AP-931

Streaming SIMD Extensions - LU Decomposition

June 1999

Order Number: 245046-001

Table of Contents

Revision History

Revision	Revision History	Date
1.0	First external publication	3/99
0.99	Internal publication	1/99

References

The following documents are referenced in this application note, and provide background or supporting information for understanding the topics presented in this document.

1. *Increasing the Accuracy of the Results from the Reciprocal and Reciprocal Square Root Instructions using the Newton-Raphson Method*, Intel Application Note AP-803, Order No: 243637-001.

2. Numerical Linear Algebra. Computational Science Education Project. http://csep1.phy.ornl.gov/csep/TEXTOC.html

3. *Using the RDTSC Instruction for Performance Monitoring*, http://www.intel.com/drg/pentiumII/appnotes/RDTSCPM1.HTM

4. *Strang, Gilbert; Linear Algebra and its Applications"*, Third Edition 1988, Harcourt Brace Jovanovich Publishers, San Diego, 1988, pp 31-39

1 Introduction

This application note describes LU Decomposition of matrices with arbitrary dimensions using Intel's Streaming SIMD Extensions.

The performance of the code, which uses the Streaming SIMD Extensions for LU Decomposition, is approximately 2.6x times faster (for 15 x 15 matrices) than a generic C code implementation (See section 5.1). With increasing matrix dimension, the performance ratio (for 30 x 30 – 3.5, 40 x 40 – 4.0) increases as well. These measurements are based on tests run on a 450MHz Pentium® III processor.

2 LU Decomposition

LU Decomposition is used to solve a system of linear equations:

$Ax = b$,

where A is the coefficient matrix, b is the vector which specifies the right-hand side of the system of equations and x is the vector of unknown values.

The coefficient matrix is decomposed into the product of an upper-triangle matrix and a lower-triangle matrix (LU Decomposition); P is some matrix of permutations of the original matrix A.

$PA = LU$

Then the reference system of equations will have the form:

$LUx = b$

To compute the vector x we will use the substitution:

$Ux = y$

In the first step, we will find y from:

$Ly = b$,

then find x from $Ux = y$.

LU Decomposition turns out to be very convenient for the solution of a large number of systems of linear equations that have the same coefficient matrix A but different right-part vectors b.

It is possible to store upper-triangle and lower-triangle matrices in the original matrix, because both the upper and the lower parts of the corresponding matrices are all equal to zero and diagonal elements of L (which are not stored) are equal to 1.The Gaussian Elimination method is used for LU Decomposition.

Algorithm for LU Decomposition:

for k = 1:n-1

Pivot by choosing l so $|A(l,k)| = max_{k<=i<=n}|A(i,k)|$*, and swapping rows l and k of A;*

Exit if A(k,k) = 0;

for i = k + 1:n

A(i,k) = A(i,k)/A(k,k)

for i = k+1:n

for j = k+1:n

*A(i,j) = A(i,j) - A(i,k) * A(k,j)*

When the algorithm is completed, the diagonal and the upper triangle of matrix *A* contain matrix *U*, while the triangle below the diagonal contains the corresponding lower part of *L* (the diagonal elements of *L* all contains 1). The matrix of permutations *P* is defined by rearrangements in the second line of the algorithm [4].

2.1 Implementing LU Decomposition with Level 2 BLAS

Let's now consider vector implementation of LU Decomposition. This algorithm uses Level 2 BLAS operations (multiplication of 2 vectors and subtraction of 2 matrices).

for k = 1:n-1

Pivot by choosing l so $|A_{lk}| = max_{k<=i<=n}|A_{ik}|$*, and swapping rows l and k of A;*

Exit if A(k,k) = 0;

$A(k+1:n,k) = A(k+1:n,k)/A_{kk}$

*A(k+1:n, k+1:n) = A(k+1:n, k+1:n) - A(k+1:n, k) * A(k, k+1:n)*

As compared to typical implementation, this algorithm is using some level of parallelism. It allows for efficient implementation on Pentium® III processors. Note, that vector algorithms are characterized by longer start-up times (preparation of data for processing in vector registers) and thus a usual algorithm is more suitable for matrices with small dimensions.

For our implementation, performance gains are achieved for matrices of size 4 x 4 and larger.

3 Performance

The performance of the LU Decomposition algorithm can be significantly increased by using Streaming SIMD Extensions. The Streaming SIMD Extensions improve the performance of LU Decomposition relative to scalar floating-point code due to the single-instruction-multiple-data processing capability of the Pentium III processor. When the data is stored row or column order, one instruction can operate on 4 data elements. This allows processing of 4 elements of a matrix row or matrix column in one instruction.

An additional increase in performance may be achieved by substituting the `divps` instruction, characterized by rather high latency, with the low-latency `RCPPS` instruction. An RCPPS instruction can be followed, if high accuracy is required, with a Newton-Raphson approximation. For more information, refer to [1] the Intel Application Note AP-803, *Increasing the Accuracy of the Results from the Reciprocal and Reciprocal Square Root Instructions using the Newton-Raphson Method.*

Table 1 compares the performance of scalar floating-point code and mixed C++ and assembly code using Streaming SIMD Extensions (vector implementation of LU Decomposition algorithm) for matrices with dimensions from 1 to 38. Processor cycles were measured by using the `rdtsc` instruction (see http://www.intel.com/drg/pentiumII/appnotes/RDTSCPM1.HTM).

Table 1: Performance Gains Using Streaming SIMD Extensions [1]

Matrix Dimension	Generic C code	C code with Streaming SIMD Extensions
1	683	115
2	735	254
3	421	466
4	913	703
5	1666	1149
6	2732	1760
7	4144	2306
8	6034	3143
9	8185	3965
10	11119	4930
11	16338	6224
12	20928	7479
13	22583	8999
14	27584	10728
15	33551	12560
16	40466	14261
17	48251	18203
18	56356	18780
19	71060	21989
20	76374	25969
21	88237	28469
22	100872	31614
23	114901	35268
24	135289	37997
25	146367	43470
26	163734	50312
27	182747	52371
28	203635	57472
29	224743	63468
30	248454	69855
31	273064	76571
32	300245	79800
33	335096	89864
34	359126	93924
35	390228	104851
36	432500	112565
37	458817	121080
38	497169	128225
39	534806	138102
40	576129	145212

[1] These measurements are based on tests run on a 450MHz, 64MB SDRAM, 100MHz bus Pentium® III processor. This is the first Pentium® III processor release. Performance on future releases of Pentium® III processor may vary.

4 Conclusion

Using Streaming SIMD Extensions provides considerable increase in the performance of the LU Matrix Decomposition. The key reasons for the performance gain are the following:

- Use of single-instruction-multiple-data commands of the Pentium® III processor;
- `rcpps` instruction with the following Newton-Raphson algorithm may be used as a substitution for `divps`, which is characterized by higher latency, in those cases when complete accuracy is not required.

5 Source Code

Below two different examples are provided. The first example is a generic implementation of LU Decomposition, and the second one is vector operation-oriented LU Decomposition using Streaming SIMD Extensions. This code is included in Intel's Small Matrix Library.
These examples require the Intel® C/C++ Compiler
(http://support.intel.com/support/performancetools/c/).

5.1 C/C++ Implementation Code

The following code performs LU Decomposition without the Streaming SIMD Extensions.

```
#include <stdio.h>
#include <stdlib.h>

#include <xmmintrin.h>

const int sz = 20;

__declspec(naked) float __fastcall FastAbs(float a)
{
        __asm {
                fld DWORD PTR [esp+4]
                fabs
                ret 4
        }
}

bool  PII_LUDecomposition(float m[sz][sz], int n, double &det, int* ri, int* irow)
{
        // Factors "m" matrix into A=LU where L is lower triangular and U is upper
        // triangular.  The matrix is overwritten by LU with the diagonal elements
        // of L (which are unity) not stored.  This must be a square n x n matrix.
        // ri[n] and irow[n] are scratch vectors used by LUBackSubstitution.
        // d is returned +-1 indicating that the
        // number of row interchanges was even or odd respectively.
        //

        int i, j, k;

        int     size, last, end, pe;
        int     last8, end8, pe8;
        float   frcp, tmp, pivel;
        register float *tmpptr;
        float   *ptr2, *ptr;

        float* pdata = m[0];

        det = 1.0;

        // Initialize the pointer vector.

        for (i = 0 ; i < n; i++)
                ri[i] = i;

        // LU factorization.

        for (int p = 1 ; p <= n - 1; p++) {

                // Find pivot element.
```

```
                for (i = p + 1 ; i <= n; i++) {
                        if (FastAbs(m[ri[i-1]][p-1]) > FastAbs(m[ri[p-1]][p-1])) {
                                // Switch the index for the p-1 pivot row if necessary.
                                int t = ri[p-1];
                                ri[p-1] = ri[i-1];
                                ri[i-1] = t;
                                det = -det;
                        }
                }

                if (m[ri[p-1]][p-1] == 0) {
                        // The matrix is singular.
                        return false;
                }

                // Multiply the diagonal elements.
                det = det * m[ri[p-1]][p-1];

                // Form multiplier.
                for (i = p + 1 ; i <= n; i++) {
                        m[ri[i-1]][p-1] /= m[ri[p-1]][p-1];

                        // Eliminate [p-1].
                        for (int j = p + 1 ; j <= n; j++)
                                m[ri[i-1]][j-1] -= m[ri[i-1]][p-1] * m[ri[p-1]][j-1];
                }

        }

        det = det * m[ri[n-1]][n-1];
        return det != 0.0;
}


void printout(float m[sz][sz])
{
        int   i, j;
        printf("{");
        for (i = 0 ; i < sz; i++) {
                printf("{");
                for (j = 0 ; j < sz; j++) {
                        printf("%f%c", m[i][j], j == sz-1? ' ':',');
                }
                printf("}%c\n", i == sz-1? ' ':',');
        }
        printf("}\n\n");
}

int main(int argc, char* argv[])
{
        float m[sz][sz];
        int   v1[sz];
        int   v2[sz];
        int   i, j;
        for (i = 0 ; i < sz; i++) {
                for (j = 0 ; j < sz; j++) {
                        m[i][j] = (float)rand() / RAND_MAX;
                }
        }

        printout(m);

        double det;
        PII_LUDecomposition(m, sz, det, v1, v2);

        printout(m);
}
```

5.2 Mixed C++ and Assembly Code with Streaming SIMD Extensions

The following mixed C++ and assembly code performs LU Decomposition using Streaming SIMD Extensions.

```
#include <stdio.h>
#include <stdlib.h>
```

```
#include <xmmintrin.h>

const int sz = 20;

__declspec(naked) float __fastcall FastAbs(float a)
{
        __asm {
                fld DWORD PTR [esp+4]
                fabs
                ret 4
        }
}

bool  PIII_LUDecomposition(float m[sz][sz], int n, double &det, int* ri, int* irow)
{
        // Factors "m" matrix into A=LU where L is lower triangular and U is upper
        // triangular.  The matrix is overwritten by LU with the diagonal elements
        // of L (which are unity) not stored.  This must be a square n x n matrix.
        // ri[n] and irow[n] are scratch vectors used by LUBackSubstitution.
        // d is returned +-1 indicating that the
        // number of row interchanges was even or odd respectively.
        //

        int i, j, k;

        int     size, last, end, pe;
        int     last8, end8, pe8;
        float   frcp, tmp, pivel;
        register float *tmpptr;
        float   *ptr2, *ptr;

        float* pdata = m[0];

        det = 1.0;

        // Initialize the pointer vector.
        for (i = 0 ; i < n; i++) {
                ri[i] = i;
                irow[i] = i * n;
        }

        // LU factorization.
        for (int p = 1 ; p <= n-1; p++) {
                // Find pivot element.
                for (i = p + 1 ; i <= n; i++) {
                        if (FastAbs((pdata + irow[i-1])[p-1]) > FastAbs((pdata + irow[p-1])[p-1])) {
                                // Switch the index for the p-1 pivot row if necessary.
                                int t = ri[p-1];
                                ri[p-1] = ri[i-1];
                                ri[i-1] = t;
                                t = irow[p-1];
                                irow[p-1] = irow[i-1];
                                irow[i-1] = t;
                                det = -det;
                        }
                }

                pivel = *(pdata + irow[p-1] + p-1);

                if (pivel == 0) {
                        // The matrix is singular.
                        return false;
                }

                // Multiply the diagonal elements.
                det = det * pivel;

                // Form multiplier.
                __asm {
                        movss    xmm1, DWORD PTR pivel
                        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
                } // calculates 1/pivel using reciprocal division

                //1. A[p+1:n][n] = A[p+1:n][n] / A[p][p]
```

```
size    = n - p;
last8   = size&7;
end8    = size - last8;
pe8     = n - last8;
last    = size & 3;
end     = size - last;
pe      = n - last;

for (i = p + 1 ; i <= pe; i+=4) {
        (pdata + irow[i-1])[p-1] *= frcp;
        (pdata + irow[i])[p-1]   *= frcp;
        (pdata + irow[i+1])[p-1] *= frcp;
        (pdata + irow[i+2])[p-1] *= frcp;
}       // end 1 and form multiplier
if(last)
{
        for (i = p + 1 + end; i <= n; i++) {
                (pdata + irow[i-1])[p-1] *= frcp;
        }
}// end 1 and form multiplier


//2. A[p+1:n][p+1:n] = A[p+1:n][p+1:n] - A[p+1:n][p] * A[p][p+1:n]
ptr2 = pdata + irow[p-1] - 1;
for (j = p + 1 ; j < pe8; j += 8) { //loop for 8 columns
        tmpptr = ptr2 + j;
        __asm mov    eax,  DWORD PTR [tmpptr]
        __asm movups xmm0, XMMWORD PTR[eax]
        __asm movups xmm7, XMMWORD PTR[eax+16]////!!!
        for (i = p + 1 ; i < pe; i += 4) {  //loop for 4 rows
                ptr     = pdata + irow[i-1];
                tmpptr  = ptr+p-1;
                __asm mov       eax,  DWORD PTR [tmpptr]
                __asm movss     xmm1, DWORD PTR[eax]
                __asm shufps    xmm1, xmm1, 0
                __asm movaps    xmm3, xmm1
                __asm mulps     xmm1, xmm0
                __asm mulps     xmm3, xmm7
                tmpptr  = ptr+j-1;
                __asm mov       eax,  DWORD PTR [tmpptr]
                __asm movups    xmm2, XMMWORD PTR[eax]
                __asm subps     xmm2, xmm1
                __asm movups    xmm4, XMMWORD PTR[eax+16]
                __asm subps     xmm4, xmm3
                __asm movups    XMMWORD PTR[eax], xmm2
                __asm movups    XMMWORD PTR[eax+16], xmm4

                ptr     = pdata + irow[i];
                tmpptr  = ptr+p-1;
                __asm mov       eax,  DWORD PTR [tmpptr]
                __asm movss     xmm1, DWORD PTR[eax]
                __asm shufps    xmm1, xmm1, 0
                __asm movaps    xmm3, xmm1
                __asm mulps     xmm1, xmm0
                __asm mulps     xmm3, xmm7
                tmpptr  = ptr+j-1;
                __asm mov       eax,  DWORD PTR [tmpptr]
                __asm movups    xmm2, XMMWORD PTR[eax]
                __asm subps     xmm2, xmm1
                __asm movups    xmm4, XMMWORD PTR[eax+16]
                __asm subps     xmm4, xmm3
                __asm movups    XMMWORD PTR[eax], xmm2
                __asm movups    XMMWORD PTR[eax+16], xmm4

                ptr     = pdata + irow[i+1];
                tmpptr  = ptr+p-1;
                __asm mov       eax,  DWORD PTR [tmpptr]
                __asm movss     xmm1, DWORD PTR[eax]
                __asm shufps    xmm1, xmm1, 0
                __asm movaps    xmm3, xmm1
                __asm mulps     xmm1, xmm0
                __asm mulps     xmm3, xmm7
                tmpptr  = ptr+j-1;
                __asm mov       eax,  DWORD PTR [tmpptr]
                __asm movups    xmm2, XMMWORD PTR[eax]
                __asm subps     xmm2, xmm1
                __asm movups    xmm4, XMMWORD PTR[eax+16]
                __asm subps     xmm4, xmm3
                __asm movups    XMMWORD PTR[eax], xmm2
                __asm movups    XMMWORD PTR[eax+16], xmm4
```

```
                ptr     = pdata + irow[i+2];
                tmpptr  = ptr+p-1;
                __asm mov       eax,  DWORD PTR [tmpptr]
                __asm movss     xmm1, DWORD PTR[eax]
                __asm shufps    xmm1, xmm1, 0
                __asm movaps    xmm3, xmm1
                __asm mulps     xmm1, xmm0
                __asm mulps     xmm3, xmm7
                tmpptr  = ptr+j-1;
                __asm mov       eax,  DWORD PTR [tmpptr]
                __asm movups    xmm2, XMMWORD PTR[eax]
                __asm subps     xmm2, xmm1
                __asm movups    xmm4, XMMWORD PTR[eax+16]
                __asm subps     xmm4, xmm3
                __asm movups    XMMWORD PTR[eax], xmm2
                __asm movups    XMMWORD PTR[eax+16], xmm4
        }
        if(last)
        {
                for (i = p + 1 + end; i <= n; i ++) {// calculates last rows
                        ptr     = pdata + irow[i-1];
                        tmpptr  = ptr+p-1;
                        __asm mov       eax,  DWORD PTR [tmpptr]
                        __asm movss     xmm1, DWORD PTR[eax]
                        __asm shufps    xmm1, xmm1, 0
                        __asm movaps    xmm3, xmm1
                        __asm mulps     xmm1, xmm0
                        __asm mulps     xmm3, xmm7
                        tmpptr  = ptr+j-1;
                        __asm mov       eax,  DWORD PTR [tmpptr]
                        __asm movups    xmm2, XMMWORD PTR[eax]
                        __asm subps     xmm2, xmm1
                        __asm movups    xmm4, XMMWORD PTR[eax+16]
                        __asm subps     xmm4, xmm3
                        __asm movups    XMMWORD PTR[eax], xmm2
                        __asm movups    XMMWORD PTR[eax+16], xmm4
                }// end calculates last rows
        }// end loop for rows
}
if(last8 > 3)
{
        tmpptr = ptr2 + p + 1 + end8;
        __asm mov       eax,  DWORD PTR [tmpptr]
        __asm movups    xmm0, XMMWORD PTR[eax]
        for (i = p + 1 ; i < pe; i += 4) {  //loop for rows
                ptr     = pdata + irow[i-1];
                tmpptr  = ptr+p-1;
                __asm mov       eax, DWORD PTR [tmpptr]
                __asm movss     xmm1, DWORD PTR[eax]
                __asm shufps    xmm1, xmm1, 0
                __asm mulps     xmm1, xmm0
                tmpptr  = ptr+pe8;
                __asm mov       eax, DWORD PTR [tmpptr]
                __asm movups    xmm2, XMMWORD PTR[eax]
                __asm subps     xmm2, xmm1
                __asm movups    XMMWORD PTR[eax], xmm2

                ptr     = pdata + irow[i];
                tmpptr  = ptr+p-1;
                __asm mov       eax, DWORD PTR [tmpptr]
                __asm movss     xmm3, DWORD PTR[eax]
                __asm shufps    xmm3, xmm3, 0
                __asm mulps     xmm3, xmm0
                tmpptr  = ptr+pe8;
                __asm mov       eax, DWORD PTR [tmpptr]
                __asm movups    xmm4, XMMWORD PTR[eax]
                __asm subps     xmm4, xmm3
                __asm movups    XMMWORD PTR[eax], xmm4

                ptr     = pdata + irow[i+1];
                tmpptr  = ptr+p-1;
                __asm mov       eax, DWORD PTR [tmpptr]
                __asm movss     xmm5, DWORD PTR[eax]
                __asm shufps    xmm5, xmm5, 0
                __asm mulps     xmm5, xmm0
                tmpptr  = ptr+pe8;
                __asm mov       eax, DWORD PTR [tmpptr]
                __asm movups    xmm6, XMMWORD PTR[eax]
                __asm subps     xmm6, xmm5
                __asm movups    XMMWORD PTR[eax], xmm6
```

```
                                ptr     = pdata + irow[i+2];
                                tmpptr  = ptr+p-1;
                                __asm mov       eax, DWORD PTR [tmpptr]
                                __asm movss     xmm3, DWORD PTR[eax]
                                __asm shufps    xmm3, xmm3, 0
                                __asm mulps     xmm3, xmm0
                                tmpptr  = ptr+pe8;
                                __asm mov       eax, DWORD PTR [tmpptr]
                                __asm movups    xmm4, XMMWORD PTR[eax]
                                __asm subps     xmm4, xmm3
                                __asm movups    XMMWORD PTR[eax], xmm4
                        }
                        if(last)
                        {
                                for (i = p + 1 + end; i <= n; i ++) {// calculates last rows
                                        ptr     = pdata + irow[i-1];
                                        tmpptr  = ptr+p-1;
                                        __asm mov       eax, DWORD PTR [tmpptr]
                                        __asm movss     xmm1, DWORD PTR[eax]
                                        __asm shufps    xmm1, xmm1, 0
                                        __asm mulps     xmm1, xmm0
                                        tmpptr  = ptr+pe8;
                                        __asm mov       eax, DWORD PTR [tmpptr]
                                        __asm movups    xmm2, XMMWORD PTR[eax]
                                        __asm subps     xmm2, xmm1
                                        __asm movups    XMMWORD PTR[eax], xmm2
                                }// end calculates last rows
                        }// end loop for rows
                }// end loop for columns
                switch(last) {// calculates last columns
                        case 0: break;
                        case 3:
                                ptr2 = pdata + irow[p-1] + p + end;
                                for (i = p + 1 ; i <= n; i++) {
                                        ptr     = pdata + irow[i-1] + p;
                                        tmp     = *(ptr - 1);
                                        ptr     += end;
                                        *ptr    -= tmp * (*ptr2);
                                        *(ptr+1) -= tmp * (*(ptr2+1));
                                        *(ptr+2) -= tmp * (*(ptr2+2));
                                }
                                break;
                        case 2:
                                ptr2 = pdata + irow[p-1] + p + end;
                                for (i = p + 1 ; i <= n; i++) {
                                        ptr     = pdata + irow[i-1] + p;
                                        tmp     = *(ptr - 1);
                                        ptr     += end;
                                        *ptr -= tmp * (*ptr2);
                                        *(ptr+1) -= tmp * (*(ptr2+1));
                                }
                                break;
                        case 1:
                                ptr2 = pdata + irow[p-1] + p + end;
                                for (i = p + 1 ; i <= n; i++) {
                                        ptr     = pdata + irow[i-1] + p;
                                        tmp     = *(ptr - 1);
                                        ptr     += end;
                                        *ptr    -= tmp * (*ptr2);
                                }
                                break;
                }// end 2 and calculates last columns
        }
        det = det * (pdata + irow[n-1])[n-1];
        return det != 0.0;
}


void printout(float m[sz][sz])
{
        int   i, j;
        printf("{");
        for (i = 0 ; i < sz; i++) {
                printf("{");
                for (j = 0 ; j < sz; j++) {
                        printf("%f%c", m[i][j], j == sz-1? ' ':',');
                }
                printf("}%c\n", i == sz-1? ' ':',');
        }
        printf("}\n\n");
```

```
}

int main(int argc, char* argv[])
{
        float m[sz][sz];
        int   v1[sz];
        int   v2[sz];
        int   i, j;
        for (i = 0 ; i < sz; i++) {
                for (j = 0 ; j < sz; j++) {
                        m[i][j] = (float)rand() / RAND_MAX;
                }
        }

        printout(m);

        double det;
        PIII_LUDecomposition(m, sz, det, v1, v2);

        printout(m);
}
```