数値計算 第2回課題
(大石先生)

G99P043-4
河邊昌彦

出題日　　:　2001/05/18
提出期限　:　2001/06/02
提出日　　:　2001/06/02

1 課題

1000×1000 行列 (double) の積を高速に計算するプログラムを作れ。10 回別の行列のペアに適用して，平均の計算時間を出せ。

2 方針

行列の積に関する高速化の手法は、以下のものが一般的である。

- ループ変換
- ループのアンローリング
- 行列のブロック化

基本的に、これらの組み合わせによって高速な計算を目指すことにする。また、行列のサイズを 1000 に固定した場合と、自由に変えられるようにした場合でどれほどの差が出るのか、ということも見てみることにする。

3 高速化

3.1 行列のブロック化

トップダウン的に見ていくと、行列のブロック化が最も大きな部分となっている。ただし、実際にはボトムの部分に現れる、ループ変換やループのアンローリングのことを考慮しつつ、行列のブロック化を行うことになる。

行列のブロック化で次のようなものを考える。

$$A \cdot B = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} = \begin{pmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{pmatrix} \quad (1)$$

この右辺の計算の中で、各 A_{ij}, B_{ij} は 2 回ずつ使われている。さらに、$n \times n$ 個のブロックに分割した場合は、各々が n 回ずつ使われることになる。従って、A_{ij}, B_{ij} を使うたびに元の行列からコピーするのは、無駄が多い。そこで、全ての分割されたブロックを並べ、ブロックの配列として予め作っておくことにする。

3.2 転置

(1) で積を計算するときに、A_{ij} の要素は行方向に走査され、B_{ij} の要素は列方向に走査される。C で行列を表現する時は、二次元配列で第一添字に行番号を、第二添字に列番号を割り振ることが多い。従って、A_{ij} の走査時にはストライド (要素と要素のアドレス間の距離) は小さいが、B_{ij} の場合には大きくなる。ストライドが大きいと、データアクセスの局所性が悪くなるため、キャッシュミスが発生しやすくなる。そこで、B_{ij} の走査時のストライドを小さくすることを考える。

ブロック化の時点で、全てのブロックをあらかじめ作っておくことにしてあった。よって、そこで B の要素全てにアクセスすることになる。従って、その時に、B を転置させておけば良いこと

になる。つまり、B のブロックを転置させて作ることによって、B_{ij} の列方向のストライドを小さくすることができる。

以上によって、ループ変換は明確になる。即ち、内側のループほど必ずどのブロックもストライドが小さくなるように、ループの順番を決めることができる。

3.3 ループのアンローリング

ループのアンローリングによって、ループ本体に対するループの分岐判定部分の処理量の割合を減らすことができる。それによって、パイプライン化が進み、スループットが向上する。

ループのアンローリングを行うべき場所は最も内側のループである。さらに、ループのアンローリングによってプログラムコードのサイズが増加するので、それ以外のループでは行うべきではない (コードが L1 キャッシュに入りきらなくなる危険性がある)。

ここで、最も内側のループとは、ブロック同士の積の計算の部分である。全体の行列のサイズが可変であっても、ブロックのサイズは固定することができる。従って、アンローリングするループのループ回数が固定であるので、そのループ全てを展開することができる。それによって、一番内側のループで終了判定が必要なくなり、分岐によるスループットの低下を抑えることができる。また、普通、ループカウンタ用にレジスタが一つ使われているので、それが他の用途に解放されることで、更なる高速化も期待できる。

4 Intel Pentium アーキテクチャの場合

4.1 更なる高速化

以上のことをプログラムにし、コンパイルして実行してみたところ、何も施していないプログラムに比べて 6 倍ほど速くなった。しかし、コンパイラのよって生成されたコードを見てみると、かなりデータ依存のあるコードになっていた。つまり、行列の積の計算の基本は、数の積和計算であるが、加算のデスティネーションが全て同じレジスタになっていた。

そこで、その部分を、複数のレジスタを使い、データ依存を減らしたところ、さらに 1 割ほど高速になった。ただしこのような高速化は、プロセッサのアーキテクチャに強く依存しており、必ずしも適切な方法とは限らない。

4.2 理論的な限界値

行列の積の基本である積和計算は、二つのデータをロードし、二つの演算 (積と和) をする必要がある。また、Pentium アーキテクチャでは、内部の μop においては、ロードユニットと FPU 演算ユニットが、それぞれ 1 サイクルで一つの μop をパイプライン化することができる。従って、積和計算を実行するには、最低で 2 サイクルが必要になる。1000×1000 の行列ならば、1000^3 回の積和計算が必要なので、2G サイクルかかることになる。よって、例えば、プロセッサの動作周波数が 500MHz ならば、必ず 4 秒以上かかることになる (浮動小数添計算に FPU ユニットを使う限り)。

実際には、よほどキャッシュヒット率が高くない限り、このような数字に近づくことはできない。しかし、理論的な限界値を知った上で実行結果を見れば、それがどの程度の最適化であるのか、推測することができる。

5　結果の検証

いくら高速化しても、正しい答えが得られないプログラムでは意味が無い。そこで、行列の積が正しく計算されているか、確かめる必要がある。ところが、足し算の結合順序を変えただけで、浮動小数点計算の結果は変わることがある。従って、加算を行う順番が違う、定義通りの計算と最適化された計算では、結果が僅かに違っている。しかし、このことは計算自体が間違っている、ということではない。

そこで、区間演算によってそれぞれの結果の範囲を求め、その区間が重なり合っていない場合に、計算が正しくないと判断するようにした。

6　実行結果

表 1 に、1000×1000 行列の積の計算を Intel Pentium III 733MHz, FSB 133MHz, SD-RAM 384MB PC133 上で実行した時の結果を載せた。プログラムのコンパイルは Visual C++ Ver.6.0 で行い、最適化に関しては "/G5 /Gr /Ox /Ot /Oa /Og /Oi /Ob2" のオプションを付けた。

一回目の実行は、コードがキャッシュに入っていないなどの理由で、二回目以降より明らかに時間がかかっていたので、一回目を除いた九回の平均時間も載せた。

Algorithm の definition というのは、定義通りの計算方法である。また、c-const, c-var は C の範囲内で最適化したものであり、c-const の方がサイズ固定、c-var の方がサイズ可変のものである。そして、asm-const, asm-var が、さらにアセンブラレベルでの最適化をしたものである。

この結果によると、サイズ可変であってもサイズ固定とほとんど変わらないことが分かる。ただし、1000×1000 の場合は、行列のサイズがブロックサイズの倍数となっているので、ほとんど影響が出なかったものと思われる。実際、999×999 の行列で計算させてみると、1000×1000 の場合より若干遅い結果になった。

この計算機での理論的な限界値は約 2.7 秒であるが、最適化後のプログラムでは、その 2 倍以内に収まるほどになっている。従って、それなりに良い結果となっているが、まだ最適化の余地は残されているのであろう。

Algorithm	10 回の平均時間 (msec)	9 回の平均時間 (msec)
definition	31360	31360
c-const	5404	5358
c-var	5366	5358
asm-const	4785	4730
asm-var	4740	4736

表 1: 実行時間

7　ソースコード

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <memory.h>
#include <math.h>
```

```
#include <float.h>

/*****************************************************************************/

void    up( void ){ _controlfp( _RC_UP, _MCW_RC ); }
void    down( void ){ _controlfp( _RC_DOWN, _MCW_RC ); }

/*****************************************************************************/

#define DIM     1000

#define BLOCK_DIM       50
#define BLOCK_NUM       ( DIM / BLOCK_DIM )
#define BLOCK_INNER_LOOP( x )   { LOOP50( x ) }

typedef double          scalar_t;
typedef scalar_t        matrix_t[DIM][DIM];
typedef scalar_t        block_t[BLOCK_DIM][BLOCK_DIM];
typedef block_t         blockarray_t[BLOCK_NUM][BLOCK_NUM];

typedef void    ( *mul_matrix_t )( matrix_t c, const matrix_t a, const matrix_t b );
typedef void    ( *mul_matrix_size_t )( matrix_t c, const matrix_t a, const matrix_t b, int dim );

/*****************************************************************************/

#define LOOP2( x )      x x
#define LOOP4( x )      x x x x
#define LOOP5( x )      x x x x x
#define LOOP8( x )      LOOP2( LOOP4( x ) )
#define LOOP10( x )     LOOP2( LOOP5( x ) )
#define LOOP20( x )     LOOP2( LOOP10( x ) )
#define LOOP25( x )     LOOP5( LOOP5( x ) )
#define LOOP40( x )     LOOP4( LOOP10( x ) )
#define LOOP50( x )     LOOP5( LOOP10( x ) )
#define LOOP80( x )     LOOP8( LOOP10( x ) )
#define LOOP100( x )    LOOP10( LOOP10( x ) )

/*****************************************************************************/

void    clear_block( block_t blk )
{
        int     i, j;

        for ( i = 0 ; i < BLOCK_DIM ; ++i )
                for ( j = 0 ; j < BLOCK_DIM ; ++j )
                        blk[i][j] = 0.0;
}

void    clear_block_size( block_t blk, const int row, const int col )
{
        int     i, j;

        for ( i = 0 ; i < row ; ++i )
                for ( j = 0 ; j < col ; ++j )
                        blk[i][j] = 0.0;
}

/*****************************************************************************/

void    copy_to_block( block_t blk, const matrix_t mat )
{
        int     i, j;

        for ( i = 0 ; i < BLOCK_DIM ; ++i )
                for ( j = 0 ; j < BLOCK_DIM ; ++j )
                        blk[i][j] = mat[i][j];
}
```

```
void    copy_to_block_size( block_t blk, const matrix_t mat, const int row, const int col )
{
        int     i, j;

        for ( i = 0 ; i < row ; ++i )
                for ( j = 0 ; j < col ; ++j )
                        blk[i][j] = mat[i][j];
}

void    copy_to_block_array( blockarray_t array, const matrix_t mat, const int i, const int j )
{
        copy_to_block( array[i][j], *( matrix_t * )&mat[ i * BLOCK_DIM ][ j * BLOCK_DIM ] );
}

void    copy_to_block_array_size( blockarray_t array, const matrix_t mat, const int i, const int j, const int row, const
{
        copy_to_block_size( array[i][j], *( matrix_t * )&mat[ i * BLOCK_DIM ][ j * BLOCK_DIM ], row, col );
}

void    copy_to_block_transpose( block_t blk, const matrix_t mat )
{
        int     i, j;

        for ( i = 0 ; i < BLOCK_DIM ; ++i )
                for ( j = 0 ; j < BLOCK_DIM ; ++j )
                        blk[j][i] = mat[i][j];
}

void    copy_to_block_transpose_size( block_t blk, const matrix_t mat, const int row, const int col )
{
        int     i, j;

        for ( i = 0 ; i < row ; ++i )
                for ( j = 0 ; j < col ; ++j )
                        blk[j][i] = mat[i][j];
}

void    copy_to_block_array_transpose( blockarray_t array, const matrix_t mat, const int i, const int j )
{
        copy_to_block_transpose( array[j][i], *( matrix_t * )&mat[ i * BLOCK_DIM ][ j * BLOCK_DIM ] );
}

void    copy_to_block_array_transpose_size( blockarray_t array, const matrix_t mat, const int i, const int j, const int
{
        copy_to_block_transpose_size( array[j][i], *( matrix_t * )&mat[ i * BLOCK_DIM ][ j * BLOCK_DIM ], row, col );
}

/*****************************************************************************/

void    copy_from_block( matrix_t mat, const block_t blk )
{
        int     i, j;

        for ( i = 0 ; i < BLOCK_DIM ; ++i )
                for ( j = 0 ; j < BLOCK_DIM ; ++j )
                        mat[i][j] = blk[i][j];
}

void    copy_from_block_size( matrix_t mat, const block_t blk, const int row, const int col )
{
        int     i, j;

        for ( i = 0 ; i < row ; ++i )
                for ( j = 0 ; j < col ; ++j )
                        mat[i][j] = blk[i][j];
}
```

```
void    copy_from_block_to( matrix_t mat, const block_t blk, const int i, const int j )
{
        copy_from_block( *( matrix_t * )&mat[ i * BLOCK_DIM ][ j * BLOCK_DIM ], blk );
}

void    copy_from_block_size_to( matrix_t mat, const block_t blk, const int i, const int j, const int row, const int col
{
        copy_from_block_size( *( matrix_t * )&mat[ i * BLOCK_DIM ][ j * BLOCK_DIM ], blk, row, col );
}

/*****************************************************************************/

void    mul_block( block_t blk_c, const block_t blk_a, const block_t blk_b )
{
        int     ii, jj, kk;

        for ( ii = 0 ; ii < BLOCK_DIM ; ++ii )
        {
                for ( jj = 0 ; jj < BLOCK_DIM ; ++jj )
                {
                        kk = 0;
                        BLOCK_INNER_LOOP( blk_c[ii][jj] += blk_a[ii][kk] * blk_b[jj][kk]; ++kk; )
                }
        }
}

__declspec( naked ) void        __fastcall mul_block_asm( block_t blk_c, const block_t blk_a, const block_t blk_b )
{
        // edx          : blk_a : call by register
        // ebx, eax     : blk_b
        // ecx          : blk_c : call by register

        __asm
        {
                push    ebx

                mov     ebx, DWORD PTR [esp+8]
                push    esi
                push    edi
                add     edx, 16                                 ; 00000010H
                mov     edi, 50                                 ; 00000032H

                sub     ecx, 8

        ROW:
                mov     eax, ebx
                mov     esi, 50                                 ; 00000032H

        COL:
                fld     QWORD PTR [eax+16+376]
                fmul    QWORD PTR [edx+376]

                fld     QWORD PTR [eax+16+368]
                fmul    QWORD PTR [edx+368]

                fld     QWORD PTR [eax+16+360]
                fmul    QWORD PTR [edx+360]
                faddp   ST(2), ST(0)

                fld     QWORD PTR [eax+16+352]
                fmul    QWORD PTR [edx+352]
                faddp   ST(1), ST(0)

                fld     QWORD PTR [eax+16+344]
                fmul    QWORD PTR [edx+344]
                faddp   ST(2), ST(0)
```

```
fld     QWORD PTR [eax+16+336]
fmul    QWORD PTR [edx+336]
faddp   ST(1), ST(0)

fld     QWORD PTR [eax+16+328]
fmul    QWORD PTR [edx+328]
faddp   ST(2), ST(0)

fld     QWORD PTR [eax+16+320]
fmul    QWORD PTR [edx+320]
faddp   ST(1), ST(0)

fld     QWORD PTR [eax+16+312]
fmul    QWORD PTR [edx+312]
faddp   ST(2), ST(0)

fld     QWORD PTR [eax+16+304]
fmul    QWORD PTR [edx+304]
faddp   ST(1), ST(0)

fld     QWORD PTR [eax+16+296]
fmul    QWORD PTR [edx+296]
faddp   ST(2), ST(0)

fld     QWORD PTR [eax+16+288]
fmul    QWORD PTR [edx+288]
faddp   ST(1), ST(0)

fld     QWORD PTR [eax+16+280]
fmul    QWORD PTR [edx+280]
faddp   ST(2), ST(0)

fld     QWORD PTR [eax+16+272]
fmul    QWORD PTR [edx+272]
faddp   ST(1), ST(0)

fld     QWORD PTR [eax+16+264]
fmul    QWORD PTR [edx+264]
faddp   ST(2), ST(0)

fld     QWORD PTR [eax+16+256]
fmul    QWORD PTR [edx+256]
faddp   ST(1), ST(0)

fld     QWORD PTR [eax+16+248]
fmul    QWORD PTR [edx+248]
faddp   ST(2), ST(0)

fld     QWORD PTR [eax+16+240]
fmul    QWORD PTR [edx+240]
faddp   ST(1), ST(0)

fld     QWORD PTR [eax+16+232]
fmul    QWORD PTR [edx+232]
faddp   ST(2), ST(0)

fld     QWORD PTR [eax+16+224]
fmul    QWORD PTR [edx+224]
faddp   ST(1), ST(0)

fld     QWORD PTR [eax+16+216]
fmul    QWORD PTR [edx+216]
faddp   ST(2), ST(0)

fld     QWORD PTR [eax+16+208]
fmul    QWORD PTR [edx+208]
```

```
faddp   ST(1), ST(0)

fld     QWORD PTR [eax+16+200]
fmul    QWORD PTR [edx+200]
faddp   ST(2), ST(0)

fld     QWORD PTR [eax+16+192]
fmul    QWORD PTR [edx+192]
faddp   ST(1), ST(0)

fld     QWORD PTR [eax+16+184]
fmul    QWORD PTR [edx+184]
faddp   ST(2), ST(0)

fld     QWORD PTR [eax+16+176]
fmul    QWORD PTR [edx+176]
faddp   ST(1), ST(0)

fld     QWORD PTR [eax+16+168]
fmul    QWORD PTR [edx+168]
faddp   ST(2), ST(0)

fld     QWORD PTR [eax+16+160]
fmul    QWORD PTR [edx+160]

add     eax, 400                                ; 00000190H
add     ecx, 8
dec     esi

faddp   ST(1), ST(0)

fld     QWORD PTR [eax+16-248]
fmul    QWORD PTR [edx+152]
faddp   ST(2), ST(0)

fld     QWORD PTR [eax+16-256]
fmul    QWORD PTR [edx+144]
faddp   ST(1), ST(0)

fld     QWORD PTR [eax+16-264]
fmul    QWORD PTR [edx+136]
faddp   ST(2), ST(0)

fld     QWORD PTR [eax+16-272]
fmul    QWORD PTR [edx+128]
faddp   ST(1), ST(0)

fld     QWORD PTR [eax+16-280]
fmul    QWORD PTR [edx+120]
faddp   ST(2), ST(0)

fld     QWORD PTR [eax+16-288]
fmul    QWORD PTR [edx+112]
faddp   ST(1), ST(0)

fld     QWORD PTR [eax+16-296]
fmul    QWORD PTR [edx+104]
faddp   ST(2), ST(0)

fld     QWORD PTR [eax+16-304]
fmul    QWORD PTR [edx+96]
faddp   ST(1), ST(0)

fld     QWORD PTR [eax+16-312]
fmul    QWORD PTR [edx+88]
faddp   ST(2), ST(0)
```

```
        fld     QWORD PTR [eax+16-320]
        fmul    QWORD PTR [edx+80]
        faddp   ST(1), ST(0)

        fld     QWORD PTR [eax+16-328]
        fmul    QWORD PTR [edx+72]
        faddp   ST(2), ST(0)

        fld     QWORD PTR [eax+16-336]
        fmul    QWORD PTR [edx+64]
        faddp   ST(1), ST(0)

        fld     QWORD PTR [eax+16-344]
        fmul    QWORD PTR [edx+56]
        faddp   ST(2), ST(0)

        fld     QWORD PTR [eax+16-352]
        fmul    QWORD PTR [edx+48]
        faddp   ST(1), ST(0)

        fld     QWORD PTR [eax+16-360]
        fmul    QWORD PTR [edx+40]
        faddp   ST(2), ST(0)

        fld     QWORD PTR [eax+16-368]
        fmul    QWORD PTR [edx+32]
        faddp   ST(1), ST(0)

        fld     QWORD PTR [eax+16-376]
        fmul    QWORD PTR [edx+24]
        faddp   ST(2), ST(0)

        fld     QWORD PTR [eax+16-416]
        fmul    QWORD PTR [edx-16]
        faddp   ST(1), ST(0)

        fld     QWORD PTR [eax+16-384]
        fmul    QWORD PTR [edx+16]
        faddp   ST(2), ST(0)

        fld     QWORD PTR [eax+16-408]
        fmul    QWORD PTR [edx-8]
        faddp   ST(1), ST(0)

        fld     QWORD PTR [eax+16-392]
        fmul    QWORD PTR [edx+8]
        faddp   ST(2), ST(0)

        fld     QWORD PTR [edx]
        fmul    QWORD PTR [eax+16-400]
        faddp   ST(1), ST(0)

        fadd    QWORD PTR [ecx]
        faddp   ST(1), ST(0)
        fstp    QWORD PTR [ecx]

        jne     COL

        add     edx, 400                                ; 00000190H
        dec     edi
        jne     ROW

        pop     edi
        pop     esi
        pop     ebx
        ret     4
}
```

```
}

void    mul_block_size( block_t blk_c, const block_t blk_a, const block_t blk_b, const int row, const int colrow, const
{
        int     ii, jj, kk;

        for ( ii = 0 ; ii < row ; ++ii )
        {
                for ( jj = 0 ; jj < col ; ++jj )
                {
                        for ( kk = 0 ; kk < colrow ; ++kk )
                                blk_c[ii][jj] += blk_a[ii][kk] * blk_b[jj][kk];
                }
        }
}

/*****************************************************************************/

void    clear_matrix( matrix_t m )
{
        int     i, j;

        for ( i = 0 ; i < DIM ; ++i )
                for ( j = 0 ; j < DIM ; ++j )
                        m[i][j] = 0.0;
}

void    clear_matrix_size( matrix_t m, const int row, const int col )
{
        int     i, j;

        for ( i = 0 ; i < row ; ++i )
                for ( j = 0 ; j < col ; ++j )
                        m[i][j] = 0.0;
}

void    rand_matrix( matrix_t m )
{
        int     i, j;

        for ( i = 0 ; i < DIM ; ++i )
                for ( j = 0 ; j < DIM ; ++j )
                        m[i][j] = ( scalar_t )( ( rand() & 0x3fffffff ) + 1 ) / 1000.0;
}

void    rand_matrix_size( matrix_t m, const int row, const int col )
{
        int     i, j;

        for ( i = 0 ; i < row ; ++i )
                for ( j = 0 ; j < col ; ++j )
                        m[i][j] = ( scalar_t )( ( rand() & 0x3fffffff ) + 1 ) / 1000.0;
}

/*****************************************************************************/

void    mul_matrix( matrix_t c, const matrix_t a, const matrix_t b )
{
        int     i, j, k;

        clear_matrix( c );

        for ( i = 0 ; i < DIM ; ++i )
                for ( j = 0 ; j < DIM ; ++j )
                        for ( k = 0 ; k < DIM ; ++k )
                                c[i][j] += a[i][k] * b[k][j];
}
```

```
void    mul_matrix_size( matrix_t c, const matrix_t a, const matrix_t b, const int dim )
{
        int     i, j, k;

        clear_matrix_size( c, dim, dim );

        for ( i = 0 ; i < dim ; ++i )
                for ( j = 0 ; j < dim ; ++j )
                        for ( k = 0 ; k < dim ; ++k )
                                c[i][j] += a[i][k] * b[k][j];
}

/*****************************************************************************/

void    mul_matrix_block( matrix_t c, const matrix_t a, const matrix_t b )
{
        static blockarray_t     blk_a, blk_b;
        static block_t  blk_c;

        int     i, j, k;

        for ( i = 0 ; i < BLOCK_NUM ; ++i )
        {
                for ( j = 0 ; j < BLOCK_NUM ; ++j )
                {
                        copy_to_block( blk_a[i][j], *( matrix_t * )&a[ i * BLOCK_DIM ][ j * BLOCK_DIM ] );
                        copy_to_block_transpose( blk_b[j][i], *( matrix_t * )&b[ i * BLOCK_DIM ][ j * BLOCK_DIM ] );
                }
        }

        for ( i = 0 ; i < BLOCK_NUM ; ++i )
        {
                for ( j = 0 ; j < BLOCK_NUM ; ++j )
                {
                        clear_block( blk_c );

                        for ( k = 0 ; k < BLOCK_NUM ; ++k )
                                mul_block_asm( blk_c, blk_a[i][k], blk_b[j][k] );       /******/

                        copy_from_block( *( matrix_t * )&c[ i * BLOCK_DIM ][ j * BLOCK_DIM ], blk_c );
                }
        }
}

void    mul_matrix_block_size( matrix_t c, const matrix_t a, const matrix_t b, const int dim )
{
        static blockarray_t     blk_a, blk_b;
        static block_t  blk_c;

        const int       bn = dim / BLOCK_DIM, br = dim - bn * BLOCK_DIM;
        int     i, j, k;

        for ( i = 0 ; i < bn ; ++i )
        {
                for ( j = 0 ; j < bn ; ++j )
                {
                        copy_to_block_array( blk_a, a, i, j );
                        copy_to_block_array_transpose( blk_b, b, i, j );
                }
        }

        if ( br > 0 )
        {
                /* 右端 */
                j = bn;
                for ( i = 0 ; i < bn ; ++i )
```

```
{
        copy_to_block_array_size( blk_a, a, i, j, BLOCK_DIM, br );
        copy_to_block_array_transpose_size( blk_b, b, i, j, BLOCK_DIM, br );
}

/* 下端 */
i = bn;
for ( j = 0 ; j < bn ; ++j )
{
        copy_to_block_array_size( blk_a, a, i, j, br, BLOCK_DIM );
        copy_to_block_array_transpose_size( blk_b, b, i, j, br, BLOCK_DIM );
}

/* 右下隅 */
i = j = bn;
copy_to_block_array_size( blk_a, a, i, j, br, br );
copy_to_block_array_transpose_size( blk_b, b, i, j, br, br );


for ( i = 0 ; i < bn ; ++i )
{
        for ( j = 0 ; j < bn ; ++j )
        {
                clear_block( blk_c );
                for ( k = 0 ; k < bn ; ++k )
                        mul_block_asm( blk_c, blk_a[i][k], blk_b[j][k] );       /******/
                mul_block_size( blk_c, blk_a[i][k], blk_b[j][k], BLOCK_DIM, br, BLOCK_DIM );

                copy_from_block_to( c, blk_c, i, j );
        }
}


/* 右端 */
j = bn;
for ( i = 0 ; i < bn ; ++i )
{
        clear_block_size( blk_c, BLOCK_DIM, br );

        for ( k = 0 ; k < bn ; ++k )
                mul_block_size( blk_c, blk_a[i][k], blk_b[j][k], BLOCK_DIM, BLOCK_DIM, br );
        mul_block_size( blk_c, blk_a[i][k], blk_b[j][k], BLOCK_DIM, br, br );

        copy_from_block_size_to( c, blk_c, i, j, BLOCK_DIM, br );
}

/* 下端 */
i = bn;
for ( j = 0 ; j < bn ; ++j )
{
        clear_block_size( blk_c, br, BLOCK_DIM );

        for ( k = 0 ; k < bn ; ++k )
                mul_block_size( blk_c, blk_a[i][k], blk_b[j][k], br, BLOCK_DIM, BLOCK_DIM );
        mul_block_size( blk_c, blk_a[i][k], blk_b[j][k], br, br, BLOCK_DIM );

        copy_from_block_size_to( c, blk_c, i, j, br, BLOCK_DIM );
}

/* 右下隅 */
i = j = bn;
clear_block_size( blk_c, br, br );

for ( k = 0 ; k < bn ; ++k )
        mul_block_size( blk_c, blk_a[i][k], blk_b[j][k], br, BLOCK_DIM, br );
mul_block_size( blk_c, blk_a[i][k], blk_b[j][k], br, br, br );
```

```
                copy_from_block_size_to( c, blk_c, i, j, br, br );
        }
        else
        {
                for ( i = 0 ; i < bn ; ++i )
                {
                        for ( j = 0 ; j < bn ; ++j )
                        {
                                clear_block( blk_c );
                                for ( k = 0 ; k < bn ; ++k )
                                        mul_block_asm( blk_c, blk_a[i][k], blk_b[j][k] );       /******/

                                copy_from_block_to( c, blk_c, i, j );
                        }
                }
        }
}

/*****************************************************************************/

int     verify_mul_matrix( mul_matrix_t mul1, mul_matrix_t mul2 )
{
        static matrix_t a, b, cu1, cd1, cu2, cd2;
        int     i, j;

        rand_matrix( a );
        rand_matrix( b );

        up();
        ( *mul1 )( cu1, a, b );
        ( *mul2 )( cu2, a, b );

        down();
        ( *mul1 )( cd1, a, b );
        ( *mul2 )( cd2, a, b );

        for ( i = 0 ; i < DIM ; ++i )
                for ( j = 0 ; j < DIM ; ++j )
                        if ( cu1[i][j] < cd2[i][j] || cu2[i][j] < cd1[i][j] )
                                return 0;

        return 1;
}

int     verify_mul_matrix_size( mul_matrix_size_t mul1, mul_matrix_size_t mul2, const int dim )
{
        static matrix_t a, b, cu1, cd1, cu2, cd2;
        int     i, j;

        rand_matrix_size( a, dim, dim );
        rand_matrix_size( b, dim, dim );

        up();
        ( *mul1 )( cu1, a, b, dim );
        ( *mul2 )( cu2, a, b, dim );

        down();
        ( *mul1 )( cd1, a, b, dim );
        ( *mul2 )( cd2, a, b, dim );

        for ( i = 0 ; i < dim ; ++i )
                for ( j = 0 ; j < dim ; ++j )
                        if ( cu1[i][j] < cd2[i][j] || cu2[i][j] < cd1[i][j] )
                                return 0;

        return 1;
}
```

```
/*****************************************************************************/

void    test_mul_matrix( mul_matrix_t mul )
{
        static matrix_t a, b, c;

        const int       loopcount = 10;
        int             i;
        clock_t total_clk;

        printf( "constant size version\n" );

        total_clk = 0;
        for ( i = 0 ; i < loopcount ; ++i )
        {
                clock_t start_clk, end_clk, clk;

                rand_matrix( a );
                rand_matrix( b );

                start_clk = clock();
                ( *mul )( c, a, b );
                end_clk = clock();

                clk = end_clk - start_clk;
                total_clk += clk;

                printf( "%f msec.\n", ( double )clk * 1000.0 / CLOCKS_PER_SEC );
        }

        printf( "average : %f msec.\n\n", ( double )total_clk * 1000.0 / CLOCKS_PER_SEC / loopcount );

        printf( "verifying... " );
        if ( verify_mul_matrix( mul_matrix, mul ) )
                printf( "ok.\n\n" );
        else
                printf( "error.\n\n" );

}

void    test_mul_matrix_size( mul_matrix_size_t mul, const int dim )
{
        static matrix_t a, b, c;

        const int       loopcount = 10;
        int             i;
        clock_t total_clk;

        printf( "variable size version\n" );

        total_clk = 0;
        for ( i = 0 ; i < loopcount ; ++i )
        {
                clock_t start_clk, end_clk, clk;

                rand_matrix_size( a, dim, dim );
                rand_matrix_size( b, dim, dim );

                start_clk = clock();
                ( *mul )( c, a, b, dim );
                end_clk = clock();

                clk = end_clk - start_clk;
                total_clk += clk;

                printf( "%f msec.\n", ( double )clk * 1000.0 / CLOCKS_PER_SEC );
```

```
        }

        printf( "average : %f msec.\n\n", ( double )total_clk * 1000.0 / CLOCKS_PER_SEC / loopcount );

        printf( "verifying... " );
        if ( verify_mul_matrix_size( mul_matrix_size, mul, dim ) )
                printf( "ok.\n\n" );
        else
                printf( "error.\n\n" );

}

/*****************************************************************************/

int     __cdecl main( void )
{
        srand( time( NULL ) );

        test_mul_matrix( mul_matrix );
        test_mul_matrix( mul_matrix_block );
        test_mul_matrix_size( mul_matrix_block_size, 1000 );

        return 0;
}

/*****************************************************************************/
```