Enhanced Parallel FDTD Method Using SSE Instruction Sets

Lihong Zhang [1, 2], Xiaoling Yang [3], and Wenhua Yu [3]

[1] School of Information Engineering
Communication University of China, Beijing, 100024, China
pzyzlh@yahoo.cn

[2] Fundamentals Department
Chinese People's Armed Police Force Academy, Langfang Hebei, 065000, China

[3] Penn State University
University Park, PA, 16802, USA
ybob@2comu.com, wxy6@psu.edu

Abstract ─ To accelerate the simulation of the parallel FDTD method, this paper proposes an effective hardware acceleration technique based on the SSE instruction sets, and puts forward a three-level data parallel algorithm based on MPI, OpenMP and SSE instructions. To demonstrate the acceleration effect of this technique, this paper develops two types of codes using C language: one is based on MPI + OpenMP, another is based on MPI + OpenMP + SSE, and then draws a comparison between the computing time of the two types of codes in the numerical experiments for the same electromagnetic radiation problems. The experimental results show that this acceleration technique can achieve an acceleration rate of 2.44 for the ideal case on a PC cluster and 2.37 for the practical problem on a 2-CPU workstation without requiring any extra hardware investment, and provide an efficient and economical technique for the electromagnetic simulations.

Index Terms ─ CPML, FDTD, MPI, OpenMP, and SSE.

I. INTRODUCTION

Finite difference time domain (FDTD) method is firstly proposed by Yee in 1966 [1], and has grown into a relatively complete method system after development through decades. In the FDTD method, the electric (magnetic) field somewhere in space can be calculated by the explicit way through its previous value at the same location and the four magnetic (electric) fields around it at the half time step earlier. The updating equation of magnetic field component along the *z*-axis is given in (1) [2]. The updating equations of the other two components are similar to (1).

$$H_z\Big|_{i+\frac{1}{2},j+\frac{1}{2},k}^{n+1} = D_a\Big|_{i+\frac{1}{2},j+\frac{1}{2},k} H_z\Big|_{i+\frac{1}{2},j+\frac{1}{2},k}^{n} - D_b\Big|_{i+\frac{1}{2},j+\frac{1}{2},k} \left(\frac{E_y\Big|_{i+1,j+\frac{1}{2},k}^{n+\frac{1}{2}} - E_y\Big|_{i,j+\frac{1}{2},k}^{n+\frac{1}{2}}}{\Delta x} - \frac{E_x\Big|_{i+\frac{1}{2},j+1,k}^{n+\frac{1}{2}} - E_x\Big|_{i+\frac{1}{2},j,k}^{n+\frac{1}{2}}}{\Delta y}\right). \quad (1)$$

Compared with other numerical methods, the FDTD method becomes more and more popular for the practical and complex problems because of its simplicity and flexibility. Moreover, the main advantage of the FDTD method is that it is parallel in nature and it can be parallelized more efficiently than finite element method (FEM) or method of moments (MoM) [3]. Therefore, a variety of parallel algorithms have been proposed to reduce the computation time of FDTD electromagnetic simulation[4-5], such as parallel techniques based on message passing interface (MPI) [6] and OpenMP [7]. Recently, a large number of publications have been on the graphic processing unit (GPU) acceleration [8-13].

In this paper, we propose an effective hardware acceleration technique of parallel FDTD simulation using streaming SIMD (single

Submitted On: July 27, 2011
Accepted On: Dec 18, 2011

instruction multiple data) extensions (SSE) instruction sets [14] and develop a 3D parallel FDTD procedure based on C language, MPI library, OpenMP, and SSE instruction sets. The procedure has been validated through an ideal case and a practical problem.

II. SSE INSTRUCTION SETS

Each core in the multi-core processor has its own cache, floating point unit (FPU) and vector arithmetic logic unit (VALU), as shown in Fig. 1. Unlike the FPU, the VALU allows us to operate on four data at the same time. We use the VALU that includes a 128-bit vector unit through the SSE instruction sets to accelerate the parallel conformal FDTD code, as shown in Fig. 2[15].

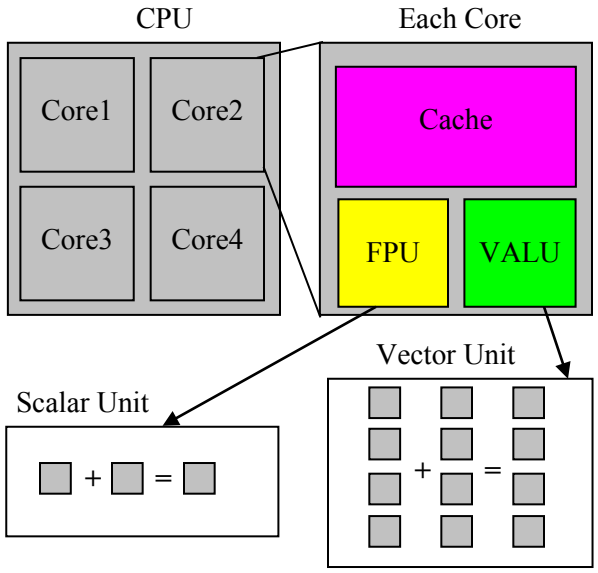


Fig. 1. CPU architecture including FPU and VALU.

SIMD was introduced into the Intel architecture with the MultiMedia eXtensions (MMX) technology. MMX technology allows SIMD computations to be performed on the packed byte, word, and double word integers. The Pentium III processor extended the SIMD computation model with the introduction of the SSE. SSE allows the SIMD computations to be performed on operands that contain four packed single-precision floating-point data elements. The operands can be in memory or in a set of eight 128-bit registers [14]. Figure 3 is a typical SIMD computation procedure.

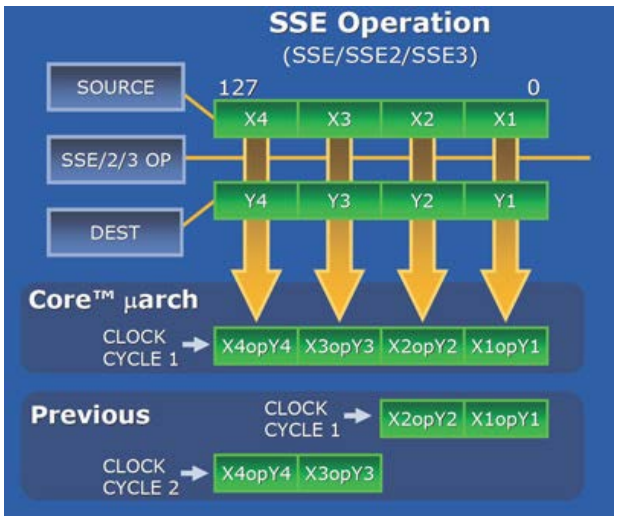


Fig. 2. Concept flowchart in VALU.

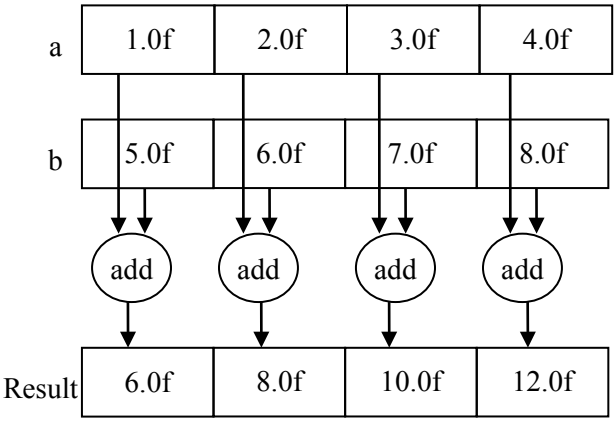


Fig. 3. Flowchart of the SIMD computation.

Recently, Intel Corporation extends previous SIMD offerings (MMX instructions and Intel streaming SIMD extensions) to advanced vector extensions (AVX). The 128-bit SIMD registers for SSE have been expanded to 256 bits. By this mean, SIMD computation procedure works as shown in Fig. 4[16]. Intel AVX is designed to support 512 or 1024 bits in the future.

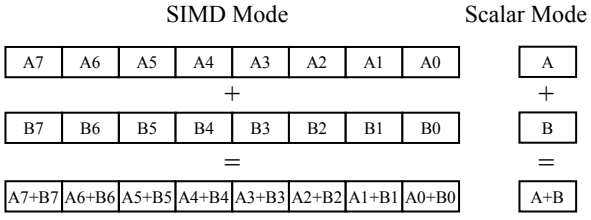


Fig. 4. SIMD computation for AVX.

III. FDTD CODE IMPLEMENTATION

A. FDTD memory allocation

A 3-D array in the FDTD code is allocated using the *malloc* function in C language, and *_aligned_malloc*(*size*, *alignment*) function in this paper. The parameter *size* in this function is the size of the requested memory allocation; the parameter *aligment* is the alignment value and is equal to 16 because that the SSE instruction sets require their memory operands to be aligned to 16-byte (16B) boundaries. For example, if we need a 3-D array *array_name*[*x_size*, *y_size*, *z_size*], we can first define a 1D array *array_name_tmp*[*N*] whose size is *N=x_size*y_size*z_size*, and then map the 1-D memory address to 3-D array *array_name*. The pseudo-code segment is demonstrated as below:

```
// allocate the 1-D memory
array_name_tmp = (float*)_aligned_malloc(
   sizeof ( float ) * x_size * y_size * z_size, 16 );
array_name =  ( float *** )_aligned_malloc (
  sizeof ( float** ) * x_size, 16 );
    for( i = 0; i < x_size; i++){
       array_name[i] = ( float ** )_aligned_malloc
          ( sizeof ( float* ) * y_size, 16 );
            for( j = 0; j < y_size; j++){
              // map the 1-D memory address to 3-D array
         map_address = i * y_size * z_size + j * z_size;
        array_name[i][j] =
 &array_name_tmp[map_address];
       }
    }
```

In the C programming language, the data inside the memory is contiguous in *y-z* plane. Suppose that the *y_size* and *z_size* are equal to 8, the data structure of the array *array_name* in the *y-z* plane is shown in Fig. 5. The memory addresses of the data elements (0,0,0), (0,0,1)…(0,0,7) is contiguous, likewise, the addresses of (0,1,0), (0,1,1)…(0,1,7) is contiguous too. The address of (0,0,7) is contiguous with the element (0,1,0). When we calculate the electric and magnetic fields in the *y-z* plane, we only need to know the address of the first element (supposed to be (0,0,0)) and the total number of elements (supposed to be 64), and then the 64 elements (0,0,0), (0,0,1), (0,0,2)…(0,7,7) are sequentially read in memory. In this case, the memory access is contiguous and therefore the cache hit ratio is relatively high.

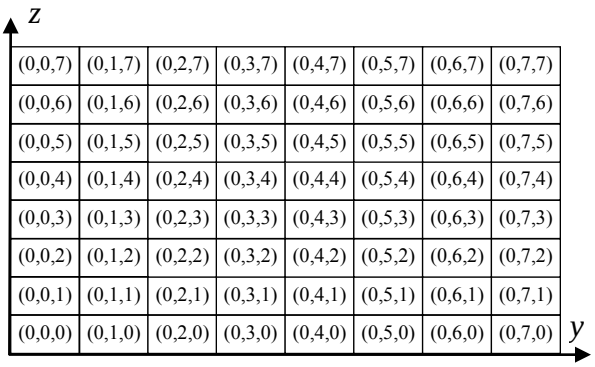


Fig. 5. Data structure in the y-z plane.

B. The partition of CPML boundary

The updating equation given in (1) is used to calculate the fields in the computational domain. However, the updating equation inside the PML layers should include two more extra terms, as shown in (2) [2]. The K value inside the computational domain equals to 1, and two Ψ terms in (2) are related to the PML material. Firstly, we update the electric and magnetic fields in the entire domain. We use the SSE instruction sets to load 4 float data into a SSE register at the same time. Secondly, we compute two Ψ terms and update the electric and magnetic fields only in the PML layers. The data division inside the PML layers is shown in Fig. 6.

$$H_z\Big|_{i+\frac{1}{2},j+\frac{1}{2},k}^{n+1} = D_a\Big|_{i+\frac{1}{2},j+\frac{1}{2},k} H_z\Big|_{i+\frac{1}{2},j+\frac{1}{2},k}^{n} - D_b\Big|_{i+\frac{1}{2},j+\frac{1}{2},k} \left[\frac{E_y\Big|_{i+1,j+\frac{1}{2},k}^{n+\frac{1}{2}} - E_y\Big|_{i,j+\frac{1}{2},k}^{n+\frac{1}{2}}}{\mathrm{K}_{x_{i+\frac{1}{2}}}\Delta x} - \frac{E_x\Big|_{i+\frac{1}{2},j+1,k}^{n+\frac{1}{2}} - E_x\Big|_{i+\frac{1}{2},j,k}^{n+\frac{1}{2}}}{\mathrm{K}_{y_{i+\frac{1}{2}}}\Delta y} + \Psi_{H_{z,x}}\Big|_{i+\frac{1}{2},j+\frac{1}{2},k}^{n+\frac{1}{2}} - \Psi_{H_{z,y}}\Big|_{i+\frac{1}{2},j+\frac{1}{2},k}^{n+\frac{1}{2}} \right]. \quad (2)$$

C. Three-level parallel architecture

The ordinary parallel FDTD code based on the MPI library or OpenMP is the one-level or the two-level parallel technique. In this paper, the FDTD code is the three-level parallel in which SSE is involved.

The first level parallelism is based on MPI in which the computational domain is broken into small sub-domains according to the number of available CPUs or nodes. The field update on the interface of each sub-domain is not independent,

namely, the field update on the interface requires the information from its neighbours through the MPI functions. However, the internal field update is independent which results in the high efficient parallel performance.

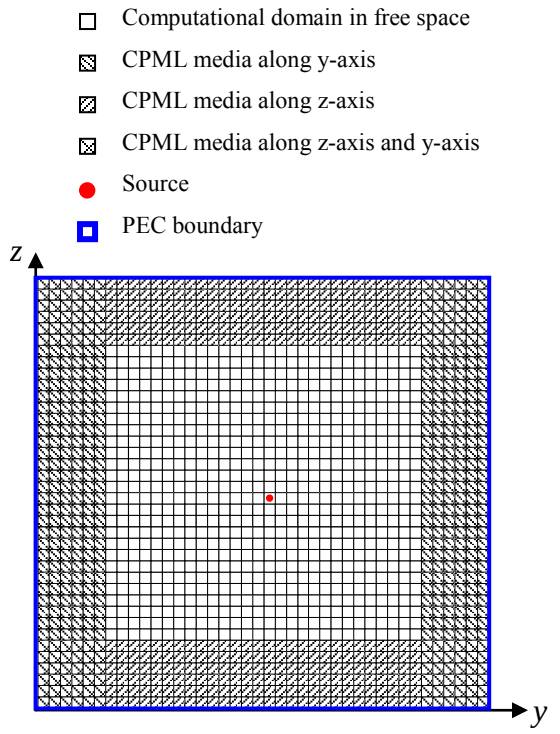


Fig. 6. Division of different regions inside the entire domain.

The second level parallelism is based on OpenMP. In the first place, several threads are generated by OpenMP based on the number of available cores; in the second place, the each thread is assigned to each core for the simulation. The framework of the algorithm is described as follows (In this paper, the parameters *imin* and *imax* are the lowest suffix and largest suffix respectively along *x*-axis of a computational domain; *jmin* and *jmax* are the lowest suffix and largest suffix respectively along *y*-axis; likewise, *kmin* and *kmax* are the lowest suffix and largest suffix respectively along *z*-axis. These six parameters define a cuboid computational domain which will be assigned to core, CPU or node.):

```
#pragma omp parallel private( num_threads, thread_num) {
        thread_num = omp_get_thread_num();
        num_threads = omp_get_num_threads();
        float imaxf = (float)imax / (float)num_threads;
        for ( i = imin + (int)((float)thread_num * imaxf);
              i <= (int)((float)(thread_num + 1) * imaxf);
              i ++) {
                for( j = jmin; j <= jmax; j++){
                  for( k = kmin; k <= kmax; k ++ ){
                    //Calculation
                  }
            }
      }
}
```

The third level parallelism is based on the SSE instruction sets. As mentioned earlier, ordinary arithmetic operations get one result but the SIMD computations get four results at the same time. This paper will concentrate on the SSE acceleration technique and code implementation.

D. SSE acceleration implementation

We here implement the SSE instruction sets to field update in the FDTD method and the field update inside the PML layers. Due to the data discontinuity inside the memory as shown in Fig. 6, the implementation of SSE inside the PML layer will degrade the performance of SSE. Firstly, we compute the magnetic components in the *z*-axis inside the entire computational domain following steps described below [17-18]:

(1) Define some __m128 variables that SSE requires and assign values to them (as operands of SSE computations);

(2) Load the coefficient into the SSE registers;

(3) Convert the float pointer to the SSE 128 bit pointer;

(4) Unroll the inner loop and reduce the cycle index to one-fourth of original numbers;

(5) Calculate the magnetic fields. The code implementation is described below:

```
// define __m128 variables
__m128 *vhz;
__m128 *vex, *vex_max, *vey, *vey_max;
__m128 vpHi, vpHj;
__m128 vDA = _mm_load1_ps( &DA );
for ( i = imin; i <= imax; i ++){
  // Load coefficient
  vpHi = _mm_load1_ps( &pHi [i] );
  for( j = jmin; j <= jmax; j++){
    // Load coefficient
    vpHj = _mm_load1_ps( &pHj [j] );
    // Convert float pointer to SSE 128 bit pointer
```

```
      vhz = ( __m128 * )hz[i][j];
            vex = ( __m128 * )ex[i][j];
            vex_max = ( __m128 * )ex[i][j+1];
            vey = ( __m128 * )ey[i][j];
            vey_max = ( __m128 * )ey[i+1][j];
            // reduce the cycle index
      for( k = 0, vk = 0; k <= kmax / 4; k += 4, vk++){
     // calculate the magnetic field
       vhz[vk]=_mm_sub_ps(_mm_mul_ps(vDA,vhz[vk]),
       _mm_sub_ps(_mm_mul_ps(vpHi,_mm_sub_ps(vey_max
      [vk],vey[vk])),_mm_mul_ps(vpHj,_mm_sub_ps(vex_max
       [vk], vex[vk])));
       }
     }
}
```

The update processing of the electric and magnetic fields inside the PML region is similar to those inside the entire computational domain. For example, when we calculate the magnetic field component along *y*-axis in the PML region, we can reference to the steps earlier and the pseudo-code is shown as follows:

```
__m128 *vhx;
__m128 *vez, *vez_max;
__m128 vBeta_PML, vpHj_PML;
__m128 vDB = _mm_load1_ps( &DB );
__m128 *vpusai_hxz;
for ( i = imin; i <= imax; i ++) {
   for( j = jmin_pml; j <= jmax_pml; j++){
      vBeta_PML = _mm_load1_ps(&Beta_PML [j]);
      vpHj_PML = _mm_load1_ps(&pHj_PML [j]);
            vhx = (__m128 *)hx[i][j];
            vez = (__m128 *)ez[i][j];
            vez_max = (__m128 *)ez[i][j+1];
            vpusai_hxz = (__m128 *)pusai_hxz[i][j-jshift];
            for( k = 0, vk = 0; k <= kmax / 4; k += 4, vk++){
       // Calculate Ψ
      vpusai_hxz[vk]=_mm_add_ps(_mm_mul_ps(vBeta_PML
,
    vpusai_hxz[vk]),_mm_mul_ps(vpHj_PML,mm_sub_ps(vez
       _max [vk], vez[vk])));
     // Update the magnetic field in PML domain
       vhx[vk]=_mm_sub_ps(vhx[vk],_mm_mul_ps(vDB,
       vpusai_hxz[vk]));
       }
     }
 }
```

To optimize the procedure and to improve the cache hit ratio, we can combine the calculation of electric and magnetic fields with the treatment of the PML boundary in the following scheme.

```
for ( i = imin; i <= imax; i ++) {
   for( j = jmin; j <= jmax; j++){
            while (vk < vkmax) {
       // calculate the electric or magnetic field
       }
       if ( the value of  j belongs to CPML domain){
        // add the PML boundary
       }
   }
}
```

IV. EXPERIMENTAL RESULTS

To demonstrate the acceleration efficiency by using the SSE instruction sets, we use the FDTD code enhanced with the SSE instruction sets to simulate the simple example that includes only one point source and field distribution output in one surface. The computational domain is separately divided into 40×40×40, 80×80×80, 120×120×120, 160×160×160, and 200×200×200 uniform cells, respectively, and is truncated by a 6-layer CPML. The excitation pulse is taken to be a pure Gaussian pulse and the excitation source is located at the center of the computational domain, as shown in Fig. 6. The numerical experiments were carried out on a PC cluster with Gigabit Ethernet. Every PC of the PC cluster is installed with an Intel Core 2 Duo CPU E7500, 2.93GHz. The experimental results for 600 time steps are summarized in Table 1. We observe from Table 1 that the SSE instruction sets can reduce the computing time of the FDTD simulation. The acceleration factor increases with the growth of the number of cells since the PML region will have less relative contribution to the simulation time when the problem size becomes larger if we fix the number of PML layers to be 6.

The ideal acceleration factor should be 4 since the vector unit based on the SSE instruction sets is four times faster than the floating point unit. However, due to the discontinuous data in the PML boundary, communication between processes and so on, the performance of the code will be reduced. That is to say, without MPI and OpenMP, the performance of the code only based on SSE instruction sets will increase to some extent. In this paper we achieve an acceleration factor of 2.44 when the number of cells is 8

million and a 6-layer PML is applied to truncate the computational domain.

Table 1: Acceleration factor by using SSE instruction sets (Time: Second)

Number Of cells	Coed Based on	Computing time	Acceleration factor
0.064 Mcells	MPI+ OpenMP	3.59	1.65
	MPI+ OpenMp +SSE	2.18	
0.512 Mcells	MPI+ OpenMP	18.21	1.93
	MPI+ OpenMp +SSE	9.45	
1.728 Mcells	MPI+ OpenMP	50.27	2.17
	MPI+ OpenMp +SSE	23.20	
4.096 Mcells	MPI+ OpenMP	110.89	2.36
	MPI+ OpenMp +SSE	46.92	
8 Mcells	MPI+ OpenMP	196.86	2.44
	MPI+ OpenMp +SSE	80.84	

In the practical problems, the simulation factors such as outputs, dispersive media, and near-to-far field transformation will influence the SSE performance due to the discontiguous data structure inside memory. However, it can be improved by optimizing the cache hit ratio.

V. ENGINEERING APPLICATION

In this part, we use the parallel FDTD code accelerated by using the SSE acceleration to simulate a waveguide (WR75) filter problem [19]. The filter includes five cavities and is excited by TE10 mode at one end, as shown in Fig. 7. The output parameter transmission coefficient for the TE10 mode measured at another end. The purpose is to investigate the performance of SSE acceleration on the 2-CPU (16 threads) workstation for the practical problem.

For the sake of comparison, we use the FEM method [20] to simulate the same problem and plot the results in the Fig. 8. It is evident from Fig. 8 to observe the good agreement. It is worthwhile to mention that the results are the same with and without the SSE acceleration.

The parallel FDTD performance with the SSE acceleration is summarized in Table 2. It is observed from Table 2 that the SSE can accelerate the FDTD code 2.37 times for this practical problem.

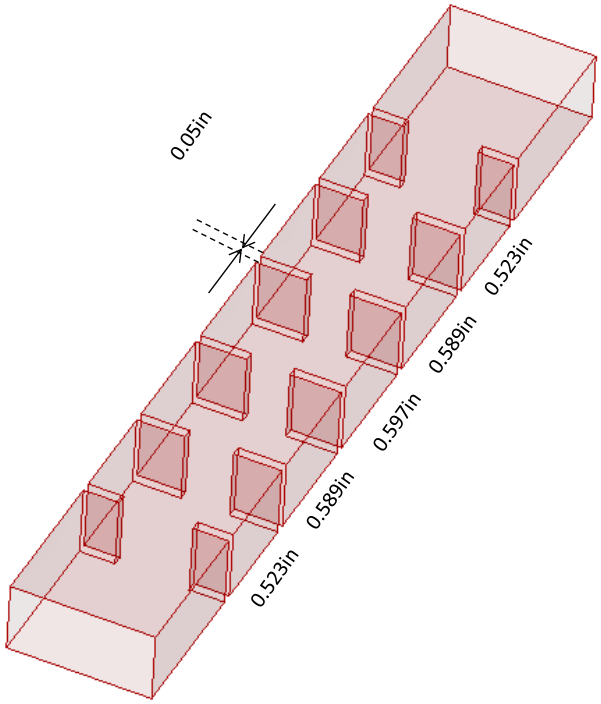


Fig. 7. Configuration of waveguide filter.

VI. CONCLUSION

In this paper, we propose a new hardware acceleration technique based on the SSE instruction sets and gives an implementation on both the PC cluster and workstation platforms. The result shows that this technique can improve the computing efficiency without any extra hardware investment, and provide an efficient and economical technique for the electromagnetic simulations. The further work will be to optimize the data structure inside the memory to further improve the SSE performance and to accelerate the FDTD simulation using AVX.

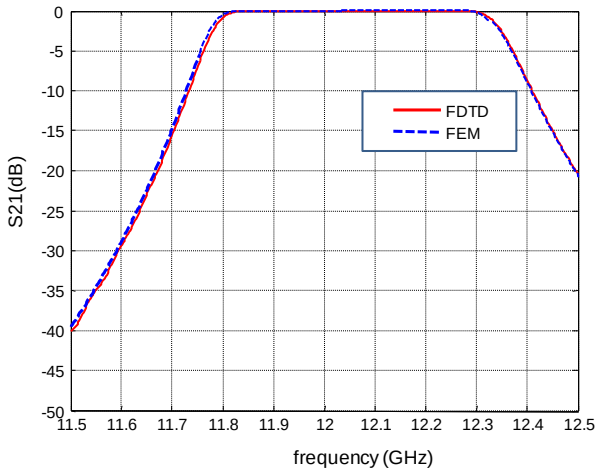


Fig. 8. Transmission coefficient of waveguide filter.

Table 2: Parallel FDTD performance with the SSE acceleration

	FDTD with SSE Acceleration	**FDTD without SSE Acceleration**
Workstation	2×AMD Opteron 6128 2.0GHz	
Memory Usage	37 MB	37 MB
Simulation time	145 sec.	345 sec.

REFERENCES

[1] K. S. Yee, "Numerical Solution of Initial Boundary Value Problems Involving Maxwell´s Equations in Isotropic Media," *IEEE Trans. Antennas Propagat*., vol. AP-14, pp. 302-307, 1966.

[2] A. Taflove and S. Hagness, *Computational Electrodynamics: The Finite-Difference Time Domain Method*, Artech House, Norwood, May 2005.

[3] W. Yu, R. Mittra, T. Su, et al., *Parallel Finite Difference Time Domain Method*, Communication University of China Press, July, 2005.

[4] W. Yu, et al., "A Robust Parallel Conformal FDTD Processing Package using the MPI Library," *IEEE Antennas and Propagation Magazine*, vol. 47, no. 3, pp. 39-59, June 2005.

[5] Y. Zhang, W. Ding, and C. H. Liang, "Study on the Optimum Virtual Topology for MPI Based Parallel Conformal FDTD Algorithm on PC Clusters," *J. of Electromagn. Waves and Appl.*, vol. 19, no. 13, pp. 1817-1831, 2005.

[6] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: Portable Parallel Programming with the Message-Passing Interface. 2nd ed.*, MIT Press, Cambridge, Nov., 1999.

[7] https://computing.llnl.gov/tutorials/openMP/

[8] V. Demir and A. Z. Elsherbeni, "Compute Unified Device Architecture (CUDA) based Finite-Difference Time-Domain (FDTD) Implementation," *Applied Computational Electromagnetic Society (ACES) Journal*, vol. 25, no. 4, pp. 303-314, Apr. 2010.

[9] N. Takada, T. Shimobaba, N. Masuda, and T. Ito, "Improved Performance of FDTD Computation using a Thread Block Constructed as a Two-Dimensional Array with CUDA," *Applied Computational Electromagnetic Society (ACES) Journal*, vol. 25, no. 12, pp. 1061-1069, Dec. 2010.

[10] M. Ujaldon, "Using GPUs for Accelerating Electromagnetic Simulations," *Applied Computational Electromagnetic Society (ACES) Journal*, vol. 25, no. 4, pp. 294-302, Apr. 2010.

[11] M. Weldon, L. Maxwell, D. Cyca, M. Hughes, C. Whelan, and M. Okoniewski, "A Practical Look at GPU-Accelerated FDTD Performance," *Applied Computational Electromagnetic Society (ACES) Journal*, vol. 25, no. 4, pp. 315-322, Apr. 2010.

[12] M. J. Inman, A. Z. Elsherbeni, J. G. Maloney, and B. N. Baker, "Practical Implementation of a CPML Absorbing Boundary for GPU Accelerated FDTD Technique," *Applied Computational Electromagnetic Society (ACES) Journal*, vol. 23, no. 1, pp. 16-22, Mar. 2008.

[13] N. Takada, N. Masuda, T. Tanaka, Y. Abe, and T. Ito, "A GPU Implementation of the 2-D Finite-Difference Time-Domain Code using High Level Shader Language," *Applied Computational Electromagnetic Society (ACES) Journal*, vol. 23, no. 4, pp. 309-316, Dec. 2008.

[14] Intel Corporation, Intel Architecture Optimization Reference Manual, Available: http://www.intel.com/design/pentiumii/manuals/245127.htm.

[15] http://www.tecchannel.de/server/hardware/437111/wechsel_an_der_spitze_intels_neue_core_prozessoren/index9.html

[16] http://software.intel.com/en-us/articles/introduction-to-intel-advanced-vector-extensions/

[17] W. Yu, "A Novel Hardware Acceleration Technique for High Performance Parallel FDTD Method," *Microwave Technology & Computational Electromagnetics (ICMTCE), 2011 IEEE International Conference on*, pp. 441-444, May 2011.

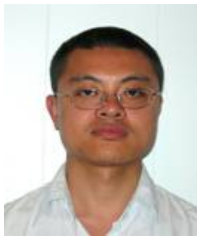
[18] W. Yu, X. Yang, Y. Liu, et al., *Advanced FDTD Methods: Parallelization, Acceleration, and*

Engineering Applications, Artech House, Boston, June 2011.

[19] M. Yu, "Power-Handling Capability for RF Filters," *IEEE Microwave Magazine*, vol. 8, no. 5, pp. 89-97, Oct. 2007.

[20] J. M. Jin, *The Finite Element Method in Electromagnetics*, New York: John Wiley & Sons, 2002.

Lihong Zhang is presently working on her Ph.D. in parallel computing and will graduate next year from Communication University of China. Her research interests include parallel processing techniques, numerical methods and software development.



Xiaoling Yang is a research associate in Material Research Institute of Pennsylvania State University. He received his B.S. and M.S. in Communication and Mathematics from Tianjin University in 2001 and 2004, respectively. He has published three books related to the FDTD method, parallel processing techniques, software development technique, and simulation techniques. He has published over 20 technical papers. His research interests include numerical methods, visual languages and software development.

Wenhua Yu is with 2COMU, Inc. and serves as the president of 2COMU. He was with Pennsylvania State University from 1996 to 2011. He received his Ph.D. in Electrical Engineering from the Southwest Jiaotong University in 1994. He worked at the Beijing Institute of Technology as a Postdoctoral Research Associate from February 1995 to August 1996. He has published six books related to the FDTD method, parallel processing techniques, software development technique, and simulation techniques from 2003 to 2011. He has published over 150 technical papers and four book chapters. He is a senior member of IEEE. His research interests include computational electromagnetic methods, software development techniques, parallel processing techniques, and simulation and design of the antennas, antenna arrays and microwave circuits.