 16	16	10.10	50.10
32 × 32	4	12.92	50.02

These calculations show a promising result but this kind of solution is not the best option for the standard PC architecture. This is due to the granularity of the data CPUs and GPUs use which is usually in chunks of 8, 16, 32 or larger. So there is no way of really benefiting from saving a few bits for every data element, unlike when working with an FPGA.

Another interesting characteristic of the algorithm is the ratio of reads and writes and also the ratio of total memory accesses when compared to the number of operations (additions and multiplications). When using the 8 × 8 tile, the number of writes per iteration is 7224, while the number of reads is 20152, so almost 3 times as more. This difference is a good indicator of the large amount of data needed for computation. An even better perspective is given once all the operations (additions and multiplications) are included. Together, the additions and multiplications amount to the ratio of 0.58 operations per single byte of data, for the 8 × 8 tile size. If we approximate the GPUs FLOPS (floating-point operations per second) with the bandwidth we see just how little compute potential of the GPU is used. Here are the specs of the now average, graphics card such as an AMD Radeon R7 265. According to the manufacturer, this GPU achieves 1843.2 GFLOPS with the maximal theoretical bandwidth of 179.2 GB/s. If we divide the two we get the ratio of 10.28 FLOPS per byte which is 17 times more than that the algorithm needs. If we move further up the market price and see the specs of a high end GPU such as the GeForce GTX 1070, we see that the result is even worse in terms of effective computation power. The GTX 1070 achieves 7816.4 GFLOPS with the theoretical bandwidth reaching 256 GB/s, making the ratio 30.52 FLOPS per byte which is over 50 times of that which is needed for the algorithm. The operation per byte ratio of the algorithm is just far below of that the GPU can achieve which makes them not the optimal choice when it comes to power efficiency (a big part of the logic remains idle). This is a confirmation that the FDTD is a memory intensive problem.

The main advantage of the FPGA in this case is that it is very well suited for bit-exact operation, having no bits being unused. Custom buses can be configured to have just the right width. To utilize spatial locality, data can be read from the memory in bursts, saving precious time. Instead of reading the data elements 1 by 1, the entire of 64 elements is read during a single access.

For a real world use case, an FPGA from the previous research [19] which was deemed optimal for memory usage can be used. It is the XC7A15T-1FGG484C from Xilinx, Artix 7 family. When calculations are compared with the specs from the manufacturer's data sheet, we can see that even this low end FPGA is suited for the job. According to our assumptions, all the bandwidth of the FPGA could be effectively used. As far as computation power is concerned, less than 70 % of the DSP slices are required. Even more importantly, the DSP frequency needed to perform all the operations in 20 milliseconds is around 20 MHz–25 MHz which is a lot less when compared to the working frequencies of modern GPUs. This way the power consumption is greatly reduced which is one of the virtues of an FPGA solution. The BRAM usage depends mostly on the tile size and is around 26 % when using 8 × 8 tiles. Smaller values are not a problem because they only require slightly more external memory without increasing the number of operations because they are dependant on the size of the Yee grid. But even the next in line which is the 16 × 16 tile would fill up the entire internal memory. In conclusion, an FPGA solution should in theory be a lower costing viable alternative if the goal is efficient memory usage and power consumption.

IV. CONCLUSIONS

The novelty presented in this paper is using the block floating-point arithmetic in an area where it is not usually applied. By using BFP the performance of the FDTD algorithm was improved by better memory utilization (requiring almost 50 % less memory with a relative error of around 5 %–6 %) than those of existing PC based solutions. The authors have tackled the fundamental issue of the problem which is intensive memory usage. The results presented show a viable alternative for bit-exact platforms such as FPGAs and provide a good foundation for future research on the subject.

Future work branches off in two directions – optimizing the algorithm and implementing the solution in some hardware description language. The algorithm could be improved with the optimization techniques as explained in chapter 2 such as pre-fetching, pipelining etc. On the other hand, as a follow up on previous research, a custom board with low-end FPGAs could be made to fully test the limit of the approach in order to draw final conclusions.

REFERENCES

[1] Kane Yee, "Numerical solution of initial boundary value problems involving maxwell's equations in isotropic media", *IEEE Trans. Antennas Propag.*, vol. 14, no. 3, pp. 302–307, 1966. DOI: 10.1109/tap.1966.1138693.

[2] R. Lee, D. Kingsland, J. F. Lee, "A perfectly matched anisotropic absorber for use as an absorbing boundary condition", *IEEE Trans. Antennas and Propagation*, vol. 43, no. 12, pp. 1460–1463, 1995. DOI: 10.1109/8.477075.

[3] S. D. Gedney, "An anisotropie perfectly matched layer-absorbing medium for the truncation of FDTD lattices", *IEEE Trans. Antennas and Propagation*, vol. 44, pp. 1630–1639, 1996. DOI: 10.1109%2F8.546249.

[4] A. Taflove, S. Hagness, *Computational Electrodynamics: The Finite-difference Time-domain Method, ser. Antennas and Propagation Library*. Artech House, 2000.

[5] M. Subotic, L. Palfi, N. U. Pjevalica, "Efficient antenna modeling method for microwave tomography", in *2016 24th Telecommunications Forum Telfor (TELFOR),* IEEE, 2016. DOI: 10.1109/telfor. 2016.7818828.

[6] S. C. Hagness, A. Taflove, J. E. Bridges, "Three-dimensional FDTD analysis of a pulsed microwave confocal system for breast cancer detection: design of an antenna-array element", *IEEE Trans. Antennas and Propagation*, vol. 47, no. 5, pp. 783–791, 1999. DOI: 10.1109/8.774131.

[7] S. M. Aguilar, M. A. Al-Joumayly, M. J. Burfeindt, N. Behdad, S. C. Hagness, "Multiband miniaturized patch antennas for a compact, shielded microwave breast imaging array", *IEEE Trans. Antennas and Propagation*, vol. 62, no. 3, pp. 1221–1231, 2014. DOI: 10.1109/TAP.2013.2295615.

[8] F. Zepparelli, P. Mezzanotte, F. Alimenti, L. Roselli, R. Sorrentino, G. Tartarini, P. Bassi, "Rigorous analysis of 3D optical and optoelectronic devices by the compact-2D-FDTD method", *Optical and quantum electronics*, vol. 31, no. 9, pp. 827–841, 1999. DOI: 10.1023/A:1006904629078.

[9] P. Kosmas, Y. Wang, C. M. Rappaport, "Three-dimensional FDTD model for GPR detection of objects buried in realistic dispersive soil", in *Proc. SPIE 4742, Detection and Remediation Technologies for Mines and Minelike Targets VII*, 2002, pp. 330–338. DOI: 10.1117/12.479104.

[10] S. Noghanian, A. Sabouni, T. Desell, A. Ashtari, *Microwave Tomography: Global Optimization, Parallelization and Performance Evaluation*. Springer, 2014. DOI: 10.1007/978-1-4939-0752-6.

[11] J. Mix, G. Haussmann, M. Piket-May, K. Thomas, "EMC/EMI design and analysis using FDTD", *IEEE EMC Symposium, Int. Symposium on Electromagnetic Compatibility. Symposium Record*, Denver, CO, 1998, pp. 177–181. DOI: 10.1109/ISEMC.1998.750081.

[12] W. Chen, P. Kosmas, M. Leeser, C. Rappaport, "An FPGA implementation of the two-dimensional finite-difference time-domain (FDTD) algorithm", in *Proc. 12th Int. symposium on Field programmable gate arrays (ACM/SIGDA 2004)*, 2004, pp. 213–222, DOI: 10.1145/968280.968311.

[13] J. Frances, S. Bleda, A. Marquez, C. Neipp, S. Gallego, B. Otero, A. Belendez, "Performance analysis of SSE and AVX instructions in multi-core CPUs and GPU computing on FDTD scheme for solid and fluid vibration problems", *The Journal of Supercomputing*, vol. 70, no. 2, pp. 514–526, 2014. DOI: 10.1007/s11227-013-1065-x.

[14] J. Chi, F. Liu, E. Weber, Y. Li, S. Crozier, "GPU-accelerated FDTD modeling of radio-frequency field–tissue interactions in high-field MRI", *IEEE Trans. Biomedical Engineering*, vol. 58, no. 6, pp. 1789–1796, 2011. DOI: 10.1109%2Ftbme.2011.2116020.

[15] E. L. Tan, "Acceleration of lod-fdtd method using fundamental scheme on graphics processor units", *IEEE Microwave and Wireless Components Letters*, vol. 20, no. 12, pp. 648–650, 2010. DOI: 10.1109/lmwc.2010.2079922.

[16] W. Chen, M. Leeser, C. Rappaport, "Acceleration of the 3D FDTD Algorithm in Fixed-point Arithmetic using Reconfigurable Hardware", PhD dissertation, Northeastern University, 2007. [Online]. Available: http://www.ece.neu.edu/groups/rcl/theses/wchen_phd2007.pdf

[17] R. N. Schneider, L. E. Turner, M. M. Okoniewski, "Application of FPGA technology to accelerate the finite-difference time-domain (FDTD) method", in *Proc. 10th Int. symposium on Field-programmable gate arrays (ACM/SIGDA 2002)*, 2002, pp. 97–105. DOI: 10.1145/503048.503063.

[18] H. M. Waidyasooriya, T. Endo, M. Hariyama, Y. Ohtera, "OpenCL-based FPGA accelerator for 3D FDTD with periodic and absorbing boundary conditions", *Int. J. Reconfigurable Comput.*, pp. 1–11, 2017. DOI: 10.1155/2017/6817674.

[19] T. Kenter, J. Forstner, C. Plessl, "Flexible FPGA design for FDTD using OpenCL", in *27th Int. Conf. Field Programmable Logic and*

Applications (FPL 2017), Ghent, Belgium, 2017, pp. 1–7. DOI: 10.23919/fpl.2017.8056844.

[20] S. Pijetlovic, M. Subotic, N. Pjevalica, "An approach to finding an optimal FPGA for memory intensive problems", in *4th Int. Conf. (IcETRAN 2017)*, 2017. [Online]. Available: https://www.etran.rs/common/pages/proceedings/IcETRAN2017/EKI/IcETRAN2017_paper_EKI1_6.pdf

[21] K. Kalliojarvi, J. Astola, "Roundoff errors in block-floating-point systems", *IEEE Trans. Signal Processing*, vol. 44, no. 4, pp. 783–790, 1996. DOI: 10.1109/78.492531

[22] D. Elam, C. Lovescu. A block floating point implementation for an N-point FFT on the TMS320C55X DSP. Texas Instruments Application Report, 2003. [Online]. Available: http://www.ti.com/litv/pdf/spra948

[23] E. Bidet, D. Castelain, C. Joanblanq, P. Senn, "A fast single-chip implementation of 8192 complex point fft", *IEEE Journal of Solid-State Circuits*, vol. 30, no. 3, pp. 300–305, 1995. DOI: 10.1109/4.364445.

[24] H. K. Boyapati, R. K. Elubudi, S. Ungati, S. Y. Chaudhari, M. Jain, "Transmit evm improvement of ofdm based wireless lan through rescaling and block floating point ifft implementation", *Procedia Computer Science*, vol. 115, pp. 635–642, 2017. DOI: 10.1016/j.procs.2017.09.162.

[25] Z. Song, Z. Liu, D. Wang, "Computation error analysis of block floating point arithmetic oriented convolution neural network accelerator design", 2017. [Online]. Available: https://arxiv.org/pdf/1709.07776

[26] E. H. D'Hollander, "High-level synthesis optimization for blocked floating-point matrix multiplication", *ACM SIGARCH Computer Architecture News*, vol. 44, no. 4, pp. 74–79, 2017. DOI: 10.1145/3039902.3039916.

[27] A. Oppenheim, "Realization of digital filters using block-floating-point arithmetic", *IEEE Trans. Audio and Electroacoustics*, vol. 18, no. 2, pp. 130–136, 1970. DOI: 10.1109/tau.1970.1162085.

[28] K. Ralev, P. Bauer, "Realization of block floating-point digital filters and application to block implementations", *IEEE Trans. Signal Processing*, vol. 47, no. 4, pp. 1076–1086, 1999. DOI: 10.1109/78.752605.

[29] Y. F. Choo, B. L. Evans, A. Gatherer, "Complex block floating-point format with box encoding for wordlength reduction in communication systems", 2017. [Online]. Available: https://arxiv.org/pdf/1705.05217