Progress In Electromagnetics Research, Vol. 139, 655–670, 2013

PERFORMANCE OF FDTD METHOD CPU IMPLEMENTATIONS FOR SIMULATION OF ELECTROMAGNETIC PROCESSES

Dmitry L. Markovich[1, *], Konstantin S. Ladutenko[1, 2], and Pavel A. Belov[1]

[1]St. Petersburg National Research University of Information Technologies, Mechanics and Optics, 49 Kronverskii Ave., St. Petersburg 197101, Russian Federation

[2]Ioffe Physical-Technical Institute of the Russian Academy of Sciences, 26 Polytekhnicheskaya Str., St. Petersburg 194021, Russian Federation

Abstract—We analyze the performance of finite-difference time-domain (FDTD) method implementations for 2D and 3D problems. Implementations in Fortran, C and C++ (with Blitz++ library) languages and performance tests on several hardware setups (AMD, Intel i5, Intel Xeon) are considered. The performance of implementations using traditional FDTD algorithm for the largest size of test problem is limited by the bandwidth of computer random-accessed memory (RAM). Our implementations are compared with a commercial simulation software package Lumerical FDTD Solutions and an open source project Meep.

1. INTRODUCTION

Nowadays, numerical modeling is being widely used for both engineering and research activities. The finite-difference time-domain (FDTD) method for electromagnetic phenomena simulations was introduced by Kane Yee in 1966 in the pioneer paper [1]. Various modifications of the method are often used for simulations of acoustics, seismic and earthquake studies [2], or even heat-acoustic phenomena [3] and bio-electromagnetic problems [4].

FDTD method requires the components of magnetic and electric field to be placed in the computational grid with half time step shift and half spatial step shift (for magnetic field components) and

Received 19 March 2013, Accepted 18 April 2013, Scheduled 19 May 2013

* Corresponding author: Dmitry L. Markovich (dmmrkovich@gmail.com).

Maxwell's equations in differential form to be explicitly discretized. This procedure enables to calculate the values of magnetic and electric field components for the "next" time step using the values from the "previous" time step. The simplest 1D case with only two field components $E_z(x,t)$ and $H_y(x,t)$ results in the following update equations [5]:

$$H_y^{n+\frac{1}{2}}\left(i+\frac{1}{2}\right)=H_y^{n-\frac{1}{2}}\left(i+\frac{1}{2}\right)+\frac{\Delta_t}{\mu\Delta_x}\left[E_z^n\left(i+1\right)-E_z^n\left(i\right)\right] \quad (1)$$

$$E_z^{n+1}\left(i\right)=E_z^n\left(i\right)+\frac{\Delta_t}{\varepsilon\Delta_x}\left[H_y^{n+\frac{1}{2}}\left(i+\frac{1}{2}\right)-H_y^{n+\frac{1}{2}}\left(i-\frac{1}{2}\right)\right] \quad (2)$$

Indices i and n enumerate spatial step and time step correspondingly, Δ_x and Δ_t stand for the magnitude of steps, ε and μ — for relative dielectric and magnetic permeability, $E_z^n(i)$ is the magnitude of the electric field z-component at $t = n\Delta_t$ with spatial coordinate $x = i\Delta_x$, magnetic field y-component uses the same notations. The algorithm can be generalized for 2D and 3D cases [5, 6], that are of particular interest for programming.

FDTD method can be also applied to a huge variety of parabolic equations and is often used for numerical solution of wave propagation in anisotropic [7], nonlinear [8], and dispersive [9, 10] media.

A special feature of the method is its high computational load, that distinctly restricts its application area. At the same time, relative simplicity of the method led to the development of a number of commercial (Lumerical FDTD Solutions [11], XFDTD [12], Acceleware FDTD [13]) and open source (Meep [14], EMTL [15], Angora [16]) software packages that perform computations with its help.

Comparing FDTD method implementations is usually obstructed for a number of reasons:

- Data found in literature are usually fragmentary, especially those related to hardware used for testing. Moreover, one can find computer architectures that have already become extinct [17, 18].
- Different FDTD method implementations are considered, that may or may not simulate such material properties as dispersion, nonlinearity, anisotropy, etc., and use various numerical schemes [5, 6, 19–23].
- Free implementations (that can be accessed via the Internet) are usually inapplicable for simulation of complicated tasks and suffer from inconsistent documentation.
- Commercial packages do not provide access to the source code and hide details about the method implementation, that are important for comparison.

The need for detailed and fair comparison led to the development of several FDTD implementations by the authors [24].

2. FORMULATION OF THE PROBLEM

In this paper we compared the performance of implementations simulating a bare-bones test problem: electromagnetic wave propagation in anisotropic media with perfect reflecting walls. This allows to fully evaluate the efficiency of computational core of FDTD method implementation.

To develop such an implementation, one has to choose a programming language and a target hardware setup. We developed several implementations of FDTD method computational core in Fortran, C, and C++ (using Blitz++ library) [25]. The implementations of 2D and 3D cases were tested on Intel i5-2400, AMD Phenom II X4 965 and Intel Xeon e5345 processors.

Intel and AMD processors were chosen for testing due to the widely spread x86 architecture. The abovementioned programming languages possess high level of portability, that's why any of our developed FDTD method implementations can be relatively easy ported to SPARC and Power architectures, that are present in TOP500 supercomputer list [26]. Another popular solution is specialized accelerators (for example, NVidia Tesla and Intel Xeon Phi), that are capable of reducing execution time dramatically. A number of studies [10, 27] shows FDTD method GPU implementation usage for various problems, but it requires more complicated programming and therefore is not considered in this article.

3. FDTD METHOD IMPLEMENTATION ARCHITECTURE

There are several approaches to implement FDTD method. One option is to store as little data as possible in the computer memory (according to memory consumption values this approach is used by Lumerical FDTD Solutions [11]). It requires storage of at least six numbers (three magnetic field components and three electric field components) for each of the grid points. Tensor components of material parameters such as magnetic and electric permeabilities, conductivities, etc are defined parametrically and are computed many times for every grid point at every time step. In case of an object with complex geometry features, these additional calculations may significantly increase computational load on the system.

Another option is a single-time evaluation of all the necessary components of material parameter tensors for every grid point and storing the result in computer memory (this approach is used by MEEP [14]). In this case, the computational load is reduced, but the required amount of memory is increased. For example, for the problem of electromagnetic wave propagation in dielectric medium, only a single material parameter is required — electric permeability value, which results in additional 15% memory consumption. Problems with complex media may require up to several times more memory consumption. In 3D case, the amount of memory required for computing is cubically proportional to the model spatial size, and that's why the maximum model size is limited by the amount of computer RAM. Keeping in mind that additional memory consumption preserves time needed for stepping when increasing the complexity of model geometry, this option was chosen for developing FDTD method implementations. Moreover, this approach allows to separate model parameters setting procedure and computational core programming and makes the process of developing easier.

Actually, FDTD method programming is just executing a number of basic mathematical operations such as addition, subtraction, multiplication, and division on arrays, that contain the magnitudes of electric and magnetic field components and material parameters. Note that the division operation should be used as rarely as possible, because it takes several times longer than other operations and one should perform "zero division" error check.

Modern processors have up to three levels of cache. The first level of cache has the smallest capacity, but the highest data access speed. Capacity of the second cache level is higher than capacity of the first one, but its speed is lower, and so on. Putting data components accurately in computer memory significantly increases the performance of FDTD method due to the reduction of cache miss rate (cache miss occurs when the data being addressed is not located in cache). A standard way to enhance the performance of computational algorithms by optimizing the cache usage is array cache blocking technique, that increases the locality of data used for computations. The next optimization step is increasing the efficiency of register usage [28]. Register blocking for high-order accurate FDTD method implementations is a relatively easy-to-realize technique, however, it failed to increase the performance of our implementations sufficiently.

3.1. Fortran Implementation

Fortran basic syntax enables iterating over chosen array slice without loops for incrementing spatial indices. Listing 1 shows code fragment

of H_x update equation in 2D case. The source code was compiled using `-O2` optimization flag of `gfortran 4.4` compiler.

```
data1(1:size,1:size,Hx) = data0(1:size,1:size,Chxh) *
 data0(1:size,1:size,Hx) - data0(1:size,1:size,Chxe) *
  (data0(1:size,2:(size+1),Ez) - data0(1:size,1:size,Ez))
```

Listing 1. Fortran code fragment of H_x update equation in 2D case.

A special feature of Fortran is columnwise memory layout of matrix elements, or, generalizing for multidimensional arrays, the most rapidly changing index within nested loops is the first index of the array. Incrementing the first array index one gets a continuous layout of array elements in memory. In the above implementation arrays `data0` and `data1` are used for $n\Delta_t$ and $(n+1)\Delta_t$ moments of time, physical quantities are addressed via a number of enumerated constants (`Hx`, `Ez`, `Chxh` and `Hx`). Correct addressing is pretty important, because changing its order from `data(x,y,F)` to `data(F,x,y)` results in up to several times performance reduction. Alternatively, one can use separate arrays for every required physical quantity, but in this case data layout in memory may become fragmentary.

3.2. C++ Implementation Using Blitz++

Blitz++ library provides a lot of useful functions to operate on arrays effectively [29]. Using template metaprogramming technique, Blitz++ does not waste processor resources on returning temporary results when operating on objects in run time, all substitutions are done at compile time. In addition, it also ensures good cache use with the help of Hilbert space filling curve array traverse order [30]. C++ source codes were compiled using `-O2` and `ftemplate-depth-30` optimization flags of `g++ 4.4` compiler.

There are two ways of FDTD method implementation in Blitz++: using `Range` or `Stencil` functions. `Range` generates a range of integers that act like an array index within a loop, performing operations on array elements, see Listing 2 for details. `Range` enables to program finite-difference equations in almost explicit "mathematical" form, whereas

```
const blitz::Range i(1,length_x);
const blitz::Range j(1,length_y);
data1(kHx,i,j) = data0(kChxh,i,j) * data0(kHx,i,j) -
 data0(kChxe,i,j) * (data0(kEz,i,j+1) - data0(kEz,i,j));
```

Listing 2. C++ code fragment of H_x update equation in 2D case using Range.

defining index increment range lightens the code.

`Stencil` is a more complicated operation that does not use explicit indexing. Instead, `Stencil` automatically determines the range of indices, analyzing the dimensions of its array arguments. A small drawback of this function is the restriction only to 3D and lower-dimension arrays, however one can easily tackle this issue. Listing 3 shows the code fragment of H_x update equation. In order to use `Stencil`, one has to define and describe the function (in this case `update_field_Hx`) using special Blitz++ macros and then call it in the proper place with `applyStencil()`. Note again that when using `Stencil` one has no access to indices range, and this may cause array boundaries determine algorithm to malfunction, especially in case of complex definitions. Nevertheless, on AMD and Intel i5 setups `Stencil` computes 2D problems significantly faster than `Range`.

```
BZ_DECLARE_STENCIL5(update_field_Hx, kHx_next, kHx, kChxh,
                    kChxe, kEz)
kHx_next = kChxh * kHx - kChxe * (kEz(0, 1) - kEz);
BZ_END_STENCIL_WITH_SHAPE(blitz::shape(0, 0),
                          blitz::shape(0, 1))
 ...
applyStencil(update_field_Hx(), kHx_next,
             kHx, kChxh, kChxe, kEz);
```

Listing 3. C++ code fragment of H_x update equation in 2D case using Stencil.

3.3. C Implementation

C code fragment of H_x update equation in 2D case is shown in Listing 4. Source code was compiled using `-O3 -fstrict-aliasing` optimizing flags of `gcc 4.4` compiler. This implementation turned out to have the highest potential for optimization, that is, however, rather tricky to realize. C implementation can be made fast only provided that the source code is scrupulously optimized by the programmer. Special code optimizations will be described further.

```
for (int i = x1; i <= x2; ++i) {
  for (int j = y1; j <= y2; ++j) {
    data1[kHx][i][j] = data0[kChxh][i][j] *
     data0[kHx][i][j] - data0[kChxe][i][j] *
      (data0[kEz][i][j+1] - data0[kEz][i][j]);}}
```

Listing 4. C code fragment of H_x update equation in 2D case without optimizations.

Unlike Fortran, C's most rapidly changing array index is the last one. In the above-mentioned code fragment data arrays have three indices. However, from the computer's point of view, it is just a continuous layout of elements in memory, i.e., a 1D array. Consequently, it is possible to exclude all indices except one, redefining it like `kHx*size_xy + i*size_y + j`, see Listing 5. Analogous representation is applicable to 3D case, reducing four indices to a single one. It allows to increase performance by a few percent because of a simpler array indexation.

```
for (int i = x1; i <= x2; ++i) {
  for (int j = y1; j <= y2; ++j) {
    data1[kHx * size_xy + i * size_y + j] =
     data0[kChxh * size_xy + i * size_y + j] *
      data0[kHx * size_xy + i * size_y + j] -
       data0[kChxe * size_xy + i * size_y + j] *
        (data0[kEz * size_xy + i * size_y + (j + 1)] -
         data0[kEz * size_xy + i * size_y + j]);}}
```

Listing 5. C code fragment of H_x update equation in 2D case with pointer arithmetics.

The next trick presented in Listing 6, is loop unroll. It helps the compiler to optimize the list of instructions and reduces the number of conditional statements to be checked within a loop. The technique is the following: update equations are copied several times within the innermost loop with incremented loop index, whereas the loop increment is adjusted to process all the elements of the array. It is no use to make more than four copies due to special features of x86 microarchitecture. Our tests showed that significant increase in performance can be obtained using only one copy of update equation. Additional performance increase can be provided by separating the innermost loop for each field component update equation.Corresponding C code fragment with all mentioned optimizations is presented in Listing 6.

Concluding the section it is worth mentioning that C code optimizations made this implementation much faster than non-optimized version. For example, on AMD setup optimizations granted 50% performance increase.

```
for (int i = x1; i <= x2; ++i) {
  for (int j = y1; j <= y2; j += 2) {
    data1[kHx * size_xy + i * size_y + j] =
     data0[kChxh * size_xy + i * size_y + j] *
      data0[kHx * size_xy + i * size_y + j] -
       data0[kChxe * size_xy + i * size_y + j] *
        (data0[kEz * size_xy + i * size_y + (j + 1)] -
         data0[kEz * size_xy + i * size_y + j]);
    data1[kHx * size_xy + i * size_y + j + 1] =
     data0[kChxh * size_xy + i * size_y + j + 1] *
      data0[kHx * size_xy + i * size_y + j + 1] -
       data0[kChxe * size_xy + i * size_y + j + 1] *
        (data0[kEz * size_xy + i * size_y + (j + 1) + 1] -
         data0[kEz * size_xy + i * size_y + j + 1]);}
  for (int j = y1; j <= y2; j += 2) {
    data1[kHy * size_xy + i * size_y + j] = ...}
  ...}
```

Listing 6. C code fragment of H_x update equation in 2D case with pointer arithmetics and innermost loop unroll.

4. COMPARING THE PERFORMANCE OF IMPLEMENTATIONS ON DIFFERENT HARDWARE SETUPS

Performance of different implementations has to be evaluated on a number of test problems with specified launch parameters in order to understand if any of them can outrace the others. To accomplish that, one has to measure the stepping time of electric and magnetic fields update algorithms of a problem with specified parameters — number of time steps and grid spatial size. It is convenient to introduce parameters `max_size` and `max_steps` and fixate the "general" task size ($\mathtt{C_{2D}} = \mathtt{max\_size}^2 \times \mathtt{max\_steps}, \mathtt{C_{3D}} = \mathtt{max\_size}^3 \times \mathtt{max\_steps}$) and then alter parameters, retaining the value of `C`. For further convenience, stepping time can be divided by an arbitrary value of grid points, in this contribution — by million grid points.

Implementations were tested on three different processors (CPU). Table 1 shows their specification, values of sequential memory read and write speed were obtained using bandwidth64 [30] test.

Stepping times for all implementations on all setups are presented in Fig. 1. For the majority of tasks, C and Fortran implementations are the best. Blitz++ implementation low performance on small spatial sizes is easily explainable — using library classes and Hilbert space filling curve traversal order takes dramatically long time compared to the acceleration it provides. However, with the growth of arrays' size, especially in 3D case, improvement of data locality and cache usage

Table 1. Specification of processors and measured performance of memory (read-write).

Processor		RAMr	Speed, [GB/s]		
			read	-	write
AMD Phenom II X4 965	3.4 GHz	DDR2-667	6.74	-	3.37
Intel Core i5-2400	3.1 GHz	DDR3-1333	14.27	-	8.61
Intel Xeon e5345	2.3 GHz	DDR2-667	3.6	-	1.92

provided by Blitz++ optimizations results in enhanced performance that allows the implementation using `Range` compete with others and even outrace them.

Due to different memory bandwidths for AMD and Intel (see Table 1) FDTD method implementations' performance showed approximately two times lower value on AMD processor comparing to Intel i5, whereas Linpack performance test gives AMD credit for performance.

Within every time and every spatial step, FDTD method's computational core performs 4 memory write and 10 memory read operations for 2D algorithm, 7 and 19 operations for 3D algorithm correspondingly, cache operations excluded. Using the general value of algorithm iterations and practically obtained sequential memory read and write speeds it is possible to evaluate problem stepping time and compare it to the obtained data. Results are presented in Table 2.

Table 2. Efficiency calculation for our best result implementations on different setups.

Problem size	t theory/practice, [s]			Efficiency, %		
	AMD	i5	Xeon	AMD	i5	Xeon
$2048^2 \times 100$	8.4/11.3	3.6/3.8	15.2/15.6	74	96	97
$128^3 \times 100$	7.7/11.2	3.4/4.0	13.9/17.0	68	85	82

Performance, obtained with the use of vector extension (single instruction multiple data or SIMD) SSE and AVX processor instructions, also turns out to be limited by memory bandwidth. Additional performance data for Intel Core i5-2400 using SSE is depicted in Fig. 1(d) with orange curve, AVX results are pretty much the same. Maximum acceleration, compared to non-SIMD implementation, does not exceed 16%, whereas when using SSE or AVX processor arithmetic performance for double precision (64b)

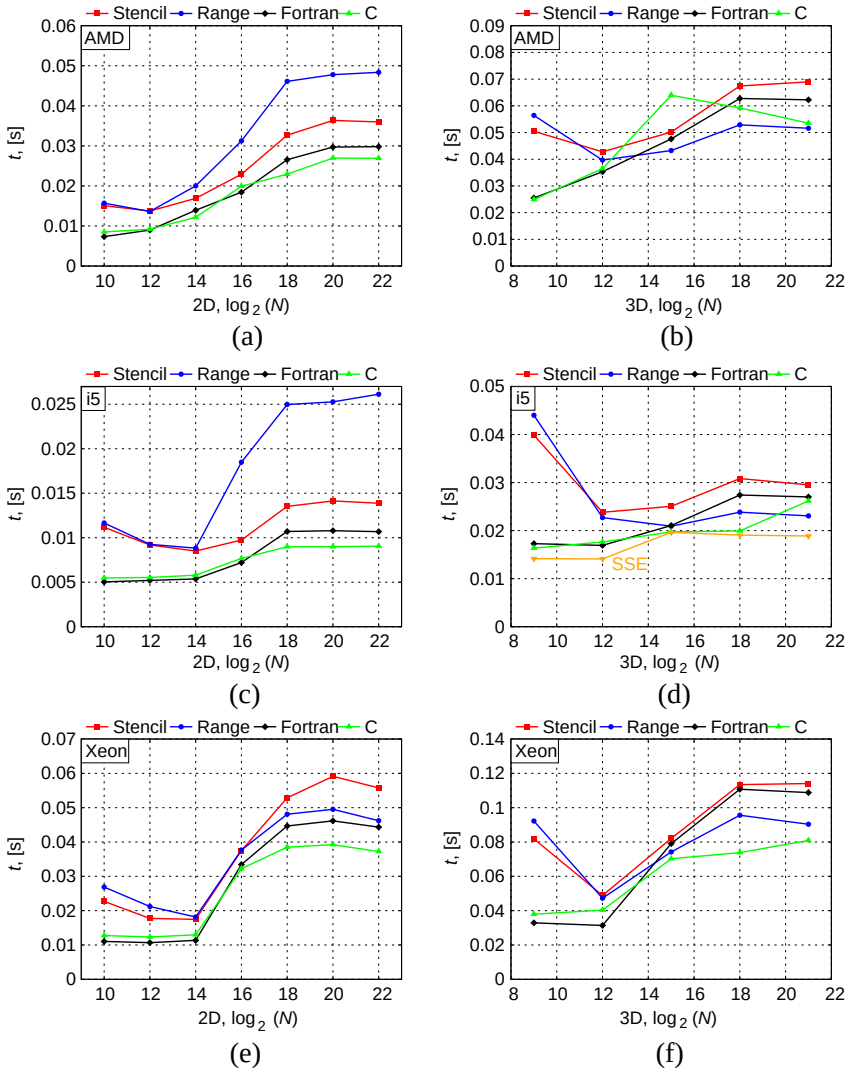

Figure 1. Performance comparison of 2D and 3D algorithms on (a), (b) AMD Phenom II X4 965, (c), (d) Intel Core i5-2400, and (e), (f) Intel Xeon e5345. t — single iteration stepping time for 10^6 grid points, N — general amount of spatial grid points.

computations it becomes 2 and 4 times higher, correspondingly. Memory bandwidth efficiency usage for this case slightly exceeds 85%.

The interrelation between processor performance, memory

bandwidth and FDTD method implementation efficiency is presented in Table 3. Specification of the computer with Intel i5-2400 processor was altered in BIOS and FDTD method. C implementation was tested on every configuration along with sequential memory read and write speed test and Intel Linpack test. Namely, processor frequency and memory bandwidth were tuned from 3.1 GHz to 2.6 GHz and from 1333 MHz to 800 MHz correspondingly. Relative performance variances were calculated with respect to the quickest configuration. The data obtained proved once again that FDTD method implementation performance strongly depends on computer memory bandwidth, whereas processor raw computational power only slightly affects it.

From the programmer's point of view, 2D and 3D FDTD algorithms differ only in the number of variables used. Both cases show about 90% memory bandwidth usage efficiency for Intel i5. This means that for other FDTD method implementations using the same amount of variables, similar memory bandwidth usage efficiency level can also be accessible. For example, it is reported in [10] that computing a $128^3 \times 200$ isotropic dispersive FDTD problem using GPU Nvidia Tesla C2050 took 3.24 seconds. Analogous CPU implementation will store material parameters of air and gold nanosphere in cache

Table 3. Interrelation between CPU frequency and memory bandwidth, stepping time for $128^3 \times 100$ of C implementation with SSE, measured sequential read from RAM speed and Linpack CPU performance test results.

Memory and processor settings				
RAM	high	high	low	low
CPU	high	low	low	high
Absolute values				
Stepping time, [s]	3.74	4.02	5.78	5.7
RAM read, [GB/s]	13.9	13.5	9.23	9.46
CPU frequency, [GHz]	3.1	2.6	2.6	3.1
Linpack, [Gflops]	87.5	74.5	71.1	82.1
Relative variance, %				
Stepping time	0	−7	−35	−34
RAM read	0	−3	−33	−32
CPU frequency	0	−16	−16	−0
Linpack	0	−15	−19	−6

and will require as much memory read and write operations as our implementation for anisotropic case. Under such restrictions, GPU implementation will perform two to three times faster than ours.

Concluding the section it is worth noticing once again that the major influence on FDTD method implementation stepping time is made by computer memory bandwidth. Consequently, the best usage of implementations is possible only on setups with maximum memory subsystem bandwidth, like multichannel high-frequency RAM.

5. COMPARISON WITH MEEP AND LUMERICAL FDTD SOLUTIONS

Let us compare our C implementation (Listing 6) to similar software packages. In order to make comparison as clear as possible, it is desired to test a specified problem using a chosen setup.

MEEP [14] is a well-known open source FDTD method implementation, an ideal candidate for comparison. Test problem was simplified to wave propagation in isotropic dielectric media with perfect reflecting walls, and stepping time of implementations was compared for maximum spatial sizes in 2D and 3D cases, see Table 4 for details. Our implementations proved to be 1.6 and 1.7 times faster correspondingly.

Table 4. Comparison of C implementation with Meep software package on AMD.

Problem spatial size	Meep	C implementation
$2048^2 \times 100$	11.15 s	7.09 s
$128^3 \times 100$	12.44 s	7.22 s

The comparison of our C implementation with commercial software simulation package Lumerical FDTD Solutions [11] was also made (Table 5). Stepping time of parallel computing using four cores was also checked. Our C implementation was parallelized with the help of OpenMP technology for the outermost loop `for` using `#pragma omp parallel for` directive. For single core computations, Lumerical FDTD Solutions shows similar performance for maximum spatial size and 1.7–2.2 times lower performance for smaller sizes. In the parallel four cores regime Lumerical FDTD Solutions shows 1.1 times faster stepping time for maximum spatial size and 3.3 slower for minimum size.

In parallel four cores regime both implementations show similar acceleration less than 25% with respect to stepping time of the largest

Table 5. Comparison of C implementation with Lumerical FDTD Solutions on Intel i5.

Problem spatial size	Lumerical FDTD		C implementation	
	4 cores	1 core	4 cores	1 core
$128^3 \times 1000$	21 s	26 s	24 s	26 s
$64^3 \times 8000$	29 s	42 s	28 s	25 s
$32^3 \times 64000$	29 s	50 s	8.8 s	23 s

problem spatial size. This proves one more time that the performance of FDTD method implementations is limited by computer memory bandwidth.

In parallel regime the problem is spatially decomposed and each core executes a part of computations. Each core of Intel i5-2400 processor has personal L1 and L2 cache memory and therefore the overall amount of used cache is increased. This results in enhanced performance of our implementation for small problem spatial sizes, that outraces Lumerical FDTD Solutions 2.6 times.

Lumerical FDTD Solutions low performance for small problem spatial sizes and a little higher performance for maximum problem spatial size is caused by the fact that the package uses the amount of memory required only for storing the electric and magnetic field components. Consequently, it is necessary to compute dielectric permeability for every grid point within every algorithm step. Our implementation does only single computation and stores the value in RAM, which results in additional memory read operations when updating field components and proportional reduction in performance for large problem spatial sizes. For small problem spatial sizes, when overall performance is limited to processor performance, additional calculations in Lumerical FDTD Solutions cause dramatical stepping time increase with respect to our implementation.

6. CONCLUSION

In this paper we made the comparison of FDTD method implementations in Fortran, C and C++ programming languages. The implementations were tested for 2D and 3D electromagnetic problems on AMD Phenom II X4 965, Intel Core i5-2400 and Intel Xeon e5345. Obtained results showed that C implementation is several percent faster than Fortran for the majority of problem spatial task sizes, and C++ implementation is much slower on small spatial sizes, but rapidly improves performance with spatial size growth.

Performance of implementations for maximum problem spatial size changes in several times for different setups, whereas the arithmetical performance of tested processors differs a lot less. This occurs because of the limitation by RAM bandwidth. Calculations and tests showed that our best implementation uses up to 90% of bandwidth limitation, which means that further improvement of source code is ineffective. It is only possible to improve performance using setups with better RAM bandwidth.

Theoretically, it is possible to program a FDTD method implementation that is capable of providing up to 40 times better performance. Our C implementation uses only about 10% of Intel i5-2400 single core computational capacity and about 2.5% of overall CPU computational capacity. To increase this percentage, FDTD method implementations have to be modified to reduce the amount of RAM addressing operations dramatically. Such modifications, called time-skewing, were presented in [31] for 1D parabolic equation finite-difference scheme. About 90% of overall CPU computational capacity usage was reported to had been reached.

ACKNOWLEDGMENT

This work was supported by the Ministry of Education and Science of Russian Federation, projects 11.G34.31.0020 and 14.B37.21.0942, and Russian Foundation for Basic Research (RFBR).

REFERENCES

1. Kane, Y., "Numerical solution of initial boundary value problems involving Maxwell's equations in isotropic media," *IEEE Transactions on Antennas and Propagation*, Vol. 14, 302–307, 1966.
2. Okamoto, T., H. Takenaka, T. Nakamura, and T. Aoki, "Large-scale simulation of seismic-wave propagation of the 2011 Tohoku-Oki M9 earthquake," *Proc. of the International Symposium on Engineering Lessons Learned from the 2011 Great East Japan Earthquake*, 349, Mar. 1–4, 2012.
3. Hallaj, I. M. and R. O. Cleveland, "FDTD simulation of finite-amplitude pressure and temperature fields for biomedical ultrasound," *J. Acoust. Soc. Am.*, Vol. 105, No. 5, L7–L12, 1999.
4. Kong, L.-Y., J. Wang, and W.-Y. Yin, "A novel dielectric conformal FDTD method for computing SAR distribution of the human body in a metallic cabin illuminated by an intentional

electromagnetic pulse (IEMP)," *Progress In Electromagnetics Research*, Vol. 126, 355–373, 2012.

5. Schneider, J. B., "Understanding the finite-difference time-domain method," www.eecs.wsu.edu/∼schneidj/ufdtd, 2012.
6. Taflove, A. and S. C. Hagness, *Computational Electrodynamics: The Finite-difference Time-domain Method*, Artech House, Inc., 685 Canton Street Nordwood, MA 02062, 2005.
7. Wang, M.-Y., J. Xu, J. Wu, B. Wei, H.-L. Li, T. Xu, and D.-B. Ge, "FDTD study on wave propagation in layered structures with biaxial anisotropic metamaterials," *Progress In Electromagnetics Research*, Vol. 81, 253–265, 2008.
8. Kung, F. and H. T. Chuah, "Stability of classical finite-difference time-domain (FDTD) formulation with nonlinear elements — A new perspective," *Progress In Electromagnetics Research*, Vol. 42, 49–89, 2003.
9. Chun, K., H. Kim, H. Kim, and Y. Chung, "PLRC and ADE implementations of Drude-critical point dispersive model for the FDTD method," *Progress In Electromagnetics Research*, Vol. 135, 373–390, 2013.
10. Lee, K. H., I. Ahmed, R. S. M. Goh, E. H. Khoo, E. P. Li, and T. G. G. Hung, "Implementation of the FDTD method based on Lorentz-Drude dispersive model on GPU for plasmonics applications," *Progress In Electromagnetics Research*, Vol. 116, 441–456, 2011.
11. http://www.lumerical.com.
12. http://www.remcom.com/xf7.
13. http://www.acceleware.com/fdtd-solvers.
14. Oskooi, A. F., D. Roundy, M. Ibanescu, P. Bermel, J. D. Joannopoulos, and S. G. Johnson, "MEEP: A flexible free-software package for electromagnetic simulations by the FDTD method," *Computer Physics Communications*, Vol. 181, 687702, 2010.
15. http://fdtd.kintechlab.com/ru/start.
16. http://www.angorafdtd.org.
17. Perlik, A. T., T. Opsahl, and A. Taflove, "Predicting scattering of electromagnetic fields using FDTD on a connection machine," *IEEE Trans. on Magnetics*, Vol. 2, No. 4. 2910–2912, 1989.
18. Chew, K. C. and V. F. Fusco, "A parallel implementation of the finite difference time-domain algorithm," *Int. J. Numer. Model. El.*, Vol. 8. 293–299, 1995.
19. Wang, J.-B., B.-H. Zhou, L.-H. Shi, C. Gao, and B. Chen, "A

novel 3-D weakly conditionally stable FDTD algorithm," *Progress In Electromagnetics Research*, Vol. 130, 525–540, 2012.

20. Mao, Y., B. Chen, H.-Q. Liu, J.-L. Xia, and J.-Z. Tang, "A hybrid implicit-explicit spectral FDTD scheme for oblique incidence problems on periodic structures," *Progress In Electromagnetics Research*, Vol. 128, 153–170, 2012.
21. Kong, Y.-D. and Q.-X. Chu, "Reduction of numerical dispersion of the six-stages split-step unconditionally-stable FDTD method with controlling parameters," *Progress In Electromagnetics Research*, Vol. 122, 175–196, 2012.
22. Sirenko, K., V. Pazynin, Y. K. Sirenko, and H. Bağci, "An FFT-accelerated FDTD scheme with exact absorbing conditions for characterizing axially symmetric resonant structures," *Progress In Electromagnetics Research*, Vol. 111, 331–364, 2011.
23. Izadi, M., M. Z. A. Ab Kadir, and C. Gomes, "Evaluation of electromagnetic fields associated with inclined lightning channel using second order FDTD-hybrid methods," *Progress In Electromagnetics Research*, Vol. 117, 209–236, 2011.
24. http://onzafdtd.org.
25. http://sourceforge.net/projects/blitz/.
26. www.top500.org.
27. Stefanski, T. P., "Implementation of FDTD-compatible Green's function on heterogeneous CPU-GPU parallel processing system," *Progress In Electromagnetics Research*, Vol. 135, 297–316, 2013.
28. Dursun, H., K. Nomura, W. Wang, M. Kunaseth, L. Peng, R. Seymour, R. K. Kalia, A. Nakano, and P. Vashishta, "In-core optimization of high-order stencil computations," *Proc. PDPTA*, 533–538, 2009.
29. Veldhuizen, T. L., "Scientific computing: C++ versus Fortran," *Dr. Dobb's Journal of Software Tools*, Vol. 22, No. 11, 34, 36, 38, 91, Nov. 1997.
30. http://zsmith.co/bandwidth.html.
31. Zumbusch, G., "Tuning a finite difference computation for parallel vector processors," *2012 11th International Symposium on Parallel and Distributed Computing, CPS*, 63–70, IEEE Press, 2012.