Streaming SIMD Extensions - LU Decomposition
# Table of Contents

1. Introduction ..............................................................................................................................1
2. LU Decomposition ...................................................................................................................1
   2.1 Implementing LU Decomposition with Level 2 BLAS ...................................................2
3. Performance .............................................................................................................................2
4. Conclusion ...............................................................................................................................4
5. Source Code ...........................................................................................................................4
   5.1 C/C++ Implementation Code ...........................................................................................4
   5.2 Mixed C++ and Assembly Code with Streaming SIMD Extensions .........................5
Revision History

<table>
<thead>
<tr>
<th>Revision</th>
<th>Revision History</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>First external publication</td>
<td>3/99</td>
</tr>
<tr>
<td>0.99</td>
<td>Internal publication</td>
<td>1/99</td>
</tr>
</tbody>
</table>

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.

   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

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:

\[ \text{for } k = 1:n-1 \]

\[ \text{Pivot by choosing } l \text{ so } |A(l,k)| = \max_{k \leq i \leq n} |A(i,k)|, \text{ and swapping rows } l \text{ and } k \text{ of } A; \]

\[ \text{Exit if } A(k,k) = 0; \]

\[ \text{for } i = k + 1:n \]

\[ A(i,k) = A(i,k)/A(k,k) \]

\[ \text{for } i = k+1:n \]
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 $|A_{lk}| = \max_{k \leq i \leq n} |A_{ik}|$, 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](http://www.intel.com/drg/pentiumII/appnotes/RDTSCPM1.HTM)).
Table 1: Performance Gains Using Streaming SIMD Extensions

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

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.

```c
#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 PIILUDecomposition(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 *ptrz, *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("\n");
    for (i = 0; i < sz; i++){
        printf("%f", m[i][j]);
        printf("%c", j == sz-1? ' ':',');
    }
    printf("\n\n");
}

int main(int argc, char* argv[]) {
    float m[sz][sz];
    int v1[sz];
    int v2[sz];
    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 <xmmintrin.h>

const int sz = 20;

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

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; 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
            rcpps xmm1, xmm1
            movss xmm3, xmm1
            mulss xmm1, xmm1
            mulss xmm2, xmm1
            adxss xmm3, xmm3
            subss xmm3, xmm2
            movss DWORD PTR frcp, xmm3
        } // calculates 1/pivel using reciprocal division

Streaming SIMD Extensions - LU Decomposition

```c
size = n - p;
l last8 = size % 8;
end8 = size - last8;
pe8 = n - last8;
l ast = 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];
}

if (last) {
    for (i = p + 1; i <= n; i++) {
        (pdata + irow[i-1])[(p-1) * frcp];
    }
}

ptr2 = pdata + irow[p-1] - 1;
for (j = p + 1; j <= 8; j++) {
    tmpptr = ptr2 + j - 1;
    __asm mov eax, DWORD PTR [tmpptr]
    __asm movups xmm0, XMMWORD PTR[eax]
    __asm mulps xmm0, xmm7
    __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 1 and form multiplier

```c
for (i = p + 1; i <= n; i++) {
    (pdata + irow[i-1])[(p-1) * frcp];
}
```

```c
ptr2 = pdata + irow[p-1] - 1;
for (j = p + 1; j <= 8; j++) {
    tmpptr = ptr2 + j - 1;
    __asm mov eax, DWORD PTR [tmpptr]
    __asm movups xmm0, XMMWORD PTR[eax]
    __asm mulps xmm0, xmm7
    __asm movups xmm4, XMMWORD PTR[eax+16]
    __asm subps xmm4, xmm3
    __asm movups XMMWORD PTR[eax], xmm2
    __asm movups XMMWORD PTR[eax+16], xmm4
}
```

```c
ptr = pdata + irow[i+1];
```
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 mulps xmm1, xmm0

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+j-1;
	__asm mov eax, DWORD PTR [tmpptr]
	__asm movups xmm2, XMMWORD PTR [eax]
	__asm subsps xmm2, xmm1
	__asm movups xmm4, XMMWORD PTR [eax+16]
	__asm subsps 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++)
{
	//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+j-1;
__asm mov eax, DWORD PTR [tmpptr]
__asm movups xmm2, XMMWORD PTR [eax]
__asm subsps xmm2, xmm1
__asm movups xmm4, XMMWORD PTR [eax+16]
__asm subsps xmm4, xmm3
__asm movups XMMWORD PTR [eax], xmm2
__asm movups XMMWORD PTR [eax+16], xmm4

ptr = pdata + irow[i];

for (i = p + 1; i <= pe; i++)
{
	//loop for rows
	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+j-1;
__asm mov eax, DWORD PTR [tmpptr]
__asm movups xmm4, XMMWORD PTR [eax]
__asm subsps 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+j-1;
__asm mov eax, DWORD PTR [tmpptr]
__asm movups xmm6, XMMWORD PTR [eax]
__asm subsps xmm6, xmm5
__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+j-1;
__asm mov eax, DWORD PTR [tmpptr]
__asm movups xmm6, XMMWORD PTR [eax]
__asm subsps xmm6, xmm5
__asm movups XMMWORD PTR [eax], xmm4

ptr = pdata + irow[i+1];

tmpptr = ptr+p-1;
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
}
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));
}
blood;

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);
}
blood;

// 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("\n");
for (i = 0; i < sz; i++)
printf("\n");
for (j = 0; j < sz; j++) {
printf("%f\n", m[i][j], j == sz-1 ? ' ' : '');
}
printf("%d\n", i == sz-1 ? ' ' : ');
}
print("\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);
}