David B. Davidson
Dept. E&E Engineering
University of Stellenbosch
Stellenbosch 7600, South Africa
Tel: +27 21 808 4458;
Fax: +27 21 808 4981
E-mail: davidson@sun.ac.za

Foreword by the Editor

Many recent papers on accelerating computations (in particular for the FDTD, which lends itself readily to parallel computation) have used GPGPUs [general-purpose graphics processing units], but contemporary CPUs offer a variety of options for performance acceleration, too. Often, these are somewhat easier to code. This month's contribution provides a detailed investigation of the use of steaming SIMD extensions instructions on x86 architectures. The authors carefully analyze hardware aspects, in particular the impact of cache alignment on performance, and provide interesting results.

As always, the authors are thanked for their contribution.

Performance of Streaming SIMD Extensions Instructions for the FDTD Computation

***Matthew Livesey*[1]*, Fumie Costen*[2]*, and Xiaoling Yang*[3]**

[1]Accenture
Kingsley Hall, 20 Bailey Lane, Manchester Airport,Manchester,M90 4AN, UK
Tel: +44 161 435 5865; E-mail: matthew.livesey@accenture.com

[2]School of Electrical and Electronic Engineering
University of Manchester
Sackville Street Building, Sackville Street, Manchester, M13 9PL, UK
Tel: +44 161 306 4717; Fax: +44 161 306 4644; E-mail: fc@cs.man.ac.uk

[3]2COMU, Inc.
200 Innovation Blvd, State College, PA 16803, USA
Tel: +1 814 4417409; Fax: +1 814 234 1829; E-mail: ybob@2comu.com

Abstract

The utilization of vector-arithmetic logic units is a promising way to speed up FDTD computations from the viewpoint of hardware acceleration. This paper studies how a streaming SIMD extensions (SSE) implementation can be efficiently developed, and the situation where SSE is beneficial for FDTD computations.

Keywords: Finite difference methods; FDTD methods; time domain analysis; hardware; acceleration; high performance computing; parallel programming; parallel architectures; streaming SIMD extensions instructions; SSE

1. Introduction

There are multiple parallel implementations of the Finite-Difference Time-Domain (FDTD) method [1] using the single-program multiple-data (SPMD) paradigm. In the SPMD paradigm, each hardware execution unit has its own instruction fetch unit, and can therefore simultaneously execute a different instruction. Shared-memory thread-based parallel programming (e.g., using *OpenMP*) and message-passing-based parallel programming (e.g., using *MPI*) are both instances of the SPMD paradigm. The progressive increase in CPU clock speed over time has recently ceased, and instead, multi- or many-core CPUs have become the norm. The trend in high-performance computing has thus become the utilization of many-core CPUs. An alternative method of many-core computation has been proposed using general-purpose computation on graphics-processing units (GPGPU). While GPGPU is attracting much attention in computational electromagnetics, this requires new hardware to be purchased.

On the other hand, most researchers engaged in FDTD coding have access to CPUs based on the x86 architecture (those from AMD and Intel). All recent AMD and Intel processors are equipped with parallel single-instruction multiple-data (SIMD) capabilities, offered by a single processor core with streaming single-instruction multiple-data extensions (SSE) instructions. In the single-instruction multiple-data paradigm, multiple hardware execution units are simultaneously issued a single instruction, and synchronously execute this instruction on multiple data items [2]. SSE optimization may thus allow performance improvement without requiring the purchase of any new hardware.

However, little attention is paid to using the vector-arithmetic logic units (VALU) capability provided by SSE, as single-instruction multiple-data is not a frequently used method of acceleration. It is understood that the major top commercial codes have used only the SSE speedups automatically offered through compiler options, without the introduction of vectorization of their code. The compiler does not automatically perform the memory allocation and alignment. The maximum benefit of SSE vector instructions may thus only be gained when hand-written vectorization is introduced into the code.

Reference [3] presented a single-precision implementation of the FDTD method using packed SSE instructions, and reported a speedup of close to four times. In reality, for some material parameters and implementations, calculations in double precision may be needed. This work extends [3] to a double-precision case using the instructions available with the second generation of SSE, and studies the situation when SSE is effective and how SSE should be practically used.

2. The FDTD Method

Maxwell's curl equations for free space without sources yield the discretized equations, such as

$$\begin{aligned} H_y^{n+1}(i,j,k) &= H_y^n(i,j,k) \\ &+ \frac{\Delta t}{\mu \Delta x}\left[E_z^n(i+1,j,k) - E_z^n(i,j,k)\right] \\ &- \frac{\Delta t}{\mu \Delta z}\left[E_x^n(i,j,k+1) - E_x^n(i,j,k)\right], \end{aligned} \quad (1)$$

where the variables and notation are the same as those in [4]. Reference [1] presents the rest of the core equations. As is seen in Equation (1), the computation is spatially localized to the one-cell neighbors. The FDTD method is thus well known to be suitable for parallel computing.

3. SSE Instructions

SSE adds eight 128-bit-wide registers to the x86 architecture [5]. Each register may hold four single-precision, 32-bit floating-point values. SSE also extends the x86 instruction set with instructions that operate on the additional registers. As Figure 1 illustrates, with packed SSE instructions [5, 6] each 32-bit section of each input register is treated as a separate operand. Four instances of the instruction are executed on four sets of data. SSE therefore provides a single-instruction multiple-data capability. A second generation of SSE, commonly known as SSE2 [7], provides 144 new instructions, which include support for double-precision floating-point values. Since double-precision floating-point values are 2^6 bits in length, a single packed instruction can perform two double-precision floating-point operations at once. Subsequent generations of SSE gradually add additional instructions [8]. Since the double-precision instructions added in SSE2 provide the required arithmetic operations for calculating the FDTD equations in double precision, this study focuses on the SSE2 instruction set.

4. The FDTD Implementation with SSE

The most direct method of using SSE instructions in an implementation is to create an assembly-language sequence of the required instructions to produce the desired result. As an

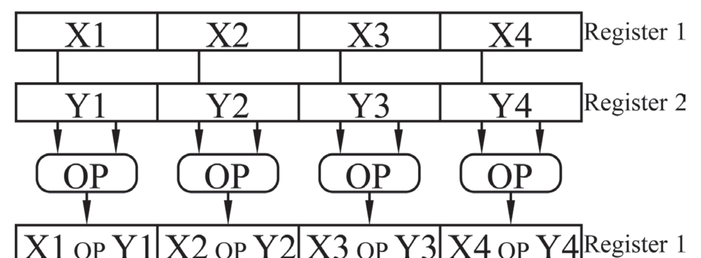


Figure 1. The behavior of a packed SSE instruction [5]. Each rectangle has single precision. X and Y are 32-bit operands, and OP means "operation."

```
//Calculate hy
// ez(i+1,j,k) -ez(i,j,k)
__m128d temp1 = _mm_sub_pd(*dez_shift_i, *dez);
divide by dx
temp1 = _mm_div_pd(temp1, ddx);
//ex(i,j,k+1) - ex(i,j,k)
__m128d temp2= _mm_sub_pd(*dex_shift_k, *dex);
//divide by dz
temp2 = _mm_div_pd(temp2, ddz);
// subtract the two clauses
temp2 = _mm_sub_pd(temp2, temp1);
// multiply by dt/pma
temp2 = _mm_mul_pd(temp2, ddtbpma);
// add calculation to hy(i,j,k) and store
*dhy = _mm_add_pd(*dhy, temp2)
```

Figure 2. The implementation of SSE using intrinsic functions.

alternative, there is a library of functions available allowing packed SSE instructions to be expressed entirely in *C*. These are known as intrinsic functions. This report focuses on intrinsic functions as the method for implementing the FDTD method using SSE, because it was found that manually coded assembly-language sequences could not easily be ported between machines with different variations of the x86 architecture.

The intrinsic library provides the data type `_m128d`, which represents a pair of 64-bit double-precision values stored in a 128-bit register. This greatly simplifies loading and storing to the SSE processor registers from the matrices representing the FDTD problem space in memory. A pointer of type `_m128d` pointing to the address of $E_x(2,1,1)$ covers the data at this location and $E_x(3,1,1)$. A single increment of this pointer changes its target to the 128-bit location of $E_x(4,1,1)$ and $E_x(5,1,1)$, stepping over two elements. The code sample in Figure 2 shows how Equation (1) is represented using intrinsic functions. Since the `_m128d` data type holds two double-precision values, this code simultaneously performs the equation for two elements of H_y.

As with many scientific computations, the original FDTD program was written in *FORTRAN* rather than *C*. It was desirable to avoid a wholesale rewrite of the existing implementation. It was necessary to use *C* code to produce the intrinsic sequences, but procedures written in *C* could be called from a *FORTRAN* program by following a particular naming convention and using the *GCC* compiler to link together the compilation units. The compiler option for integration of *FORTRAN* and *C* code was `-O2 -msse2 -mfpmath=sse`. Prior to this integration, `-c -std=c99` was used for compiling *C* code. Porting the standard code to the SSE version was not trivial, mainly because of lack of documentation on it. Particular attention was required for the pointer arithmetic at the end of the triply nested loops.

5. Numerical Experiments

5.1 Computational Environment

The experiments were performed on four machines. Table 1 gives the specifications of each machine. Each had a varying number of available cores, but only the single-instruction multiple-data hardware available on a single core was used in these experiments. The number of cores was therefore not expected to impact the results. Figure 3 shows the cache organization of each processor presented in Table 1.

Figure 4 generalizes the correspondence of the memory address to the level-1 cache-line address when a computer had an *N*-byte level-1 cache, and there was an *M*-way set associated in the level-1 cache. Each matrix was assigned to a 2^{12} B (2^{12} byte) boundary in memory. Each level-1 cache-line could hold a contiguous 2^6 B block from main memory. The level-1 cache-line address into which each 2^6 B block in memory was copied was a whole number of

$$\left\lfloor \left(\text{memory address modulo } \frac{N}{M} \right) \Big/ 2^6 \right\rfloor .$$

Since each set in level-1 cache had identical cache-line addresses, there were *M* locations for each cache-line address and *M* competing blocks could reside in cache together, one in each set. When the $(M+1)$ th competing block was loaded, one of the existing blocks had to be evicted from its cache-line to make room for the new block.

5.2 Expectation

Figure 4 with $(M,N)=\left(2,2^{16}\right)$ and $\left(8,2^{15}\right)$ represents the AMD processor and the Intel processor, respectively. With the AMD processors, if memory addresses of the beginning of each of the six matrices E_x, E_y, E_z, H_x, H_y, and H_z happened to be congruent modulo 2^{15}, the matrices could be said to be aligned with respect to the level-1 data-cache. Since all six matrices were identical in size and structure, any two elements with the same indices but from different matrices would compete for the same position in cache when the matrices were aligned with respect to the level-1 data-cache. Figure 5 shows how the cache-line eviction could occur in the level-1 cache of the AMD processors when executing a calculation for $H_y(i,j,k)$. The memory addresses of $E_x(i,j,k)$ and $E_z(i,j,k)$ resolved to the first cache-line, and therefore each was loaded into one of the two sets available. When the value of $H_y(i,j,k)$ was subsequently required to complete the cal-

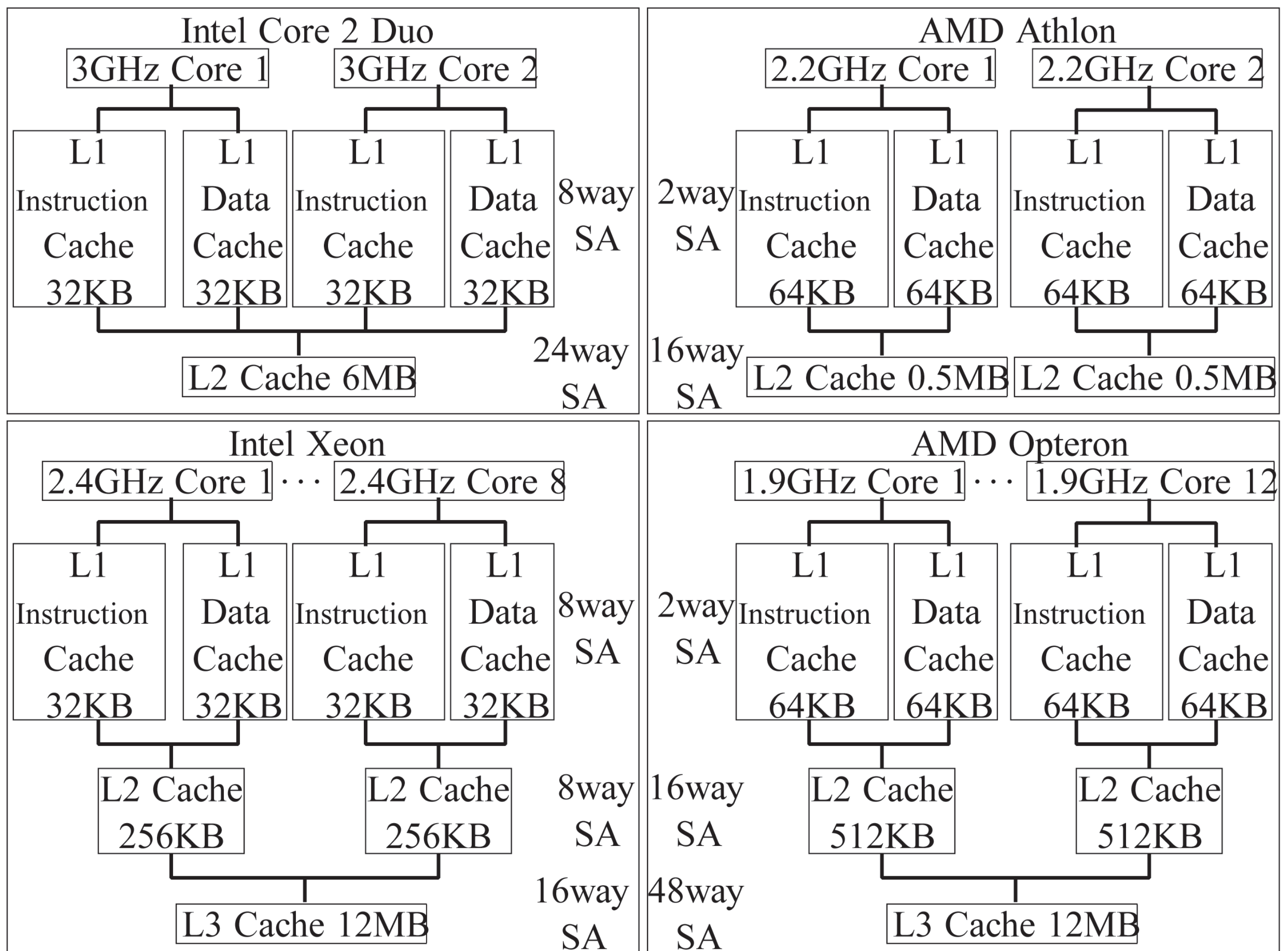


Figure 3. The cache organization of the processors in Table 1. SA stands for "set associative."

Table 1. The details of the architectures used for the numerical experiments. The cache details are in Figure 3. SL stands for "*Scientific Linux*."

	AMD Athlon Dual-Core 4200+	**AMD Opteron 6168**	**Intel Core2Duo E8400**	**Intel Xeon E5620**
Type	64 Bit	64 Bit	32 Bit	64 Bit
Operating System Kernel	OpenSuse 10.2 2.6.18.8	CentOS5.5 2.6.18	Fedora11 2.6.30.10	SL 5.5 2.6.18
Core number	2	48	2	8
Core speed	2.2 GHz	1.9 GHz	3 GHz	2.4 GHz
GCC version	4.3.0	4.1.2	4.4.1	4.1.2
Level of SSE support	SSE2	SSE4a	SSE4.1	SSE4.2

N-Byte Level 1 cache with M way set associative

Set 1
64 Bytes
0 7 8 16 24 39 47 55 63 line 1
64 line 2
$\frac{N}{64 \cdot M}$

Set 2
64 Bytes
0 7 8 16 24 39 47 55 63 line 1
64 line 2
$\frac{N}{64 \cdot M}$

Set M

main memory

64 Bytes for line 1
64 Bytes for line 2
64 Bytes for line 64
64 Bytes for line 65
64 Bytes for line 66
64 Bytes for line 128
for line $\frac{N}{64M}$ 192
for line $\frac{N}{64M}$ 191
for line $\frac{N}{64M}$ 129
64 Bytes for line $\frac{N}{64M}$ 128
64 Bytes for line $\frac{N}{64M}$ 64
64 Bytes for line $\frac{N}{64M}$ 63
64 Bytes for line $\frac{N}{64M}$ 1
64 Bytes for line $\frac{N}{64M}$
64 Bytes for line 1
64 Bytes for line 63
for line $\frac{N}{64M}$ 257
for line $\frac{N}{64M}$ 256
for line $\frac{N}{64M}$ 194
64 Bytes for line $\frac{N}{64M}$ 193
H_x
H_y

Figure 4. The relationships between the addresses in main memory and the *N*-byte Level-1 cache with *M*-way set associative.

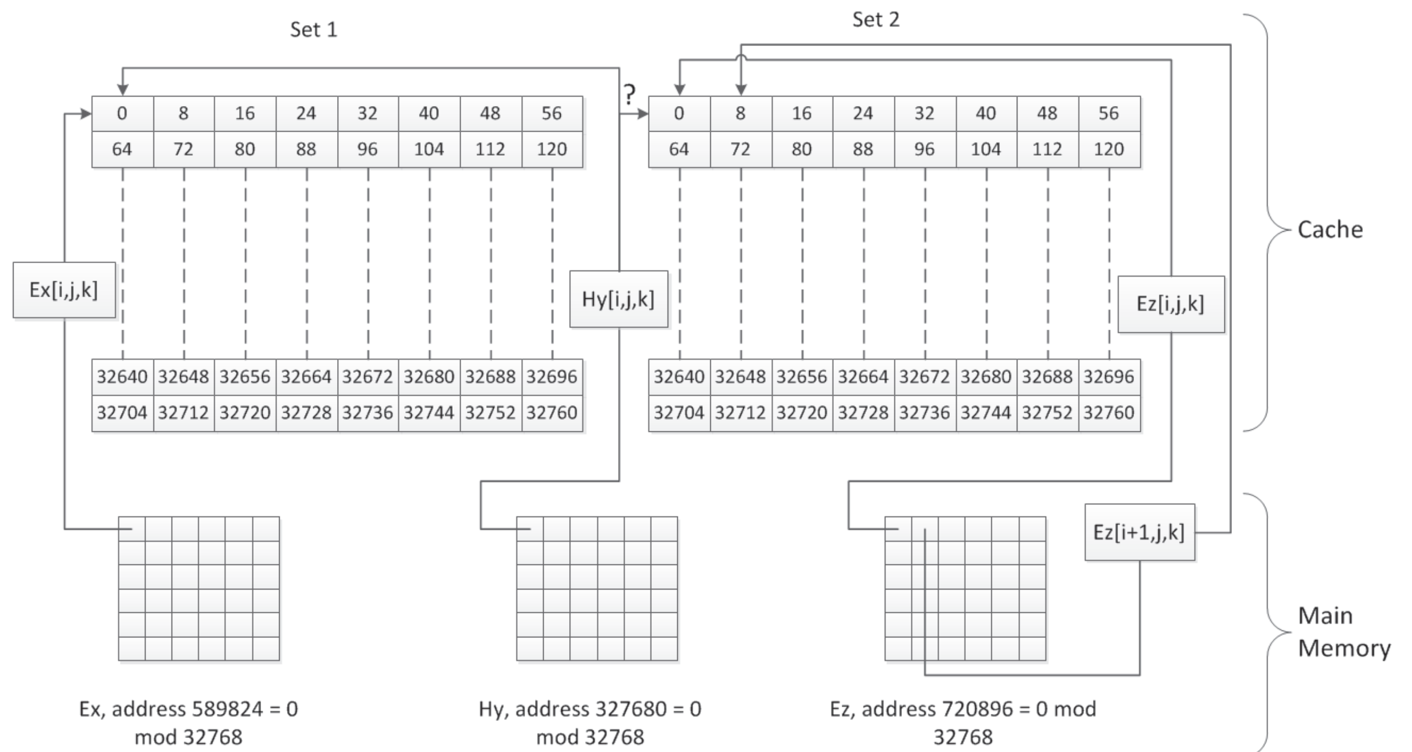


Figure 5. The cache behavior of the AMD Athlon processor.

Figure 6. The performance of FDTD implementations on AMD architectures: (a) AMD Athlon; (b) AMD Opteron.

Figure 7. The performance of FDTD implementations on AMD architectures, following the memory-alignment fix: (a) AMD Athlon; (b) AMD Opteron.

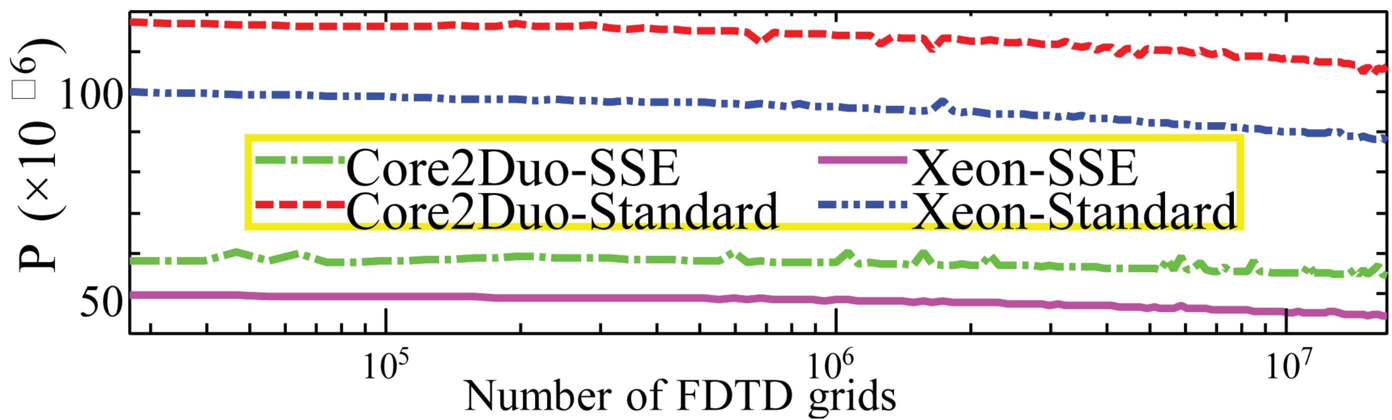


Figure 8. The performance of FDTD implementations on Intel architectures.

culation, it had to be placed in the first cache-line of one of the two sets. This led to either the values of $E_x(i,j,k)$ to $E_x(i+7,j,k)$ or $E_z(i,j,k)$ to $E_z(i+7,j,k)$ being evicted, even though these values would be required in several calculations immediately following the calculation of $H_y(i,j,k)$. The resulting cache misses led to main memory accesses. A performance deterioration was thus expected, depending on the FDTD grid size, on AMD architectures.

This memory alignment should not have been an issue on the Intel machines for our simple FDTD algorithm. Since the level-1 cache was eight-way set associative, each cache-line address corresponded to eight slots in the cache. Since there were only six matrices in this simple FDTD algorithm, the 2^6 B cache-lines containing the elements indexed by a particular `(i,j,k)` index from each matrix could all simultaneously reside in cache, using six of the available eight positions for that cache-line address.

5.3 Results on AMD Architectures

5.3.1 Performance Measurement

The standard *FORTRAN* implementation and the SSE implementation were each executed with an FDTD grid number of $(2l)^3$, where l is an integer from eight to 128. The execution was run for 1000 FDTD time steps, and each result was taken four times to produce an average. The timing result measured was the overall execution time of the 1000 time steps without initialization and data output. The performance, P, is defined as the execution time in seconds for 1000 time steps divided by the number of FDTD grids, $(2l)^3$. A smaller P thus represents better performance.

5.3.2 Intermittent Performance Degradation

Figure 6 shows the performance results from the AMD machines. The 倖 and △ symbols are the execution time with the standard *FORTRAN* implementation and with the SSE implementation, respectively. As expected, both the standard *FORTRAN* implementation and the SSE implementation showed frequent and significant drops in performance at various problem sizes on both AMD machines. Since the drops in performance happened at particular problem sizes, but the performance returned to normal for subsequent problem sizes, this was not related to the scalability of the algorithm. Calculating the matrix size but adjusting for memory alignment to the next 2^{12} B boundary, we found that those dimensions that led to a dramatic loss in throughput perfectly correlated with those dimensions calculated to be aligned on a 2^{15} B boundary. This proved our expectation that alignment in memory – leading to excessive eviction of cache-lines – was the cause of the drop in performance.

5.3.3 Poor Performance on AMD Athlon

The AMD Athlon is based on the K8 micro-architecture, which features a 64-bit data path. This means that 128-bit SSE instructions must be split in two, effectively removing all concurrency from double-precision execution. In contrast, the AMD Opteron is based on the K10 micro-architecture, and both Intels are based on the Core micro-architecture, both of which feature a 128-bit data path. This difference is the most likely cause of the lack of performance on the AMD Athlon. All processors that are SSE capable but have a 64-bit data path are likely to suffer from a similar lack of speedup when executing double-precision SSE vector instructions.

5.4 Remedy for Localized Performance Deterioration

One solution to the memory-alignment problem is to introduce memory-allocation logic into the code, in order to increase the space between each matrix and ensure that alignment does not occur. Having calculated that a particular problem size would cause alignment and required adjustment, a simple solution was to add redundant elements to the beginning of each matrix. The extra space was given negative indices and

therefore was ignored by the code, which traversed the matrices using positive indices. Figure 7 shows the performance results for the standard and SSE implementations on the AMD machines after the memory alignment fix was in place. Compared with Figure 6, there were many fewer instances of loss of throughput in Figure 7. Where the drop in performance still occurred, this was caused by problem sizes where adding an extra layer altered the size of the matrices by an exact 2^{15} B amount. A more-practical solution may be to be aware of the FDTD grid sizes that cause cache contention due to alignment on a particular machine, and to avoid those sizes.

5.5 Results on Intel Architectures

Figure 8 shows the performance results on the Intel Xeon and Intel Core 2 Duo machines. On both Intel machines, the differences between the performance of the standard *FORTRAN* implementation and the SSE implementation represented a speedup of around two, the ideal speedup achievable when using the packed SSE instructions to perform two arithmetic operations at once. As expected, the Intel machines did not exhibit the same pattern of performance drops seen with the AMD machines.

6. Conclusion

This paper investigated the application of SSE to double-precision computation and its efficiency in commonly available computer architectures. It was found that using intrinsic functions, an SSE implementation could accelerate the double-precision computation on machines that featured a 128-bit data path. When a level-1 cache was M-way set associate and computational equations required more than M matrices with an identical size, we needed to consider memory alignment leading to cache collisions in order not to suffer sporadic dips in performance. This paper proposed and demonstrated one of the ways to avoid this when $M = 2$. An alternative approach could be an application of memory interleaving to reduce the cache misses. Introduction of boundary conditions, Huygens excitation, and more complex situations such as handling frequency-dependent material will produce more computation locally and in the entire FDTD space. These computations can also use SSE instructions. While we expect similar speedups can be achieved if the issues of memory allocation and code structure are observed, performance will vary on a case-by-case basis.

7. References

1. A. Taflove and S. Hagness, *Computational Electrodynamics: The Finite-Difference Time-Domain Method, Third Edition*, Norwood, MA, Artech House Publishers, 2005.

2. D. A. Patterson and J. L. Hennessy, *Computer Organization and Design, Third Edition: The Hardware/Software Interface*, Burlington, MA, Morgan Kaufmann, 2004.

3. W. Yu, X. Yang, Y. Liu, R. Mittra, D. Chang, C. Liao, A. Muto, W. Li, and L. Zhao, ``New Development of Parallel Conformal FDTD Method in Computational Electromagnetic Engineering," *IEEE Antennas and Propagation Magazine*, **53**, 3, June 2011, pp. 15-41.

4. F. Costen, J.-P. Berenger, and A. Brown, "Comparison of FDTD Hard Source with FDTD Soft Source and Accuracy Assessment in Debye Media," *IEEE Transactions on Antennas and Propagation*, **AP-57**, 7, July 2009, pp. 2014-2022.

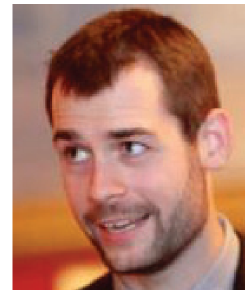
5. S. Thakkur and T. Huff, ``Internet Streaming SIMD Extensions," *Computer*, **32**, 12, December 1999, pp. 26-34.

6. J. Huynh, "The AMD Athlon MP Processor with 512KB L2 Cache," Technical Report, AMD, May 2003.

7. G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel, "The Microarchitecture of the Pentium 4 Processor," *Intel Technology Journal*, **5**, 2001.

8. Intel Corporation, "Intel 64 and IA-32 Architectures Software Developer's Manual Volume 1: Basic Architecture," March 2012.

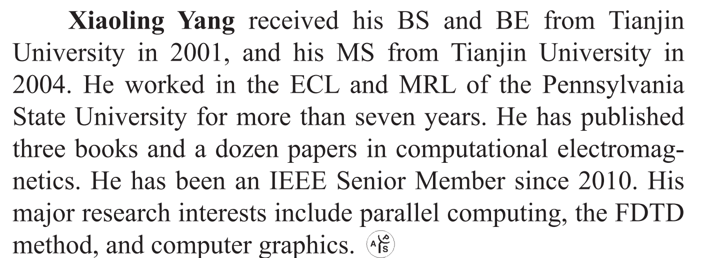
Introducing the Authors



Matthew Livesey graduated from the University of Manchester in 2005 with a BEng Software Engineering (First Class Honors). He subsequently joined Accenture, working as an IT

consultant. In 2011, he returned to the University of Manchester and received an MSc with distinction in Computer Science, and was awarded the Peter Jones prize as the highest achiever in the year. Matthew continues to work at Accenture as an IT Project Manager and Solution Architect. Matthew is interested in parallel and distributed systems, and is currently working with Big Data in Hadoop.

Fumie Costen received the BSc and MSc in Electrical Engineering, and the PhD in Informatics, all from Kyoto University, Japan. From 1993 to 1997, she was with Advanced Telecommunication Research International, Kyoto, where she was engaged in research on direction-of-arrival estimation based on the Multiple SIgnal Classification (MUSIC) algorithm for three-dimensional laser microvision. She received an academic invitation from Kiruna Division, Swedish Institute of Space Physics, Sweden, in 1996, and received three patents from the research in 1999. From 1998 to 2000, she was with Manchester Computing in the University of Manchester, UK, where she was engaged in research on metacomputing and has been a Lecturer since 2000. Her research interests include computational electromagnetics for such topics as a variety of the Finite-Difference Time-Domain methods for the microwave frequency range, and high spatial resolution and FDTD subgridding and boundary conditions. Her work extends to hardware acceleration of the computations using general-purpose computing on graphics processing units, streaming single-instruction multiple-data-extension (SSE) and advanced-vector-extensions instructions. Dr. Costen received an ATR Excellence in Research Award in 1996, and a best paper award from the 8th International Conference on High Performance Computing and Networking Europe in 2000.



Xiaoling Yang received his BS and BE from Tianjin University in 2001, and his MS from Tianjin University in 2004. He worked in the ECL and MRL of the Pennsylvania State University for more than seven years. He has published three books and a dozen papers in computational electromagnetics. He has been an IEEE Senior Member since 2010. His major research interests include parallel computing, the FDTD method, and computer graphics.