SSE Based Parallel Solution for Power Systems Network Equations

Y.F. Fung[1], M. Fikret Ercan[2] ,T.K. Ho[1], and W.L. Cheung[1]

[1] Dept. of Electrical Eng., The Hong Kong Polytechnic University, Hong Kong SAR
{eeyffung, eetkho, eewlcheung}@polyu.edu.hk

[2] School of Electrical and Electronic Eng., Singapore Polytechnic, Singapore
mfercan@sp.edu.sg

Abstract. Streaming SIMD Extensions (SSE) is a unique feature embedded in the Pentium III class 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 power systems network equations with SSE and discuss advantages and disadvantages of this approach.

1 Introduction

Personal Computer (PC) or workstation is currently the most popular computing system for solving various engineering problems. A major reason is the cost-effectiveness of a PC. With the advanced integrated circuit manufacturing processes, the computing power that can be delivered by a microprocessor is increasing. Currently, processor with a working frequency of 1GHz is available. The computing performance of a microprocessor is primarily dictated by two factors, namely the operating frequency (or clock rate), and the internal architecture.

The Streaming SIMD Extensions (SSE) is a special feature available in the Intel Pentium III class of microprocessors. As its name implies, the SSE enables the execution of SIMD (Single Instruction Multiple Data) operations inside the processor and therefore, the overall performance of an algorithm can be improved significantly.

The power network problem is computationally intensive and in order to reduce the computation time many researchers have proposed solutions [1,2] based on parallel hardware systems. However, 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 power network problem utilizing SSE will benefit to research in this area. In the next section, details of the SSE mechanism will be described, followed by a discussion on the problem of power systems network solution. The parallel algorithm using SSE and its performance will be discussed consecutively.

2 SSE Mechanism

The SSE can be considered as an extension of the MMX technology implemented by the Intel Pentium processors [3]. It provides a set of 8 64-bit wide MMX registers and 57 instructions for manipulating packed data stored in the registers.

2.1 Register and Data Storage

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 [4]. 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. Applications relying heavily on floating-point operations, such as 3D geometry, and video processing can be substantially accelerated [5]. Moreover, the support of floating-point values in the SSE operations has tremendously widened its applications in other problems including the power systems network problem described in this paper.

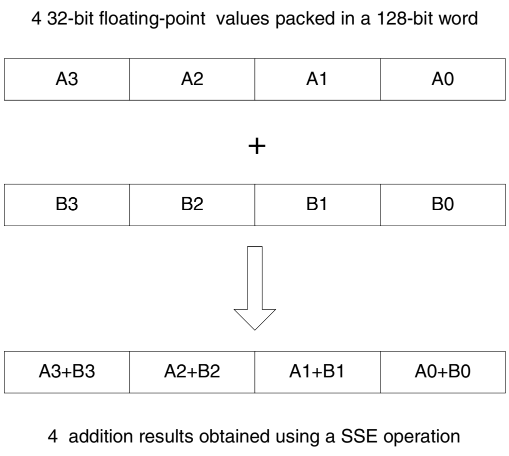
4 32-bit floating-point values packed in a 128-bit word
A3
A2
A1
A0
+
B3
B2
B1
B0
A3+B3
A2+B2
A1+B1
A0+B0
4 addition results obtained using a SSE operation

Fig. 1. Parallelism based on SSE operation

2.2 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 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*** [4]. 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 are 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 Power System Network Equations

The power systems network equations usually involve identifying solutions for a set of linear equations in the form of:

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

where A is an incidence symmetric sparse matrix of order n, b is a given independent vector and x is an unknown solution vector. As discussed in the introduction, the problem is computationally intensive. In addition, for some applications such as real-time power systems simulation, solution for equation (1) must be determined in a short time-interval [5], e.g. 10 ms, this also demands a very fast computation.

A common procedure [6] for solving (1) is to factor A into lower and upper triangular matrices L and U such that

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

and this 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.

A realistic power system network is comprising of a number of sub networks A_i connected via t_i -lines A_{ic} , as shown in Fig. 2, to a group of busbars known as cut-

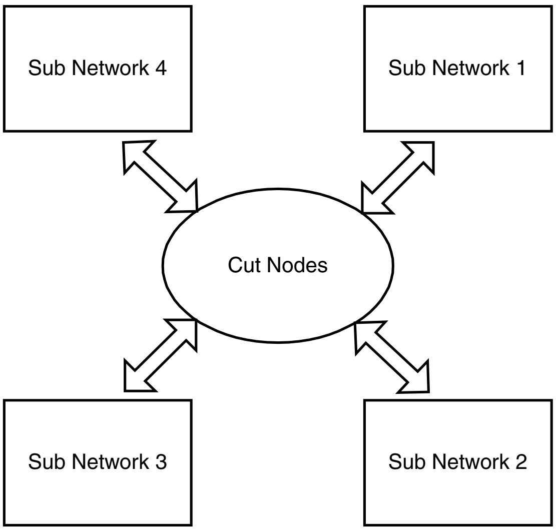

Fig. 2. Block diagram of power systems networks

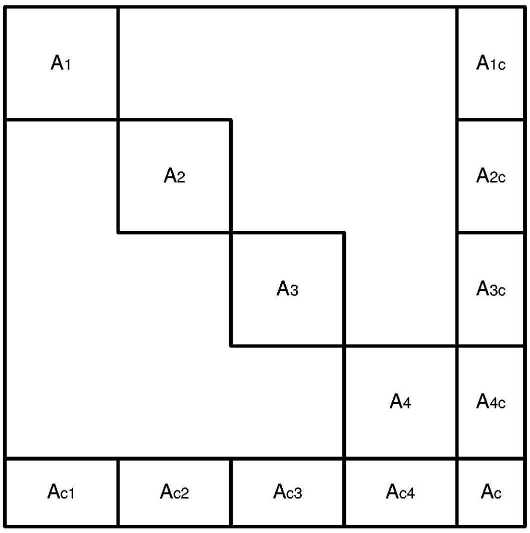

Fig. 3. Bordered block diagonal form for a power network system

nodes A_c [5]. If the network admittance matrix is arranged to follow the sub network configuration, it can be re-arranged into the Bordered Block Diagonal Form (BBDF) as shown in Fig. 3.

The BBDF matrix can now be grouped into sub-matrices, as shown in Fig. 4. Each matrix can be solved by LU decomposition. The solution for the A_c (the cut-node block) is determined by

$$L_c U_c = A_c - \sum_{i=1}^{n} A_{ic} \tag{5}$$

Referring to Fig.4, the sub-matrix is now a dense matrix and therefore, traditional dense matrix algorithm can be applied to determine the L, U triangular matrices. On the other hand, the BBDF, which is a sparse matrix, should be solved by sparse matrix solutions, such as the Choleski method [7].

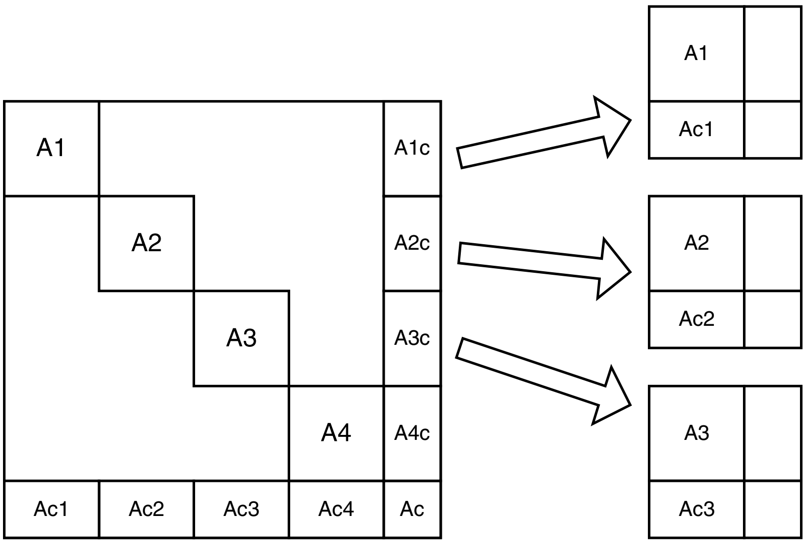

Fig. 4. Partitioning the BBDF matrix into sub-matrices

4 Parallel LU Decomposition Based on SSE

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

```
For  k = 0 to n − 2                                        (6)
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 A matrix.

According to (6), 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 and therefore, four elements in a row can be evaluated in one single step.

Based on (6), the term $\frac{a_{i,k}}{a_{k,k}}$ 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 pseudo codes shown in following illustrate the steps performed in order to implement equation (6) 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 = $\frac{a_{i,k}}{a_{k,k}}$ ; _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:

$$x'_i = b_i - \sum_{j=1}^{i-1} x'_j \bullet L_{i,j} \tag{7}$$

where $[x'_i]$ represents element in the $[x']$ matrix as shown in equation (4); b_i represents elements in the $[b]$ matrix $L_{i,j}$ represents elements in the $[L]$ matrix. SSE operations are applied in the operation $x'_j \bullet L_{i,j}$. 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

$$x_j = \frac{x'_j - \sum_{n=j+1}^{m} x_n \bullet U_{j,n}}{U_{j,j}} \tag{8}$$

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 $x_n \bullet U_{j,n}$ can be executed by SSE functions with four elements of x_n and $U_{j,n}$ being operated on at the same instead.

5 Experimental Results

In Sections 3 and 4, the processing requirements for the power system network equations and basic features of SSE have been described. In this section, results obtained from the equation $Ax = b$ based on LU decomposition and forward and backward substitutions are given. Processing time obtained for different dimensions of the matrix A are given in Table 1. Three different cases are being compared, namely, (1) conventional 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, by taking the processing time of the traditional approach as reference, are illustrated in Fig. 5.

Table 1. Processing time for solution of $Ax = b$ in (ms)

	Size of Matrix A		
	100	200	400
Traditional	9	58.83	525,2
SSE	6	40	417
SSE with alignment	5	36.83	367.8

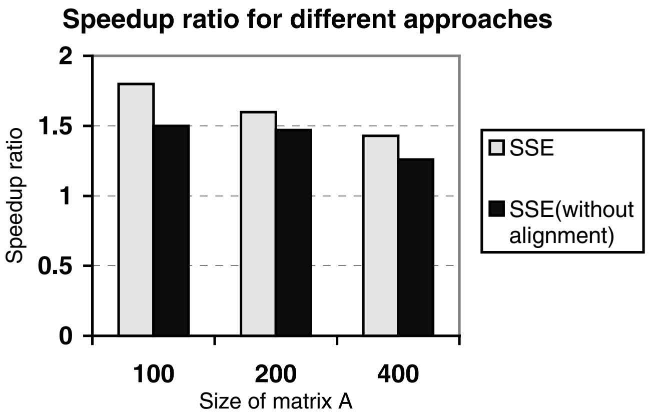

Fig. 5. Speedup ratio for three different approaches

Referring to Fig. 5, the speedup ratio obtained in the case of using SSE is slightly better than non-aligned case. For a better performance, data should be 16-byte aligned form as described in Section 2.2. 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 100x100 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 4, 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

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

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

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

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

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.

The equation (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, therefore it is neglected in the current model.

According to equations (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.

6 Conclusions

In this paper, we have presented the basic operations involved in utilizing the SSE features of the Pentium III processors. In order to examine the effectiveness of SSE, the power systems network equations problem was solved by applying SSE functions. According to our results, a speedup ratio around 1.5 can be easily obtained. The results are satisfactory and only minor modifications of the original program are needed in order to utilize the SSE features. Most importantly, additional hardware is not required for this performance enhancement. SSE is a valuable tool for improving the performance of computation intensive problems, and the power systems network equations problem is one of the ideal applications.

In this paper, we have only considered the DC power systems, where computations involve only real numbers. Currently, we are investigating AC systems that require processing of complex numbers. Moreover, the applications of SSE in a dual-CPU system will also be studied. These studies will lead to cost-effective high performance computation of power system applications.

References

1. Taoka, H., Iyoda, I., and Noguchi, H.: Real-time Digital Simulator for Power System Analysis on a Hybercube Computer. IEEE Trans. On Power Systems. 7 (1992) 1-10
2. Guo Y., Lee H.C., Wang X., and Ooi B.: A Multiprocessor Digital Signal Processing System for Real-time Power Converter Applications, IEEE Trans. On Power Systems, 7 (1992) 805-811
3. The Complete Guide to MMX Technology, Intel Corporation, McGraw-Hill (1997)
4. Conte G., Tommesani S., Zanichelli F.: The Long and Winding Road to High-performance Image Processing with MMX/SSE, IEEE Int'l Workshop for Computer Architectures for Machine Perception (2000), 302-310.
5. Wang K.C.P., and Zhang X.: Experimentation with a Host-based Parallel Algorithm for Image Processing, Proc. 2nd Int'l conf on Traffic and Transportation Studies (2000) 736-742.
6. Intel C/C++ Compiler Class Libraries for SIMD Operations User's Guide (2000)
7. Chan K.W. and Snider, L.A.: Development of a Hybrid Real-time Fully Digital Simulator for the Study and Control of Large Power Systems, Proc. of APSCOM 2000, Hong Kong, (2000) 527-531

8. Wu J.Q., and Bose A.: Parallel Solution of Large Sparse Matrix Equations and Parallel Power Flow, IEEE Trans. On Power Systems, 10 (1995) 1343-1349
9. Jess J.A., and Kees, G.H.: A Data Structure for Parallel LU Decomposition, IEEE Trans., C-31 (1992) 231-239