数値計算レポート 2

G99P057-3
斎藤卓也

出題日：2001 年 5 月 21 日
提出期限：2001 年 6 月 2 日正午
提出日：2001 年 6 月 2 日午前

1 問題

1000x1000 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。尚，自分の計算機環境を示し，最も早い計算時間を出したものは試験を免除する（試験の点の配分分だけ満点をあげる)。

2 計算機環境

表 1: コンピュータ

モデル名	自作マシン
CPU	Intel Pentium 4 1.4GHz (Willamette)
2 次キャッシュ	256KB
Chip Set	Intel 850
メモリ	Direct RDRAM 128MB
HDD	80GB
OS	Microsoft Windows 2000 + Service Pack 1
C Compiler	Microsoft Visual C++ 6.0 + Service Pack 5 + Processor Pack

3 高速化への道程

私は、上記のコンピュータ環境でプログラムを作り、速度を上げる努力をした。他にも celeron を搭載したマシンもあるのだが、Pentium 4 と celeron では、キャッシュの量もコア内部の構造も違うため、celeron で高速化したプログラムは Pentium 4 では逆に遅くなってしまったり、また、その逆になったりなどの現象が起きたため、Pentium 4 でもっとも

速くなるようにプログラムを作った。以降、Pentium 4 用に最適化した主な経緯を記すことにする。

まず、大石先生のホームページからリンクされている「行列高速化プロジェクト」なるものを参考にし、ほぼそのプログラムと同じようなことをやってみた。

具体的には、計算の内積化とブロック化である。ここまでやると、ほぼ掲載されているプログラムと同じ程度の実行時間が得られた。そこで、これをベースにして高速化することを考えた。

まず最初に試みたのは、行列 a と b の各要素を掛けている部分、例えば、

c[y][x] = a[y][k] * b[k][x];

のようなことをしている部分を、そのまま x87 FPU の命令をアセンブラで記述してみた。すると …!

これが驚いたことに、異常に遅いのである。高速化するどころか、大幅に速度が低下してしまったのである！C で最適化したときには、celeron 600MHz で 200 MFLOPS 程出ていたのが、なんと “ 2” MFLOPS になってしまったのである !! この速度低下は尋常ではない。それにつけても遅すぎる！

最初、私は Visual C++のコンパイラが吐き出してくるオブジェクト・コードなど、所詮 C の文法を機械語レベルに自動変換したものであるから、最適化などしても余り効率の良いものではなく、人間がちょっと考えてアセンブラで書いてやれば簡単にその速度を上回ることができるだろう、と高を括っていた。だが、実際にはそんなに甘いものではなかったのだ。

そこで、仕方なく x87 FPU を使った、なるべく効率の良いアセンブラ・プログラムを何度も考え、改良を加えて行った。fxch 命令を使い、レジスタ・リネーミングを活用すると非常に高速になる、という Intel の最適化マニュアルなども参考にした。P6 アーキテクチャにおける Pentium スケジューリングや、x87 命令がどれとどれをどの順で実行させると、並列実行が可能になり、何クロックでどれだけ実行できるのかまで考慮した（なぜ P6 アーキテクチャなのかというと、Willamette プロセッサ用の x87 FPU 命令最適化の本格的な資料が無いからである。ここが歯がゆいところだ）。

ここまでやると、さすがに高速になり、C で最適化したものとも遜色のないレベルまで高速化に成功した。といっても、非常に多くの苦労をした割には、結局何も考えずに C で書いて「Win32 Release」（最適化）を選んだものと速度的には変わりないのだが … 。

Microsoft Visual C++ 6.0 の最適化は大したものである。アセンブラレベルで人間が頭をひねりまくっても、そう簡単にはかなわないほどのコードを自動的に生成してくれるのである。

ただ、インライン・アセンブラを使用すると、その周辺部分の C コードにも影響を与え、その部分が最適化されなくなってしまうので、それでアセンブラ版は結局遅くなっているということも考えられるのだが、それにしても大したものだと思う。

3.1 Streaming SIMD Extensions 2

というわけで、x87 FPU の命令を使っての高速化は、ある程度のところで諦めた。問題は、私の CPU Pentium 4 に搭載されている Streaming SIMD Extensions 2 (SSE2) 命

令セットである。これを使うと、１回の掛け算命令 (mulpd) の実行で、２つの倍精度浮動小数点数を掛け合わせることができるのだ。こんなに良いものがある以上、x87 FPU で最適化などしていても仕方がない。今までのは高速化の序の口に過ぎない。ここからが本番だ！

しかし、これもまた当然のことながら、単に C 言語のみでプログラムを組んだだけでは SSE2 の命令を使うことは出来ない。その部分をインライン・アセンブラを使って機械語レベルで直接プログラムしなければならないのである。となると、上述の x87 FPU の時と同様に、インライン・アセンブラが C 言語側の最適化を阻害し、思ったほどのパフォーマンスが得られない可能性がある。

とグチを言っていても始まらないので、早速 x87 FPU アセンブラ版を元にして、SSE2 命令を使用してプログラムしてみた、が … 。やはりというか何と言うか … 、遅い … 。

x87 FPU の時程ではないにせよ、この時点で C 言語版が約 500MFLOPS くらいの速度を出していたにもかかわらず、350MFLOPS 程度しか出ないのである。これは x87 FPU アセンブラ版よりも遅い。

SSE2 はそもそも、倍精度浮動小数点演算を高速に行うために用意された命令なので、こんな筈はない。何か速度が遅くなる原因がある筈だ。そこで、Intel の Pentium 4 Optimization Manual なる文献を参照することにした。

その文献によると、まず第一に、SSE2 の倍精度浮動小数点演算ロード命令では、読み込むデータのメモリ上でのアライメントを１６バイト境界に合わせると非常に高速にメモリから読み込みが出来るようになるとある。また、キャッシュの制御を考えると、メモリのアライメントは３２バイト境界に合わせるほうが良いだろう。

また、SSE2 では倍精度浮動小数点数を２つの連続したデータとして扱うので、今のままでは都合が悪い。行列 a,b の乗算を考えた場合、行列 a の方は２つの要素を連続したアドレスから取り出せるので問題ないが、行列 b の方は、次のデータが次の行になってしまうので、一度でデータを読み込めず、効率が悪い。

そこで、行列 b は内部表現では転地行列化することにした。こうすれば、行列 a も行列 b も、１度に２つの連続したデータを読み込むことが出来るようになる。また、この配置はキャッシュを有効に活用するのにも役立つ筈である。

以上より、配列の取り方を工夫し、メモリのアライメントを３２バイト境界に合わせて確保するように変更した。また、行列 b は内部表現では転地行列化することにした。そして、SSE2 命令の方も、アライメントが合っているデータへのアクセス専用命令に変更した。すると …!

速い！遂に３秒台の壁を突破し、２秒台へと突入した！速いっ！

しかし、まだまだである。理論的には２倍の速度が出る（もちろん、浮動小数点演算以外何もしていない訳ではないから、どう頑張っても２倍にはならないのだが … ）はずであるから、理論的には１秒台も夢ではない！

そこで、for 文の最適化なども行い、XMM レジスタ同士の掛け算（行列の要素の掛け算）以外の、アドレスを計算するための掛け算命令をアセンブラ内で１個だけになるように減少させた（この１個を取り除くのは難しい。ループを逆にすれば取れるが、そうするとキャッシュミスが起きる）。SSE2 はもちろん、それ以外の命令の実行順序もスケジューリングを考えて全て煮詰めなおし、レジスタを有効に活用し、徹底的に高速に実行できる

ように改良しまくった。

これらの細かい改良を加え、最終的には遂に夢の 1.5 秒台（約 1300 MFLOPS）にまで高速化に成功した !! これは速い ! 圧倒的に速い ! さすが Streaming SIMD Extensions 2!! である。

1.5 秒台に突入してからも、SSE2 アセンブラ・プログラムの関数を行列計算メイン関数内にインライン化してみたり、結果の足し算を x87 FPU と SIMD とで並列実行させてみたり、いろいろと細かい改良を試してみたが、これ以上の速度向上には至らなかった。というより、逆に遅くなったりした。

ここで、Pentium III などでは、prefetch 等のキャッシュ制御命令を使うことにより、さらなる高速化が可能かもしれないが、Pentium 4 はキャッシュのプリフェッチ動作も CPU が先読みして自動的にやってしまうため、いろいろ試してみたのだが、効果は無く、かえって遅くなることの方が多かった。

おそらく、このアルゴリズムでの行列乗算では、Pentium 4 1.4GHz でのほぼ限界値に達したのではないかと考えている。

約 1.3GFLOPS に到達したということは、CPU のクロック周波数が 1.4GHz であるから、ほぼ 1 クロックにつき 1 個の浮動小数点数演算をしていることになる。実際には浮動小数点数をメモリから「読み込み」、「乗算」、「加算」の処理を繰り返し、さらにメモリと大量のデータ転送を行っている訳だから、CPU の性能から考えても限界に近いと思われる。

3.2 更なる高速化へ向けて

これ以上の高速化の為には、キャッシュの動きを完全に把握し、それを最適になるよう直接制御したり、また、乗算アルゴリズム自体の見直しなどが考えられる。

行列の配列をを SSE2 演算用にデータ順を完全に並べ替えて、それで実行させたらもう少し高速化が可能なような気もする。

そもそも、Windows 2000 というプリエンプティブなマルチタスク OS 上で実行しているのが良くない。当然、他の処理に CPU の時間が割かれているわけだ。よって、コンピュータを MS-DOS で起動し、プログラムの最初で CPU をリアルモードから 386 プロテクトモードへと移行し、後はひたすら計算する、という風にすれば当然もっと速くなるだろう。

しかし、時間的制約があるため、今回はこの辺で諦めることにしよう。

4　高速化に使った主なテクニック

前節にて、高速化への主な道筋を示したが、ここでは、行列乗算を高速に行うために、凝らした工夫を説明する。

4.1　アンローリング

例えば、下のように足し算を複数に分けて計算させ、後で一つにまとめている。これが案外効果がある。一見、沢山変数を用意せずに、一度で掛け算を足して代入してしまえば速いように思われるが、なぜかこのように一度変数に代入した方が速くなる。

```
for (y2=0; y2 < block; y2++) {
    s1 = 0.0;
    s2 = 0.0;
    s3 = 0.0;
    s4 = 0.0;
    s5 = 0.0;
    for (k2=0; k2 < block; k2 += 5) {
        s1 += a2[y2][k2]   * b2[x2][k2];
        s2 += a2[y2][k2+1] * b2[x2][k2+1];
        s3 += a2[y2][k2+2] * b2[x2][k2+2];
        s4 += a2[y2][k2+3] * b2[x2][k2+3];
        s5 += a2[y2][k2+4] * b2[x2][k2+4];
    }
    c2[y2][x2] += s1+s2+s3+s4+s5;
}
```

似たようなことは、x87 FPU 版や SSE2 版のプログラムでも行っている。SSE2 版では、特に１０個の浮動小数点の掛け算を１回のループで行うようにしている。下に SSE2 版の演算部分のソースを示す。

```
Loop1:movapd    xmm2, xmmword ptr [esi][8*eax]
      mulpd     xmm2, xmmword ptr [edi][8*eax]
      movapd    xmm3, xmmword ptr [esi][8*eax]+10h
      mulpd     xmm3, xmmword ptr [edi][8*eax]+10h
      movapd    xmm4, xmmword ptr [esi][8*eax]+20h
      mulpd     xmm4, xmmword ptr [edi][8*eax]+20h
      movapd    xmm5, xmmword ptr [esi][8*eax]+30h
      mulpd     xmm5, xmmword ptr [edi][8*eax]+30h
      movapd    xmm6, xmmword ptr [esi][8*eax]+40h
      mulpd     xmm6, xmmword ptr [edi][8*eax]+40h

      addpd     xmm2, xmm3
```

```
addpd     xmm4, xmm5
addpd     xmm1, xmm6
addpd     xmm2, xmm4

add       eax, ebx

addpd     xmm1, xmm2

jnz       Loop1
```

基本的に、アンローリングによる高速化のポイントは、ループの部分のオーバーヘッドを取り除けるところにある。それ以外にも、複数の演算を１度で行ったほうが、機械語を前の命令とペアになるように配置できたりなどするので、高速化できる。

4.2 行列のブロック化

キャッシュメモリを有効に活用するために、一度に演算する行列のサイズを 100×100 程度に小さく出来るようにブロック化した。ブロック化する際に、どのように配列を確保するかが問題となる。

例えば、

```
double A[1000][1000];
```

のように配列を宣言することも出来るのだが、これだとブロック化したときに都合が悪い。

というのも、例えば 10×10 分割したとする。この場合、一番最初の 100×100 行列を演算するためには、その部分を別の 100×100 サイズの行列（配列）にコピーしてから、演算を行わなければ意味が無い。ということは、$10 \times 10 = 100$ 回配列のコピーを行列 A と行列 B の両方で行わなければならない。これは非常に効率が悪い。

そこで、私はメモリの取り方を工夫することにより、別の配列にコピーすることなく、しかも連続した 100×100 などの行列用メモリ空間を取れるようにした。

具体的には、まず、

```
double**** a;
```

のように、４重のポインタを用意する。これは、a[][][][] というように、４次元配列として扱うことができる。a[行ブロック][列ブロック] のようにすることにより、どのブロックなのかを指定できる。

また、a[行ブロック][列ブロック][行][列] のように、４つ指定すると、配列の１つ１つの要素を特定できる。

ここで、ブロック化して、しかもメモリをブロック的に、キャッシュに収まるように確保するために、****a について、下のように BlockNum 個のメモリ空間を確保する。

```
a = (double****) malloc (sizeof(double***) * BlockNum);
```

次に、***a についても同様に BlockNum 個メモリ空間を確保する。これは、上で確保したメモリ１つ１つに対して全て行う。

```
a[i] = (double***) malloc (sizeof(double**) * BlockNum);
```

そして、ここからがポイントである。まず、以上までと同じようにして、**a について BlockWidth 個分のメモリを確保する。

```
a[i][j] = (double**) malloc (sizeof(double*) * BlockWidth);
```

その後、

```
a[i][j][0] = (double*) malloc (sizeof(double) * BlockWidth * BlockWidth);
```

というようにして、BlockWIdth * BlockWidth サイズ分のメモリを一度に確保してしまう。こうすることにより、BlockWidth * BlockWidth だけのサイズの連続したメモリ領域が確保できる。この後、

```
for (k=1; k < BlockWidth; k++) {
    a[i][j][k] = a[i][j][k-1] + BlockWidth;
}
```

のようにして、３つ目までの [] のアドレスを指定してやれば、後は全部 a[][][][] と４次元配列として使用することが出来る。

また、a[][][0] と最初の２つ＋０を指定してやると、この値は、そのブロックの先頭アドレスを示すようになる。こうすれば、ブロック化したデータを処理するために、毎回小さな配列にデータをコピーしてやる必要はなくなる。

4.3 メモリのアライメント

メモリはただそのサイズだけ確保すれば良いというものではない。メモリとのアクセス速度を考慮すると、メモリが確保される開始アドレスのアライメントを１６バイト境界や３２バイト境界に合わせてやる必要がある。実際、SSE2 では１６バイト境界に合っていないと大幅にメモリ転送が遅くなり、また、キャッシュメモリは３２バイト境界に合っていないとパフォーマンスが低下する。

そこで、私のプログラムでは３２バイト境界にメモリアライメントを合わせることにした。具体的には、以下のようにして３２バイト境界にアライメントを合わせている。

```
ar[i][j][0] = (double*) malloc (sizeof(double) * BlockWidth * BlockWidth + 31);
a[i][j][0] = (double*) (((unsigned)ar[i][j][0] + 31) & (-32));
```

まず、最初に３１バイト分余計にメモリを確保しておき、確保されたメモリの開始アドレス＋３１バイトの値と 0xffffffe0(=-32) との論理積を取り、それを実際の開始アドレスとして設定することにより、これを実現している。

4.4 行列 b の転地行列化

SSE2 の場合、１度のロード命令で２つの倍精度浮動小数点数（１６バイト）を取り込むことができる。それには、この２つの倍精度浮動小数点数がメモリ上で隣り合っている必要がある。そうでないと、別々に取り込むことになり、パフォーマンス低下を招く。

これを解消するために、行列 b の方を転地行列化した。これにより、行列 b への２つの要素のアクセスも隣り合ったメモリ上に存在するようになる。

また、これにより、行列 b のデータを連続したメモリ領域から取り出すことになるので、結果的にキャッシュメモリの有効利用にもつながるものと考えられる。

4.5 Streaming SIMD Extensions 2 (SSE2)

これは、Willamette プロセッサに採用された、２つの倍精度浮動小数点演算を１回の命令実行で可能にする命令群である。

```
movapd  xmm1, xmmword ptr [esi]
mulpd   xmm1, xmmword ptr [edi]
```

最初の movapd 命令で、xmm1 レジスタ（xmm レジスタは１２８ビット）に esi レジスタの示すアドレスから１６バイト（倍精度浮動小数点２個分）のデータが一度でロードでき、また mulpd 命令で、xmm1 レジスタの内容と edi レジスタの示すアドレスのメモリの内容との積をとることが出来る。

つまり、２命令で、

$$a = a * b$$
$$c = c * d$$

を行うような効率の良いコーディングが可能になる。私のプログラムでは、この機能を使い、どれだけ高速化できるかに主眼を置き、プログラムを作成した。

5 作成したプログラム

以下に作成したプログラムを記す。SSE2 版、C 言語版、x87 FPU 版の 3 つがある。

5.1 SSE2 版

```
/*

  行列高速乗算プログラム
  with Streaming SIMD Extensions 2

  2001-06-01
  Programmed by Takuya Saito

*/

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

#define    BlockWidth    130
#define    BlockWidth8   (BlockWidth * 8)
#define    BlockNum      (1000 / BlockWidth + (1000 % BlockWidth == 0 ? 0 : 1))

/*
     関数のプロトタイプ宣言
*/
void init_matrix(double**** a, double**** b, double**** c);
void mul_matrix(double**** a, double**** b, double**** c);
void mul_matrix_sub(int, int, int, double* a, double* b, double* c);
int mflops(double flops, double sec);
void set_matrix_ac(int x, int y, double**** a, int value);
void set_matrix_b (int x, int y, double**** b, int value);
double get_matrix_ac(int x, int y, double**** a);
double get_matrix_b (int x, int y, double**** b);


int main()
{
    clock_t tic, toc;
    double sec, flops, ave_sec;
```

```
double ****a,  ****b,  ****c;
double ****ar, ****br, ****cr;
int  i, j, k;

flops = 2.0*1000*1000*1000;

/*
    a, b, c のメモリ領域確保（ar, br, crはreal pointer）
    （３２バイト境界にアライメントを合わせてメモリを確保する）
*/
a = (double****) malloc (sizeof(double***) * BlockNum);
b = (double****) malloc (sizeof(double***) * BlockNum);
c = (double****) malloc (sizeof(double***) * BlockNum);
ar = (double****) malloc (sizeof(double***) * BlockNum);
br = (double****) malloc (sizeof(double***) * BlockNum);
cr = (double****) malloc (sizeof(double***) * BlockNum);
for (i=0; i < BlockNum; i++) {
    a[i] = (double***) malloc (sizeof(double**) * BlockNum);
    b[i] = (double***) malloc (sizeof(double**) * BlockNum);
    c[i] = (double***) malloc (sizeof(double**) * BlockNum);
    ar[i] = (double***) malloc (sizeof(double**) * BlockNum);
    br[i] = (double***) malloc (sizeof(double**) * BlockNum);
    cr[i] = (double***) malloc (sizeof(double**) * BlockNum);
    for(j=0; j < BlockNum; j++) {
        a[i][j] = (double**) malloc (sizeof(double*) * BlockWidth);
        b[i][j] = (double**) malloc (sizeof(double*) * BlockWidth);
        c[i][j] = (double**) malloc (sizeof(double*) * BlockWidth);
        ar[i][j] = (double**) malloc (sizeof(double*) * BlockWidth);
        br[i][j] = (double**) malloc (sizeof(double*) * BlockWidth);
        cr[i][j] = (double**) malloc (sizeof(double*) * BlockWidth);
        ar[i][j][0] = (double*) malloc (sizeof(double) * BlockWidth * BlockWidth + 31
        br[i][j][0] = (double*) malloc (sizeof(double) * BlockWidth * BlockWidth + 31
        cr[i][j][0] = (double*) malloc (sizeof(double) * BlockWidth * BlockWidth + 31

        a[i][j][0] = (double*) (((unsigned)ar[i][j][0] + 31) & (-32));
        b[i][j][0] = (double*) (((unsigned)br[i][j][0] + 31) & (-32));
        c[i][j][0] = (double*) (((unsigned)cr[i][j][0] + 31) & (-32));

        for (k=1; k < BlockWidth; k++) {
            a[i][j][k] = a[i][j][k-1] + BlockWidth;
            b[i][j][k] = b[i][j][k-1] + BlockWidth;
```

```
            c[i][j][k] = c[i][j][k-1] + BlockWidth;
        }
    }
}

printf("Streaming SIMD Extensions 2 version\n\n");
printf("*** Type ***   Time[sec]    Speed[MFLOPS]\n");

ave_sec = 0;

for (i=0; i < 10; i++) {

    init_matrix(a, b, c);

    tic = clock();
    mul_matrix(a, b, c);
    toc = clock();
    sec = (double)(toc - tic)/CLOCKS_PER_SEC;
    ave_sec += sec;

    printf("  Mul_SSE2   %10.2f %13d\n", sec, mflops(flops, sec));

}

ave_sec /= 10.;

printf("----------------------------------------\n");
printf("  Average    %10.2f %13d\n", ave_sec, mflops(flops, ave_sec));

/*
    a, b, c, ar, br, cr のメモリ領域開放

*/
for (i=0; i < BlockNum; i++) {
    for (j=0; j < BlockNum; j++) {
        free(ar[i][j][0]);
        free(br[i][j][0]);
        free(cr[i][j][0]);
        free(a[i][j]);
        free(b[i][j]);
        free(c[i][j]);
```

```
            free(ar[i][j]);
            free(br[i][j]);
            free(cr[i][j]);
        }
        free(a[i]);
        free(b[i]);
        free(c[i]);
        free(ar[i]);
        free(br[i]);
        free(cr[i]);
    }
    free(a);
    free(b);
    free(c);
    free(ar);
    free(br);
    free(cr);

    return 0;
}
int mflops(double flops, double sec)
{
    int mf;

    if (sec > 1.0e-14) mf = (int)(flops/sec/1000000);
    else mf = 0;

    return mf;
}
/*

   行列の初期化

*/
void init_matrix(double**** a, double**** b, double**** c)
{
    int i, j, k, l;

    srand( (unsigned)time( NULL ) );

    for (i=0; i < BlockNum; i++) {
```

```
        for (j=0; j < BlockNum; j++) {
            for (k=0; k < BlockWidth; k++) {
                for (l=0; l < BlockWidth; l++) {
                    a[i][j][k][l] = (double)rand()/1000.0;
                    b[i][j][k][l] = (double)rand()/1000.0;
                    c[i][j][k][l] = 0.;
                }
            }
        }
    }
}
/*

   行列 a, c に値をセットする関数

*/
void set_matrix_ac(int x, int y, double**** a, int value)
{
    int i, j, k, l;

    i = x / BlockWidth;
    j = y / BlockWidth;
    k = x % BlockWidth;
    l = y % BlockWidth;

    a[i][j][k][l] = value;
}
/*

   行列 b に値をセットする関数

*/
void set_matrix_b(int x, int y, double**** b, int value)
{
    int i, j, k, l;

    i = x / BlockWidth;
    j = y / BlockWidth;
    k = x % BlockWidth;
    l = y % BlockWidth;
```

```
    b[j][i][l][k] = value;
}
/*

   行列 a, c から値を取り出す関数

*/
double get_matrix_ac(int x, int y, double**** a)
{
    int i,j,k,l;

    i = x / BlockWidth;
    j = y / BlockWidth;
    k = x % BlockWidth;
    l = y % BlockWidth;

    return a[i][j][k][l];
}
/*

   行列 b から値を取り出す関数

*/
double get_matrix_b(int x, int y, double**** b)
{
    int i,j,k,l;

    i = x / BlockWidth;
    j = y / BlockWidth;
    k = x % BlockWidth;
    l = y % BlockWidth;

    return b[j][i][l][k];
}
/*

   行列の乗算メインルーチン

*/
void mul_matrix(double**** a, double**** b, double**** c)
{
```

```
    int x, y, k;
    int    nobk, rem;
    int block_x, block_y, block_k;

    nobk = 1000 / BlockWidth; /* number of block */
    rem  = 1000 % BlockWidth;

    if (rem == 0) {
        for (y=0; y < nobk; y++) {
            for (x=0; x < nobk; x++) {
                for (k=0; k < nobk; k++) {
                    mul_matrix_sub(BlockWidth, BlockWidth, BlockWidth,
                        a[y][k][0], b[x][k][0], c[y][x][0]);
                }
            }
        }
    } else {
        /* when rem != 0 */
        for (y=0; y <= nobk; y++) {

            if (y == nobk) block_y = rem;
            else block_y = BlockWidth;

            for (x=0; x <= nobk; x++) {

                if (x == nobk) block_x = rem;
                else block_x = BlockWidth;

                for (k=0; k <= nobk; k++) {

                    if (k == nobk) block_k = rem;
                    else block_k = BlockWidth;

                    mul_matrix_sub(block_x, block_y, block_k,
                                   a[y][k][0], b[x][k][0], c[y][x][0]);

                }
            }
        }
    }
}
```

```
/*

   Streaming SIMD Extensions 2 行列乗算ルーチン

*/
void mul_matrix_sub(int block_x, int block_y, int block_k,
                    double* a2, double* b2, double* c2)
{
    int x2, y2;
    int blockwidth;
    int block_k8;

    blockwidth = BlockWidth;
    block_k8   = block_k * 8;

    for (y2=0; y2 < block_y * BlockWidth * 8; y2 += BlockWidth * 8) {
        for (x2=0; x2 < block_x * 8; x2 += 8) {

            __asm {
                ;
                ;  esi = a2 + y
                ;  edi = b2 + x * BlockWidth
                ;
                mov        eax, dword ptr [x2]
                mov        edi, dword ptr [b2]      ; for edi
                mul        dword ptr [blockwidth]   ; eax * BlockWidth
                add        edi, eax                 ; for edi

                mov        esi, dword ptr [a2]      ; for esi
                add        esi, dword ptr [y2]      ; for esi
                ;
                ;  esi += block_k * 8
                ;  edi += block_k * 8
                ;
                mov        eax, dword ptr [block_k8]
                mov        ebx, 0ah                 ; ebx の増分
                add        esi, eax
                add        edi, eax
                ;
                ;  eax = - block_k
                ;
```

```
        mov         eax, dword ptr [block_k]
        xorpd       xmm1, xmm1              ; xmm1 = 0;
        neg         eax                     ; eax = - eax
        ;jz         Next1                   ; when eax==0
        ;
        ;  Streaming SIMD Extensions 2 Main Loop
        ;
  Loop1:movapd      xmm2, xmmword ptr [esi][8*eax]
        mulpd       xmm2, xmmword ptr [edi][8*eax]
        movapd      xmm3, xmmword ptr [esi][8*eax]+10h
        mulpd       xmm3, xmmword ptr [edi][8*eax]+10h
        movapd      xmm4, xmmword ptr [esi][8*eax]+20h
        mulpd       xmm4, xmmword ptr [edi][8*eax]+20h
        movapd      xmm5, xmmword ptr [esi][8*eax]+30h
        mulpd       xmm5, xmmword ptr [edi][8*eax]+30h
        movapd      xmm6, xmmword ptr [esi][8*eax]+40h
        mulpd       xmm6, xmmword ptr [edi][8*eax]+40h

        addpd       xmm2, xmm3
        addpd       xmm4, xmm5
        addpd       xmm1, xmm6
        addpd       xmm2, xmm4

        add         eax, ebx

        addpd       xmm1, xmm2

        jnz         Loop1

  Next1:movapd        xmm2, xmm1
        unpckhpd      xmm2, xmm2
        addsd         xmm1, xmm2

        mov           esi, dword ptr [c2]
        add           esi, dword ptr [y2]
        mov           eax, dword ptr [x2]

        addsd         xmm1, qword ptr [esi][eax]
        movsd         qword ptr [esi][eax], xmm1
    }
}
```

```
    }
}
```

5.2 C言語版

```
/*

  高速行列乗算プログラム1

  Programmed by Takuya Saito

*/
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>

#define    BlockWidth    100
#define    BlockNum    (1000 / BlockWidth + (1000 % BlockWidth == 0 ? 0 : 1))

/*  関数のプロトタイプ宣言  */
void init_matrix(double**** a, double**** b, double**** c);
void mul_matrix(double**** a, double**** b, double**** c);
int mflops(double flops, double sec);
void set_matrix_ac(int x, int y, double**** a, int value);
void set_matrix_b (int x, int y, double**** b, int value);
double get_matrix_ac(int x, int y, double**** a);
double get_matrix_b (int x, int y, double**** b);
/*

   main 関数

*/
int main()
{
    clock_t    tic, toc;
    double    sec, flops, ave_sec;
    double****    a;
    double****    b;
    double****    c;
    double****    ar;
    double****    br;
    double****    cr;
    int  i, j, k;
```

```
/*
    a, b, c のメモリ領域確保（ar, br, cr は real pointer）
    （３２バイト境界にアライメントを合わせてメモリを確保する）
*/
a = (double****) malloc (sizeof(double***) * BlockNum);
b = (double****) malloc (sizeof(double***) * BlockNum);
c = (double****) malloc (sizeof(double***) * BlockNum);
ar = (double****) malloc (sizeof(double***) * BlockNum);
br = (double****) malloc (sizeof(double***) * BlockNum);
cr = (double****) malloc (sizeof(double***) * BlockNum);
for (i=0; i < BlockNum; i++) {
    a[i] = (double***) malloc (sizeof(double**) * BlockNum);
    b[i] = (double***) malloc (sizeof(double**) * BlockNum);
    c[i] = (double***) malloc (sizeof(double**) * BlockNum);
    ar[i] = (double***) malloc (sizeof(double**) * BlockNum);
    br[i] = (double***) malloc (sizeof(double**) * BlockNum);
    cr[i] = (double***) malloc (sizeof(double**) * BlockNum);
    for(j=0; j < BlockNum; j++) {
        a[i][j] = (double**) malloc (sizeof(double*) * BlockWidth);
        b[i][j] = (double**) malloc (sizeof(double*) * BlockWidth);
        c[i][j] = (double**) malloc (sizeof(double*) * BlockWidth);
        ar[i][j] = (double**) malloc (sizeof(double*) * BlockWidth);
        br[i][j] = (double**) malloc (sizeof(double*) * BlockWidth);
        cr[i][j] = (double**) malloc (sizeof(double*) * BlockWidth);
        ar[i][j][0] = (double*) malloc (sizeof(double) * BlockWidth * BlockWidth + 31
        br[i][j][0] = (double*) malloc (sizeof(double) * BlockWidth * BlockWidth + 31
        cr[i][j][0] = (double*) malloc (sizeof(double) * BlockWidth * BlockWidth + 31

        a[i][j][0] = (double*) (((unsigned)ar[i][j][0] + 31) & (-32));
        b[i][j][0] = (double*) (((unsigned)br[i][j][0] + 31) & (-32));
        c[i][j][0] = (double*) (((unsigned)cr[i][j][0] + 31) & (-32));

        for (k=1; k < BlockWidth; k++) {
            a[i][j][k] = a[i][j][k-1] + BlockWidth;
            b[i][j][k] = b[i][j][k-1] + BlockWidth;
            c[i][j][k] = c[i][j][k-1] + BlockWidth;
        }
    }
}
ave_sec = 0;
```

```
init_matrix(a, b, c);

printf("*** Type ***   Time[sec]    Speed[MFLOPS]\n");

for (i=0; i < 10; i++) {
    tic = clock();
    mul_matrix(a, b, c);
    toc = clock();
    sec = (double)(toc - tic)/CLOCKS_PER_SEC;
    flops = 2.0*1000*1000*1000;
    printf(" multmats    %10.2f %13d\n", sec, mflops(flops, sec));
    ave_sec += sec;
}

ave_sec /= 10.0;

printf("----------------------------------------\n");
printf(" Average     %10.2f %13d\n",ave_sec, mflops(flops, ave_sec));

/*
    a, b, c, ar, br, cr のメモリ領域開放

*/
for (i=0; i < BlockNum; i++) {
    for (j=0; j < BlockNum; j++) {
        free(ar[i][j][0]);
        free(br[i][j][0]);
        free(cr[i][j][0]);
        free(a[i][j]);
        free(b[i][j]);
        free(c[i][j]);
        free(ar[i][j]);
        free(br[i][j]);
        free(cr[i][j]);
    }
    free(a[i]);
    free(b[i]);
    free(c[i]);
    free(ar[i]);
    free(br[i]);
    free(cr[i]);
```

```
    }
    free(a);
    free(b);
    free(c);
    free(ar);
    free(br);
    free(cr);

    return 0;
}
int mflops(double flops, double sec)
{
    int mf;

    if (sec > 1.0e-14) mf = (int)(flops/sec/1000000);
    else mf = 0;

    return mf;
}
/*

   行列の初期化

*/
void init_matrix(double**** a, double**** b, double**** c)
{
    int i, j, k, l;

    srand( (unsigned)time( NULL ) );

    for (i=0; i < BlockNum; i++) {
        for (j=0; j < BlockNum; j++) {
            for (k=0; k < BlockWidth; k++) {
                for (l=0; l < BlockWidth; l++) {
                    a[i][j][k][l] = (double)rand()/1000.0;
                    b[i][j][k][l] = (double)rand()/1000.0;
                    c[i][j][k][l] = 0.;
                }
            }
        }
    }
```

```
}
/*

   行列 a, c に値をセットする関数

*/
void set_matrix_ac(int x, int y, double**** a, int value)
{
    int i, j, k, l;

    i = x / BlockWidth;
    j = y / BlockWidth;
    k = x % BlockWidth;
    l = y % BlockWidth;

    a[i][j][k][l] = value;
}
/*

   行列 b に値をセットする関数

*/
void set_matrix_b(int x, int y, double**** b, int value)
{
    int i, j, k, l;

    i = x / BlockWidth;
    j = y / BlockWidth;
    k = x % BlockWidth;
    l = y % BlockWidth;

    b[j][i][l][k] = value;
}
/*

   行列 a, c から値を取り出す関数

*/
double get_matrix_ac(int x, int y, double**** a)
{
    int i,j,k,l;
```

```
    i = x / BlockWidth;
    j = y / BlockWidth;
    k = x % BlockWidth;
    l = y % BlockWidth;

    return a[i][j][k][l];
}
/*

   行列 b から値を取り出す関数

*/
double get_matrix_b(int x, int y, double**** b)
{
    int i,j,k,l;

    i = x / BlockWidth;
    j = y / BlockWidth;
    k = x % BlockWidth;
    l = y % BlockWidth;

    return b[j][i][l][k];
}
/*

   行列の乗算を行う関数

*/
void mul_matrix(double**** a, double**** b, double**** c)
{
    int    nobk, rem;
    double s1, s2, s3, s4, s5;
    int block, block_x, block_y, block_k;
    int x, y, k, x2, y2, k2;
    double **a2, **b2, **c2;

    block = BlockWidth;
    nobk = 1000 / BlockWidth; /* number of block */
    rem  = 1000 % BlockWidth;
```

```
if (rem == 0) {
    for (x=0; x < nobk; x++) {
        for (y=0; y < nobk; y++) {
            for (k=0; k < nobk; k++) {

                a2 = a[y][k];
                b2 = b[x][k];
                c2 = c[y][x];
                for (x2=0; x2 < block; x2++) {
                    for (y2=0; y2 < block; y2++) {
                        s1 = 0.0;
                        s2 = 0.0;
                        s3 = 0.0;
                        s4 = 0.0;
                        s5 = 0.0;
                        for (k2=0; k2 < block; k2+=5) {
                            s1 += a2[y2][k2]   * b2[x2][k2];
                            s2 += a2[y2][k2+1] * b2[x2][k2+1];
                            s3 += a2[y2][k2+2] * b2[x2][k2+2];
                            s4 += a2[y2][k2+3] * b2[x2][k2+3];
                            s5 += a2[y2][k2+4] * b2[x2][k2+4];
                        }
                        c2[y2][x2]   += s1+s2+s3+s4+s5;
                    }
                }
            }
        }
    }
} else {
    /* when rem != 0 */
    for (x=0; x <= nobk; x++) {

        if (x == nobk)
            block_x = rem;
        else
            block_x = BlockWidth;

        for (y=0; y <= nobk; y++) {

            if (y == nobk)
                block_y = rem;
```

```
                else
                    block_y = BlockWidth;

                for (k=0; k <= nobk; k++) {

                    if (k == nobk)
                        block_k = rem;
                    else
                        block_k = BlockWidth;

                    a2 = a[y][k];
                    b2 = b[x][k];
                    c2 = c[y][x];
                    for (x2=0; x2 < block_x; x2++) {
                        for (y2=0; y2 < block_y; y2++) {
                            s1 = 0.0;
                            s2 = 0.0;
                            s3 = 0.0;
                            s4 = 0.0;
                            s5 = 0.0;
                            for (k2=0; k2 < block_k; k2+=2) {
                                s1 += a2[y2][k2]   * b2[x2][k2];
                                s2 += a2[y2][k2+1] * b2[x2][k2+1];
                                s3 += a2[y2][k2+2] * b2[x2][k2+2];
                                s4 += a2[y2][k2+3] * b2[x2][k2+3];
                                s5 += a2[y2][k2+4] * b2[x2][k2+4];
                            }
                            c2[y2][x2] += s1+s2+s3+s4+s5;
                        }
                    }
                }
            }
        }
    }
}
```

5.3 x87 FPU 版

```
/*

   行列高速乗算プログラム
   x87 FPU アセンブラ版

   Programmed by Takuya Saito

*/


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

#define    BlockWidth    120
#define    BlockNum    (1000 / BlockWidth + (1000 % BlockWidth == 0 ? 0 : 1))

void init_matrix(double**** a, double**** b, double**** c);
void mul_matrix(double**** a, double**** b, double**** c);
void mul_matrix_sub(int, int, int, double** a, double** b, double** c);
int mflops(double flops, double sec);


int main()
{
    double sec, flops, ave_sec;
    clock_t tic, toc;

    double****    a;
    double****    b;
    double****    c;
    double****    ar;
    double****    br;
    double****    cr;
    int  i, j, k;

    /*
        a, b, c のメモリ領域確保（ar, br, cr は real pointer）
        （３２バイト境界にアライメントを合わせてメモリを確保する）
```

```
*/
a = (double****) malloc (sizeof(double***) * BlockNum);
b = (double****) malloc (sizeof(double***) * BlockNum);
c = (double****) malloc (sizeof(double***) * BlockNum);
ar = (double****) malloc (sizeof(double***) * BlockNum);
br = (double****) malloc (sizeof(double***) * BlockNum);
cr = (double****) malloc (sizeof(double***) * BlockNum);
for (i=0; i < BlockNum; i++) {
    a[i] = (double***) malloc (sizeof(double**) * BlockNum);
    b[i] = (double***) malloc (sizeof(double**) * BlockNum);
    c[i] = (double***) malloc (sizeof(double**) * BlockNum);
    ar[i] = (double***) malloc (sizeof(double**) * BlockNum);
    br[i] = (double***) malloc (sizeof(double**) * BlockNum);
    cr[i] = (double***) malloc (sizeof(double**) * BlockNum);
    for(j=0; j < BlockNum; j++) {
        a[i][j] = (double**) malloc (sizeof(double*) * BlockWidth);
        b[i][j] = (double**) malloc (sizeof(double*) * BlockWidth);
        c[i][j] = (double**) malloc (sizeof(double*) * BlockWidth);
        ar[i][j] = (double**) malloc (sizeof(double*) * BlockWidth);
        br[i][j] = (double**) malloc (sizeof(double*) * BlockWidth);
        cr[i][j] = (double**) malloc (sizeof(double*) * BlockWidth);
        ar[i][j][0] = (double*) malloc (sizeof(double) * BlockWidth * BlockWidth + 31
        br[i][j][0] = (double*) malloc (sizeof(double) * BlockWidth * BlockWidth + 31
        cr[i][j][0] = (double*) malloc (sizeof(double) * BlockWidth * BlockWidth + 31

        a[i][j][0] = (double*) (((unsigned)ar[i][j][0] + 31) & (-32));
        b[i][j][0] = (double*) (((unsigned)br[i][j][0] + 31) & (-32));
        c[i][j][0] = (double*) (((unsigned)cr[i][j][0] + 31) & (-32));

        for (k=1; k < BlockWidth; k++) {
            a[i][j][k] = a[i][j][k-1] + BlockWidth;
            b[i][j][k] = b[i][j][k-1] + BlockWidth;
            c[i][j][k] = c[i][j][k-1] + BlockWidth;
        }
    }
}

printf("*** Type ***   Time[sec]    Speed[MFLOPS]\n");

ave_sec = 0;
```

```
for (i=0; i < 10; i++) {

    init_matrix(a, b, c);

    tic = clock();
    mul_matrix(a, b, c);
    toc = clock();
    sec = (double)(toc - tic)/CLOCKS_PER_SEC;
    ave_sec += sec;

    flops = 2.0*1000*1000*1000;
    printf("  Mult_x87   %10.2f %13d\n", sec, mflops(flops, sec));
}

ave_sec /= 10.;

printf("----------------------------------------\n");
printf("  Average    %10.2f %13d\n", ave_sec, mflops(flops, ave_sec));

/*
    a, b, c, ar, br, cr のメモリ領域開放

*/
for (i=0; i < BlockNum; i++) {
    for (j=0; j < BlockNum; j++) {
        free(ar[i][j][0]);
        free(br[i][j][0]);
        free(cr[i][j][0]);
        free(a[i][j]);
        free(b[i][j]);
        free(c[i][j]);
        free(ar[i][j]);
        free(br[i][j]);
        free(cr[i][j]);
    }
    free(a[i]);
    free(b[i]);
    free(c[i]);
    free(ar[i]);
    free(br[i]);
    free(cr[i]);
```

```
    }
    free(a);
    free(b);
    free(c);
    free(ar);
    free(br);
    free(cr);

    return 0;
}

int mflops(double flops, double sec)
{
    int mf;

    if (sec > 1.0e-14) mf = (int)(flops/sec/1000000);
    else mf = 0;

    return mf;
}
void init_matrix(double**** a, double**** b, double**** c)
{
    int i, j, k, l;

    srand( (unsigned)time( NULL ) );

    for (i=0; i < BlockNum; i++) {
        for (j=0; j < BlockNum; j++) {
            for (k=0; k < BlockWidth; k++) {
                for (l=0; l < BlockWidth; l++) {
                    a[i][j][k][l] = (double)rand()/1000.0;
                    b[i][j][k][l] = (double)rand()/1000.0;
                    c[i][j][k][l] = 0.;
                }
            }
        }
    }
}
void mul_matrix(double**** a, double**** b, double**** c)
{
    int x, y, k;
```

```
int x2, y2;
int    nobk, rem;
int block_x, block_y, block_k;
int blockwidth;
double *a2, *b2, *c2;
blockwidth = BlockWidth;

nobk = 1000 / BlockWidth; /* number of block */
rem  = 1000 % BlockWidth;

if (rem == 0) {
    for (x=0; x < nobk; x++) {
        for (y=0; y < nobk; y++) {
            for (k=0; k < nobk; k++) {
                mul_matrix_sub(BlockWidth, BlockWidth, BlockWidth,
                    a[y][k], b[x][k], c[y][x]);
            }
        }
    }
} else {
    /* when rem != 0 */
    for (x=0; x <= nobk; x++) {

        if (x == nobk) block_x = rem;
        else block_x = BlockWidth;

        for (y=0; y <= nobk; y++) {

            if (y == nobk) block_y = rem;
            else block_y = BlockWidth;

            for (k=0; k <= nobk; k++) {

                if (k == nobk) block_k = rem;
                else block_k = BlockWidth;
                /*
                mul_matrix_sub(block_x, block_y, block_k, a[y][k], b[x][k], c[y][x])
                */
                a2 = a[y][k][0];
                b2 = b[x][k][0];
                c2 = c[y][x][0];
```

```
        for (x2=0; x2 < block_x*8; x2+=8) {
                for (y2=0; y2 < block_y * BlockWidth * 8; y2 += BlockWidth *

__asm {
    ;
    ; esi += y * nblc * 8
    ;
    mov         esi, dword ptr [y2]
    add         esi, dword ptr [a2]
    ;
    ; edi += x * nblc * 8
    ;
    mov         eax, dword ptr [x2]
    mul         dword ptr [blockwidth]
    mov         edi, dword ptr [b2]
    add         edi, eax
    ;
    ; Loading c[y][x][y2][x2] to x87 FPU Stack
    ;
    mov         eax, dword ptr [y2]
    mov         ecx, dword ptr [x2]
    add         eax, dword ptr [c2]
    add         ecx, eax
    fld         qword ptr [ecx]

    mov         eax, dword ptr [block_k]
    mov         ebx, eax
    shl         ebx, 3h                 ; ecx = ecx * 8
    add         esi, ebx
    neg         eax                     ; eax = -eax
    ;jz         Next1
    add         edi, ebx

    mov         ebx, 2h                 ; eax の増分
    ;
    ;  x87 FPU Calculate Main Loop
    ;
Lop1:fld        qword ptr [esi+eax*8]
    fmul        qword ptr [edi+eax*8]
```

```
                fld         qword ptr [esi+eax*8]+8h
                fmul        qword ptr [edi+eax*8]+8h

                faddp       st(1), st

                add         eax, ebx

                faddp       st(1), st

                jnz         Lop1

          Next1:fstp        qword ptr [ecx]
                            }
                        }
                    }
                }
            }
        }
    }
}
void mul_matrix_sub(int block_x, int block_y, int block_k,
                    double** a, double** b, double** c)
{
    int x, y;
    int blockwidth;
    double *a2, *b2, *c2;
    blockwidth = BlockWidth;
    a2 = a[0];
    b2 = b[0];
    c2 = c[0];

    for (x=0; x < block_x*8; x+=8) {
        for (y=0; y < block_y * BlockWidth * 8; y += BlockWidth * 8) {
            /*
            s = 0.0;
            for (k=0; k < block_k; k++) {
                s += a[y][k] * b[x][k];
            }
            c[y][x] += s;
            */
            __asm {
```

```
        ;
        ; esi += y * nblc * 8
        ;
        mov         esi, dword ptr [y]
        add         esi, dword ptr [a2]
        ;
        ; edi += x * nblc * 8
        ;
        mov         eax, dword ptr [x]
        mul         dword ptr [blockwidth]
        mov         edi, dword ptr [b2]
        add         edi, eax

        mov         eax, dword ptr [block_k]
        mov         ecx, eax
        shl         ecx, 3
        add         esi, ecx
        neg         eax                     ; eax = -eax
        ;jz         Next1
        add         edi, ecx

        fldz

        mov         ebx, 5h     ; ecx の増分

Lop1:fld        qword ptr [esi+eax*8]
        fmul        qword ptr [edi+eax*8]

        fld         qword ptr [esi+eax*8]+8h
        fmul        qword ptr [edi+eax*8]+8h

        fld         qword ptr [esi+eax*8]+10h
        fmul        qword ptr [edi+eax*8]+10h

        fld         qword ptr [esi+eax*8]+18h
        fmul        qword ptr [edi+eax*8]+18h

        fld         qword ptr [esi+eax*8]+20h
        fmul        qword ptr [edi+eax*8]+20h

        faddp       st(1), st
```

```
                fxch        st(1)
                faddp       st(2), st
                fxch        st(2)
                faddp       st(3), st
                fxch        st(1)
                faddp       st(2), st

                add         eax, ebx            ; 上の faddp とペアにして動作
速度向上

                faddp       st(1), st

                jnz         Lop1

          Next1:mov         esi, dword ptr [y]
                mov         eax, dword ptr [x]
                add         esi, dword ptr [c2]
                fld         qword ptr [esi][eax]
                faddp       st(1), st
                fstp        qword ptr [esi][eax]
            }
        }
    }
}
```

6 実行結果

以下に実行結果（実行時間、MFLOPS）を示す。SSE2 版、C 言語版、x87 FPU 版の 3 つがあり、それぞれ Visual C++ 6.0 でコンパイル時に最適化を行っている。

メインは Streaming SIMD Extensions 2 版である。高速化の労力の大半はこのプログラムでの実行結果向上のために費やしている。ほぼ限界の速度に達していると思う。

下の結果を見ての通り、**平均 1.52 秒、1313 MFLOPS** という速度を叩き出している。

C 言語版は、いずれ SSE2 用のプログラムにするための元のプログラムとして、一番最初に作ったプログラムである。こちらでは、平均 2.60 秒、769 MFLOPS まで高速化できた。

また、x87 FPU 版は、C 言語版を元にして、インラインアセンブラで直接 x87 FPU の命令を操作し、SSE2 版へ変更する為に実験的に作成したものである。これでは、平均 2.90 秒、689 MFLOPS という結果が得られた。

6.1 Streaming SIMD Extensions 2 版の結果

```
Streaming SIMD Extensions 2 version

*** Type ***   Time[sec]    Speed[MFLOPS]
  Mul_SSE2         1.52          1314
  Mul_SSE2         1.52          1314
  Mul_SSE2         1.52          1314
  Mul_SSE2         1.53          1304
  Mul_SSE2         1.52          1314
  Mul_SSE2         1.51          1322
  Mul_SSE2         1.52          1314
  Mul_SSE2         1.50          1330
  Mul_SSE2         1.54          1297
  Mul_SSE2         1.52          1314
---------------------------------------
  Average          1.52          1313
```

6.2 C 言語版の結果

```
C Language version
*** Type ***   Time[sec]    Speed[MFLOPS]
 multmats          2.60           768
 multmats          2.60           768
 multmats          2.59           771
 multmats          2.60           768
 multmats          2.59           771
 multmats          2.60           768
 multmats          2.60           768
 multmats          2.59           771
 multmats          2.60           768
 multmats          2.59           771
----------------------------------------
 Average           2.60           769
```

6.3 x87 FPU 版の結果

```
x87 FPU version
*** Type ***   Time[sec]    Speed[MFLOPS]
  Mult_x87         2.90           688
  Mult_x87         2.90           688
  Mult_x87         2.89           691
  Mult_x87         2.91           688
  Mult_x87         2.89           691
  Mult_x87         2.90           688
  Mult_x87         2.89           691
  Mult_x87         2.91           688
  Mult_x87         2.89           691
  Mult_x87         2.89           691
----------------------------------------
  Average          2.90           689
```

7　考察＆感想

今回の課題は大変面白いものであった。このように、いろいろと工夫して演算速度を向上させるという作業はプログラミングの楽しみの醍醐味であると言えよう。久しぶりにプログラムの楽しさを思い起こさせてくれた。

また、特に浮動小数点演算のプログラムなどは、機会がないと中々組むことがないので（特にアセンブラでは）今回はその面でも大変勉強になった。今回の課題で学んだことは、今後のプログラミング作業でも生かされるであろうと思う。

ちなみに、私の作成したSSE2版のプログラムはWillametteプロセッサ以外では動作しない。また、コンパイルにはVisual C++ 6.0 Processor Packが必要になる。

C言語版やx87 FPU版はどのCPUでも動作するが、Pentium 4用に最適化してあるため、他のCPUでは想像以上にパフォーマンスが発揮できない可能性がある。事実、celeronでそのまま実行してみると、ブロックサイズが大きすぎるため、小さくしてやるなどしないと、大幅にパフォーマンスが低下する。celeronで実行する場合は、BlockWidthを80位にするよ良いようである。

また、Pentium 4ではC言語版の方がx87 FPU版よりも速いが、同じものをceleronでやってみると、どうやってもx87 FPU版の方が速い。

この辺が面白いところである。celeronで実行するのであれば、celeron用にもっとチューニングすれば、さらに好タイムを出すことが出来るだろう。というのも、現在のプログラムは、一時celeronで出していたタイムよりも悪いタイムしか出なくなってしまったからである。

8　他 CPU との比較用データ

他の CPU の人との比較がしにくいと思うので、参考までに、私の環境 (Pentium 4 1.4GHz) で、大石先生のホームページからリングが張られていた「行列乗算高速化プロジェクト」に載っているプログラム (mltmatsc.lzh) を Visual C++ 6.0 でコンパイルし、最適化しない（Win32 Debug モードのまま）場合と、最適化あり (Win32 Release) の場合の実行結果を以下に示す。

mltmatsc.lzh は

http://www.geocities.co.jp/Technopolis/6872/math/sub_c.htm

からダウンロードできる。

8.1　数値計算 HP のプログラム (mltmatsc.lzh) の実行結果　最適化なし

```
input n (<= 1000)
 n = 1000
*** Type ***   Time[sec]    Speed[MFLOPS]    Check[Error]
 definition        41.67              47             ***
 multmats          37.62              53        1.2e-009
```

8.2　数値計算 HP のプログラム (mltmatsc.lzh) の実行結果　最適化あり

```
input n (<= 1000)
 n = 1000
*** Type ***   Time[sec]    Speed[MFLOPS]    Check[Error]
 definition        40.76              49             ***
 multmats           4.42             452        1.2e-009
```