CUDAを用いたガウシアンフィルタの実装とパフォーマンス検討

CUDA Implemention of Gaussian Filtering with examinating performance

09/12

※CUDAの勉強用にGaussianFilteringをやってみたときの文書です

2D Gaussian Filter → 1D Gaussian FIlter x2

実装対象

2D Gaussian Filter

$$\frac{1}{sum_weight}\sum_y \sum_x e^{-\frac{\Delta x^2+\Delta y^2}{2\sigma^2}} L(x,y)$$

は2つの1Dフィルタに分離できる.

L(x,y)は座標(x,y)での画素値,
(Δx,Δy)は着目画素位置からの(x,y)の相対座標,
sum_weightはフィルタ窓内でのフィルタ値の総和,
σはガウス関数の標準偏差.

分離に関係ないところを省略して書くと,

$$\sum_y \sum_x e^{\Delta x^2+\Delta y^2} L(x,y)$$

$$= \sum_y \sum_x e^{\Delta x^2} e^{\Delta y^2} L(x,y)$$

$$= \sum_y e^{\Delta y^2} \left(\sum_x e^{\Delta x^2} L(x,y) \right)$$

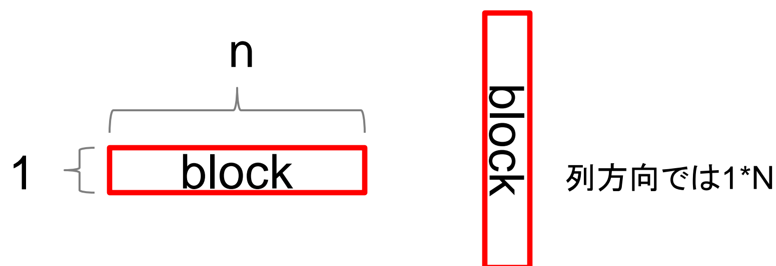
となり,
X方向に1D Gaussain Filterを施し,
その結果に
Y方向の1D Gaussian Filterを施したもの
と等しい.

目的等

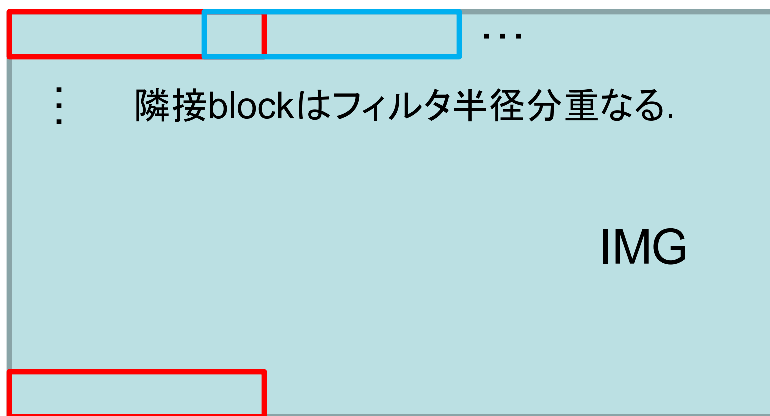
SDKサンプルconvolutionSeparable等があるが，
学習目的であるため，
自分で実装して，計算時間短縮方法等を学ぶ．

今回実装したkernel関数では，blockサイズはシンプルにN*1.

1threadが1pixel分のぼけた画素値を計算．



列方向では1*N



処理対象画像データは
host側ではOpenCVの
IplImage（8bit3ch）で持っている．

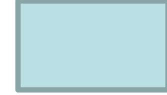
今回はdevice memory上でも
ch毎の画素値は8bitで扱う．

比較対象として
OpenCVのcvSmooth()の
処理時間を計測．

実装1：素直な実装

まずは原理通りに実装.
行方向，つづけて列方向にフィルタリング

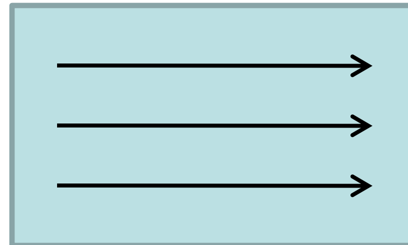
使用device memoy



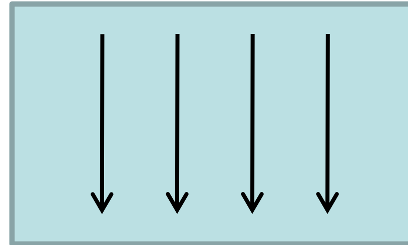
kernel関数call

1. row-dir filter
2. colmn-dir filter

第１のkernel関数が
行方向フィルタリングを行い



第２のkernel関数が
列方向フィルタリングを行う



result

•行方向にくらべて，列方向がかなり遅い

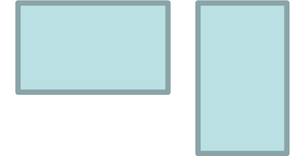
block

列方向では，ブロック内threadが離れた位置をアクセスするために
別々のトランザクションになってしまうためと思われる.

実装2:よくない転置方法

列方向の離れた位置へのアクセスの解消をめざす

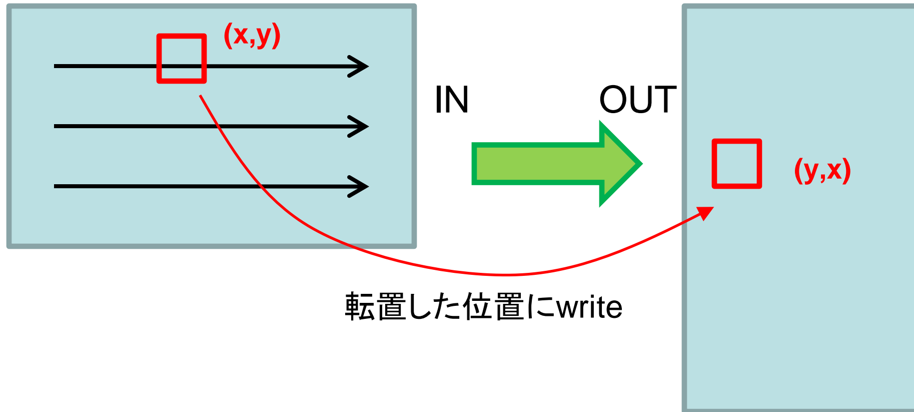
使用device memoy



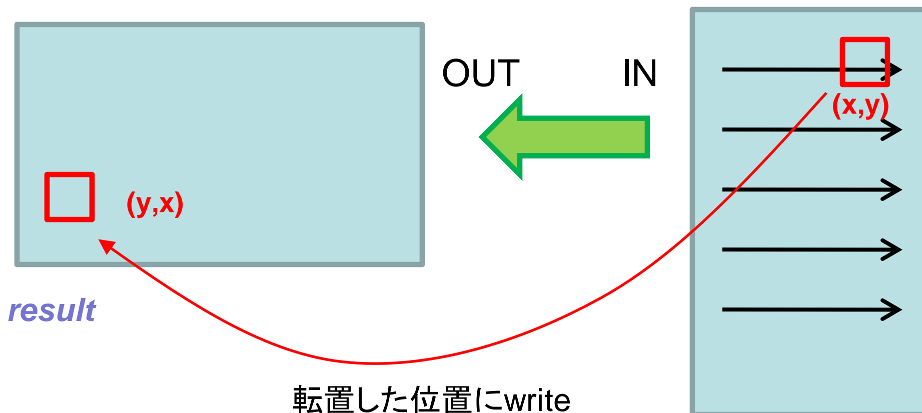
kernel関数call

1. row-dir filter+transpose
2. row-dir filter+transpose

行方向の結果を結果格納領域の転置した位置に保存し,



同じkernelを再度使用して結果を生成



アイデアの実装方法が誤っている.

しかもそのことに実装後に気付いた…

「転置した位置」はy方向に連続
なのだから,
パフォーマンスが改善するハズがない

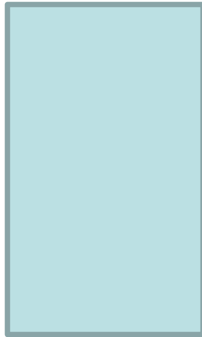
…のだが,
フィルタリング処理の計算時間が
何故か 数%短縮された.
(理由不明)

実装3: 良い転置方法

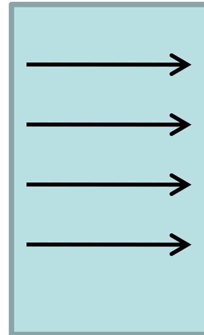
フィルタリング処理と転置処理を別々のkernelに分離し,
{行方向フィルタ→転置→行方向フィルタ→転置}の順で処理.

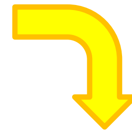
(1)行方向フィルタリング

(2)転置

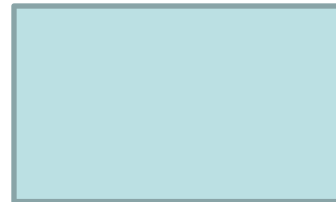


(3)行方向フィルタリング



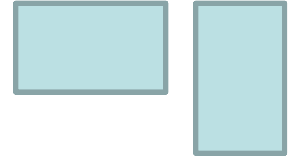


(4)転置



result

使用device memoy



kernel関数call

1. row-dir filter
2. transpose
3. row-dir filter
4. transpose

転置時のblockは16*16

列方向への飛びとびのアクセスを避けた結果,
フィルタリング処理時間が改善.
実装1と比べるとdeviceメモリ割り当てに要する時間はやや増加しているが,
全体の処理時間で見ても改善している.

実装4：8U4C

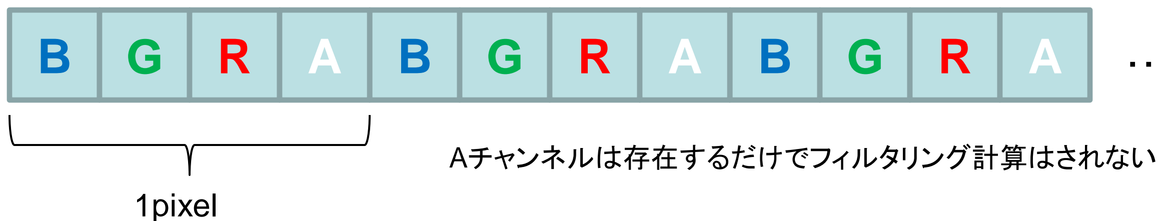
実装3と処理自体は同一.

これまでの実装では画像データは3byte/pixelであり

使用device memoy 4/3 倍 4/3 倍

kernel関数call
1. row-dir filter(4B/pix)
2. transpose(4B/pix)
3. row-dir filter(4B/pix)
4. transpose(4B/pix)

という形であった．この実装4は
画素毎に1byte増やして4byte/pixelとした並びとすることで
globalメモリアクセスをuchar4型（__align__(4)な構造体）で行うものである.



Aチャンネルは存在するだけでフィルタリング計算はされない

※host側の持つ原画の時点で8U4Cであるものとする.

今回，8U4Cの画像データはcvLoadImage→cvCvtColorで用意した.

nvccが出力したPTXコードを見ると，これまで1byte*3回/pixelとなっていたglobalメモリアクセスが，4byte*1回 となったようである.
実装3と比べてフィルタリング処理時間がさらに短縮できている.
ただし，メモリサイズが（単純計算で）4/3倍となったため転送時間は増えている.

※__align__(4)を指定しない struct{ unsigned char B,G,R,A; }型 を用いた場合では，1byte*4回となってしまい，逆にパフォーマンスが悪化する.

時間計測結果

400x300，640x480の2種類の画像サイズでの処理時間を計測．
（時間計測にはcutilのstopwatchを使用）

フィルタカーネルサイズ7(σ =1.4)　　blockDim = (256,1)

実装（下に行くほど新しい版）	処理工程（上から順にやる）	処理時間[ms](5回試行の平均) 400x300[pixel]	640x480[pixel]
cvSmooth(※CPU,OpenCV)	ガウシアンフィルタリング計算	4.485	11.6228
[rowフィルタ→colmnフィルタ] (8U3C)	device memory alloc + 原画データ転送	0.2856	0.5602
	ガウシアンフィルタリング計算	0.5404	1.215
	結果画像データをhostに転送	0.268	0.6016
	device memory de-alloc	0.0776	0.0878
[rowフィルタが転置位置にwrite]×2 (8U3C)	device memory alloc + 原画データ転送	0.3508	0.6344
	ガウシアンフィルタリング計算	0.4958	1.1502
	結果画像データをhostに転送	0.2694	0.6072
	device memory de-alloc	0.1402	0.1498
[rowフィルタ→転置]×2 (8U3C)	device memory alloc + 原画データ転送	0.35	0.626
	ガウシアンフィルタリング計算	0.337	0.6622
	結果画像データをhostに転送	0.268	0.613
	device memory de-alloc	0.142	0.147
[rowフィルタ→転置]×2 (8U4C) (※cvCvtColorの時間は未考慮)	device memory alloc + 原画データ転送	0.4032	0.751
	ガウシアンフィルタリング計算	0.2422	0.4312
	結果画像データをhostに転送	0.3392	0.7744
	device memory de-alloc	0.1428	0.1492

全行程時間合計	
1.1716	2.4646
1.2562	2.5416
1.097	2.0482
1.1274	2.1058

フィルタリング結果画像

画像サイズ:400x300[pixel]
(PowerPointへの挿入時サイズで等倍)

※WebへのUP時の対処:
BMPやPNGで画像貼ると
PowerPointファイルサイズが悪魔的に巨大なので
JPG圧縮したものを貼ってあります

原画

cvSmooth()

CUDA実装

※差分等を取っての確認はしていないが，見た目には同等の結果が得られている．

フィルタリング結果画像

画像サイズ:640x480[pixel]
(入りきらないため
PowerPointへの挿入時サイズから
70%に縮小)

原画

cvSmooth()

CUDA実装

まとめ

Gaussian Filteringの実装を行い，パフォーマンス改善のための検討を行った.

- なるべくcoalescedアクセスとなるようにすることでkernel関数のパフォーマンス向上
- 8U3C→8U4Cで扱えば，globalメモリアクセスが効率化

その他

- 8U3Cの実装でも，shared memoryは8U4Cとすればbank_conflictが無くなると思い試したが，時間計測値に明確な差異が確認できなかった
- フィルタカーネル値配列をconstantメモリからglobalメモリに変えてみても同様に差を確認できなかった

未検討項目が各種ある.

- execution configurationによりどの程度差が出るか？
- kernel関数内での型変換も時間を食うらしいので，最初から32F4C等でdevice memory上にデータ配置すれば高速になるか？
- pinned memoryを使うと転送時間が改善するか？
- 大きいフィルタカーネル１回　vs　小さいフィルタカーネル複数回　等

適応型輝度補正

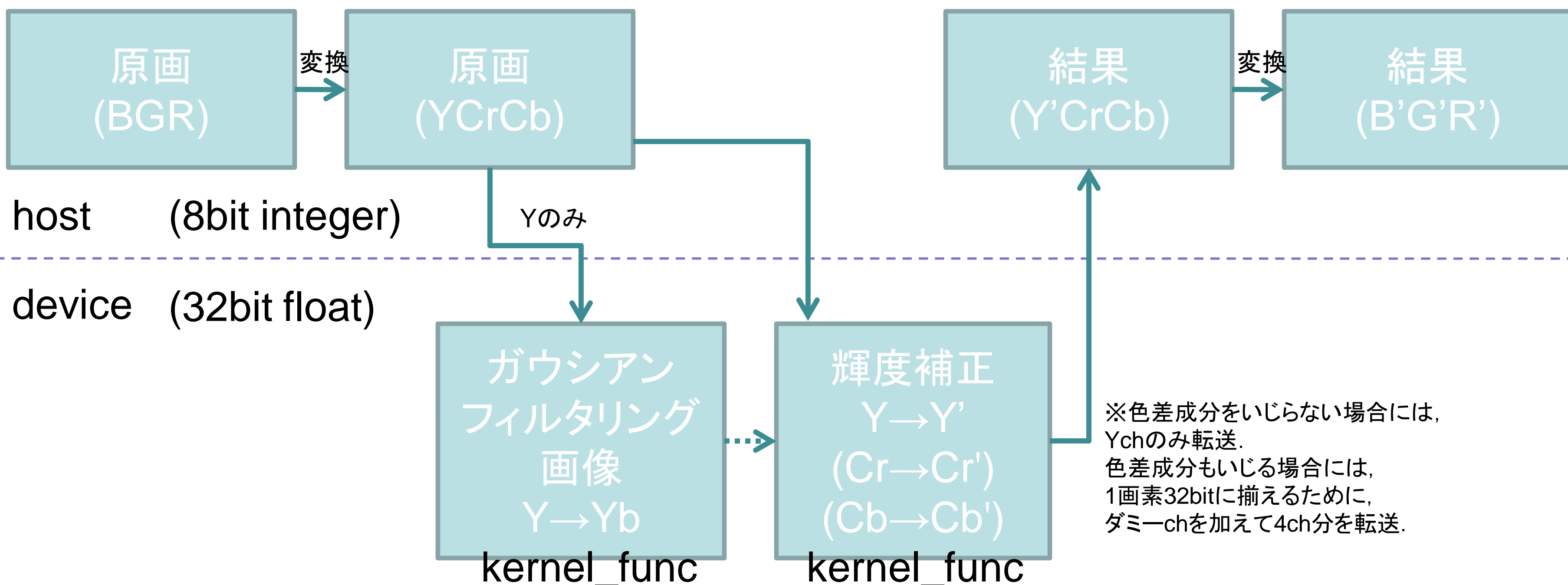
09/12

適応型輝度補正：
画像全体に一様な補正をするのではなくて，
場所毎の明るさ具合に合わせた補正をすることで
暗いところも明るいところも諧調が潰れないという補正処理.

CUDAによる適応型輝度補正処理の実装

GaussianFilteringを使うアプリケーションの１例として，
適応型輝度補正処理の実装を行ってみた.

- 実装するアルゴリズムは以前に作成したアルゴリズム
- 検討時8bit integerだったGaussianFilteringのコードを32bit float化して使用



アルゴリズム

for each pixel (Y = [0,255])
{

$Yb = GaussianFiltered(Y)$ ……ぼかし輝度画像

Y'=（Ybを使って輝度を補正した値）

※アルゴリズムは秘密です

$Y' = (1-\alpha)Y + \alpha Y'$ ……過剰なようなら原画とブレンド

}

必要なら後処理（今は未実装）

ある範囲Ymin～Ymaxを0～255に引き延ばす処理

Ymin,Ymaxの値は
- 補正画像の最小と最大
- ヒストグラム等から決定
- 10,245等の固定値

等，なにかしらの方法で決める.

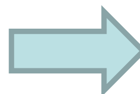
※元々低コントラストな画像等への対応

処理例

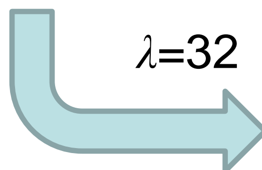
※WebへのUP時の対処：
BMPやPNGで画像貼ると
PowerPointファイルサイズが悪魔的に巨大なので
JPG圧縮したものを貼ってあります

原画

λ=0



λ=32

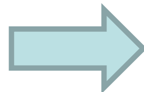


今回の実装では
ガウシアンフィルタを使用している関係で
ハロー効果が目立つ.
（例えばカレンダーの絵の縁等）

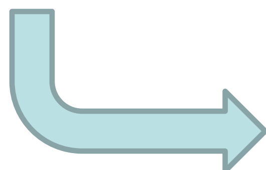
※α=1.0（原画とのブレンドなし）である
※スライドに収まるように適当に縮小して貼っている

色差ch補正の検討

λ=32



Yのみ



輝度の補正量に合わせてCr,Cbも補正してみた例.
上側はYのみ補正結果（前ページと同一）.
下側は色差成分も補正してみた結果.
暗所での灰色感が減った.