## 数値計算、Apache、Samba、ソフト開発に挑戦

# PS2 Linux

株式会社ソニー・コンピュータエンタテインメントから、Linuxコミュニティに向けてリリースされた「プレイステーション2」用Linuxキット（β版）。すでに入手した方、また今後どうにか入手したいと考えている方のために、このキットを楽しむための第1歩となりそうな情報を掲載させていただきました。今後は連載としてさまざまな関連情報を掲載していきます。



# EE（VU0）で数値計算

清水尚彦　nshimizu@keyaki.cc.u-tokai.ac.jp

ここでは、PS2 Linuxを使って数値計算を行うための基本的な事項をまとめてみます。皆さんもご存知のように、PS2はメモリの搭載容量が少なく大規模数値計算には不向きであるにも関わらず、大変多くの方がPS2 LinuxKitを入手されております。メモリが大して必要のない分野でも数値計算はいろいろと行われており（といって具体的なアプリケーションを挙げられないのですが）、この章では「簡単に」PS2の性能を引き出すための工夫を見ていきましょう。「簡単に」というのが大切で、複雑な方法は結局利用者が少なくなってしまいます。ここで示す以上の性能を求めたい方は複雑な方法をぜひご検討いただきたいと思いますが、手軽に数値計算を高速化したいという向きにはこの程度でもいろいろと面白いことが出来るのではないかと考えています。

【リスト1】　g77用サンプルプログラム

```
real*4 function foo(x)
  real*4 x
  foo=sin(x)
  return
end
```

【リスト2】

```
subu    $sp,$sp,48
.cprestore 16
sw      $31,36($sp)
l.s     $f12,0($4)
sw      $28,32($sp)
jal     fptodp          <-- 倍精度へ変換
move    $4,$2
move    $5,$3
jal     sin         <-- 倍精度関数コール
move    $4,$2
move    $5,$3
jal     dptofp          <-- 単精度へ変換
mov.s   $f12,$f0
jal     fptodp          <-- 倍精度へ変換
lw      $31,36($sp)
#nop
.set    noreorder
.set    nomacro
j       $31
addu    $sp,$sp,48
```

ここでは、

- PS2を数値計算で使うための注意事項
- VU0の利用の勧め
- VU0を使ったサンプルプログラム

の順で「簡単な」PS2の利用方法を説明して行こうと思います。

## PS2を数値計算で使うための注意事項

そんなことは、言われなくても良いと思っている方も多いかもしれませんが、ここではあえて基本的な注意事項を説明させてもらいます。Fortranを普段お使いの方は不満かもしれませんが、PS2は今のところFortranの利用には不向きのようです。g77はC99の機能を利用していない上に、古いCを用いたf2cというプログラム（特にライブラリ）を引きずっており、関数呼び出しでどうしても倍精度に変換するルーチンが呼ばれてしまいます。g77を使って数値計算を高速化するには組み込み関数を用いないで自前の単精度関数を用意する必要があるので、倍精度のハードウェアを持たないPS2ではこれは致命傷になってしまいます。C言語も以前は単精度の関数は用意されていなかったのですが、C99ではちゃんと単精度の関数も標準ライブラリに定義されてgccもこれに対応しているため、単精度で計算するという最低限のレベルはクリアしております。

例えば、g77で簡単なプログラムをコンパイルしてこのことを確認してみましょう（**リスト1**）。

このFortranプログラムを`foo.f`として

```
g77 -O2 -S foo.f
```

とコンパイルするとアセンブラソースが得られ、その主用部分を示すと**リスト2**のようになります。

わざわざ単精度で来た引数を倍精度に変換して倍精度の関数を呼び出し、帰ってきた値を単精度に変換した後、再び関数値を倍精度としてリターンするという、ちょっととんでもないコードが生成されています。これを少しでも改善するには、コンパイル時に

```
gcc -fno-f2c -O2 -S foo.f
```

とすると、最後に関数値を倍精度に変換するところは避けられます。しかし、組み込み関数はf2c互換のライブラリを用いるため相変わらず倍精度で呼び出します。

一方、同じことをCで書くと、C99の単精度関数を用いて**リスト3**のように書けます。gccでコンパイルしてみましょう。

【リスト3】

```
#include <math.h>

float foo(float x)
 {
   return sinf(x);
 }
```

```
gcc -O2 -S foo.c
```

こちらも主用部分を示します（**リスト4**）。違いがお分かりですか？　Cでは単精度の関数コールを用いてそのまま単精度の値を関数値として返しています。

【リスト4】

```
subu    $sp,$sp,48
.cprestore 16
sw      $31,36($sp)
sw      $28,32($sp)
jal     sinf        --> 単精度関数コール
lw      $31,36($sp)
#nop
.set    noreorder
.set    nomacro
j       $31
addu    $sp,$sp,48
```

ということで、PS2で数値計算を行う場合にはCを用いて、関数計算はすべて単精度の関数を明示的に呼び出すことにしましょう。単精度関数は一般の関数名の後にfloatを意味する「f」が付きます。「`sin -> sinf`」、「`cos -> cosf`」などになります。

## EEcoreへの苦言

さて、いわゆるCPUと呼ばれるEEcoreですが、MIPS R5900相当のプロセッサということになっています。MIPSアーキテクチャではコプ

ロセッサ命令を使って浮動小数点演算を実行しますが、これは浮動小数点プロセッサが外付けだったR3000時代を引きずっているからです。EEcoreを調べていくと、どうも1980年代で進化が止まったような錯覚に陥ります。分岐予測は怪しいし、ヒットアンダーミスも1ミスだけだし、浮動小数点データの方が一般にミス率が高いにも関わらず、コプロセッサのミスはメインパイプもストールさせる。これなら私が学生教育用に公開しているプロセッサの方が、少なくとも基本部分はましだというのが素直な感想です。

よほど慌てて作ったのか、1990年代のアーキテクチャの進歩を知らずに作ったのか、せっかく詰め込むだけ詰め込める財政的な余裕があるプロジェクトなんだからもう少し真面目に作ればいいと思うのは私だけでしょうか?

プロセッサを作るのはどこでもできます。しかし、それを支えるマーケティングや資本がないのが多くの会社の実情で、量産プロセッサを作る機会を持っている人たちはがっかりさせるようなものを出さないでほしいです。

## EEを使いこなすには?

それではこのプロセッサで性能を出すための工夫を考えてみましょう。まずは、キャッシュミスをコプロセッサで起こすとストールが発生することから、コプロセッサではキャッシュミスが発生しないような工夫が必要になります。こう書くと一言で終わりますが、キャッシュ側は結構厄介です。CPUもキャッシュミスを1つしか保留できないため、プリフェッチにCPUのキャッシュロードを使うのは難しいのです。そこで、オンチップの小規模メモリであるスクラッチパッドメモリを利用するのが良さそうなのですが、スクラッチパッドメモリはDMAコントローラでしか非同期の転送を起動できません。そのためユーザープログラムからの利用はあまり現実的ではありません。

DMAコントローラがアドレス変換を行ってユーザープログラムから簡単に駆動できるような仕組みがあればすむのですが、Cray T3EのEレジスタや、私のところのSCALTのような簡便さは望めません(もっともゲーム機としてはこれでもいいのかもしれませんけど。DSPなどでも、内部メモリへの転送はDMAを用いるのが主流のようですしね)。

ではどうしたら良いかというと、スクラッチパッドにCPUの命令で明示的にデータを移動して、その領域に対して演算を行うのが良さそうです。これにはスクラッチパッド領域を仮想アドレスに登録しておく必要があります。マッピング方法はEEcoreのマニュアルに書いていますが、Linuxのシステムコールを新設してメモリ要求時にスクラッチパッドを要求できるようにするのがいいですね。

今回はあくまでも「簡単に」を目指すため、アイデアはいろいろあっても複雑な手続きやカーネルの変更が必要なものはとりあえず置いておきます。そこで、スクラッチパッドの利用は後日のお楽しみということにして、プログラムでデータのブロック化を行って、CPUがデータキャッシュにあらかじめブロックを配置してから演算を開始する方が性能が出やすいだろうという指針の提示に留めておきます。

## VU0の利用の勧め

VU0とVU1という2つのベクトルユニットがあるPS2は、演算の並列性を生かして高速演算を行えます(というかこれらを使わないと単精度の演算だけのただの遅いマシンに過ぎません)。そこで、数値計算を行う場合にはこれらのベクトルユニットをいかに活用するかがポイントになります。ところが、これらのユニットをマイクロ命令で活用するには、「マイクロ命令のアセンブラを使って作成したオブジェクトを、ベクトルユニットのメモリに転送して起動する」という面倒な方法が必要になります。この記事ではあくまでも「簡単な」使い方を想定していますので、VU1の利用はあきらめて、「簡単に」使えるVU0をどのように使うかを説明します。

## ベクトルユニットとは?

PS2では2つのベクトルユニットを利用できるとなっています。これは一体何物なのでしょうか? 簡単に言ってしまうと、浮動小数点積和演算器をたくさん積んだ演算エンジンです。32本の128ビット浮動小数点レジスタ、16本の16ビット整数レジスタ、浮動小数点アキュムレータや平方根や除算などの実行時間の長い命令の結果を返すQレジスタなどがあります。実装している命令はFPU(これはコプロセッサ1=COP1)と大差ないのですが、ベクトルユニットは128ビットのレジスタに4つの浮動小数点を格納して4つのデータを並列に演算できます。そのため、同じ命令でもスループットは4倍になるわけです。浮動小数点レジスタ(ベクトルレジスタ)になるべくデータを残して演算を何度も繰り返すような計算手法が高性能のカギになります。一般的なベクトル機と呼ばれるマシンではベクトルレジスタはもっとずっと大量のデータを保持できるので、並列度を引き出しやすいのですが、1本のレジスタに4つのデータしか入らないPS2ではそれなりに良く考えてデータを配置しないと性能を出すのは難しいことになります。とりあえずは「簡単に使ってみる」ことを目標にしましょう。

ところで、VU0とVU1はCPUへの接続方法が微妙に異なります。MIPSの命令セットの中でコプロセッサへのインターフェイスが決められているのですが、このインターフェイスで接続されているのはVU0のみで、VU1は特別な起動シーケンスで起動しないと利用できないのです。この結果、VU0の起動は普通のFPUと同じく命令から直接起動できるのに、VU1は煩雑な方法でしか起動できないことになります。ここではあくまでも「簡単な」方法を追求するため、VU0だけを利用していくことにします。複雑さにめげない方は、積極的にVU1を使うことをお勧めしますが、とりあえずVU0でベクトル計算に慣れてからでも遅くないと思います。

コプロセッサとしてVU0を使うモードのことをマクロモードと呼びます。これに対してVUが独自に命令を読み出して実行するモードをマイクロモードと読んでいます。マイクロモードでは2つの命令を同時に発行できるので並列度は最大になりますが、マクロモードでも4つの演算器を同時に動かせるので、使い方の簡単さを考えれば捨てがたいものがあります。

VU0のコプロセッサ命令はCPUの演算パイプラインのI1を用いるので、この命令とCPUのI0パイプを用いる命令は同時に実行可能です。例えば、FPUの命令はI0パイプを用いるので、VUとFPUを同時に動かせるわけです。

VPUで注意しなくてはならない点として、データを16バイト境界にあわせておかなくてはならないということが挙げられます。そこでプログラムの作成時にalignedのアトリビュートを付けておきましょう。

## VU0を動かしてみよう

さて、お話しばかりではつまらないので、実際にVU0を動かしてみましょう。ユーザーモードでVU0を使うためには、VU0のデバイスファイルをユーザーから読み書きできるように設定する必要があります。スーパーユーザーから

```
# chmod a+rw /dev/ps2vpu0
```

としてアクセス権を設定します。公のところで使う場合にはこの方法は少し気持ち悪いのですが、デバイスを1人がオープンすれば他の人は使えなくなるので、それほど悪い方法でもないでしょう。少なくとも自分1人しか使わない環境では全く問題ありません。何もないところからデバイスの使い方を見付け出すのは大変なのですが、幸い、

```
/usr/doc/PlayStation2/vpu_kick
```

にサンプルプログラムがLGPLで置かれています。これを参考に「簡単な」ところだけを利用していきます。まず、vpu_dev.c、vpu_dev.hはそのまま利用させてもらいましょう。これはデバイスのオープンとデバイス情報をioctlで取り出すルーチンが入っていて、中身を知らなくて

も利用できます。アプリケーションインターフェイス（ＡＰＩ）は同じディレクトリのvpu_dev_jp.txtに書かれていますからご参照ください。ここで使う関数は

```
vpudev_open
vpudev_close
vpudev_reset
```

の3個だけです。このディレクトリの中の説明はVPUのマクロモードとマイクロモードが入れ違ったりしていますが、必要な情報は少ないので本記事で書いている程度の知識があれば大丈夫です。

さて、最初の例題は大変簡単なものです（**リスト5**）。この例題はVPU0をオープンして、リセットした後、CPUの汎用レジスタの値をVPU0のVF1レジスタに転送します（qmtc2）。次に、VF1レジスタのに入っている整数値を浮動小数点数に変換します（vitof0）。そして、VF1レジスタの値同士を加算して（vadd）、再度浮動小数点数を整数値に変換し（vftoi0）、最後に汎用レジスタに値を戻しています。その後、画面に表示してVPU0をクロースして終了します。インラインアセンブラを用いて作成していますが、gccのインラインアセンブラにパラメータを与えるため、レジスタで値書き込みを行う「=r」のパラメータとして整数変数jを指定し、読み出しだけのパラメータとして整数変数iを指定します。「\t」と「\n」はアセンブラに変換したときに読みやすいように入れてあります。なくても動作上は問題ありません。

コンパイルは**実行例1**のようにvpu_devを合わせてリンクします。実行すると無事入力値の2倍の値が得られることが分かると思います。

次は、このプログラムがどのようにコンパイルされるのか、コンパイラにアセンブラコードを生成させてみましょう。

```
$ cc -O2 -S sample1.c
```

とすると、サンプルコードのアセンブラ（sample1.s）が出力されます。主用部分を抜き出します（**リスト6**）。

ここでは最適化をかけているためにプログラムの字面と少し異なるコードが生成されています。まず、vpudev_resetでVPUのリセットの手続きが呼ばれています。次に変数iを$6レジスタに割り当てて定数ロードが行われます。変数jは後で使用されないため、ここでは初期化されていません。#APPから#NO_APPまではインラインアセンブラのコードがそのまま出ています。ここで、%0、%1として指定したレジスタにどちらも$6が代入されていることが分かります。その後$5に再度変数iの値を定数でロードしてからprintfを呼び出しています。依存関係を見ながらメモリを読み出すより、定数として設定した方が高速なコードとなるとコンパイラが判断した結果です。これはコンパイラオプションの指定によって大きく変わってきますので、いろいろなコンパイラオプションを試して命令列がどのように変化するか実験すると面白いでしょう。

さて、CPUからデータを送るだけでは演算のスループットに追いつかない処理はたくさんあります。せっかく128bitのデータバス幅があるので、今度はVPUに自らデータを読み出してもらいましょう。とはいっても、マイクロ命令を起動するなんて大げさな話ではなくて、VPUのマクロ命令のLQC2、SQC2を使ってデータを読み出すという話です。

gccのインラインアセンブラではメモリを扱うときには引数のタイプとして「m」を用います。ところが、vpu_kickのサンプルコードではわざわざ「r」でレジスタ値を渡しています。なんでこんな回りくどいことをしているのかと思って、**リスト7**のようにしてみると、コンパイルは通るけれどもアセンブラでエラーになります。コンパイラが吐き出すコードは、**リスト8**を見ると特に問題になりそうなところがないので、どうやらアセンブラの問題のようです。レジスタ渡しでも動くものは作れるはずですが、メモリオブジェクトの依存性をコンパイラが認識するようになったら、メモリオブジェクトが変更されているかどうかを認識できないベースレジスタでの指定方法は問題が大きそうです。しかし、ここは深く考えずとにかく「簡単に」動かすための方法を探しましょう。

ベースレジスタ渡しとしたときにはベースレジスタの値そのものは変更しないため、インラインアセンブラへの引数は読み出し変数のみになります。**リスト9**を見てください。

このプログラムは16バイト境界に置いた2つの浮動小数点配列をvf1、vf2のベクトルレジスタに読み出し、加算した結果をaの配列に書き出すものです。アトリビュートで配列のアラインを指定してVPUの制限であるオペランドが16バイト境界であることを確保しています。コンパイル実行してみましょう（**実行例2**）。1命令で連続する4つの

【リスト5】

```
/*
  sample1.c: sample code for vpu usage
          Copyright (C) 2001  Naohiko Shimizu
*/

  #include <stdio.h>
  #include <unistd.h>
  #include "vpu_dev.h"

  int main()
  {
          VPUDEV *vdev;
          int i, j;
          vdev = vpudev_open(0);
          if (vdev == (void *)-1) { perror("vpu0"); }
  vpudev_reset(vdev);

  i=314;
  j=123;
asm __volatile__ (
   "qmtc2   %1, vf1;\n"
   "\tvitof0  vf1, vf1;\n"
   "\tvadd    vf1, vf1, vf1;\n"
   "\tvftoi0  vf1, vf1;\n"
   "\tqmfc2   %0, vf1;\n"
   : "=r"(j) : "r"(i) );
printf("%d %d\n", i,j);
vpudev_close(vdev);
return 0;
}
```

【実行例1】

```
$ cc sample1.c vpu_dev.o -o sample1
$ ./sample1
```

【リスト6】

```
$L19:
        jal     vpudev_reset
li      $6,314              # 0x13a
 #APP
qmtc2   $6, vf1;
vitof0  vf1, vf1;
vadd    vf1, vf1, vf1;
vftoi0  vf1, vf1;
qmfc2   $6, vf1;
 #NO_APP
la      $4,$LC1
li      $5,314              # 0x13a
jal     printf
move    $4,$16
jal     vpudev_close
```

【リスト7】

```
asm __volatile__ (
"lqc2      vf1, %0;\n"
"\tlqc2    vf2, %1;\n"
"\tvadd    vf1, vf1, vf2;\n"
"\tsqc2    vf1, %0;\n"
: "+m"(a) : "m"(b) );
```

【リスト8】

```
#APP
        lqc2      vf1, a.33;
        lqc2    vf2, $LC1;
        vadd    vf1, vf1, vf2;
        sqc2    vf1, a.33;
#NO_APP
```

【リスト9】

```
/*
  sample2.c: sample code for vpu usage
           Copyright (C) 2001  Naohiko Shimizu
*/

#include <stdio.h>
#include <unistd.h>
#include "vpu_dev.h"


int main()
{
        static float a[]  __attribute__((aligned(16))) = {1.0, 2.0, 3.0, 4.0};
        static float b[]  __attribute__((aligned(16))) = {5.0, 6.0, 7.0, 8.0};
        VPUDEV *vdev;
        int i;
        vdev = vpudev_open(0);
        if (vdev == (void *)-1) { perror("vpu0"); }
        vpudev_reset(vdev);
        asm __volatile__ (
                "lqc2      vf1, (%0);\n"
                "\tlqc2    vf2, (%1);\n"
                "\tvadd    vf1, vf1, vf2;\n"
                "\tsqc2    vf1, (%0);\n"
                :  : "r"(a),"r"(b) );
        for(i=0; i< 4; i++) printf("%d %f %f\n", i,a[i],b[i]);
        vpudev_close(vdev);
        return 0;
}
```

浮動小数点に対する演算を行っています。

## 積和演算器

基本的なインラインアセンブラの使い方に慣れたら、次はいよいよ積和演算器を使ってみましょう。積和演算器は乗算と加算を1命令で実行するものです。乗算器が加算のツリーを使って計算する最後にもう1つの加算をマージすることで、比較的簡単にハードウェアが実現できるため多くのプロセッサでサポートされています。この積和演算は内積演算の基本となっており応用範囲が広いのも特徴的です。例えば

```
A ＝ B ＋ C × D
```

という演算を行え、ベクトル積和演算器を使うことで演算レイテンシの長さをカバーしてより効率の良い計算ができます。

ただし、VPUの積和演算器には大きな制約があります。上の式のBがアキュムレータという専用レジスタに固定されているのです。アキュムレータは、そのままではロードもストアもできないレジスタなのであまり使い勝手がよくありません。そこで、積和演算を行う場合にはあらかじめ前準備と後処理が必要になってきます。普通は前準備の段階は後続の演算と併せてしまってプログラムには出さないのですが、分かりやすくするために記事のサンプルは前準備も明示的に行います（**リスト10**）。

**リスト10**は4×4のrowメジャーな行列aにベクトルbを掛けてベクトルcに格納しています。まず、最初のインラインコードはベクトルbを読み出し、ACCレジスタをクリアしています。値が0の`vf0x`を使って乗算して作成した0のベクトルをACCに格納します。余談ですが、アキュムレータのACCをアセンブラオペランドに記述するときに小文字を使うとエラーになります。当初これが分からなくて、なんでエラーなのかとマニュアルを行ったり来たり時間をかけて調べてしまいました。どうみても間違っていないので、もしかしてと思って大文字に変換したらすんなりと通ったという苦い経験をしました。先ほどのメモリオペランドの件といい、アセンブラの出来はさすがにベータ版らしきものですね。

次に、行列aから4つの要素を読み込んで、アキュムレータに積和演算をしていきます。これは行列aの始めの3行に対して行っており、最後の行に対する積和演算はディスティネーションをアキュムレータではなくベクトルレジスタとしてレジスタの値をベクトルcに書き込んでいます。

コンパイルと実行は**実行例3**のようになります。行列のデータの並びが、通常のCのプログラムに比べて行と列を交換したようになっていることにご注意ください。

## 高速化

せっかくVPUを使っても、これでどれだけ高速化の効果があったのか分からないと面白くないですよね。そこで、例によって実効時間の測定をしてみましょう。**リスト10**のコードを繰り返し実行した時間をFPUで同様の演算を行ったときの時間と比べてみます。時間測定と繰り返し部分を加えたソースをsample4.cとして作成しました（**リスト11**）。REPEATは10000000を指定しています。

**実行例4**のようにすると、同一結果を得るのに約7倍の高速化が達成されているのが分かります。インラインアセンブラは多少面倒ですが、やってみるだけの価値はありそうですね。

多くの資料はPS2 LinuxのDVD-ROMに入っていますので、ここに挙げたようなことはみなさんももう試されているかもしれません。もっといろいろと試してみたい方は、

```
mount /mnt/cdrom
```

としてDVD-ROMをマウントした後、

```
/mnt/cdrom/sm_pdf
```

の下のPDFファイルを読んでみてください。日本語でも読めますので安心です。

それでは、Happy Hacking！

【実行例2】

```
$ cc sample2.c vpu_dev.o -o sample2
$ ./sample2
0 6.000000 5.000000
1 8.000000 6.000000
2 10.000000 7.000000
3 12.000000 8.000000
```

【実行例3】

```
$ cc sample3.c vpu_dev.o -o sample3
$ ./sample3
0 90.000000
1 100.000000
2 110.000000
3 120.000000
```

【実行例4】

```
$ cc sample4.c vpu_dev.o -o sample4 -O2
$ ./sample4
0 90.000000
1 100.000000
2 110.000000
3 120.000000
0 90.000000
1 100.000000
2 110.000000
3 120.000000
VU elaps 1.359262e+00: EE elaps
8.756338e+00
```

【リスト10】

```
/*
  sample3.c: sample code for vpu usage
         Copyright (C) 2001  Naohiko Shimizu
*/

#include <stdio.h>
#include <unistd.h>
#include "vpu_dev.h"

int main()
{
        static float a[][4] __attribute__((aligned(16)))
           = { {1.0, 2.0, 3.0, 4.0},
               {5.0, 6.0, 7.0, 8.0},
               {9.0, 10.0, 11.0, 12.0},
               {13.0, 14.0, 15.0, 16.0}} ;
        static float b[]  __attribute__((aligned(16))) = {1.0, 2.0, 3.0, 4.0};
        static float c[4]  __attribute__((aligned(16)));
        VPUDEV *vdev;
        int i;

        vdev = vpudev_open(0);
        if (vdev == (void *)-1) { perror("vpu0"); }
        vpudev_reset(vdev);
        asm __volatile__ (
              "lqc2    vf1, (%0);\n"
              "\tvmulax    ACC, vf0, vf0x;\n"
               : : "r"(b) );
        for(i=0;i<3;i++)
        asm __volatile__ (
                "lqc2      vf2, (%0);\n"
                "\tvmaddax    ACC, vf2, vf1x;\n"
                "\tvmr32      vf1, vf1;\n"
                :  : "r"(a[i]));
        asm __volatile__ (
                "lqc2      vf2, (%0);\n"
                "\tvmaddx  vf3, vf2, vf1x;\n"
                "\tsqc2    vf3, (%1);\n"
                :  : "r"(a[i]), "r"(c));
        for(i=0; i< 4; i++)
             printf("%d %f\n", i,c[i]);
        vpudev_close(vdev);
        return 0;
}
```

【リスト11】

```
/*
  sample4.c: sample code for vpu usage
         Copyright (C) 2001  Naohiko Shimizu
*/

#include <stdio.h>
#include <unistd.h>
#include <sys/time.h>
#include "vpu_dev.h"
#define REPEAT 10000000

int main()
{
        static float a[][4] __attribute__((aligned(16)))
           = { {1.0, 2.0, 3.0, 4.0},
               {5.0, 6.0, 7.0, 8.0},
               {9.0, 10.0, 11.0, 12.0},
               {13.0, 14.0, 15.0, 16.0}} ;
        static float b[]  __attribute__((aligned(16))) = {1.0, 2.0, 3.0, 4.0};
        static float c[4]  __attribute__((aligned(16)));
        VPUDEV *vdev;
        int i,j, repeat;
        struct timeval tv1, tv2, tv3,tv4;

        vdev = vpudev_open(0);
        if (vdev == (void *)-1) { perror("vpu0"); }
        vpudev_reset(vdev);

    gettimeofday(&tv1, NULL);
    for(repeat=0; repeat < REPEAT; repeat++) {
        asm __volatile__ (
              "lqc2    vf1, (%0);\n"
              "\tvmulax    ACC, vf0, vf0x;\n"
               : : "r"(b) );
        for(i=0;i<3;i++)
        asm __volatile__ (
                "lqc2      vf2, (%0);\n"
                "\tvmaddax    ACC, vf2, vf1x;\n"
                "\tvmr32      vf1, vf1;\n"
                :  : "r"(a[i]));
        asm __volatile__ (
                "lqc2      vf2, (%0);\n"
                "\tvmaddx  vf3, vf2, vf1x;\n"
                "\tsqc2    vf3, (%1);\n"
                :  : "r"(a[i]), "r"(c));
      }
    gettimeofday(&tv2, NULL);
    for(i=0; i< 4; i++)
             printf("%d %f\n", i,c[i]);
        vpudev_close(vdev);
    gettimeofday(&tv3, NULL);
    for(repeat=0; repeat < REPEAT; repeat++) {
        for(i=0; i<4; i++)  c[i] = 0;
        for(i=0; i<4; i++)
           for(j=0; j<4; j++)
             c[j] += b[i]*a[i][j];
      }
    gettimeofday(&tv4, NULL);
    for(i=0; i< 4; i++)
             printf("%d %f\n", i,c[i]);
    printf("VU elaps %e: EE elaps %e\n",
            (tv2.tv_sec+1e-6*tv2.tv_usec)-
            (tv1.tv_sec+1e-6*tv1.tv_usec),
            (tv4.tv_sec+1e-6*tv4.tv_usec)-
            (tv3.tv_sec+1e-6*tv3.tv_usec));
        return 0;
}
```