Acceleration Study for the FDTD Method Using SSE and AVX Instructions

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

[1]Fundamentals Department, Chinese People's Armed Police Force Academy, Langfang, China

[2, 3]Penn State University, University Park, PA, USA

[*1]pzyzlh@yahoo.cn; [2]ybob@2comu.com; [3]wenyu@2comu.com

Abstract- **Parallel Finite Difference Time Domain (FDTD) method has been studied for a long time due to the expensive computation and huge memory needed for its application. In this paper, we introduce a novel hardware acceleration technique based on Vector Arithmetic Logic Unit (VALU) built in a regular CPU for parallel FDTD simulation with Convolutional Perfect Matched Layer (CPML) absorbing boundary condition (ABC). We discuss the acceleration effect for the FDTD method using SSE and AVX instructions and give an implementation on PC. We developed three types of code: code developed by C language, code accelerated by SSE instructions and code accelerated by AVX instructions. We analyse the radiation character of two models of dipole antenna and rectangle micro strip patch antenna using the accelerated code and present the results of a performance study of the FDTD algorithm on Intel 2nd core i5 2320 (quad core) named Sandy Bridge. The results show that we can improve the performance of FDTD algorithm using SSE and AVX instructions.**

Keywords- FDTD; VALU; SIMD; SSE; AVX; Acceleration

I. INTRODUCTION

FDTD method is firstly put forward by Yee in 1966 [1] and recently becomes more and more popular among students, engineers and researchers when they solve electromagnetic (EM) problems. Compared with other numerical methods, FDTD method becomes more and more popular for the practical and complex problems because of its simplicity and flexibility. Moreover, the main advantage of 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) [2]. Like other tools, FDTD method also faces with some challenges when coping with large and complex practical problems for the reason of intensive computation and large memory consumption, and for that reasons, multi-core, high performance cluster and so on are designed to accelerate the computation and to provide large size memory. Parallel technique is introduced to reduce the execution time of EM simulation and parallel code is developed based on OpenMP or MPI (Message Passing Interface) [3-6]. Recently, a large number of publications have been on the Graphic Processing Unit (GPU) acceleration [7-10]. However there are many challenging issues today for GPU to solve the practical electromagnetic problems.

In this paper, we propose an effective hardware acceleration technique of FDTD simulation using Streaming SIMD (Single Instruction Multiple Data) Extensions (SSE) and AVX (Advanced Vector Extensions) instruction sets to accelerate the 3D FDTD method. We developed three types of code: code developed by C language, code accelerated by SSE instructions and code accelerated by AVX instructions on PC environment with Windows 7 operating system.

II. FDTD METHOD AND BOUNDARY CONDITIONS

The famous Yee algorithm solves for electric and magnetic fields in time and space using the coupled Maxwell's curl equation that centred differences are realized both in the calculation of each field component with this field arrangement and in the time domain. The explicit finite difference approximations for E_z and H_z components are gotten as below [11].

$$E_z^{n+1}(i,j,k+\frac{1}{2}) = \frac{1-\frac{\Delta t\sigma_z}{2\varepsilon_z\varepsilon_0}}{1+\frac{\Delta t\sigma_z}{2\varepsilon_z\varepsilon_0}} E_z^n(i,j,k+\frac{1}{2}) + \frac{\frac{\Delta t}{\varepsilon_z\varepsilon_0}}{1+\frac{\Delta t\sigma_z}{2\varepsilon_z\varepsilon_0}} \left[\begin{array}{l} \dfrac{H_y^{n+\frac{1}{2}}(i+\frac{1}{2},j,k+\frac{1}{2}) - H_y^{n+\frac{1}{2}}(i-\frac{1}{2},j,k+\frac{1}{2})}{\Delta x} \\ -\dfrac{H_x^{n+\frac{1}{2}}(i,j+\frac{1}{2},k+\frac{1}{2}) - H_x^{n+\frac{1}{2}}(i,j-\frac{1}{2},k+\frac{1}{2})}{\Delta y} \end{array} \right] \tag{1}$$

$$H_z^{n+\frac{1}{2}}(i+\frac{1}{2},j+\frac{1}{2},k) = \frac{1-\frac{\Delta t\sigma_{Mz}}{2\mu_z\mu_0}}{1+\frac{\Delta t\sigma_{Mz}}{2\mu_z\mu_0}} H_z^{n-\frac{1}{2}}(i+\frac{1}{2},j+\frac{1}{2},k) + \frac{\frac{\Delta t}{\mu_z\mu_0}}{1+\frac{\Delta t\sigma_{Mz}}{2\mu_z\mu_0}} \left[\begin{array}{l} \dfrac{E_x^n(i+\frac{1}{2},j+1,k) - E_x^n(i+\frac{1}{2},j,k)}{\Delta y} \\ -\dfrac{E_y^n(i+1,j+\frac{1}{2},k) - E_y^n(i,j+\frac{1}{2},k)}{\Delta x} \end{array} \right] \tag{2}$$

From the Yee space lattice (Yee cell) illustrated in Fig. 1 and Fig. 2, we can see that E and H components are centred in three-dimensional space and each E component is surrounded by four circulating H components and vice versa. Moreover, the E cell and H cell interlace with each other by a half cell. This field arrangement can commendably satisfy the continuity of tangential field components automatically [11]. In FDTD method, electromagnetic wave propagation and the interaction between electromagnetic wave and medium are achieved through the differential recurrence of electric and magnetic fields in space and time. 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) and (2). The updating equations of the other four components are similar to (1) and (2).

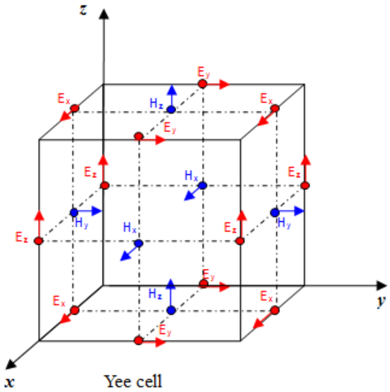

Fig. 1 Yee cell

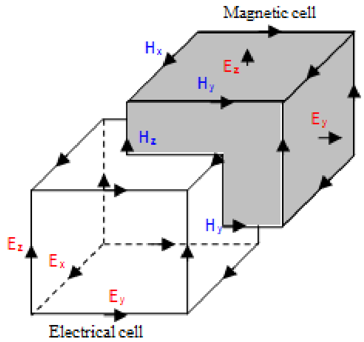

Fig. 2 Electric cell and magnetic cell

The boundary condition is very important for the solution of FDTD method. For the limitation of memory and computing speed, the computational domain of FDTD method is finite. In fact, the FDTD method develops slowly in early years until the present of the Mur ABC by G. Mur [12]. After the Mur ABC, many kinds of ABC had sprung up such as PML. With the ABC, the FDTD method can handle practical EM problems. In this paper, we use the PEC boundary condition and CPML ABC. When CPML is involved, the iterative formula must be modified. For example, Equation (1) is modified to (3) and (4) by the addition of two extra items Ψ. The same modification for other five iterative formulas is ok [11].

$$\begin{aligned} E_x^{n+1}(i+\frac{1}{2},j,k) &= \frac{1-\dfrac{\Delta t\sigma_x}{2\varepsilon_x\varepsilon_0}}{1+\dfrac{\Delta t\sigma_x}{2\varepsilon_x\varepsilon_0}} E_x^n(i+\frac{1}{2},j,k) + \frac{\dfrac{\Delta t}{\varepsilon_x\varepsilon_0}}{1+\dfrac{\Delta t\sigma_x}{2\varepsilon_x\varepsilon_0}} \frac{1}{K_y\Delta y}\begin{bmatrix} H_z^{n+\frac{1}{2}}(i+\frac{1}{2},j+\frac{1}{2},k) \\ -H_z^{n+\frac{1}{2}}(i+\frac{1}{2},j-\frac{1}{2},k)\end{bmatrix} \\ &- \frac{\dfrac{\Delta t}{\varepsilon_x\varepsilon_0}}{1+\dfrac{\Delta t\sigma_x}{2\varepsilon_x\varepsilon_0}} \frac{1}{K_z\Delta z}\begin{bmatrix} H_y^{n+\frac{1}{2}}(i+\frac{1}{2},j,k+\frac{1}{2}) \\ -H_y^{n+\frac{1}{2}}(i+\frac{1}{2},j,k-\frac{1}{2})\end{bmatrix} + \frac{\dfrac{\Delta t}{\varepsilon_x\varepsilon_0}}{1+\dfrac{\Delta t\sigma_x}{2\varepsilon_x\varepsilon_0}} \begin{bmatrix} \psi_{exy}^{n+\frac{1}{2}}(i+\frac{1}{2},j+\frac{1}{2},k) \\ -\psi_{exz}^{n+\frac{1}{2}}(i+\frac{1}{2},j,k+\frac{1}{2})\end{bmatrix} \end{aligned} \tag{3}$$

$$\psi_{exy}^{n+\frac{1}{2}}(i+\frac{1}{2},j+\frac{1}{2},k) = b_y\psi_{exy}^{n-\frac{1}{2}}(i+\frac{1}{2},j+\frac{1}{2},k) + a_y\frac{H_z^{n+\frac{1}{2}}(i+\frac{1}{2},j+\frac{1}{2},k) - H_z^{n+\frac{1}{2}}(i+\frac{1}{2},j-\frac{1}{2},k)}{\Delta y} \tag{4}$$

$$\psi_{exz}^{n+\frac{1}{2}}(i+\frac{1}{2},j,k+\frac{1}{2}) = b_z\psi_{exz}^{n-\frac{1}{2}}(i+\frac{1}{2},j,k+\frac{1}{2}) + a_z\frac{H_y^{n+\frac{1}{2}}(i+\frac{1}{2},j,k+\frac{1}{2}) - H_y^{n+\frac{1}{2}}(i+\frac{1}{2},j,k-\frac{1}{2})}{\Delta z}$$

III. SIMD PARALLELISM

The Intel Multi Media Extensions (MMX) technology was introduced into the IA-32 architecture in the Pentium II processor family and Pentium processor. The extensions introduced in MMX technology support a SIMD execution model. MMX technology allows SIMD computations to be performed on the packed byte, word, and double word integers. In 1999, the Pentium III processor extended the SIMD computation model with the introduction of the128-bit SSE to support the performing of four single-precision floating-point computations with a single instruction in parallel, implemented through of eight 128-bit data registers named xmm0, xmm1, ..., xmm7, as shown in Fig. 3. AMD also added support for SSE instructions soon after. Most available modern processors can perform SIMD operations now [13].

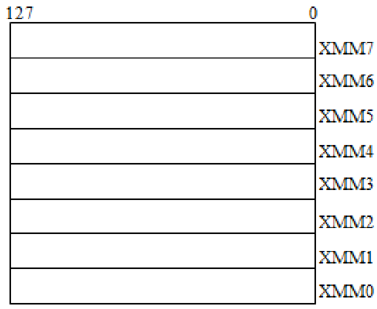

Fig. 3 SSE register set

In SSE instruction set, new methods of coding algorithms are required such as vectorization which transforms sequentially executing or scalar code into code that can execute in parallel. Traditional CPU instructions take single values as their operands but SSE instructions need new data types as their operands because of vectorization. Intel has defined two new C data types, representing 64-bit and 128-bit objects (__m64 and __m128, respectively) as operands. For example, the add operation of four couples floating-point data can be implemented by _mm_add_ps (__m128 a, __m128 b) instruction, as shown in Fig. 3. Where the variables *a* and *b* are the operands of the _mm_add_ps instruction [13]. Fig. 4 shows a typical SSE computation. The fundamental principle of the SIMD computation is based on the hardware named VALU.

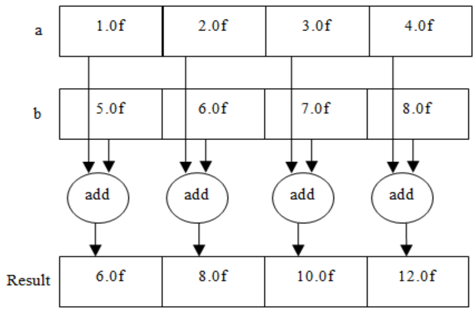

Fig. 4 Flowchart of the SIMD computation

The need for greater computing performance continues to grow across industry segments. To support the rising demand and evolving usage models, recently, Intel Corporation extends previous SIMD offerings (MMX instructions and Intel Streaming SIMD Extensions) to AVX. Intel AVX is a new 256 bit instruction set extension to SSE and is designed for Floating Point (FP) intensive applications. It was released early 2011 as part of the Sandy Bridge processor family. Intel AVX improves performance due to wider vectors, new extensible syntax, and rich functionality. These results in better management of data and general purpose applications like image, audio/video processing, scientific simulations, financial analytics and 3D modelling and analysis. The 128-bit SIMD registers for SSE have been expanded to 256 bits as shown in Fig. 5 [14]. By this mean, SIMD computation procedure works as shown in Fig. 6 [14]. Intel AVX is designed to support 512 or 1024 bits in the future.

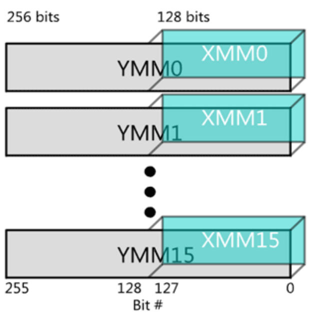

Fig. 5 AVX and SSE register set

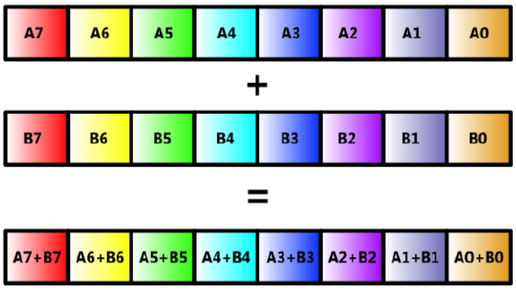

Fig. 6 SIMD computation of AVX

In modern CPU, each core in the multi-core processor has its own cache, Floating Point Unit (FPU) and VALU, as shown in Fig. 7. Unlike the FPU, the VALU allows us to operate on four or eight data at the same time. We use the VALU that separately includes a 128-bit and a 256-bit vector unit through the SSE and AVX instruction sets to accelerate the FDTD code.

Moreover, the SSE instructions or the AVX instructions operate on four or eight set of data at the same time, and when we use the VALU, we should give a special arrangement for the allocation of the data structures to increase the cache hit rate. In other words, if the cache hit rate is very low, then the data needed by SSE or AVX instructions are not ready in cache, and the total computing time will dramatically increase for the reason of memory access. The memory allocation method here is the same as that used in [15].

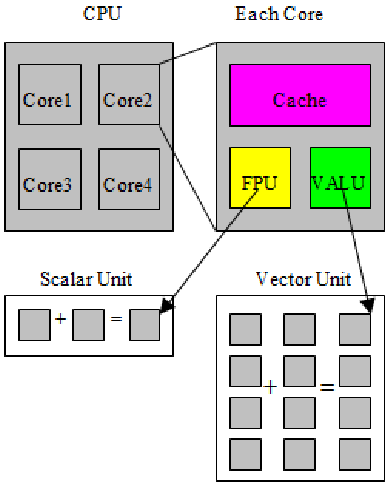

Fig. 7 SIMD computation of AVX

IV. EXPERIMENTAL RESULTS

A. Physical Model, Software and Hardware Environment

To demonstrate the acceleration efficiency by using the SSE and AVX instruction sets, we use the FDTD code developed by C language and enhanced with the SSE and AVX instruction sets to simulate the radiation character of dipole antenna model and rectangle patch antenna model using the codes below.

The numerical experiments were carried out on a PC with Intel 2nd Core i5 2320 CPU (3.0GHz; RAM: DDR3, 8GB; 1st-level Data Cache: 32KB, 8-way set associative, 64-byte line size; 1st-level Instruction Cache: 32KB, 8-way set associative, 64-byte line size; 2nd-level Cache: 256KB, 8-way set associative, 64-byte line size; 3rd-level Cache: 6MB, 12-way set associative, 64-byte line size) and Windows 7 Operating System, and the codes are compiled in visual studio 2012 professional trial version integrated development environment of Microsoft company.

To describe the use of the SSE instructions, a short piece of code is given bellow. For example, when we compute the electric field Ez, we can translate (5) into SSE instructions code as follows:

$$E_z^{n+1}(i,j,k+\frac{1}{2}) = CA\cdot E_z^{n}(i,j,k+\frac{1}{2}) + CB\cdot\begin{bmatrix}[H_y^{n+\frac{1}{2}}(i+\frac{1}{2},j,k+\frac{1}{2}) - H_y^{n+\frac{1}{2}}(i-\frac{1}{2},j,k+\frac{1}{2})] \\ -[H_x^{n+\frac{1}{2}}(i,j+\frac{1}{2},k+\frac{1}{2}) - H_x^{n+\frac{1}{2}}(i,j-\frac{1}{2},k+\frac{1}{2})]\end{bmatrix} \tag{5}$$

```
for ( i=1; i<Nx; i++){
  for ( j=1; j<Ny; j++){
    //The five lines of codes below will convert C language array into SSE data type.
     __m128* sse_ez=(__m128*)ez[i][j];
     __m128* sse_hy=(__m128*)hy[i][j];
     __m128* sse_hy_min=(__m128*)hy[i-1][j];
     __m128* sse_hx=(__m128*)hx[i][j];
     __m128* sse_hx_min=(__m128*)hx[i][j-1];
     for(k=0;k<=Nz;k+=4)        {
        __m128 m1,m2,m3;
        //Previous line defines some variables of __m128 data type for SSE instructions
        m1=_mm_sub_ps(*sse_hy,*sse_hy_min);
        //Previous line operate the computation:
```

$$H_y^{n+\frac{1}{2}}(i+\frac{1}{2},j,k+\frac{1}{2})-H_y^{n+\frac{1}{2}}(i-\frac{1}{2},j,k+\frac{1}{2}) \tag{6}$$

```
        m2=_mm_sub_ps(*sse_hx,*sse_hx_min);
        //Previous line operate the computation:
```

$$H_x^{n+\frac{1}{2}}(i,j+\frac{1}{2},k+\frac{1}{2})-H_x^{n+\frac{1}{2}}(i,j-\frac{1}{2},k+\frac{1}{2}) \tag{7}$$

```
        m3=_mm_sub_ps(m1,m2);
        //Previous line operate the computation:
        (6) – (7)
        m3=_mm_mul_ps(CB,m3);
        //Previous line operate the computation:
        CB * [(6) – (7)]
        m1=_mm_mul_ps(CA,*sse_ez);
        //Previous line operate the computation:
```

$$CA \cdot E_z^n(i,j,k+\frac{1}{2}) \tag{8}$$

```
        m2=_mm_add_ps(m1,m3);
        //Previous line get four Ez value at the same time.
        //The six lines of code below will move the pointer for next computation
        *sse_ez=m2;
        sse_ez++;
        sse_hy++;
        sse_hy_min++;
        sse_hx++;
        sse_hx_min++;
}}}
```

Here (5) is simplified version of (1). The variables N_x, N_y and N_z are the cell number in x, y and z direction respectively, and the words behind symbol "//" are detailed explanation of the codes. The updating codes of the other components of electromagnetic field and the realization of CPML ABC are similar to that one above. The AVX based code is almost similar to the SSE based one except that the data type is '__m256' other than '__m128' and the operators start with a '_mm256_' other than a '_mm_'.

Firstly, we simulate the dipole antenna using the code above. The dipole antenna model is shown as Fig. 8. In the FDTD simulation, the dipole antenna is placed along z-axis in free space, and the computational domain is truncated by CPML ABC [16]. The length of half wavelength dipole is 100 mm. The feed gap is taken to be 1 mm, and hence, the length of each arm is 49.5 mm. The material of dipole arms is selected to be PEC (perfect electric conductor) or copper. The lumped port excitation with 50 Ohms internal resistance is located in the feed gap from the lower to upper arm. The cell size is uniform in three dimension space as (9). The cross sections of arm occupy one cell. The feed point occupies two cells along z-axis. The computational domain is divided into 30×30×224 cells. The time step number is 8000.

$$\Delta x = \Delta y = \Delta z = 0.5mm \tag{9}$$

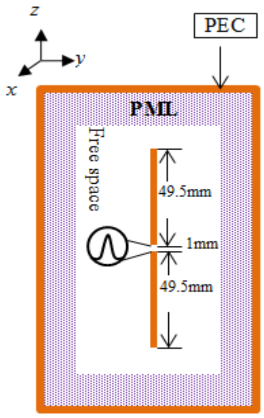

Fig. 8 Dipole antenna model

Secondly, we study the patch antenna. The patch antenna model is shown as Fig. 9 and Fig. 10. In the FDTD simulation, the size of the patch antenna is taken to be 25×0.794×50mm³. The sizes of every part of the patch antenna are marked in Fig. 9 and Fig. 10. The antenna is placed in *x-z* plane in free space for SSE and AVX, and the computational domain is truncated by 6 layers of CPML ABC, too. The feed gap is taken to be one cell in *x-z* plane and three cells along *y* axis. The material of patch and micro strip are selected to be PEC or copper. The voltage source is Gaussian pulse with 50 Ohms internal resistance. The cell size is heterogeneous in three dimension space, as shown in (10). The computational domain is divided into 98×37×160 cells. The time step number is 8000, too.

$$\Delta x = 0.389mm, \Delta y = 0.265mm, \Delta z = 0.4mm \tag{10}$$

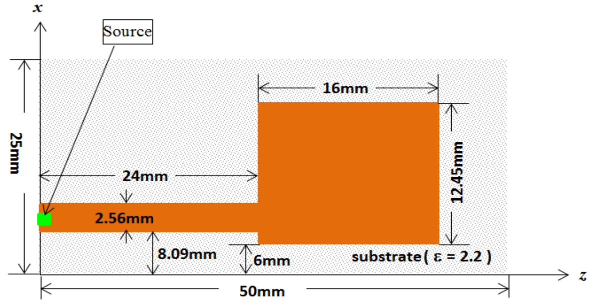

Fig. 9 Top view of rectangle patch antenna model

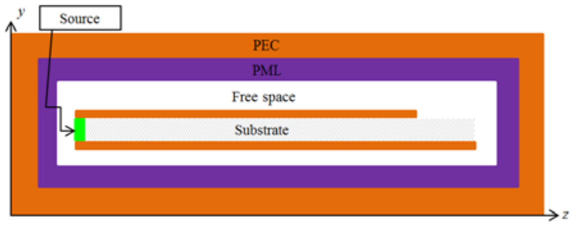

Fig. 10 Side view of rectangle patch antenna model

B. Computing Time of Code Accelerated by SSE and AVX

Table I shows the simulation results. From the table we can observe that the SSE and AVX can accelerate the simulation to some extent. For example, the speed up ratio of SSE to C code is about 3.78 and that of AVX is about 4.97 for dipole antenna simulation, and 3.87 and 5.15 for patch antenna. Here the speed up ratio is the ratio of computing time values for two kinds of codes.

TABLE I COMPUTING TIME OF THE THREE TYPES OF CODE

Number of Cells (x×y×z)		Code based on	Computing Time (Second)
Dipole antenna	30×30×224	C	89.29
		C(Auto-Vectorization)	88.03
		SSE	23.63
		AVX	17.96
Patch antenna	98×37×160	C	262.93
		C(Auto-Vectorization)	251.98
		SSE	68.01
		AVX	51.02

To intuitively evaluate the performance of the SSE and AVX codes, we define the performance as follows:

$$performance(\frac{Mcells}{s}) = \frac{(N_x \times N_y \times N_z) \times Number\_of\_timesteps}{Simulation\_time(Second)}$$

That is to say that the Performance is equal to the cell number computed per second. For example, the performance of C based code is 18 Mcells/sec, as shown in Fig. 11. If we accelerate this code using SSE based code, the performance increases to 68 Mcells/sec. When we use AVX based code, we get a performance of 90 and 91 Mcells/sec for dipole and patch antenna respectively.

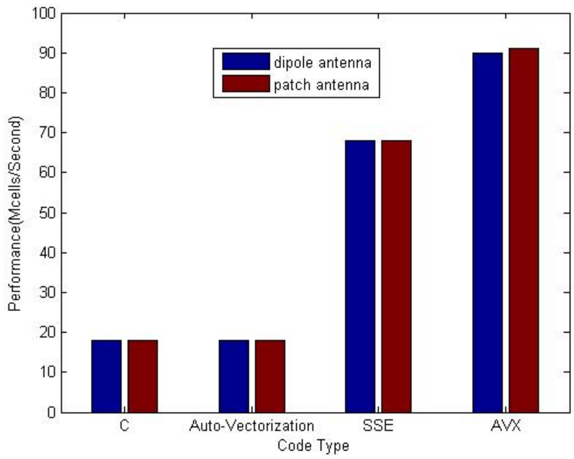

Fig. 11 The performance of the four kinds of codes

Recently, the auto-vectorization flags are available in modern compilers. To compare the degree of improvement achieved by means of the explicit implementation of SSE-AVX to the sequential code with auto-vectorization, the C based code is enhanced by using the auto-vectorization function of visual studio 2012. But the acceleration result of auto-vectorization is not ideal for this code, as shown in Table I and Fig. 11.

Because of the computation task of dipole antenna model does not have too big difference to the patch antenna, so the performance is almost the same to the patch antenna.

By the way, the computing time will linearly increase with the growth of cell number under certain conditions, e.g. the memory need by the computation task is not out of range of your computer. That is to say that the bigger the scale of the problem is, the better the acceleration effect is. Moreover, when one single dimension (especially z-axis) is increased amongst the others, this scheme is more efficient. The reason is that with the growth of the cell number, especially the growth along z-axis direction, the average cache hit rate of the SSE and AVX instructions will increase, so the total computing time will decrease [15].

The impedance of the dipole antenna is shown as Fig. 12. The red line means the real part and the green one means the imaginary part. From the Fig. 12 we can observe that the resonant frequency of this antenna is about 1.4GHz. The return loss of dipole and patch antenna is shown in Fig. 13 and Fig. 14 respectively.

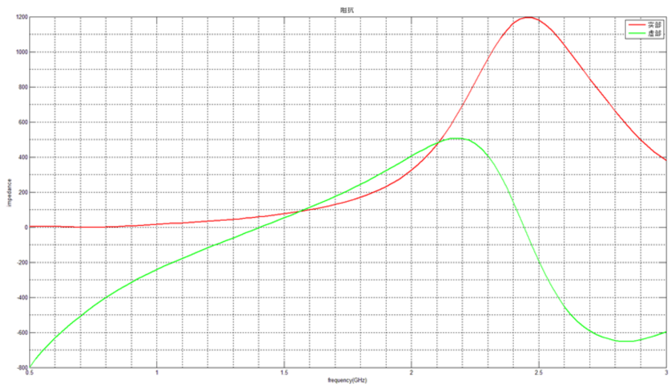

Fig. 12 The impedance of the dipole antenna

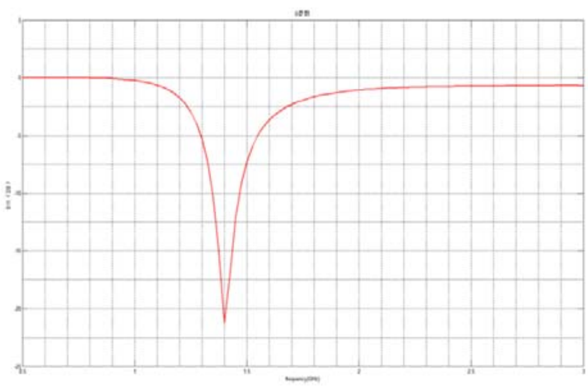

Fig. 13 The return loss of the dipole antenna

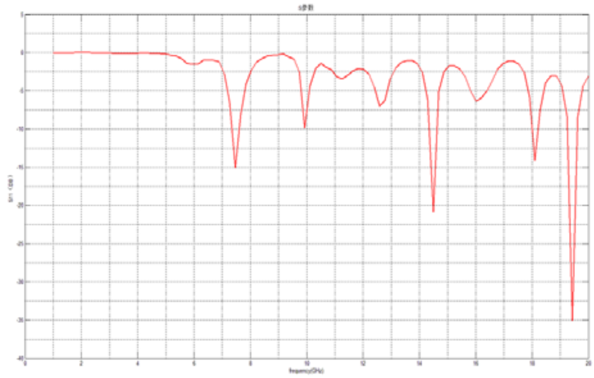

Fig. 14 The return loss of the patch antenna

V. CONCLUSIONS

In this paper, we introduce a novel hardware acceleration technique based on VALU built in a regular CPU for FDTD simulation with CPML absorbing boundary condition, and then, we compare the acceleration effect of auto-vectorization of modern compiler, SSE and AVX instructions sets and give an implementation for dipole and patch antenna model on Intel 2nd Core i5 2320 CPU and Windows 7 Operating System. The result shows that SSE and AVX instructions based code 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 AVX performance and to accelerate the FDTD simulation using AVX in engineering problem. And also we will study how to increase the auto-vectorization acceleration effect.

REFERENCES

[1] K. S. Yee. Numerical solution of initial boundary value problems involving Maxwell’s equations in isotropic media. IEEE Trans. Antennas and Propagat., 1966, 14(3): 302—307.

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

[3] 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.

[4] 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.

[5] Mehmet F. Su, Ihab El-Kady, et al. “A Novel FDTD Application Featuring OpenMP-MPI Hybrid Parallelization”, International Conference on Parallel Processing, vol.1, pp. 373-379, 2004.

[6] Ke-Di Zhang, Wei Shao, and Bing-Zhong Wang. “Parallelized ADI-FDTD Algorithm for Attenuation Constant Extraction by Using OpenMP Library”, International Conference on Microwave and Millimeter Wave Technology, pp. 786-788, 2010.

[7] V. Demir and A. Z. Elsherbeni, “Compute Unified Device Architecture (CUDA) based Finite-Difference Time-Domain (FDTD) implementation,” ACES Journal, vol.25, no. 4, pp. 303-314, Apr. 2010.

[8] 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,” ACES Journal, vol. 25, no. 12, pp. 1061-1069, Dec. 2010.

[9] M. Ujaldon, “Using GPUs for accelerating electromagnetic simulations,” ACES Journal, vol. 25, no. 4, pp. 294-302, Apr. 2010.

[10] M. Weldon, L. Maxwell, D. Cyca, M. Hughes, C. Whelan, and M. Okoniewski, “A practical look at GPU-accelerated FDTD performance,” ACES Journal, vol. 25, no.4, pp. 315-322, Apr. 2010.

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

[12] G. Mur, Absorbing Boundary Conditions for the Finite-Difference Approximation of the Time-Domain Electromagnetic Field Equations, IEEE Transactions on Electromagnetic Comptibility, Vol.23, No.3, 1981, pp.377-382.

[13] http://www.intel.com/design/pentiumii/manuals/245127.htm.

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

[15] L. Zhang, et al., “Enhanced Parallel FDTD Method Using SSE Instruction Sets”, ACES Journal, vol. 27, no. 1, pp. 1-8, Apr. 2012.

[16] Roden J. A., Gedney S. D. Convolutional PML(CPML): An efficient FDTD implementation of CFS-PML for arbitrary media. Microwave Opt. Tech. Lett., 2000, 27(12): 334-339.