Microprocessors and Microsystems 26 (2002) 39–44
MICROPROCESSORS AND MICROSYSTEMS
www.elsevier.com/locate/micpro

A parallel solution to linear systems

Y.F. Fung[a], M.F. Ercan[b,*], T.K. Ho[a], W.L. Cheung[a]

[a]Department of Electrical Engineering, The Hong Kong Polytechnic University, Hong Kong SAR, People's Republic of China
[b]Department of Electrical Engineering, Singapore Polytechnic, 500 Dover Road, Singapore, Singapore 139651

Received 26 March 2001; revised 4 November 2001; accepted 13 November 2001

Abstract

Streaming SIMD Extensions (SSE) is a unique feature embedded in the Pentium III and IV classes of microprocessors. By fully exploiting SSE, parallel algorithms can be implemented on a standard personal computer and a theoretical speedup of four can be achieved. In this paper, we demonstrate the implementation of a parallel LU matrix decomposition algorithm for solving linear systems with SSE and discuss advantages and disadvantages of this approach based on our experimental study.

Keywords: Parallel algorithms; Instruction level parallelism; LU decomposition

1. Introduction

Personal Computers (PC) or workstations are currently the most popular computing systems for solving various engineering problems due to their low cost. With the advances made in integrated circuit technology, the computing power of microprocessors is steadily increasing. Currently, processors with a working frequency of higher than 1 GHz are available. The computing performance of a microprocessor is mainly dictated by two factors, the operating frequency (or clock rate), and the internal architecture. The computing performance that can be achieved by increasing clock speed is reaching to physical limits thus making the architectural solutions more prominent. An important architectural feature added to recent microprocessors is a set of instructions that can exploit Single Instruction Multiple Data (SIMD) type of parallelism. A notable example is Intel's MMX technology and its latest Streaming SIMD Extensions (SSE) included in the Intel Pentium III class of microprocessors [6]. The SSE feature enables the execution of SIMD operations inside the processor and therefore, the overall performance of an algorithm can be improved significantly. Similar SIMD register extensions can also be found in other processors such as VIS in SunSPARC [14] processor and AltiVec in Motorola PowerPC [1].

The technologies mentioned earlier, embodied to recent processors, are mainly targeted multi-media and graphics applications, where they provide a significant speed up. The performance of Intel MMX and SSE technology in common digital signal processing and multi-media algorithms have been studied extensively in the literature (see for instance, Refs. [2,7,13]). Nevertheless, our intension in this paper is to exploit this technology in another significant engineering application, which is the solution of the linear system equations.

The solution methods of systems of linear equations have been widely studied in the research community. Gaussian Elimination, Gauss Jordan, LU decomposition are among the classical techniques [5,10]. The matrix size involved in many practical applications is quite large and this results in high computation complexity. Therefore, a major study has been devoted to parallel algorithms in the literature. Many of these parallel algorithms are basically developed considering large SIMD machines such as Connection machine and MasPar MP-2 [3] or MIMD architectures such as transputers [15] and Intel iPSC [11]. A recent survey by Duff and van der Vorst [4] gives an extensive coverage on the parallel algorithms and the latest trends in the area. Most of those hardware platforms are expensive and may not be available to most researchers. On the other hand, the cost of a PC is low and therefore, an improved solution to the linear systems solutions by exploiting SIMD extension will be beneficial to research in this area.

In this paper, we will consider parallel LU decomposition algorithm since it is a widely used and fundamental algorithm. The implementation of parallel LU decomposition

* Corresponding author. Tel.: +65-772-1503; fax: +65-772-1972.
E-mail addresses: eeyffung@polyu.edu.hk (Y.F. Fung), mfercan@sp.edu.sg (M.F. Ercan), eetkho@polyu.edu.hk (T.K. Ho), eewlcheung@polyu.edu.hk (W.L. Cheung).

0141-9331/02/$ - see front matter
PII: S0141-9331(01)00144-2

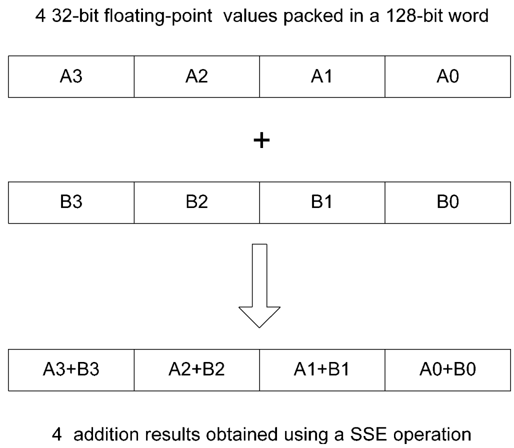


Fig. 1. Parallelism based on SSE operation.

using SIMD extension will be described in the following sections. There are C language intrinsics provided to ease programming though it is still users' role to explore available parallelism. We outline the methodology and trade offs for obtaining substantial speedups over serial code.

2. SSE registers and data storage

The SSE mechanism can be considered as an extension of the earlier MMX technology implemented in the Intel Pentium processors [6]. The major difference between SSE and MMX is in the data type that can be operated upon in parallel. In MMX, special MMX registers are provided to hold different types of data, however, it is limited to character, or integer values. On the other hand, the SSE registers are 128-bit wide and they can store floating-point values, as well as integers. There are eight SSE registers, each of which can be directly addressed using the register names. Utilization of the registers is straightforward with a suitable programming tool. In the case of integers, eight 16-bit integers can be stored and processed in parallel. Similarly, four 32-bit floating-point values can be manipulated. Therefore, when two vectors of four floating-point values have been loaded into two SSE registers, as shown in Fig. 1, SIMD operations, such as add, multiply, etc. can be applied to the two vectors in one single operation step [2,9]. Applications relying heavily on floating-point operations, such as 3D geometry, and video processing can be substantially accelerated [16]. Moreover, the support of floating-point values in the SSE operations has tremendously widened its applications in other problems.

2.1. Programming with SSE

Programming with the SSE can be achieved by two different approaches. The SSE operations can be invoked by assembly codes, included in a standard C/C++ programs. In the following, sample codes showing how to evaluate the value ($1/x$) using assembly codes are given:

```
float x, frcp;

__asm {movss xmm1, DWORD PTR x
  movss xmm2, xmm1
  rcpss xmm1, xmm1
  movss xmm3, xmm1
  mulss xmm1, xmm1
  mulss xmm2, xmm1
  addss xmm3, xmm3
  subss xmm3, xmm2
  movss DWORD PTR frcp, xmm3}
```

Alternatively, by utilizing the special data type we can develop a C/C++ program without any assembly coding. The new data type designed for the manipulation of the SSE operation is `F32vec4` [9]. It represents a 128-bit storage, which can be applied to store four 32-bit floating-point data. Similarly, there is also the type `F32vec8`, which is used to store eight 16-bit values. These data types are defined as C++ classes and they can be applied in a C/C++ program directly.

In addition to the new data types, operations are derived to load traditional data, such as floating-point, into the new data structure. As an example, to load (or pack) four floating-point values into a `F32vec4`, the function `_mm_load_ps` can be applied. When using `_mm_load_ps`, it is assumed that the original data is 16-byte aligned (16-byte aligned implies that the memory address of the data is a factor of 16) otherwise the function _mm_loadu_ps should be used instead. Once data stored into the 128-bit data structure, functions that can manipulate the `F32vec4` type data can be called. This will result in parallel processing in two sets of four floating-point values. Source codes demonstrating how to add elements stored in two arrays using the SSE features are depicted as following:

```
Float array1 [4];
Float array2 [4];
Float result [4];
F32vec4 A1, A2, A3;
A1 = _mm_load_ps(array1);
A2 = _mm_load_ps(array2);
A3 = A1 + A2;
_mm_store_ps (result, A3);
```

the variable A1 and A2 can be manipulated just like any standard data type. The function `_mm_store_ps` is used to convert (or unpack) the data from the `F32vec4` type back to floating-points and stored in an array.

3. Parallel LU decomposition on SSE

The solution for a set of linear equations is in the form of:

$$Ax = b \tag{1}$$

where A is a matrix of order n, b is a given independent vector and x is an unknown solution vector, is one of the essential problems in computing science, as well as in engineering. As discussed earlier, this problem is computationally intensive. In many engineering applications, matrix A is large and solution to Eq. (1) needed to be determined in a short time-interval (see for instance Ref. [16], where real-time power system simulation needs a solution in 10 ms).

A common procedure [9] for solving Eq. (1) is to factor A into lower and upper triangular matrices L and U such that

$$LUx = b \tag{2}$$

and this is then followed by forward/backward substitution of the form

$$Lx' = b \tag{3}$$

and

$$Ux = x' \tag{4}$$

Forward substitution first identifies the intermediate results x' and vector x is determined by backward substitution.

The calculation involved in LU decomposition can be explained by the following pseudo code and the equation:

For $k = 0$ *to* $n - 2$ (5)

Do

For $i = k + 1$ *to* $n - 1$

Do

For $j = k + 1$ *to* $n - 1$

$$a_{i,j} = a_{i,j} - \frac{a_{i,k} \times a_{k,j}}{a_{k,k}}$$

In the above equation, $a_{i,j}$ represents elements in the matrix A.

According to Eq. (5), elements in the matrix A are being processed along the diagonal and on a row-by-row basis. Data, stored in a row of the matrix, map naturally into the `F32vec4` data structure and therefore, four elements in a row can be evaluated in one single step. Eq. (5), the term $a_{i,k}/a_{k,k}$ is a constant when elements in row i are being processed. It can be, therefore, stored in a `F32vec4` value with the command `_mm_load_ps1`. The command loads a single 32-bit floating-point value, copying it into all four words. The following pseudo codes illustrate the steps performed in order to implement Eq. (5) using SSE functions:

```
F32vec C, A1, A2; /* 128-bit values */
Float x;
For (k = 0; k < n − 2; k++)
For  (i = k + 1;  i < n − 1;  i++)  {x = a_{i,k}/a_{k,k};
_mm_load_ps1 (C, x);
for (j = k + 1; j < n − 1; j + = 4){
pack four values from a(k,j) to a(k,j + 3)
into A1;
pack four value from a(i,j) to a(i,j + 3)
into A2;
A2 = A2 − (A1*C); Unpack the results from
A2 and store in output matrix
}}
```

In forward substitution, the operations can be represented by:

$$x'_i = b_i - \sum_{j=1}^{i-1} x'_j \cdot L_{i,j} \tag{6}$$

where $[x'_i]$ represents elements in the $[x']$ matrix as shown in Eq. (4); b_i represents elements in the $[b]$ matrix and $L_{i,j}$ represents elements in the $[L]$ matrix. SSE operations are applied in the operation $x'_j \cdot L_{i,j}$. Four elements of x'_j and $L_{i,j}$ can be stored in two different `F32vec4` data and multiplied at a single operation.

In backward substitution, the operations are represented by

$$x_j = \frac{x'_j - \sum_{n=j+1}^{m} x_n \cdot U_{j,n}}{U_{j,j}} \tag{7}$$

where $U_{i,j}$ are the elements in the upper matrix $[U]$; m is the size of the vector $[x]$. Similar to forward substitution, the multiplication of $x_n \cdot U_{j,n}$ can be executed by SSE functions with four elements of x_n and $U_{j,n}$ being operated on at the same instead.

4. Experimental results

In this section, results obtained for solving linear system equation $Ax = b$ using the earlier described parallel LU decomposition algorithm are given. We have randomly

Table 1
Processing time (in ms) for various matrix sizes

	Size of matrix A				
	100	200	300	400	500
Sequential	6.22	43.26	189	648	1205
SSE	4	35.86	175.1	613.9	1161
SSE with alignment	3.4	29.84	130	442	830

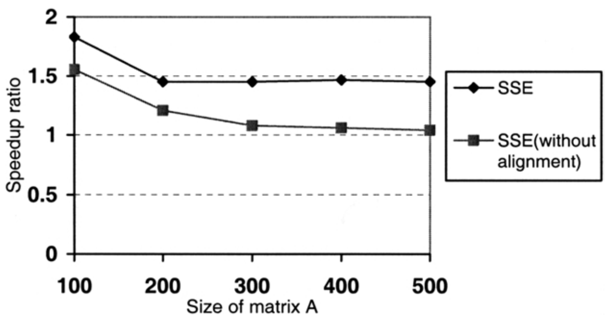


Fig. 2. Speedup ratio for three different approaches.

generated the matrix A with different sizes. In Table 1 processing time obtained for different dimensions of the matrix A are given. Three different cases are compared, namely, (1) sequential approach that is without using SSE, (2) solution obtained by SSE, (3) solution obtained by SSE (but without the 16-bit alignment.) The speedup ratios, illustrated in Fig. 2, are calculated by taking the processing time of the sequential algorithm as reference:

$$\text{Speedup} = \frac{\text{CPU time of sequential algorithm}}{\text{CPU time of parallel algorithm}} \quad (8)$$

Referring to Fig. 2, the speedup ratio obtained in the case of using SSE is better than non-aligned case. For a better performance, data should be in 16-byte aligned form as described in Section 2.1. In order to maintain the 16-byte alignment, the size of the matrix must be a multiple of 4. If this is not the case, then extra rows and columns must be added into the matrix to get around the alignment problem.

The best result is obtained for a relatively smaller matrix of 100×100, where the speedup rate is about 1.8. The performance of the SSE algorithms is affected by the overhead due to additional steps required to convert data from standard floating-point values to 128-bit `F32vec4` data and vice-versa. Referring to pseudo code given in Section 3, three packing/unpacking functions are carried out when elements in a row are being processed. Assuming that it takes the same time (t_p) to execute a packing, or unpacking function, then the overhead will be $3t_p$. And the time required to operate on four elements becomes $3t_p + t_m$, where t_m is the processing time for one multiplication, one subtraction, and one store operation with SSE. If we define $(3t_p + t_m)$ as t_{SSE}, then the total processing time becomes $t_{SSE} \times$ (total number of operations). In the case of SSE, the number of operations can be appropriated by

$$\frac{1}{4}\sum_{n=2}^{N-1}(n-1)(n) \quad (9)$$

where N is the size of the vector x given in Eq. (1).

In the case of traditional algorithm, the total number of operations is

$$\sum_{n=2}^{N-1}(n-1)^2 \quad (10)$$

and the processing time per operation is t'_m, which is the time taken to perform the multiplication, subtraction and store with the standard programming approach. And we can assume that the processing time for t_m and t'_m are the same.

Eq. (9) does not include processing in the forward and backward substitution. The forward and backward substitution only account for a very small portion (about 1%) of the total processing time, therefore it is neglected in the current model. According to Eqs. (9) and (10), the speedup ratios obtained for different sizes of the matrix A can be approximated by a constant, provided that the values of $3t_p$ and t_m are known. The values of $3t_p$ and t_m are being determined empirically and our result indicates that $3^* t_p \approx t_m$. The speedup ratios obtained by our model is close to our experimental results and therefore, we can gauge the performance of the SSE algorithm with different sizes of matrix A based on our model.

4.1. Experiments with complex systems of equations

The LU decomposition algorithm studied earlier assumed that linear system equations $Ax = b$ are real. However, in many engineering problems matrix A and vector b could be complex. The implementation of the LU decomposition algorithm for complex numbers is similar to the real-number algorithm described earlier. This time computations have to be conducted with complex arithmetic instead. In the complex algorithm, two SSE registers are used to store four complex numbers with one register to hold the real part and the other to store the imaginary part. With such an arrangement, a complex multiply of four complex numbers is performed using four multiply instructions, an addition, and a subtraction. Comparing to the real-number algorithm, the computation requirements are higher in the complex number algorithm, which makes the packing, and unpacking overhead relatively less effective. This is reflected by the speedup obtained (Fig. 3). In Table 2 results for various matrix sizes are presented.

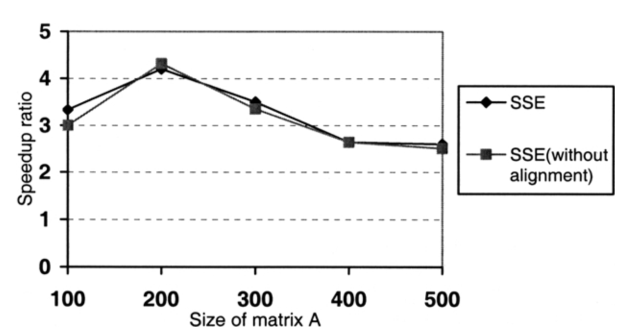


Fig. 3. Speedup ratio for various sized complex matrices.

Table 2
Processing time (in ms) for various sized complex matrices

	Size of matrix *A*				
	100	200	300	400	500
Sequential	30	285.5	1042	2611	5124.1
SSE	10	66.1	311.2	988.3	2034.9
SSE with alignment	9	68	297.7	988.4	1968.8

4.2. A comparison of performance

We have also implemented the LU decomposition algorithm on a multiple-CPU shared memory machine (the SunSparc UE10000) and the programming method is based on threads [12]. The speedup is obtained by utilizing four CPUs in the system. The speedup produced by the SMP system is slightly better than a factor of two. In Table 3 and Fig. 4, we have included results obtained by parallel machine and by SSE. Apparently, both execution time and speedup ratio are much better in the multi-CPU machine. However, for a realistic comparison we should also consider the cost of both systems. Multi-CPU machine is at least 20 times more expensive than the workstation used in our tests (a PC with a 550 MHz Pentium III microprocessor). In the case of SSE, there is no extra hardware cost or upgrading effort required that makes SSE mechanism a very cost-effective way of improving performance. In addition, better performance figures were reported in [8] for LU decomposition, where SSE portions of the code were developed in assembly language.

5. Conclusions

In this paper, we have presented the basic operations involved in utilizing the SSE features of the Pentium III processors to speed up LU decomposition algorithm that is commonly used in solution of linear system equations. In order to examine the effectiveness of SSE, a set of experiments was conducted with both real and complex valued linear system equations of various dimensions. According to our results, approximately a speedup ratio of 2.5 can easily be obtained in the case of complex number algorithm. By utilizing special data structure and related intrinsics provided in Intel's C compiler, performance of an existing LU decomposition algorithm could be improved 80% on the average. The resulting performance improvement is satisfactory if we consider that only a minor modification on the original program is needed. Most importantly, there is no additional hardware needed for such performance enhancement.

The performance obtained by SSE can be improved in various ways. One of them is by optimizing the cache memories. As data is being read from memory continuously, pre-fetching data into the cache memory of the system will improve the data access time. Pre-fetching is supported in the SSE and users can choose the cache memory (level 1 or level 2) to host the data. The experiments given earlier make use of the cache memory implicitly.

In our experiments, the speedup obtained for larger matrices tends to decrease. This was due to increasing data packing and unpacking overhead that eventually dominates the speedup gained by SIMD computations. It is not unrealistic to expect that future generation microprocessors

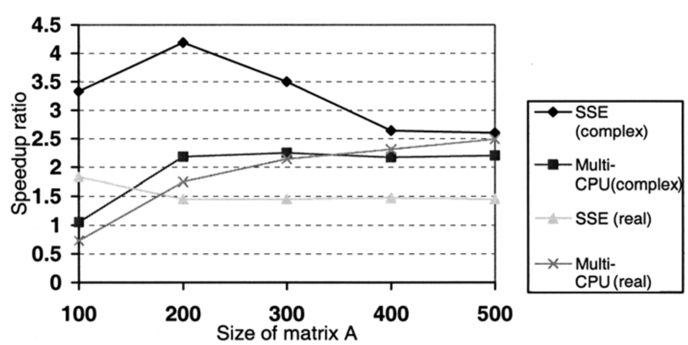


Fig. 4. Speedup ratio for various sized complex matrices.

Table 3
Processing time (in ms) for various sized complex matrices

	Matrix *A* (real)					Matrix *A* (complex)				
	100	200	300	400	500	100	200	300	400	500
Seq. (Pentium)	6.22	43.26	189	648	1205	30	285.5	1042	2611	5124.1
SSE (Pentium)	3.4	29.84	130	442	830	9	68	297.7	988.4	1968.8
Seq. (multi-CPU)	7.65	60.1	202	481	938.2	116.5	918.0	3161	7220	14320
Par. (multi-CPU)	10.5	34.2	94.2	208.0	376.5	111.1	421.3	1401	3320	6521

will have wider SSE registers. With its promising performance increase, SSE is a valuable tool for improving the performance of many computationally intensive problems.

References

[1] AltiVec Programming Environments Manual, Motorola, 2001.

[2] G. Conte, S. Tommesani, F. Zanichelli, The long and winding road to high-performance image processing with MMX/SSE, IEEE International Workshop for Computer Architectures for Machine Perception, 2000, pp. 302–310.

[3] J.M. Conroy, S.G. Kratzer, R.F. Lucas, Data-parallel sparse matrix factorization. Proceedings of the Fifth SIAM Conference on Linear Algebra, Philadelphia, 1994, pp. 377–381.

[4] I.S. Duff, H.A. van der Vorst, Developments and trends in the parallel solution of linear systems, Parallel Computing 25 (1999) 1931–1970.

[5] G.H. Golub, C.F. Van Loan, Matrix Computations, John Hopkins University Press, Baltimore, 1989.

[6] The Complete Guide to MMX Technology, Intel Corporation, McGraw-Hill, New York, 1997.

[7] Using MMX instructions in a fast IDCT algorithm for MPEG decoding, Intel Application note AP-528, 1998.

[8] Streaming SIMD extension application note n. 931 at http://developer.intel.com.

[9] Intel C/C++ Compiler Class Libraries for SIMD Operations User's Guide, 2000.

[10] S. Lakshmivarahan, S.K. Dhall, Analysis and Design of Parallel Algorithms—Arithmetic and Matrix Problems, McGraw-Hill, New York, 1990.

[11] K. Lau, D.J. Tylavsky, A. Bose, Coarse Grain Scheduling in Parallel Triangular Factorisation and Solution of Power System Matrices, IEEE Transactions on Power Systems 6 (1991) 708–714.

[12] B. Lewis, D.J. Berg, Multithreaded Programming with Pthreads, Sun Microsystems Press, Upper Saddle River, N.J., 1998.

[13] H. Nguyen, L. John, Exploiting SIMD parallelism in DSP and multimedia algorithms using the AltiVec technology, Proceedings of 13th ACM International Conference on Supercomputing, 1999, pp. 11–20.

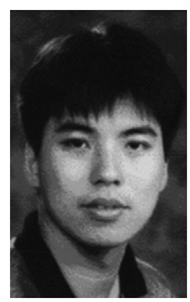
[14] VIS Instruction Set User Manual, Sun Microsystems Inc., 2001.

[15] J.G.G. Van de Vorst, R.H. Bisseling, Parallel LU-decomposition on a transputer network, Lecture Notes in Computer Science, Springer, Berlin, 1989 pp. 61–77.

[16] K.C.P. Wang, X. Zhang. Experimentation with a host-based parallel algorithm for image processing, Proceedings of second International Conference on Traffic and Transportation Studies, 2000, pp. 736–742.

Yu-Fai Fung is an Assistant Professor in the Department of Electrical Engineering at the Hong Kong Polytechnic University. He conducts research and teaches in the areas of computer architecture, software engineering, and parallel computation. He received the B.Sc. (eng) degree in 1986, and the Ph.D. in 1991, all from the University College of London.

M. Fikret Ercan received his B.Sc. and M.Sc. degrees in Electronic Engineering, respectively, in 1987 and 1991 from Dokuz Eylul University, Turkey. He obtained his Ph.D. degree in Electrical Engineering from The Hong Kong Polytechnic University in 1998. He was a research and development engineer in electronic and computer industries during 1987–1994. Currently, he is a lecturer in School of Electrical and Electronic Engineering at Singapore Polytechnic, Singapore. His research interests are in computer vision, high performance computing, computer architectures, and multiprocessor task scheduling.



Tin Kin Ho received the B.E. and Ph.D. degrees in Electronic and Electrical Engineering in 1988 and 1994, respectively, from the University of Birmingham, UK. He joined the Department of Electrical Engineering of the Hong Kong Polytechnic University in 1992 and is currently an Assistant Professor. His research interests include traffic control in various types of networks, railway signaling and operation simulation.

Raymond Wai-Leung Cheung received the Bachelor degree in Electrical Engineering from the University of Western Ontario, Canada in 1978 and obtained his Ph.D. degree in Electrical Engineering from the Hong Kong Polytechnic University in 1999. He has been an assistant professor at the Hong Kong Polytechnic University since 1995. His research interests include multimedia traffic high speed communication network, performance issues in micro-processing computing architecture, and layer protocols operational performance in wireless communication.