AP-928

Streaming SIMD Extensions - Inverse of 4x4 Matrix

March 1999

Order Number: 245043-001

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

3. *VTune™ 4.0 Performance Enhancement Environment With Support For Pentium® III Processor,* http://circuit.intel.com/emp_comm/register/corporate/ww08/vtune.htm

1 Introduction

This application note describes finding the inverse of a 4x4-matrix using Streaming SIMD Extensions.

The performance of C code with Streaming SIMD Extensions, which implements the inverse of the 4x4-matrix using Cramer's Rule, is 4x faster than a C only implementation for 450MHz Pentium® III processor.

2 The Inverse of a Matrix

The matrix A^{-1} is said to be ***inverse*** of a matrix A if it satisfies the following equality:

$A^{-1} * A = I$,

where I is the unit matrix (with main diagonal elements equal to 1 and all others equal to 0).

From this definition it follows, that $A * A^{-1} = I$, and if the inverse of matrix A exists, than inverse of A^{-1} is equal to A.

It's well known that not every matrix has an inverse. A simple criterion for matrix A to have an inverse is that its determinant is not equal to 0.

The inverse matrix can be found by solving a system of equations:

$A * \mathbf{x}^1 = \boldsymbol{e}^1$

$A * \mathbf{x}^2 = \boldsymbol{e}^2$

● ● ● ●

$A * \mathbf{x}^n = \boldsymbol{e}^n$

Here $\boldsymbol{e}^i$ are the vector-columns of a unit matrix, and $\mathbf{x}^i$ are corresponding vector-columns of the inverse matrix. There are several methods of solving this system of equations such as Gaussian elimination, the Gauss-Jordan method, Cramer's Rule and LU decomposition.

2.1 Implementing Matrix Inversion with Cramer's Rule

A 4x4 matrix can be inverted using Cramer's Rule. Let us briefly describe the algorithm of matrix inversion using Cramer's Rule.

1. Transpose the given matrix.
2. Calculate cofactors of all matrix elements. A new matrix is formed from all cofactors of the given matrix elements.
3. Calculate the determinant of the given matrix.
4. Multiply the matrix obtained in step 3 by the reciprocal of the determinant.

Cramer's Rule should not be used for large matrices due to a large number of multiplication operations to be made in the process of calculation of the determinant. If a matrix is inverted using Gaussian elimination, pivoting is required, which is rather time-consuming, because it involves if-branches.

Compared to the method of Gaussian elimination, inversion of a matrix using Cramer's Rule doesn't need pivoting and requires only one division (calculation of the reciprocal of the determinant). As a result, Cramer's Rule yields a performance gain for inversion of matrices with small dimensions (≤ 6).

3 Performance

The performance of 4x4-matrix inversion using Cramer's Rule can be increased significantly if we use Streaming SIMD Extensions. The basic data type used in these instructions is the packed-single precision floating point data type. One instruction can operate on 4 data elements. This allows processing of all elements of one 4x4-matrix row or column in a single instruction. Additional increase in performance may be achieved by substituting the `divps` instruction, characterized by rather high latency, with the low-latency `RCPPS` instruction followed, if high accuracy is required, with the Newton-Raphson approximation. For more information, refer to 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 C code based on Gaussian elimination, scalar implementation of Cramer's Rule and Streaming SIMD Extensions implementation. Processor cycles were measured by using the **rdtsc** instruction (see http://www.intel.com/drg/pentiumII/appnotes/RDTSCPM1.HTM).

Table 1: Performance Comparison [1]

Version	**Cycles**
C code with Gaussian elimination	1074
C code with Cramer's rule	846
C code with Cramer's rule & Streaming SIMD Extensions	210

4 Conclusion

Using Streaming SIMD Extensions provides considerable increase in the performance of 4x4 matrix inversion. The key reasons for the performance gain in applications which use Streaming SIMD Extensions are as follows:

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

[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.

5 Source Codes

Represented below are three different code examples. The first example is matrix inversion based on Gaussian elimination. The second code example is direct C implementation of the inverse based on Cramer's Rule (with scalar floating-point operations) and the third one is matrix inversion using Streaming SIMD Extensions (based on Cramer’s Rule).
These examples require the Intel® C/C++ Compiler (http://support.intel.com/support/performancetools/c/).

5.1 Gaussian Elimination C Code

The following C code performs the inversion of the 4x4 matrix using Gaussian elimination without using Streaming SIMD Extensions.

```
/*********************************************************************
*
* input:
*    b - pointer to array of 16 single floats (source matrix)
* output:
*    a - pointer to array of 16 single floats (invert matrix)
*
*********************************************************************/

void Invert(float b[][4], float a[][4])
{
    long indxc[4], indxr[4], ipiv[4];
    long i, icol, irow, j, ir, ic;
    float big, dum, pivinv, temp, bb;
    ipiv[0] = -1;
    ipiv[1] = -1;
    ipiv[2] = -1;
    ipiv[3] = -1;
    a[0][0] = b[0][0];
    a[1][0] = b[1][0];
    a[2][0] = b[2][0];
    a[3][0] = b[3][0];
    a[0][1] = b[0][1];
    a[1][1] = b[1][1];
    a[2][1] = b[2][1];
    a[3][1] = b[3][1];
    a[0][2] = b[0][2];
    a[1][2] = b[1][2];
    a[2][2] = b[2][2];
    a[3][2] = b[3][2];
    a[0][3] = b[0][3];
    a[1][3] = b[1][3];
    a[2][3] = b[2][3];
    a[3][3] = b[3][3];
    for (i = 0; i < 4; i++) {
        big = 0.0f;
        for (j = 0; j < 4; j++) {
            if (ipiv[j] != 0) {
                if (ipiv[0] == -1) {
                    if ((bb = (float) fabs(a[j][0])) > big) {
                        big = bb;
                        irow = j;
                        icol = 0;
                    }
                } else if (ipiv[0] > 0) {
                    return;
                }
                if (ipiv[1] == -1) {
                    if ((bb = (float) fabs((float) a[j][1])) > big) {
                        big = bb;
                        irow = j;
                        icol = 1;
                    }
                } else if (ipiv[1] > 0) {
                    return;
                }
                if (ipiv[2] == -1) {
                    if ((bb = (float) fabs((float) a[j][2])) > big) {
                        big = bb;
                        irow = j;
                        icol = 2;
                    }
                } else if (ipiv[2] > 0) {
                    return;
                }
                if (ipiv[3] == -1) {
                    if ((bb = (float) fabs((float) a[j][3])) > big) {
                        big = bb;
                        irow = j;
                        icol = 3;
                    }
                } else if (ipiv[3] > 0) {
                    return;
                }
            }
        }
        ++(ipiv[icol]);
        if (irow != icol) {
            temp = a[irow][0];
            a[irow][0] = a[icol][0];
            a[icol][0] = temp;
            temp = a[irow][1];
            a[irow][1] = a[icol][1];
            a[icol][1] = temp;
            temp = a[irow][2];
            a[irow][2] = a[icol][2];
            a[icol][2] = temp;
            temp = a[irow][3];
            a[irow][3] = a[icol][3];
            a[icol][3] = temp;
        }
        indxr[i] = irow;
        indxc[i] = icol;
        if (a[icol][icol] == 0.0) {
            return;
        }
        pivinv = 1.0f / a[icol][icol];
        a[icol][icol] = 1.0f;
        a[icol][0] *= pivinv;
        a[icol][1] *= pivinv;
        a[icol][2] *= pivinv;
        a[icol][3] *= pivinv;
        if (icol != 0) {
            dum = a[0][icol];
            a[0][icol] = 0.0f;
            a[0][0] -= a[icol][0] * dum;
            a[0][1] -= a[icol][1] * dum;
            a[0][2] -= a[icol][2] * dum;
            a[0][3] -= a[icol][3] * dum;
        }
        if (icol != 1) {
            dum = a[1][icol];
            a[1][icol] = 0.0f;
            a[1][0] -= a[icol][0] * dum;
            a[1][1] -= a[icol][1] * dum;
            a[1][2] -= a[icol][2] * dum;
            a[1][3] -= a[icol][3] * dum;
        }
        if (icol != 2) {
            dum = a[2][icol];
            a[2][icol] = 0.0f;
            a[2][0] -= a[icol][0] * dum;
            a[2][1] -= a[icol][1] * dum;
            a[2][2] -= a[icol][2] * dum;
            a[2][3] -= a[icol][3] * dum;
        }
        if (icol != 3) {
            dum = a[3][icol];
            a[3][icol] = 0.0f;
            a[3][0] -= a[icol][0] * dum;
            a[3][1] -= a[icol][1] * dum;
            a[3][2] -= a[icol][2] * dum;
            a[3][3] -= a[icol][3] * dum;
        }
    }
    if (indxr[3] != indxc[3]) {
        ir = indxr[3];
        ic = indxc[3];
        temp = a[0][ir];
        a[0][ir] = a[0][ic];
        a[0][ic] = temp;
        temp = a[1][ir];
        a[1][ir] = a[1][ic];
        a[1][ic] = temp;
        temp = a[2][ir];
        a[2][ir] = a[2][ic];
        a[2][ic] = temp;
        temp = a[3][ir];
        a[3][ir] = a[3][ic];
        a[3][ic] = temp;
    }
    if (indxr[2] != indxc[2]) {
        ir = indxr[2];
        ic = indxc[2];
        temp = a[0][ir];
        a[0][ir] = a[0][ic];
        a[0][ic] = temp;
        temp = a[1][ir];
        a[1][ir] = a[1][ic];
        a[1][ic] = temp;
        temp = a[2][ir];
        a[2][ir] = a[2][ic];
        a[2][ic] = temp;
        temp = a[3][ir];
        a[3][ir] = a[3][ic];
        a[3][ic] = temp;
    }
    if (indxr[1] != indxc[1]) {
        ir = indxr[1];
        ic = indxc[1];
        temp = a[0][ir];
        a[0][ir] = a[0][ic];
        a[0][ic] = temp;
        temp = a[1][ir];
        a[1][ir] = a[1][ic];
        a[1][ic] = temp;
        temp = a[2][ir];
        a[2][ir] = a[2][ic];
        a[2][ic] = temp;
        temp = a[3][ir];
        a[3][ir] = a[3][ic];
        a[3][ic] = temp;
    }
    if (indxr[0] != indxc[0]) {
        ir = indxr[0];
        ic = indxc[0];
        temp = a[0][ir];
        a[0][ir] = a[0][ic];
        a[0][ic] = temp;
        temp = a[1][ir];
        a[1][ir] = a[1][ic];
        a[1][ic] = temp;
        temp = a[2][ir];
        a[2][ir] = a[2][ic];
        a[2][ic] = temp;
        temp = a[3][ir];
        a[3][ir] = a[3][ic];
        a[3][ic] = temp;
    }
}
```

5.2 C Code with Cramer's rule

The following C code performs 4x4-matrix inversion with Cramer's Rule (without Streaming SIMD Extensions).

```
/************************************************************
*
*  input:
*    mat - pointer to array of 16 floats (source matrix)
*  output:
*    dst - pointer to array of 16 floats (invert matrix)
*
*************************************************************/

void Invert2(float *mat, float *dst)
```

```
{
    float     tmp[12]; /* temp array for pairs                */
    float     src[16]; /* array of transpose source matrix */
    float     det;     /* determinant                         */

    /* transpose matrix */
    for (int i = 0; i < 4; i++) {
        src[i]        = mat[i*4];
        src[i + 4]    = mat[i*4 + 1];
        src[i + 8]    = mat[i*4 + 2];
        src[i + 12]   = mat[i*4 + 3];
    }

    /* calculate pairs for first 8 elements (cofactors) */
    tmp[0]  = src[10] * src[15];
    tmp[1]  = src[11] * src[14];
    tmp[2]  = src[9]  * src[15];
    tmp[3]  = src[11] * src[13];
    tmp[4]  = src[9]  * src[14];
    tmp[5]  = src[10] * src[13];
    tmp[6]  = src[8]  * src[15];
    tmp[7]  = src[11] * src[12];
    tmp[8]  = src[8]  * src[14];
    tmp[9]  = src[10] * src[12];
    tmp[10] = src[8]  * src[13];
    tmp[11] = src[9]  * src[12];

    /* calculate first 8 elements (cofactors) */
    dst[0]  = tmp[0]*src[5] + tmp[3]*src[6] + tmp[4]*src[7];
    dst[0] -= tmp[1]*src[5] + tmp[2]*src[6] + tmp[5]*src[7];
    dst[1]  = tmp[1]*src[4] + tmp[6]*src[6] + tmp[9]*src[7];
    dst[1] -= tmp[0]*src[4] + tmp[7]*src[6] + tmp[8]*src[7];
    dst[2]  = tmp[2]*src[4] + tmp[7]*src[5] + tmp[10]*src[7];
    dst[2] -= tmp[3]*src[4] + tmp[6]*src[5] + tmp[11]*src[7];
    dst[3]  = tmp[5]*src[4] + tmp[8]*src[5] + tmp[11]*src[6];
    dst[3] -= tmp[4]*src[4] + tmp[9]*src[5] + tmp[10]*src[6];
    dst[4]  = tmp[1]*src[1] + tmp[2]*src[2] + tmp[5]*src[3];
    dst[4] -= tmp[0]*src[1] + tmp[3]*src[2] + tmp[4]*src[3];
    dst[5]  = tmp[0]*src[0] + tmp[7]*src[2] + tmp[8]*src[3];
    dst[5] -= tmp[1]*src[0] + tmp[6]*src[2] + tmp[9]*src[3];
    dst[6]  = tmp[3]*src[0] + tmp[6]*src[1] + tmp[11]*src[3];
    dst[6] -= tmp[2]*src[0] + tmp[7]*src[1] + tmp[10]*src[3];
    dst[7]  = tmp[4]*src[0] + tmp[9]*src[1] + tmp[10]*src[2];
    dst[7] -= tmp[5]*src[0] + tmp[8]*src[1] + tmp[11]*src[2];

    /* calculate pairs for second 8 elements (cofactors) */
    tmp[0]  = src[2]*src[7];
    tmp[1]  = src[3]*src[6];
    tmp[2]  = src[1]*src[7];
    tmp[3]  = src[3]*src[5];
    tmp[4]  = src[1]*src[6];
    tmp[5]  = src[2]*src[5];
    tmp[6]  = src[0]*src[7];
    tmp[7]  = src[3]*src[4];
    tmp[8]  = src[0]*src[6];
    tmp[9]  = src[2]*src[4];
    tmp[10] = src[0]*src[5];
    tmp[11] = src[1]*src[4];

    /* calculate second 8 elements (cofactors) */
    dst[8]  = tmp[0]*src[13] + tmp[3]*src[14] + tmp[4]*src[15];
    dst[8] -= tmp[1]*src[13] + tmp[2]*src[14] + tmp[5]*src[15];
    dst[9]  = tmp[1]*src[12] + tmp[6]*src[14] + tmp[9]*src[15];
    dst[9] -= tmp[0]*src[12] + tmp[7]*src[14] + tmp[8]*src[15];
    dst[10] = tmp[2]*src[12] + tmp[7]*src[13] + tmp[10]*src[15];
    dst[10]-= tmp[3]*src[12] + tmp[6]*src[13] + tmp[11]*src[15];
    dst[11] = tmp[5]*src[12] + tmp[8]*src[13] + tmp[11]*src[14];
    dst[11]-= tmp[4]*src[12] + tmp[9]*src[13] + tmp[10]*src[14];
    dst[12] = tmp[2]*src[10] + tmp[5]*src[11] + tmp[1]*src[9];
    dst[12]-= tmp[4]*src[11] + tmp[0]*src[9] + tmp[3]*src[10];
    dst[13] = tmp[8]*src[11] + tmp[0]*src[8] + tmp[7]*src[10];
    dst[13]-= tmp[6]*src[10] + tmp[9]*src[11] + tmp[1]*src[8];
    dst[14] = tmp[6]*src[9] + tmp[11]*src[11] + tmp[3]*src[8];
    dst[14]-= tmp[10]*src[11] + tmp[2]*src[8] + tmp[7]*src[9];
    dst[15] = tmp[10]*src[10] + tmp[4]*src[8] + tmp[9]*src[9];
    dst[15]-= tmp[8]*src[9] + tmp[11]*src[10] + tmp[5]*src[8];

    /* calculate determinant */
    det=src[0]*dst[0]+src[1]*dst[1]+src[2]*dst[2]+src[3]*dst[3];

    /* calculate matrix inverse */
    det = 1/det;
    for (int j = 0; j < 16; j++)
        dst[j] *= det;
}
```

5.3 C Code with Cramer's rule and Streaming SIMD Extensions

The following C code example performs 4x4-matrix inversion with Cramer's Rule and Streaming SIMD Extensions. Note: the Intel C/C++ Compiler is required to build this example.

Brief description of the program:

1.
Variables (Streaming SIMD Extensions registers) which will contain cofactors and, later, the lines of the inverted matrix are declared.

2.
Variables which will contain the lines of the reference matrix and, later (after the transposition), the columns of the original matrix are declared.

3.
Temporary variables and the variable that will contain the matrix determinant are declared.

4 - 11.
Matrix transposition.

12 - 57.

Cofactors calculation. Because in the process of cofactor computation some pairs in three-element products are repeated, it is not reasonable to load these pairs anew every time. The values in the registers with these pairs are formed using shuffle instruction**.** Cofactors are calculated row by row (4 elements are placed in 1 SP FP SIMD floating point register).

58 - 63.

Evaluation of determinant and its reciprocal value. 1/det is evaluated using a fast **rcpps** command with subsequent approximation using the Newton-Raphson algorithm.

64 - 75.
Multiplication of cofactors by 1/det. Storing the inverse matrix to the address in pointer **src**.

```
void PIII_Inverse_4x4(float* src)
{

1       __m128  minor0, minor1, minor2, minor3;
2       __m128  row0,   row1,   row2,   row3;
3       __m128  det,    tmp1;

4       tmp1    = _mm_loadh_pi(_mm_loadl_pi(tmp1, (__m64*)(src)), (__m64*)(src+ 4));
5       row1    = _mm_loadh_pi(_mm_loadl_pi(row1, (__m64*)(src+8)), (__m64*)(src+12));

6       row0    = _mm_shuffle_ps(tmp1, row1, 0x88);
7       row1    = _mm_shuffle_ps(row1, tmp1, 0xDD);

8       tmp1    = _mm_loadh_pi(_mm_loadl_pi(tmp1, (__m64*)(src+ 2)), (__m64*)(src+ 6));
9       row3    = _mm_loadh_pi(_mm_loadl_pi(row3, (__m64*)(src+10)), (__m64*)(src+14));

10      row2    = _mm_shuffle_ps(tmp1, row3, 0x88);
11      row3    = _mm_shuffle_ps(row3, tmp1, 0xDD);
        //      -----------------------------------------------
12      tmp1    = _mm_mul_ps(row2, row3);
13      tmp1    = _mm_shuffle_ps(tmp1, tmp1, 0xB1);

14      minor0  = _mm_mul_ps(row1, tmp1);
15      minor1  = _mm_mul_ps(row0, tmp1);

16      tmp1    = _mm_shuffle_ps(tmp1, tmp1, 0x4E);

17      minor0  = _mm_sub_ps(_mm_mul_ps(row1, tmp1), minor0);
18      minor1  = _mm_sub_ps(_mm_mul_ps(row0, tmp1), minor1);
19      minor1  = _mm_shuffle_ps(minor1, minor1, 0x4E);
        //      -----------------------------------------------
20      tmp1    = _mm_mul_ps(row1, row2);
21      tmp1    = _mm_shuffle_ps(tmp1, tmp1, 0xB1);

22      minor0  = _mm_add_ps(_mm_mul_ps(row3, tmp1), minor0);
23      minor3  = _mm_mul_ps(row0, tmp1);

24      tmp1    = _mm_shuffle_ps(tmp1, tmp1, 0x4E);

25      minor0  = _mm_sub_ps(minor0, _mm_mul_ps(row3, tmp1));
26      minor3  = _mm_sub_ps(_mm_mul_ps(row0, tmp1), minor3);
27      minor3  = _mm_shuffle_ps(minor3, minor3, 0x4E);
        //      -----------------------------------------------
28      tmp1    = _mm_mul_ps(_mm_shuffle_ps(row1, row1, 0x4E), row3);
29      tmp1    = _mm_shuffle_ps(tmp1, tmp1, 0xB1);
30      row2    = _mm_shuffle_ps(row2, row2, 0x4E);

31      minor0  = _mm_add_ps(_mm_mul_ps(row2, tmp1), minor0);
32      minor2  = _mm_mul_ps(row0, tmp1);

33      tmp1    = _mm_shuffle_ps(tmp1, tmp1, 0x4E);

34      minor0  = _mm_sub_ps(minor0, _mm_mul_ps(row2, tmp1));
35      minor2  = _mm_sub_ps(_mm_mul_ps(row0, tmp1), minor2);
36      minor2  = _mm_shuffle_ps(minor2, minor2, 0x4E);
        //      -----------------------------------------------
37      tmp1    = _mm_mul_ps(row0, row1);
```

```
	tmp1	= _mm_shuffle_ps(tmp1, tmp1, 0xB1);

	minor2	= _mm_add_ps(_mm_mul_ps(row3, tmp1), minor2);
	minor3	= _mm_sub_ps(_mm_mul_ps(row2, tmp1), minor3);

	tmp1	= _mm_shuffle_ps(tmp1, tmp1, 0x4E);

	minor2	= _mm_sub_ps(_mm_mul_ps(row3, tmp1), minor2);
	minor3	= _mm_sub_ps(minor3, _mm_mul_ps(row2, tmp1));
	//	-----------------------------------------------
	tmp1	= _mm_mul_ps(row0, row3);
	tmp1	= _mm_shuffle_ps(tmp1, tmp1, 0xB1);

	minor1	= _mm_sub_ps(minor1, _mm_mul_ps(row2, tmp1));
	minor2	= _mm_add_ps(_mm_mul_ps(row1, tmp1), minor2);

	tmp1	= _mm_shuffle_ps(tmp1, tmp1, 0x4E);

	minor1	= _mm_add_ps(_mm_mul_ps(row2, tmp1), minor1);
	minor2	= _mm_sub_ps(minor2, _mm_mul_ps(row1, tmp1));
	//	-----------------------------------------------
	tmp1	= _mm_mul_ps(row0, row2);
	tmp1	= _mm_shuffle_ps(tmp1, tmp1, 0xB1);

	minor1	= _mm_add_ps(_mm_mul_ps(row3, tmp1), minor1);
	minor3	= _mm_sub_ps(minor3, _mm_mul_ps(row1, tmp1));

	tmp1	= _mm_shuffle_ps(tmp1, tmp1, 0x4E);

	minor1	= _mm_sub_ps(minor1, _mm_mul_ps(row3, tmp1));
	minor3	= _mm_add_ps(_mm_mul_ps(row1, tmp1), minor3);
	//	-----------------------------------------------
	det	= _mm_mul_ps(row0, minor0);
	det	= _mm_add_ps(_mm_shuffle_ps(det, det, 0x4E), det);
	det	= _mm_add_ss(_mm_shuffle_ps(det, det, 0xB1), det);
	tmp1	= _mm_rcp_ss(det);

	det	= _mm_sub_ss(_mm_add_ss(tmp1, tmp1), _mm_mul_ss(det, _mm_mul_ss(tmp1, tmp1)));
	det	= _mm_shuffle_ps(det, det, 0x00);

	minor0	= _mm_mul_ps(det, minor0);
	_mm_storel_pi((__m64*)(src), minor0);
	_mm_storeh_pi((__m64*)(src+2), minor0);

	minor1	= _mm_mul_ps(det, minor1);
	_mm_storel_pi((__m64*)(src+4), minor1);
	_mm_storeh_pi((__m64*)(src+6), minor1);

	minor2	= _mm_mul_ps(det, minor2);
	_mm_storel_pi((__m64*)(src+ 8), minor2);
	_mm_storeh_pi((__m64*)(src+10), minor2);

	minor3	= _mm_mul_ps(det, minor3);
	_mm_storel_pi((__m64*)(src+12), minor3);
	_mm_storeh_pi((__m64*)(src+14), minor3);
}
```