適応フィルタのSIMD最適化

茂木 和洋 @ まるも製作所

自己紹介

- 今回は省略
- 初めての方は、#1 の資料を参照

発表内容

- 適応フィルタとは
- 適応フィルタの問題点（速度面で）
- SIMD比較命令でマスク処理
- ベンチマーク

適応フィルタとは [1/13]

- 固定のフィルタではなく、入力値によって処理を変更し、最適な結果を求める
- 例
  - 基準値との差異を閾値と比較して、参照画素として使うか使わないかを切り替える
  - 最小自乗法でフィッティングしてフィルタ係数自体を動的に作成する
  - 他いろいろ

適応フィルタとは [2/13]

- 次のスライドから実際の例で紹介

適応フィルタとは [3/13]

- lennaを素材に使う

適応フィルタとは [4/13]

- 今回はグレースケールで扱います

適応フィルタとは [5/13]

- 8bitグレイスケールに、±12のランダムノイズが加算されている場合について考える

適応フィルタとは [6/13]

0	1	1	1	0
1	2	2	2	1
1	2	4	2	1
1	2	2	2	1
0	1	1	1	0

- 左のガウスフィルタをノイズ加算済み画像に適用すると

適応フィルタとは [7/13]

- ノイズはある程度消えるものの、絵がぼけてしまう
- 次のスライドで並べて比較

適応フィルタとは [8/13]

適応フィルタとは [9/13]

- 絵がボケるのは、エッジ部分の輝度値に差が大きいところも考慮せずに畳み込んでいるため
- ノイズは±12なので、一定の閾値を設定し、中央画素との差が閾値を超えた場合は畳み込みに使わなければボケない

適応フィルタとは [10/13]

- 前のスライドのアルゴリズムで処理した場合
- 次のスライドで並べて表示

適応フィルタとは [11/13]

適応フィルタとは [12/13]

- 判りにくかったかもしれないので
- 前スライドは単純な5x5ガウシアンフィルタ [左] とエッジ考慮5x5ガウシアン [右]
- 次スライドはノイズ画像 [左] とエッジ考慮5x5ガウシアン [右] を並べて表示

適応フィルタとは [13/13]

適応フィルタの問題 [1/6]

- 主に、処理負荷に関して
- 実際のコードで単純な5x5ガウシアンとエッジ考慮5x5ガウシアンで比較すると

適応フィルタの問題 [2/6]

```
for (y=0;y<height;y++) {
        for (x=0;x<width;x++) {
                w  =            pr2[x-1]   + pr2[x]   + pr2[x+1];
                w += pre[x-2] + pre[x-1]*2 + pre[x]*2 + pre[x+1]*2 + pre[x+2];
                w += src[x-2] + src[x-1]*2 + src[x]*4 + src[x+1]*2 + src[x+2];
                w += nxt[x-2] + nxt[x-1]*2 + nxt[x]*2 + nxt[x+1]*2 + nxt[x+2];
                w +=            nx2[x-1]   + nx2[x]   + nx2[x+1];
                dst[x] = clip_u8((w+16)/32);
        }
        dst += width;
        src += src_pitch;
        pr2 += src_pitch;
        pre += src_pitch;
        nxt += src_pitch;
        nx2 += src_pitch;
}
```

- 単純な5x5ガウシアンの場合

適応フィルタの問題 [3/6]

```
static int thresholding(int val, int center, int threshold) {
        if (abs(val-center) > threshold) { return center; }
        return val;
}
<途中省略>
        for (y=0;y<height;y++) {
                for (x=0;x<width;x++) {

                        int center = src[x];

                        w  = thresholding(pr2[x-1], center, threshold);
                        w += thresholding(pr2[x],   center, threshold);
                        w += thresholding(pr2[x+1], center, threshold);
<以降省略>
```

- エッジ考慮ガウシアンの場合

適応フィルタの問題 [4/6]

- ループの一番奥に分岐が入る
- 参照画素毎に処理が切り替わるので、SIMDに向かない

適応フィルタの問題 [5/6]

- 最適化原則
  1. ディスクを読むな
  2. [キャッシュよりも大きな] メモリを読むな
  3. 分岐するな
  4. 除算するな
  5. 乗算するな

適応フィルタの問題 [6/6]

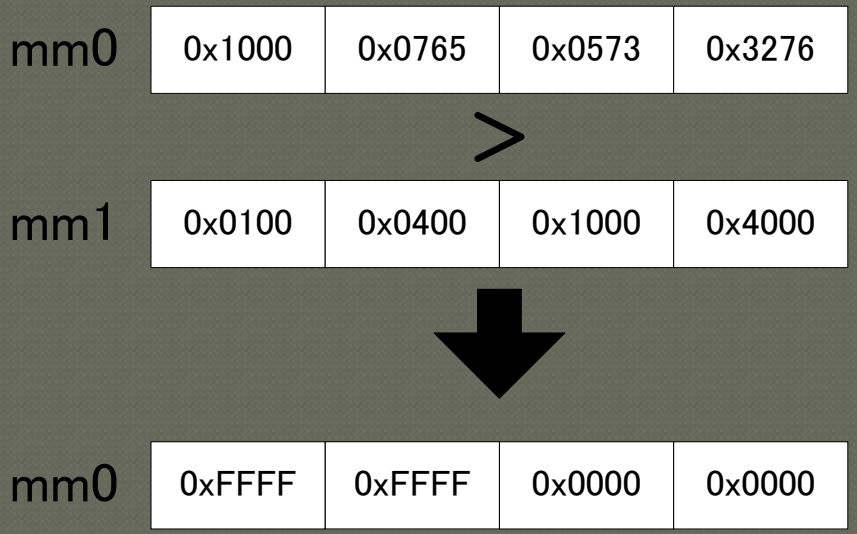
- 最適化原則
  1. ディスクを読むな
  2. [キャッシュよりも大きな] メモリを読むな
  3. **分岐するな**
  4. 除算するな
  5. 乗算するな

SIMDマスク処理 [1/5]

- 今回の適応フィルタならばSIMD化可能
- SIMD比較命令でビットマスクを作り、参照画素を選択すればよい
- 制御構造の分岐を除去でき、SIMDで並列処理できるので高速化する

SIMDマスク処理 [2/5]

PCMPGTW mm0, mm1 の例



- PCMPGTx, PCMPEQx は比較を行い、対応レジスタのビットを全て立てたり、全て落としたりする命令

SIMDマスク処理 [3/5]

```
static int thresholding(int val, int center, int threshold) {
        if (abs(val-center) > threshold) { return center; }
        return val;
}
<途中省略>
        for (y=0;y<height;y++) {
                for (x=0;x<width;x++) {

                        int center = src[x];

                        w  = thresholding(pr2[x-1], center, threshold);
                        w += thresholding(pr2[x],   center, threshold);
                        w += thresholding(pr2[x+1], center, threshold);
<以降省略>
```

- この処理をSIMD比較命令で8並列処理すると

SIMDマスク処理 [4/5]

```
;// xmm0 に threshold 設定済み
;// xmm1 に 全て 0xffff 設定済み
;// xmm2 に center 設定済み
;// esi が参照画素アドレス
PMOVZXBW xmm4, [esi]; // WORD変換
MOVDQA   xmm6, xmm4;
PSUBW    xmm4, xmm2; // val-center
PABSW    xmm4, xmm4; // 絶対値に変換
PCMPGTW  xmm4, xmm0; // 閾値と比較
MOVDQA   xmm5, xmm4;
PXOR     xmm4, xmm1; // mask 反転
PAND     xmm5, xmm2;
PAND     xmm4, xmm6;
POR      xmm4, xmm5;
```

- 左のコード断片で、centerとvalの選択処理を8画素並列に実行可能
- xmm4に選択後画素が入る

SIMDマスク処理 [5/5]

- 注意事項
  - PMOVZXBW (BYTE 配列をゼロ拡張して PACKED WORD読み込み) は SSE4.1 の命令
  - PABSW (PACKED WORD を絶対値変換) は SSSE3 の命令

ベンチマーク [1/2]

- VisualStudio 2010 (cl 16.00.30319) の場合 (SandyBridge Core i5 2500K)
  - C実装：　　13.587 sec
  - SSE4.1 実装：　　1.186 sec
- 512x512画像1024回実行の消費時間
- /Ox /Oi /arch:SSE2 /MT を指定

ベンチマーク [2/2]

- Intel Compiler XE (icl 12.0.3.175)
  - C実装：　　2.278 sec
  - SSE4.1 実装：　　1.186 sec

- 環境は同一
- /fast /MT を指定

Intel Compiler の ASM

```
$LN1004:
  pmovzxbd xmm3, DWORD PTR [eax+edi]
$LN1005:
  psubd    xmm6, xmm3
$LN1006:
  movdqa   XMMWORD PTR [80+esp], xmm0
$LN1007:
  pabsd    xmm0, xmm6
$LN1008:
  movdqa   xmm6, XMMWORD PTR [32+esp]
$LN1009:
  pcmpgtd  xmm0, xmm6
$LN1010:
  pblendvb xmm5, xmm3, xmm0
$LN1011:
  movdqa   XMMWORD PTR [96+esp], xmm5
```

- 左は /Fa で確認したASM出力の一部
- この程度のコードであれば自動でSIMD化してくれる模様
- DWORDで計算しているので、並列度は手で書いた場合の半分

まとめ

- Intel Compiler は結構がんばってくれる
- それでもまだ人間の方が賢い
- ループの奥の分岐を潰してSIMD化できれば、やらない場合よりも10倍程度速くなる (こともある)

- 今回のコード等は次のURIに置いてます。
  - http://www.marumo.ne.jp/junk/x86opti/2011_01_2nd/x86opti_02_kzmogi.zip

補足 [1/2]

- 会場で指摘してもらった以下二点の追加実験結果を記載します
  - PBLENDVB [SSE4.1] を使った方が、選択処理は楽なのでは
  - Intel Compiler が WORD 処理をしてくれないのは、stdlib.h の abs() が原因ではないか

補足 [2/2]

	CL	ICL
C [OLD]	13.587	2.278
C [NEW]	29.391	2.044
SSE4.1 [OLD]	1.186	1.186
SSE4.1 [NEW]	0.983	0.998

- PBLENDVB でかなりの高速化
- abs() 排除でIntel CもWORD処理をしてくれるようになったが、VCがさらに遅くなる
- やっぱり手で書いた方が速い