メガデモに見る小手先高速化技術

福 地　健 太 郎[†]

「メガデモ」とは、それまで誰も見た事がないような映像効果を実現しながら、1 枚のフロッピーディスクに収まるようなソフトウェアを指す。古くは Amiga や Commodore 64 上で動作するデモが数多く発表され、次第に PC へと場を移しながらも多くの若者がその技を競っていた。しかしプロセッサ能力の急激な発達と、3D グラフィックチップの急速な普及のため、低処理能力プロセッサの為の最適化技術も重要ではなくなってきた。しかし、どれ程プロセッサ能力が向上しようとも、何らかの形でパフォーマンスの不足に悩む局面は多く、こうした最適化技術は決してその意義を失ってはいない。本発表では、特に条件分岐を避けるための技法と、擬似的に並列処理を実現する技法をいくつか紹介する。

1. はじめに

「メガデモ (mega-demo)」とは、元々は家庭用コンピューター上で動作するデモンストレーションソフトウェアで、映像効果の出来を競うものである。古くは Amiga や Commodore 64 上で動作するデモが数多く発表され、当初は二次元画像のアニメーション技術を競うのが主流だった。使用されるプロセッサの能力が向上するに伴なって、三次元描画を高速に行うデモが主流となっていった。そのうちに活動の場を PC へ移しながら、多くのコーダー[☆]がその技を競い合い、コーダー達が腕を競う大会が世界各地で開催された。ちなみに、大会ではデモ全体のサイズが 1 メガバイト強のフロッピィディスク一枚に収まる事が規定となっており、メガデモの名前はこれに由来する。

しかしプロセッサ能力の急激な発達と、3D グラフィックチップの急速な普及のため、かつての職人芸的な雰囲気は薄れ、華美な映像と描画ポリゴン数を張り合うデモが増えた。それに伴い、低処理能力プロセッサの為の最適化技術も重要ではなくなってきた。メガデモ大会の開催頻度も低くなり、コーダーの多くが手を引き、こうした最適化技術はコードの可読性を低めるという批判もあってあまり顧みられなくなってきている。

とはいえ、どれ程プロセッサ能力が向上しようとも、何らかの形でパフォーマンスの不足に悩む局面はいまだ多く、こうした最適化技術は決してその意義を失ってはいない。

2. 背　　景

2.1 EffecTV

EffecTV[☆☆]は、筆者が開発しているビデオイフェクトソフトウェアで、PC 上でリアルタイムで映像を加工するものである。現在は Linux カーネル用に開発されており、市販のビデオキャプチャーカードを差した標準的な PC 上で動作する。

EffecTV の特徴は、ビデオイフェクトとメガデモを融合させ、派手な映像効果をリアルタイムに楽しめる点にある。個々の映像効果はよく知られたアルゴリズムを用いているが、背景差分抽出等の技法を採用し、人の動きにあわせた映像効果が楽しめる等、新しい映像表現を提供する。

EffecTV では従来の映像効果技術と違って、リアルタイムに効果を生成する事が要求される。機器構成上の上限である毎秒 30 フレームを達成するためには、徹底した最適化を施す必要がある。EffecTV が必要とする最適化技術はその性質上、二次元画像を扱うものが多い。筆者がこうした最適化技術に興味を抱くようになったのはこの為である。

また、EffecTV は移植性を考慮しており、そのほとんどを C 言語で記述している。その為、多くの最適化技法は、C 言語で記述可能なように考慮したものになっている。

† 東京工業大学情報理工学研究科数理・計算科学専攻
Tokyo Institute of Technology, Graduate School of Information Science and Engineering

☆ coder: プログラマーの意味だが、アルゴリズムを考案し、なおかつ徹底的なチューニング能力を兼ね備える人物という意味合いも含む

☆☆ http://effectv.sourceforge.net/

2.2 最適化技法について

最適化技法の効果は、プロセッサやシステム全体のアーキテクチャに強く依存する。旧来のメガデモでは、手作業でのループ展開や、やはり手作業で constant propagation 最適化を施したり、コードの自己書き換え等が良く使われた技法であった。しかし、近年のプロセッサアーキテクチャの変化に伴い、こうした様相は大きく変化した。パイプライン・スーパースカラアーキテクチャの採用や、様々な拡張命令 (MMX,SSE など) の追加によって、旧来の最適化技法のうち意味を成さなくなったものも多い。また、3D グラフィックスの描画技法等は、ポリゴン描画のアクセラレーション機能を持つビデオチップの普及により、コーダーが自分で記述する必要がなくなった。

しかし、こうした環境の急激な変化を経た現在でも使い回しの効く最適化技法はある。本論文ではこうした技術のうち、「条件分岐を回避する演算」と「擬似並列処理」を扱ったものを取り上げる。

3. 基 本 編

3.1 条件分岐の回避

近年の CPU はパイプラインの利用効率を高めるために、分岐予測機能を持つものが多い。予測が当たっている場合は大変効率が良い反面、予測が外れた時のペナルティは高い。分岐のパターンがあまり一定でないような局面においては、条件分岐命令は極力用いない方が良い。ここでは分岐予測を避ける為の基本的な技法で面白いものを紹介する。

二つの整数 x,y の最大値を求める命令列は、

```
max(x,y) := (x>y)? x: y;
```

となる。この命令列は通常条件分岐を含む命令列に変換される。これを、条件分岐の代りにビット演算を駆使して書き換えたものが次の命令列である。なお、以降断わりのない限り整数は 32 ビットで、2 の補数表現を用いているものとする。

```
int max(int x, int y)
{
    unsigned int a, b;
    a = x - y;
    b = (a>>31) - 1;
    return (x & y) + b;
}
```

順番に説明しよう。x と y の大小を比較する際に、まず $x-y$ を計算しておく。この値が 0 未満である場合は、$max(x,y)$ は y である。一方、この値が 0 以上の場合は $max(x,y)$ は x であるが、これは $(x-y)+y$ とも考えられる。そこで、$x-y$ が 0 未満なら 0 を、そうでなければ $x-y$ を返すような式を作ってやれば、その値に y を足すことで、望んだ結果を得られることになる。

$x-y$ の値を右に 31 ビットシフト[☆]すると、最下位ビットに元の値の最上位ビットが来る。元の値の最上位ビットは、$x-y$ が負なら 1、正なら 0 である。これらの値から 1 を引くと、$x-y$ が負なら 0、正なら 0xffffffff となる[☆☆]。この結果と、元の $x-y$ の値との論理積を取ると、負なら 0、正なら $x-y$ が得られる。

	$x-y$ が正	負
`(x-y)>>31`	0	1
`((x-y)>>31)-1`	0xffffffff	0
`((x-y)>>31)-1) & (x-y)`	x-y	0
`(((x-y)>>31)-1) & (x-y))+y`	x	y

以上の命令列はビット演算 2 回と加減算 3 回と、変更前の命令列に比べて命令数は増えている。しかし、2 個の乱数値を引数として与えるような場合だと、変更後の命令列の方が計算時間が短い場合もある。参考までに、PentiumIII500MHz の計算機で、gcc-2.95 を用いて、最適化オプションを “-O3 -funroll-loops” にして計測した結果、2 倍程度の速度向上が見られた。一方、分岐予測が全て当たる場合には、逆に 0.7 倍程度に速度が低下した。

3.2 擬似並列命令

メガデモでは画像処理が処理の大部分を占める。そのため、RGB それぞれの値を一度に処理する、パック演算のための技法が重宝される。これを汎用レジスタとビット演算で実現する様々な方法が開発された。今では MMX 等の、パック演算を行う為の拡張命令が整備されているために、この種の技法の必要性は薄れてしまったが、特殊レジスタとのロード/ストアの手間との兼ね合いから、場合によってはこうした技法が有利である事もある。ここでは、二つの画像を 1:1 で合成する「ハーフトーン合成」の技法を紹介する。

ハーフトーン合成では、二つの画素の RGB 各要素でそれぞれ相加平均をとる。RGB 各要素がそれぞれ 1 バイトで表されているとして、1 バイトづつ処理すれば簡単だが、多くのアーキテクチャではバイト単位の操作はコストが高い。そこで、RGB 各要素を一度に処理する技法が開発された。

ここで、RGB 値が 32 ビット整数値にパックされているものとする。普通に二つの整数値を足すと、繰り上りが発生した場合に、RGB 各要素の最下位ビットと干渉してしまう。また、加えた後に 2 で割った場合、RGB 各要素の最下位ビットが、隣の RGB 値の最上位ビットになってしまう。そこで、加算する前にあらかじめ両画素の RGB 各要素の最下位ビットをクリアしておく。その後に加算すれば、繰り上がりビットは干渉する事なく残る。そして右シフトすれば相加平均が得られる。(図 1)

[☆] このシフトは最上位ビットのコピーをしない。
[☆☆] なお、IA-32 の CPU では CWD 命令等を用いて同様の操作が可能であり、可能であればそちらの方が良い。

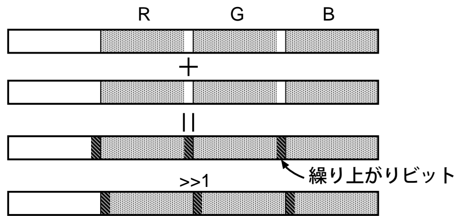

図 1　ハーフトーン合成の演算

しかしこの手法では、誤差が大きく、また即値を用いたビット演算が 2 回ある。次に述べる方法は、加算を論理演算で置き換える方法を用い、これらの欠点を補っている。

加算は論理演算では次のように置き換えられる。

$$x + y = (x \texttt{ xor } y) \texttt{ or } ((x \texttt{ and } y) \ << \ 1)$$

従って、

$$(x + y)/2 = ((x \texttt{ xor } y) \ >> \ 1) \texttt{ or } (x \texttt{ and } y)$$

と表せる。このことから、以下のような命令列でハーフトーン合成ができる。

```
((x ^ y) & 0xfefeff)>>1 | (x & y)
```

右シフトをする際に隣の要素に影響を与えないよう、各要素の最下位ビットをクリアする必要があるので、即値演算はやはり必要になるが、誤差は最小限に抑えられている。

4. 飽和加算

前節では二つの画像のハーフトーン合成を扱ったが、本節ではその応用として、飽和加算合成を扱う。飽和加算合成とは、RGB 各要素で加算を行い、上限値 (ここでは 255) を超えた場合は上限値に丸めるような加算を行う合成である。これもやはり各要素を一括して計算する技法がある。

まず、飽和加算に着目する。飽和加算は、

```
(x+y>255)?255:x+y;
```

という式で表わせるが、条件分岐を含んでおり好ましくない。そこでこれを除去することを考える。画像合成の場合、上式においては x と y の両方が 255 以下であるため、その和は最大でも `0x1ff` となる。従って、9 ビット目を見れば、加算においてその和が 255 を超えたかどうかが判定できる。このビットが 0 であれば和に対して何か操作をする必要はなく、1 であれば和を 255 に丸めれば良い。これを実現するのが次の命令列である。

```
a = x + y;
b = a & 0x100;
c = b - (b >> 8);
z = a | c;
```

2 行目で繰り上がりビットを検知し、3 行目でそこからマスクビット列を生成している。3 行目の操作により、繰り上がりビットが 0 の場合は `c = 0`、1 の場合は `c = 0xff` となる。そして 4 行目の操作で、和が 255 を超えている場合は、下位 8 ビットが全て 1 となり、飽和加算を実現している。ただし、この時点では 9 ビット目にゴミが残っているが、ひとまずこれを無視する。

次に、この操作を RGB 各要素で一括して行う。基本的には上記の操作を各要素毎に行えば良いのだが、繰り上がりビットを検知するためには、あらかじめ該当するビットを 0 にしておく必要がある。

```
a = (x & 0xfefeff) + (y & 0xfefeff);
b = a & 0x1010100;
c = b - (b >> 8);
z = a | c;
```

2 行目で繰り上がりビットの検出を RGB 各要素で同時に行っている。あとは同様の処理を RGB 各要素で一括して行えばよい。

さて、ここで今まで無視してきた、繰り上がりビットが結果に残ってしまう問題について考える。実際、上記の命令列で計算すると、9,17,25 ビット目に繰り上がりビットが残ってしまうのだが、9,17 ビット目については、それぞれは G,R 要素の最下位ビットでもある。実際の画面上では、RGB の値が 1 増減したとしても、人間の目にはわからない。そこで、これは無視してしまっても差し支えのない場面は多い。25 ビット目に関しても、処理上特に問題がなければこれも無視してしまえる。

4.1 計測結果

上記の最適化の効果を測定した。測定環境は Pentium-mIII 500MHz を使用した PC で、Linux2.4 カーネルを用いた環境で、gcc-2.95.4 を使用し、最適化オプションを “-O3 -funroll-loops” にしてコンパイルして計測したものである。計測したのは、幅 320 ピクセル高さ 240 ピクセル、計 76800 ピクセルの画像どうしの飽和加算を 500 回行うのに要した時間である。比較するのは、

(1) 条件分岐を用いてバイト単位で計算
(2) テーブル参照を用いてバイト単位で計算
(3) 前節のアルゴリズムを使用
(4) MMX 命令を使い、2 ピクセルづつ (qword 単位) 処理
(5) MMX 命令を使い、1 ピクセルづつ (dword 単位) 処理

の五種である。各ピクセルの RGB 値は乱数で生成した。その為、条件分岐で処理した場合は分岐予測の的中率は低いものと予想される。

グラフを見ると、条件分岐を含む演算はやはりコストが高い事がわかる。(2) が (3) に比べて速度向上の度合が低いのはテーブル参照のコストのためだが、計測に用いたプログラムでは、加算される数値が合せて 600 キロバイトの配列に収められているため、テーブルを参照する際のキャッシュヒット率が極めて低いと

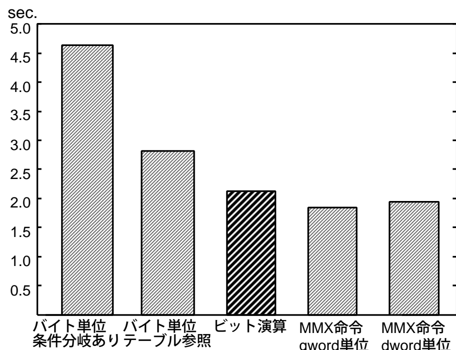

図 2 飽和加算アルゴリズムの計測結果

いう事が考えられる。したがって、使用する場面によっては充分効果のある最適化である場合もあるだろう。

グラフでは、(3) と (4),(5) とであまり差が無いように見えるが、これは計測時間のうち、メモリアクセスのコストがほとんどを占めているためである。最高速度を出しているのは (4) であるが、MMX レジスタは 64 ビット幅を持っているので、一回の演算で 2 ピクセル分処理できる。そこで、この数値を基準として、(3) と、MMX 命令を使用しながらも 1 ピクセルづつ処理した (5) とを比較してみると、実行時間の差にして、(3) は 272 ミリ秒余計にかかっているのに対し、(5) は 123 ミリ秒の増加となっており、やはり MMX 命令の方が効率が速いことがわかる。(3) が遅い原因としては、命令数が増加している事に加えて、レジスタ間の依存関係が強く、レジスタ数の少ない CPU では不利であることも挙げられる。

5. 二 値 化

RGB 画像で、RGB それぞれの値で別々に二値化を行うことを考える。言い換えると入力画像を 8 色に減色する事が目的となる。

単純には、各ピクセルに対し、`0x808080` との論理積を取る事で実現できるが、二値化の際の閾値は RGB それぞれで独立に設定できる方が好ましい☆。前章のアルゴリズムを応用すると、これを解決する事ができる。

`x` が入力画素、`y` が出力画素、`t` が各要素の閾値をパックしたものとすると、

```
a = x | 0x1010100;
b = (x - t) & 0x1010100;
y = b - (b>>8);
```

今度は前章とは逆で、各要素の 9 ビット目に相当するビットをあらかじめ立てておく (1 行目)。その後に、閾値を引くと (各要素の閾値は、最下位ビットが 0 である事が条件となる)、各要素で引き算の結果が負になった場合、9 ビット目は 0 になる。つまりこのビットはキャリーフラグを反転したものと考えることができ。その後、このビットだけ取り出し、8 ビット右シフトしたものを引けば、差が正の場合は 255、負の場合は 0 が得られ、二値化できる。

☆ 青色の閾値は低く設定した方が良いなど、様々な要求がある。

6. 背 景 差 分

入力画像から背景画像を消し、前景のみを取り出す処理を背景差分と呼び、多くの手法が知られている。ここでは、“Color-key” と呼ばれる手法で、RGB 各要素で入力画像と背景画像の差を取り、その絶対値が閾値を超えたら前景画像とみなすものを取り上げる。

このアルゴリズムを単純に実装すると、

```
dr = abs(ir - br);
dg = abs(ig - bg);
db = abs(ib - bb);
if(dr > t || dg > t || db > t) {
    ...
}
```

といったものになる。しかし、絶対値の計算は条件分岐を含むので、これを除去したい。

ここで、$-x = (\mathrm{not}\ x) + 1$ であることと、not x はこの場合 x xor 0xff で代用可能であることを利用すれば、これまでに述べてきたような方法で実現できる。

`x` が入力画素、`y` が背景画素とする。

```
a = x | 0x1010100;
b = y & 0xfefeff;
c = x - y;
d = c & 0x1010100;
e = d - (d>>8);
f = ~e; // not e
g = c ^ f;
```

1,2,3 行目の処理で、入力画素から背景画素を引いた時に各要素の値が正になったかどうかが、9,17,25 ビット目からわかる (`d`)。5 行目でマスクビット列 (`e`) を生成する。反転させたいのは負の要素なのでマスクビット列を反転させてから (`f`)、元々の差と排他論理和をとる。こうする事で、各要素の差の絶対値が取れる (`g`) ので、あとは前節のようなやり方で閾値を引くか、あるいは単純に各要素の先頭数ビットのみを抜き出して判定すればよい。

なお、正しくは排他論理和で反転させた後 1 を加えなければならないのだが、この手法ではそもそも各要素の最下位ビットは捨てられているために精度の向上には寄与しないので、無視している。

7. 直 線 描 画

Bresenham の直線描画アルゴリズムと言えば、描画アルゴリズムの初学者には深い感銘を与える、名アルゴリズムであると筆者は思っている。直線の描画とい

う、一見すると実数演算を伴わないと実現できないように思える処理を、整数演算のみを用いて実現でき、なおかつ誤差もなく実行速度も速く、アルゴリズムの花形と言っても過言ではない。アルゴリズムを簡単におさらいすると、x 切片 dx,y 切片 dy で $dy < dx$ を満すような直線を描画する際、単純に計算すると、x を 1 づつ増加する度に y に dy/dx を足していき、x, y に点を打つ。

```
for(x=0; x<=dx; x++) {
     y += dy/dx;
     plot(x,y);
}
```

ここで y の増分を計算をする部分を以下のように書き換える。

```
     t += dy/dx;
     if(t>1) {
          y++;
          t -= 1.0;
     }
```

ここで、1 行目の式の両辺を dx 倍して、残りの行もそれに倣うと

```
     t += dy;
     if(t>dx) {
          y++;
          t -= dx;
     }
```

となり、全ての変数は整数値をとるようになる。これがブレゼンハムの直線描画アルゴリズムである。

しかし、このアルゴリズムは条件分岐を含んでいる。直線の傾きによっては分岐のパターンは複雑になり、分岐予測が外れやすくなる。最内ループが非常に短いが故、このコストは大きい。

回避策としては二通りあり、一つはこれまでに述べたような、算術演算やビット演算を駆使してこれらの計算を実現する方法である。これは以下のように書ける。まず、元のアルゴリズムの変数 `t` の増減方向を逆にしておく。

```
t = dx;
for(x=0; x<= dx; x++) {
     t -= dy;
     if(t<0) {
          y++;
          t += dx;
     }
}
```

これの内部を書き換えると

```
     t -= dy;
     mask = t>>31;
     y -= mask;
     t += dx & mask;
```

2 行目の式は、$t < 0$ の時に `mask = 0xffffffff` となる事を前提としている。

もう一つの方法は、固定小数点演算で計算する方法である。この場合、精度は犠牲になり、また値域も限られる。

これらの手法を比較するための計測をしたのが、図 3 である。これは、dx=10000 ピクセル, dy は 0 から 10000 ピクセル (101 ピクセル間隔[☆]) の直線をそれぞれ 1000 本引くのに要した時間を測定したものである (20 回計測して最低値を採用)。

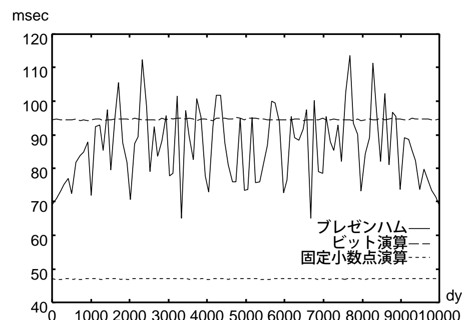

図 3 直線描画アルゴリズムの比較

まず通常のブレゼンハムアルゴリズムの計測結果を見てみると、直線の傾きによって実行時間が大きく異なっていることがわかる。最低と最大で約 2 倍程度の差があり、分岐予測の的中率によって効率が大きく異なることがよくわかる。一方、回避策 1 と回避策 2 は直線の傾きの影響は全く受けておらず、安定したアルゴリズムだと言える。しかし、回避策 1 は、大雑把に平均して比較すればブレゼンハムアルゴリズムよりも実行速度は劣っている。これは、命令数が増えている上に命令間依存度が高い命令列であるが故にコストがかかっているためである。

一方、回避策 2 では、基本的に最内ループでは一回の加算と一回のビットシフト (固定小数点数のゲタを除く処理) だけなので、回避策 1 よりかは速い。ブレゼンハムアルゴリズムで、分岐予測が全て的中した場合に比べても速い理由は、演算が単純なせいかコンパイラによって充分に最適化されているためである。しかし、ブレゼンハムアルゴリズムをきちんと最適化しても、回避策 2 より効率が良くなることはないと予測する。

8. 付録:ライフゲーム

John Horton Conway のライフゲームにおいて、セルの生死判定と更新をビット演算に置き換える手法を紹介する。ここまで来ると遊びの領域だが、面白い応用例としてご観賞いただきたい。

☆ この間隔は、素数だと分岐予測の的中率がバラけるであろうという予測の元に適当に決めた。

ライフゲームの生死判定は、

- セルが生きている (=1) 場合、近傍セルが 2 か 3 個の場合のみ生き残る
- セルが空 (=0) の場合、近傍セルが 3 個の場合に新しくセルが誕生する

となっている。さて、ここで自分を含む近傍 9 セルのうち、生きているセルの数を sum、自分のセルの状態を x とし、次の時刻のセルの状態を y とすると、この規則は

```
if(x==0) {
    if(sum==3) {
        y=1;
    } else {
        y=0;
    }
} else {
    if(sum<3 || sum>4) {
        y=0;
    } else {
        y=1;
    }
}
```

と書ける。しかしライフゲームの盤面は一般に極めて乱雑で、分岐予測の的中率は高いとは思えない。これをビット演算に置き換えると、

```
y = (sum==3) | ((x!=0)&(sum==4));
```

と表わせる。これは、C 言語では論理式の返り値が真のときは 1、偽のときは 0 になる事を利用している。上式で問題となるのは、実際にコンパイルされたコードがどれ程効率が良いかである。幸い IA-32 の CPU では、論理式の返り値は cmp 命令と set 命令との組み合せで実現されており、命令数は 10 命令と、効率の良いコードが生成されている☆。

9. 議　論

条件分岐を避けたり、擬似的に並列処理を行う技法をいくつか紹介したが、これらの技法に共通して言えるのは、適用する場面を慎重に選ぶ必要があるという点である。

一つには実行速度の問題がある。条件分岐の回避の場合、置き換える条件分岐の分岐予測の的中率を考えないと、かえって遅くなる場面もある。大抵の場合、分岐予測がよく的中するのであれば、命令数が増え命令依存が複雑になるような置き換えをするよりも、条件分岐を使った方が速い。もし仮に「分岐予測が当らない」事が実行時に予測可能であれば、実行時に二つのアルゴリズムを使い分けるといった使用法も考えられるだろう。

もう一つは誤差の問題である。紹介した技法のいくつかは誤差を無視している。これは画像処理であるが故に無視できるものであり、一般には適用できる領域はそれ程は多くないだろう。発表後の質疑応答にて、こうした最適化をコンパイラに自動化させられないかという意見があったが、これを実現するためには、「誤差をどこそこまで許す」「値域はどこそこまで狭くなってもよい」といった情報をコンパイラに与える必要が出てくるだろう。

本論文で紹介した技法は、MMX や MME 命令等の拡張命令を極力使わない、一般性のあるものを選んだ。いずれも C 言語で記述可能なので、大抵の環境においては少なくとも実行可能ではある。とはいえ、やはり専用に命令された拡張命令が使えるのであればそれを使用するに越した事はない。ただ、紹介した技法を応用する事で、拡張命令だけでは効率的に実現できない処理もあるだろう。

質疑応答で PentiumPro 以降の Pentium シリーズの CPU に追加された「CMOV(Conditional move) 命令」についてのコメントを求められたが、実際 CMOV 命令は今日のメガデモでも大変重宝される命令で、条件分岐を使わずにメモリ上の値を更新できる便利な命令である。8 章のライフゲームの例は、CMOV 命令をビット演算で代用した例である。

シンポジウムで発表した際に多く聞かれた声として、「懐しい」「みんなよくこういうのやっていた」というのがある。実際、ここで紹介した技法の基礎は、メガデモのコーダー達が 8 ビット機の頃から育ててきたものであり、さらに歴史を遡ればコンピューター創世記の頃から連綿と繰り返して使用され、再発明されてきたものでもある。残念な事に、緻密な最適化技術は往々にして企業秘密の中に隠されてしまう。ピープホール的な最適化技術でも、時にはソフトウェアの効率を極端に変えてしまう。最適化技術が少しでも広範に知れ渡り、共有の財産となる事を願う。

謝　辞

最適化技術について様々な技法を公開してくださり、また議論にお付き合いいただいている、光成滋生氏他「せっかちな人々☆☆」に感謝いたします。

☆ gcc-2.95.4 を使用

☆☆ http://homepage1.nifty.com/herumi/