Appendix A

DAPDNA-2を使ってみた！

早乙女勝昭

ここではアイピーフレックスのダイナミック・リコンフィギャラブル技術を使ったLSI「DAPDNA-2」を利用して，離散コサイン変換（DCT）の処理を実現する手順を紹介する．同処理をDNAマトリックス上に展開し，RISCプロセッサ（DAP）で実行した場合の約50倍の速度で演算できることを確認した．

（編集部）

ここ数年，ダイナミック・リコンフィギャラブル技術の話題がかなりホットになってきています．アイピーフレックスを筆頭に，それをLSI化するベンダも現れました．読者の皆さんの中にも「そろそろ評価してみようか」と思われている方がいらっしゃるのではないでしょうか．ただし，これまで雑誌などで提供されている情報は，技術の概念の紹介が多いようです．「どうやって使うのか」という点ではいまだベールに包まれており，なかなか実感がわかないのが実状だと思います．

筆者は，今回，DAPDNA-2を評価する機会に恵まれました．ここでは簡単な例題をもとに，DAPDNA-2を使用した場合のアルゴリズム設計から実装までの手順を紹介します．

● 離散コサイン変換の処理をDAPDNA-2で実現

今回，例題として取り上げるのは離散コサイン変換（以下，DCT）です．評価の“素材”としてはあまりおもしろくないのかもしれませんが，以下の理由によりDCTを取り上げました．

- **画像応用（JPEG，MPEGなど）でよく使われている**
- **チュートリアルの設計規模としては妥当**
- **だれでも知っている**

ここではDCTの詳細には触れませんが，簡単にその算出方法を説明します．以下に，画像圧縮などでよく使われる8×8ピクセルの2次元DCTの演算式を示します．

$$\boldsymbol{Y}(u,v)=\frac{1}{4}a(u)a(v)\sum_{i=0}^{7}\sum_{j=0}^{7}\boldsymbol{X}(i,j)\cdot\cos\{(2i+1)u\pi/16\}\cdot\cos\{(2j+1)v\pi/16\} \quad (1)$$

$$a(u)=\frac{1}{\sqrt{2}}\ (u=0),\ 1\ (0<u<7)$$

$$a(v)=\frac{1}{\sqrt{2}}\ (v=0),\ 1\ (0<v<7)$$

ここで$\boldsymbol{C}(p,q)=\frac{1}{2}a(p)\sum\cos\{(2q+1)p\pi/16\}$として行列式でまとめると，

$$\boldsymbol{Y}=\boldsymbol{C}\cdot\boldsymbol{X}\cdot{}^{t}\boldsymbol{C}$$ （tは行列要素の転置を表す）

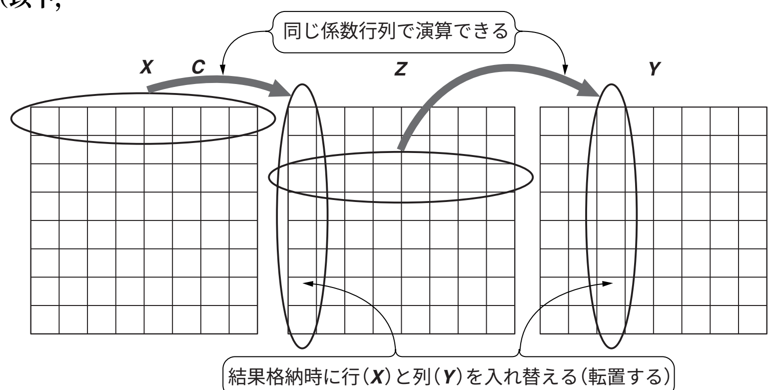

図1
1次元DCTへの分解
データ・ブロックXにおいて横方向に1次元DCT処理を行うが，その結果を格納する際には横方向ではなく縦方向に格納する（データ・ブロックZ）．同様に，その処理結果のデータ・ブロックZにおいて，横方向に1次元DCT処理を行い，結果を縦方向へ格納する（データ・ブロックY）．これにより，データ・ブロックYは元のデータ・ブロックXに縦横両方向にDCT処理（2次元DCT処理）を行った結果となる．

リスト1　DCT処理のCソース・コード(浮動小数点演算)

```
#define WIDTH           8
#define HEIGHT          8
#define IMGSIZE         WIDTH * HEIGHT
#define BytePerBlk      8
#define XCNT            WIDTH / BytePerBlk
#define YCNT            HEIGHT / BytePerBlk
#define BLKS            XCNT * YCNT

#define M_PI    3.14159265358979323846
#define M_SQRT2 1.41421356237309504880

static void dct_float8x8(int *in, int *out )
{
  int    k, j;
  int    *inptr;
  float  tmp, th;

  for( j = 0; j < BytePerBlk; j++ ) {
    inptr = in + WIDTH * j;
    for( k = 0; k < BytePerBlk; k++ ) {
      th = k * M_PI / 16;
      tmp = inptr[ 0 ] * cos( (2 * 0 + 1) * th )
        + inptr[ 1 ] * cos( (2 * 1 + 1) * th )
        + inptr[ 2 ] * cos( (2 * 2 + 1) * th )
        + inptr[ 3 ] * cos( (2 * 3 + 1) * th )
        + inptr[ 4 ] * cos( (2 * 4 + 1) * th )
        + inptr[ 5 ] * cos( (2 * 5 + 1) * th )
        + inptr[ 6 ] * cos( (2 * 6 + 1) * th )
        + inptr[ 7 ] * cos( (2 * 7 + 1) * th );
      if ( k == 0 ) tmp /=  M_SQRT2;
      out[ WIDTH * k + j ] = (int) (tmp / 2.0);
    }
  }
}
```

リスト3　余弦テーブル最適化

```
static int cos_table[] = {
   C14,   C14,   C14,   C14,   C14,   C14,   C14,   C14,
  C116,  C316,  C516,  C716, -C716, -C516, -C316, -C116,
   C18,   C38,  -C38,  -C18,  -C18,  -C38,   C38,   C18,
  C316, -C716, -C116, -C516,  C516,  C116,  C716, -C316,
   C14,  -C14,  -C14,   C14,   C14,  -C14,  -C14,   C14,
  C516, -C116,  C716,  C316, -C316, -C716,  C116, -C516,
   C38,  -C18,   C18,  -C38,  -C38,   C18,  -C18,   C38,
  C716, -C516,  C316, -C116,  C116, -C316,  C516, -C716
};
```

リスト2　DCT処理のCソース・コード(固定小数点演算)

```
#define WIDTH           8
#define HEIGHT          8
#define IMGSIZE         WIDTH * HEIGHT

#define BytePerBlk      8
#define XCNT            WIDTH / BytePerBlk
#define YCNT            HEIGHT / BytePerBlk
#define BLKS            XCNT * YCNT

static int cos_table[] = {
  0x00005a82, 0x00005a82, 0x00005a82, 0x00005a82,
  0x00005a82, 0x00005a82, 0x00005a82, 0x00005a82,
  0x00007d8a, 0x00006a6d, 0x0000471c, 0x000018f8,
  0xffffe708, 0xffffb8e4, 0xffff9593, 0xffff8276,
  0x00007641, 0x000030fb, 0xffffcf05, 0xffff89bf,
  0xffff89bf, 0xffffcf05, 0x000030fb, 0x00007641,
  0x00006a6d, 0xffffe708, 0xffff8276, 0xffffb8e4,
  0x0000471c, 0x00007d8a, 0x000018f8, 0xffff9593,
  0x00005a82, 0xffffa57e, 0xffffa57e, 0x00005a82,
  0x00005a82, 0xffffa57e, 0xffffa57e, 0x00005a82,
  0x0000471c, 0xffff8276, 0x000018f8, 0x00006a6d,
  0xffff9593, 0xffffe708, 0x00007d8a, 0xffffb8e4,
  0x000030fb, 0xffff89bf, 0x00007641, 0xffffcf05,
  0xffffcf05, 0x00007641, 0xffff89bf, 0x000030fb,
  0x000018f8, 0xffffb8e4, 0x00006a6d, 0xffff8276,
  0x00007d8a, 0xffff9593, 0x0000471c, 0xffffe708
};

void dct_fixed8x8(int *in, int *out)
{
  int k, j;
  int *inptr, *tblptr;
  int tmp;

  for( j = 0; j < BytePerBlk; j++ ) {
    inptr = in + WIDTH * j;
    for( k = 0; k < BytePerBlk; k++ ) {
      tblptr = cos_table + k * BytePerBlk;
      tmp = inptr[ 0 ] * tblptr[ 0 ]
        + inptr[ 1 ] * tblptr[ 1 ]
        + inptr[ 2 ] * tblptr[ 2 ]
        + inptr[ 3 ] * tblptr[ 3 ]
        + inptr[ 4 ] * tblptr[ 4 ]
        + inptr[ 5 ] * tblptr[ 5 ]
        + inptr[ 6 ] * tblptr[ 6 ]
        + inptr[ 7 ] * tblptr[ 7 ];
      out[ WIDTH * k + j ] = tmp >> 16;
    }
  }
}
```

となり，$X \cdot {}^{t}C$は8×8ピクセルの横(X)方向の1次元DCT処理になります．さらに，その演算結果について，Cとの行列積をとることで，縦(Y)方向の1次元DCT処理を行えます(**図1**)．

この$X \cdot {}^{t}C \,(= {}^{t}(C \cdot {}^{t}X))$を実際にC言語の関数として実装した結果が**リスト1**です．これを**リスト2**のように固定小数点化します．さらに，余弦(cos)テーブルを見ると，同じ係数値が存在しており，

C14 ＝0x5a82　C18＝0x7641　C38＝0x30fb
C116＝0x7d8a　C316＝0x6a6d　C516＝0x471c
C716＝0x18f8

と置き換えてみると，規則性があることがわかると思います(**リスト3**)．例えば，それぞれの要素を$C(i,\ j)$とすると，j行0，4は加減算のみで計算可能です．jが偶数行のとき，i列0，7およびi列1，6は，それぞれ同じ係数になります．また，jが奇数列のときは，それらの符号が反転しているだけなので，これらを先に加減算処理し，そのあと乗算処理を行うことによって演算回数を減らすことができます．これが，いわゆるバタフライ演算を用いた高速DCTアルゴリズムです(**リスト4**)．

できるだけ早い段階(C言語記述の段階)で，アルゴリズムを最適化したり，実装内容を簡素化しておくと，後でDNAマトリックスの設計を行う際にも，作業が楽になります．

● DAPでサンプル記述を動作させる

さて，いよいよDAPDNA-2への実装作業に移ります．まずは，先ほどのDCTのCプログラムをDAP(RISCプロセッサ・