 @tomoaki_teshima updated at 2020-12-08

OpenCV Advent Calendar 2020 Day 6

OpenCVの丸め誤差bit-exactnessについて

OpenCV

はじめに

- 本日はOpenCVのbit-exactnessについて解説します。
- 本記事はOpenCV アドベントカレンダー 2020、第6日目の記事です。
- 他の記事は目次を参照して下さい。

TL;DR

- 基本的にOpenCVのAPIはbit-exactnessを保証しない

丸め誤差とbit-exactness

- 一般的に、浮動小数演算は、ピッタリの結果で表すことができない。
- 演算結果が、離散的に表現された浮動小数点数の間に陥ることが多々ある。

- このとき、どちらの値に丸め込もうとも、実数とは違う結果になるため、「丸め誤差」が生じる
- この丸め誤差が十分小さければ、丸め誤差のことを気にする必要は無い。一方で、丸め誤差が目的に対して大きい場合は、ちゃんと丸め誤差を考慮に入れる必要がある
- 一方で OpenCV で話題に上がっている、bit-exactness とは、プラットフォーム(アーキテクチャ/CPU/GPU/SIMD演算などなど)が変わっても、bit単位で同じ値を出力するか、を問いている。
- 現在、OpenCVでは、OpenCV Evolutionと題して、改善案を議論している。そのOE-15はOpenCVの関数のbit-exactnessについて提案されている
- ちなみに、bit-exactnessと、実行毎に結果がブレる挙動は全く別の問題である。
  - 色んな人がよく誤解するのですが、実行するたびに結果が変わる挙動は全く別の問題です。ですが、丸め誤差と混同する人をよく見ます
  - 浮動小数点演算を含んでようと、なかろうと、プログラムの演算結果は、何度実行しても同じであることが期待されます。
  - こちらは丸め誤差と違って、deterministic(決定的)な挙動、と言ったりします。
  - non-deterministicな挙動の場合はマルチスレッドの制御に失敗してたり、未初期化変数を踏んでたりする可能性があります。

OpenCVでのbit-exactについて

- 一般的に、プラットフォームが変わるとbit-exactな結果は期待できない。
- もちろん、整数演算に限ればbit-exactな結果を期待できる。ただし、入力と出力が整数(例えばCV_8UC3)だからと言って、内部に浮動小数点演算が存在しない、とは限らない。
- 先に述べたとおり、丸め処理はプラットフォームにより固有で、残念ながらbit-exactを保証できない
- 実際には、色変換Lab2RGBがOpenCV 3.3.1から導入され、resizeでのbit-exactnessが3.4.0で導入されました。
- ここでは、調べた範囲内で、bit-exactnessが保証されてるケース、一部保証されてるケース、保証されないケースを紹介する

どんな場合でもbit-exactを保証

- `transpose`
- `split` / `merge`
- `flip`
- `reshape`
- 当たり前というか、画素値をメモリから読み込むものの、一切処理をせずに、別の並び、メモリアドレスに保存する処理です。
- こいつらはどんな入力、プラットフォームであろうとbit-exactnessが保証されます。
- 裏を返すと、多分この5個ぐらいしか無いと思う。

一部bit-exactを保証するケースがある

- `cvtColor`（`RGB2GRAY`、`RGB2HSV` など一部の色変換)
- `imwrite` (pngやbmpなど、出力ファイルの拡張子次第)
- `resize` (補間モードに `cv::INTER_LINEAR_EXACT` もしくは `cv::INTER_NEAREST_EXACT` を指定した場合)
- `GaussianBlur`
- 今回の主眼である、bit-exactnessが保証されるケースです。
- ポイントは、「保証されるケースがある」だけであって、入力やモードによっては保証されない場合があります。
- CPU版はbit-exactだけれど、OpenCL(UMat)実装とはbit-exactでは無い、などの罠もありえます。

未検証(bit-exactは保証されない)

- その他全部がこのジャンルに入ります。基本的にbit-exactは保証されない前提です。

bit-exactの落とし穴

- あまり気をつけることが無いので、bit-exactとはどんなものか、筆者がハマった落とし穴を書いておきます。

`imwrite` がbit-exactなのは `bmp` と `png` ぐらい

- 当然ですが、 `jpeg` で保存する場合は、エンコーディング処理が浮動小数点演算だらけです。
- bit-exactな結果を書き出すなら、 `png` か `bmp` で保存しましょう。

bit-exactnessを検証するには、別アーキテクチャが必要

- https://github.com/opencv/opencv/pull/10921#pullrequestreview-99294475

> "tolerance = 0" is not enough to guarantee bit-exact results. At least not between platforms.

- テストで、「許容誤差0のテストを実装して、PASSしたからbit-exactだよ！」という主張はできないのです。
- PCとラズパイ、みたいにアーキテクチャが違う2つのプラットフォームで試さないとbit-exactだとは主張できないのです

bit-exactの仕組み

- 中身はいわゆる固定小数点演算です。あまり難しいことはしていません。
- 例えば、グレースケール化を考えましょう。
- OpenCV(master)では、以下の演算が行われます。

$$gray = B \times 0.114 + G \times 0.587 + R \times 0.299$$

```
//constants for conversion from/to RGB and Gray, YUV, YCrCb according to BT.601
static const float B2YF = 0.114f;
static const float G2YF = 0.587f;
static const float R2YF = 0.299f;
```

- しかし、現状のmasterでは、これは浮動小数点演算ではなく、整数での積和が行われます。

```
num
{
    gray_shift = 15,
    yuv_shift = 14,
    xyz_shift = 12,
    R2Y = 4899, // == R2YF*16384
    G2Y = 9617, // == G2YF*16384
    B2Y = 1868, // == B2YF*16384
    RY15 =  9798, // == R2YF*32768 + 0.5
    GY15 = 19235, // == G2YF*32768 + 0.5
    BY15 =  3735, // == B2YF*32768 + 0.5
    BLOCK_SIZE = 256
};
```

```
    static const int BY = BY15;
    static const int GY = GY15;
    static const int RY = RY15;
    static const int shift = gray_shift;
    :
        const int coeffs0[] = { RY, GY, BY };
        for(int i = 0; i < 3; i++)
                coeffs[i] = (short)(_coeffs ? _coeffs[i] : coeffs0[i]);
        if(blueIdx == 0)
            std::swap(coeffs[0], coeffs[2]);

        CV_Assert(coeffs[0] + coeffs[1] + coeffs[2] == (1 << shift));
    :
        short cb = coeffs[0], cg = coeffs[1], cr = coeffs[2];
    :
            int b = src[0], g = src[1], r = src[2];
            ushort d = (ushort)CV_DESCALE((unsigned)(b*cb + g*cg + r*cr), shift); // ここ
            dst[0] = d;
```

- 最後の CV_DESCALE マクロは展開すると、以下のようになります。

```
#define  CV_DESCALE(x,n)     (((x) + (1 << ((n)-1))) >> (n))
  :
ushort d = (ushort)(((unsigned)(b*cb + g*cg + r*cr) + (1 << ((shift)-1))) >> (shift));
```

- const な値も展開すると、以下の通りです。

```
ushort d = (ushort)(((unsigned)(b*3735 + g*19235+ r*9798) + (16384)) >> (15));
```

- 最後の右15bitシフトは / 32768 に相当するので、前述の処理は以下の数式で表せます。

$$gray = b \times \frac{3735}{32768} + g \times \frac{19235}{32768} + r \times \frac{9798}{32768} + \frac{16384}{32768}$$

- 最後に$\frac{16384}{32768} = 0.5$を足してるのは、四捨五入の演算のためなので、書き換えると、以下の数式になります。

$$gray = b \times 0.114 + g \times 0.587 + r \times 0.299$$

- で、最初の数式と全く同じ形にたどり着きました。

$$gray = B \times 0.114 + G \times 0.587 + R \times 0.299$$

- え？じゃあ今までの演算は何だったの？って思うかも知れませんが、ポイントは、コード上の計算がすべて整数型で行われていることです。

```
ushort d = (ushort)(((unsigned)(b*cb + g*cg + r*cr)                  + (1 << ((shift)-1))) >
                          //  ^       ^       ^ 全て整数型の掛け算 ^ 整数型の足し算
```

- この処理には浮動小数点演算が無く、全て整数型の演算なので「丸め誤差」なるものも一切発生しません。

softfloat

- 前節みたいに、固定小数点演算を使ってる箇所もありますが、その場合は制約が一つだけあります。それは演算途中の値域がオーバーフローしない、ということです。
- 例えば色変換では使われる係数が固定であり、入力が符号なし8bitに固定されてますので、オーバーフローするしないはコンパイル時に確認できます。
- 一方で、 `resize` 関数では主に画像サイズと拡大率という2つの値で浮動小数点演算が行われることになり、どちらも範囲は定められていません。
- 固定小数点演算するには、範囲を固定する必要がありますので、このままでは固定小数点演算できません。
- という訳で、OpenCV内部では、 `softfloat` クラスが実装されています。
    - この `softfloat` クラスが何者かと言うと、 `float` をコンストラクタで受け、 `Cv32suf` という、 `float` と `uint` と `int` を `union` にした構造体で値を保持します。

softfloat.hpp

```
    /** @brief Construct from float */
    explicit softfloat( const float a ) { Cv32suf s; s.f = a; v = s.u; }
```

cvdef.h

```
typedef union Cv32suf
{
    int i;
    unsigned u;
    float f;
}
Cv32suf;
```

- そして、後の演算では、 `float` の演算を、全てコード上でエミュレート ~~という狂気の沙汰を~~ します

softfloat.cpp

```
// 浮動小数点演算の加算をエミュレートしたコード（抜粋）
static float32_t softfloat_addMagsF32( uint_fast32_t uiA, uint_fast32_t uiB )
{
    :
    /*------------------------------------------------------------------------
      第1引数と第2引数の符号ビットと指数部を取り出し
    *------------------------------------------------------------------------*/
```

```
    expA = expF32UI( uiA );
    sigA = fracF32UI( uiA );
    expB = expF32UI( uiB );
    sigB = fracF32UI( uiB );
    /*------------------------------------------------------------------------
      指数部の差分を計算
    *------------------------------------------------------------------------*/
    expDiff = expA - expB;
    if ( ! expDiff ) {
        /*--------------------------------------------------------------------
          指数部が同じ場合の処理
        *--------------------------------------------------------------------*/
         :
    } else {
        /*--------------------------------------------------------------------
          指数部が異なる場合の処理（＝桁あわせから始める）
        *--------------------------------------------------------------------*/
         :
        if ( expDiff < 0 ) {
            /*----------------------------------------------------------------
              第2引数の絶対値が第1引数より大きい場合
            *----------------------------------------------------------------*/
            :
        } else {
            /*----------------------------------------------------------------
              第1引数の絶対値が第2引数より大きい場合
            *----------------------------------------------------------------*/
            :
        }
         :
    }
    return softfloat_roundPackToF32( signZ, expZ, sigZ );
    /*------------------------------------------------------------------------
     結果がNaNになる場合
    *------------------------------------------------------------------------*/
 propagateNaN:
    uiZ = softfloat_propagateNaNF32UI( uiA, uiB );
 uiZ:
    return float32_t::fromRaw(uiZ);
}
```

- 「昔はCPUに浮動小数点演算器なんかが無くて、それでコプロセッサの x87 が出てきた。それまでは、浮動小数点演算を全てコードでエミュレーションしていた」、っ

て教科書で読んだり昔の人に聞いたことがありますが、2020年にもなり、浮動小数点演算どころか、1クロックで何FLOPs計算できるかを競う時代に、敢えて全てコード上で浮動小数点演算をエミュレーションするとは！！！
    - ちなみにOpenCVに `softfloat` がマージされたのは2017年の話
- とは言え、たくさんFLOPSを稼ぐよりも、丸め誤差freeに正確な計算を行いたい、というニーズは理解できます。

まとめ

- 丸め誤差はハマると奥が深くて闇を見ます。
- プラットフォームが変わっても、丸め誤差に影響されないAPIの、ごく一部を紹介しました。
- 抜けがあると思いますので、丸め誤差freeを気にされる型はOpenCVの実装内部を覗いてみるのが良いのではないでしょうか。

補足

- プラットフォーム間でのbit-exactnessは一部の関数で保証されますが、グレースケール化のパラメータなど、一部のパラメータは3系列、4系列で微妙に違うパラメータが使われていますので、OpenCVのバージョンを跨いでのbit-exactnessは保証されてません。
- まさにバージョン間の違い踏み抜いたissue

参考URL

- OpenCVのログを漁ってbit-exact実装を盛り込んだPRを引っ張り出したが、多分に漏れがある。
- `resize` 関数をbit-exactにしたcommit
- `resize` 関数をbit-exactにしたPR
- `Lab` と `RGB` 色空間の変換をbit-exactにしたPR