社団法人　情報処理学会　研究報告
IPSJ SIG Technical Report

2006－CG－122（14）
2006／2／21

畳み込み型顔検出器におけるGPUの利用

鈴木 裕一　　山口 泰 †
東京大学　教養学部　広域科学科広域システム分科
† 東京大学大学院　総合文化研究科 広域科学専攻　広域システム科学系
E-mail:{yuichi,yama}@graco.c.u-tokyo.ac.jp

本研究では，GPU を計算に利用して，画像の中から高速に顔を検出することを試みる．畳み込みを多用して顔を画像の中から検出する CFF(Convolutional Face Finder) という方法がある．CFF の性能は良いとされているが，畳み込みの計算量が多いので他の競合する顔検出アルゴリズムに比べてやや遅くなる．GPU は畳み込みのような計算を高速に実行できるので，GPU を利用して CFF を計算することで高速化を図ることができる．様々な大きさの画像を入力とする実験を行い，その結果を基に，GPU による高速化について議論する．

GPU Implementaion of Convolutional Face Finder

Yuichi SUZUKI　　Yasushi YAMAGUCHI†
Dept of General Systems Studies, College of Arts and Sciences
The University of Tokyo
† Dept of General Systems Studies, Graduate School of Arts and Sciences
The University of Tokyo
E-mail:{yuichi,yama}@graco.c.u-tokyo.ac.jp

In this research, the authors try to detect faces at high speed by using a GPU. There is a method named CFF(Convolutional Face Finder) that detects faces by computing a lot of convolutions. Although the detection rate of CFF said to be good, CFF is slow compared with other competing face detection algorithms because convolutions consume much computational time. Since a GPU can execute the calculation like convolutions at high speed, we attempt to speed up CFF by using a GPU. In our experiment, images of various sizes are used. This report shows that GPU can compute CFF faster than CPU.

1　はじめに

画像の中から人間の顔を発見することができれば，それを様々なアプリケーションに利用できる．これまでに，検出率がかなり高い方法がいくつか提案されているが，本研究では，誤検出率を低く維持しつつ，高い顔検出率をもつ，畳み込み型顔検出器（CFF(Convolutional Face Finder)) [1] に GPU (Graphic Processing Unit）を用いることで高速化を試みる．

CFF は LeNet-5 [2] というニューラルネットットを参考にして作られている．LeNet-5 は，手書きおよび機械出力の数字を認識する多層フィードフォワードネットワークで，Convolutional Neural Netoworks という構造をしている．これは，ネットワークの入力に画像の輝度そのものを使用し，畳み込みにより特徴量マップを作ることを繰り返し，最終的に入力画像の判別を行うというものである．CFF も LeNet-5 同様に，畳み込みを多用するため，計算量がかなり多くなる．そのため，計算速度という面では，競合する他の顔検出アルゴリズムに比べてやや劣る．

GPU のフラグメントプロセッサは一つの命令で多数のデータを並列に処理できる (SIMD-parallel)．畳み込みは，すべてのピクセルについて，同じ処理を繰り返すので，これをフラグメントプロセッサで行わせると高速に実行できると期待できる．特に本研究では，すべての処理を GPU で行うことにより高速化を図る．

2　CFF

まず CFF の基本的な構成を図 1 に示す．入力層を含めて 7 層のニューラルネットである．入力される画像は 32×36 ピクセルで，この画像が顔か非顔かを判別する．入力画像を 5×5 のマスクで畳み込みした値にニューラルネットのバイアスを加算してシグモイド関数 ($\tanh(0.5x)$) で計算された値が C1 層の特徴量マップ（以下「C1 マップ」と呼ぶ）の値となる．C1 マップの大きさは 28×32 となる．5×5 のマスク

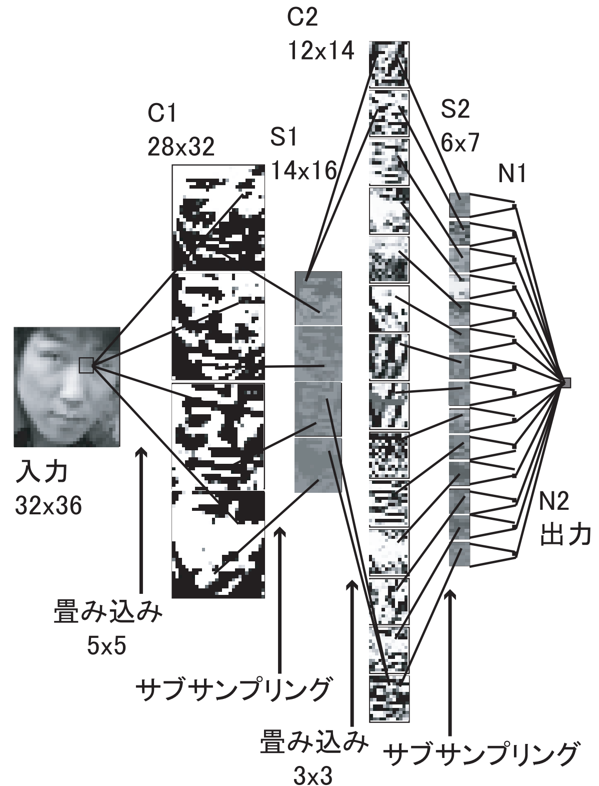

図 1　CFF の構成

は C1 マップ 1 枚につき 1 種類で合計 4 種類ある．故に，入力層と C1 層の間の結合荷重は $(5 \times 5 + 1) \times 4 = 104$ 種類となる．

C1 マップの隣接 4 ピクセルの平均に重みを掛け，バイアスを加算してシグモイド関数を通したものが S1 マップの値となる．マップの大きさは 14×16 となり，結合荷重は $(1 + 1) \times 4 = 8$ 種類となる．

S1 マップを 3×3 のマスクで畳み込みした値にバイアスを加算してシグモイド関数に通したものが C2 マップの値となる．C2 マップの大きさは 12×14 となる．C2 マップは，1 種類の S1 マップをもとに 2 枚と，2 種類の S1 マップの組み合わせをもとに 1 枚ずつ作られるので，合計 $4 \times 2 +_4 C_2 = 14$ 枚．よって結合荷重は $8 \times (3 \times 3 + 1) + 6 \times (3 \times 3 \times 2 + 1) = 194$ 種類となる．

C2 マップの隣接 4 ピクセルの値の平均に重み

を掛けて，バイアスを加算してシグモイド関数を通したものがS2マップの値となる．マップの大きさは6×7となり，結合荷重は$(1+1) \times 14 = 28$種類となる．

S2マップのすべての値が，14個のN1層のニューロンと直接接続しており，バイアスを加算してシグモイド関数に適用する．よって結合荷重は$(6 \times 7 + 1) \times 14 = 602$種類となる．

N1層の14個のニューロンはすべて，N2層の1個のニューロンに接続しており，これもバイアスを加算してシグモイド関数に適用する．故に，結合荷重は$14 + 1 = 15$種類となる．

このネットワークは，顔なら1を，非顔なら-1を出力するように訓練される．学習はバックプロパゲーションで行う．効率よく学習を行うために，非顔データは，反復ブートストラップにより集める．反復ブートストラップとは，学習の途中でネットワークに非顔を入力したとき，顔であると判別されてしまったデータのみ，非顔データとして学習に用いるという方法である．こうすることで，より少ない非顔データですむ．

認識時に，32×36の窓を入力画像全体に走査し，各窓で独立に処理すると，計算量が非常に多くなる．このニューラルネットを構成する畳み込みやサブサンプリングは，32×36の枠によらず計算可能なので，入力画像全体をこのニューラルネットに通すということも可能である．その結果，顔のある領域の値が1に近く，そうでない領域は-1に近い2次元配列が出力される（図2）．

この入力画像だけをニューラルネットでは，画像中の32×36に近い大きさの顔しか見つけられない．そこで，入力画像を縮小した画像を何枚か入力することで，大きな顔を見つけることができる．このようにすると，何枚かの結果画像が出力される．もし本当に顔が存在している場合は，複数のポジティブな出力がまとまって出てくる場合が多いので，ポジティブな出力

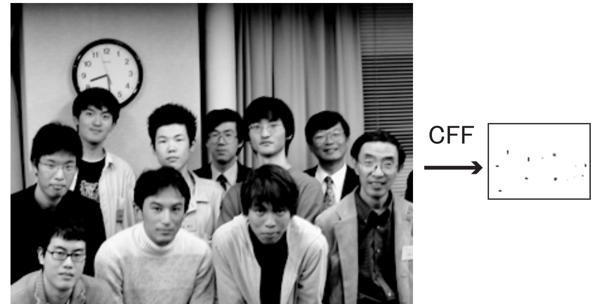

図2　CFFの入出力画像

が少ない部分は顔でないとみなして排除する．

本研究ではこのCFFの顔認識過程にGPUを利用し，顔検出の高速化を試みる．

3　GPUでの計算方法

GPUはCPUのようにアドレスを指定してメモリにアクセスできるわけではないので，CPUと同じようにプログラムを作るわけにはいかない．CPUの扱うメインメモリ上のデータは，GPUではビデオメモリ上のデータということになり，メモリデータはテクスチャとして扱われる．

テクスチャはグリッド上のデータ値を表しているといえるが，フラグメントプロセッサはgather（周りのフラグメントから情報を集めること）はできるが，scatter（周りのフラグメントに情報を渡すこと）はできない．よって，scatterを必要とするアルゴリズムは，gatherに変換しないと実現できない．

3.1　プログラム全体の構成

まずはプログラム全体の手順を説明する．

i. 入力画像の読み込み：画像幅と高さは両方とも4の倍数である必要がある．
ii. 初期化：マスクデータ（ニューラルネットの中で使用する結合荷重）の読み込み．入力画像の幅と高さから，縮小画像の幅と高さを計算．
iii. ビデオメモリにデータをロード：入力画像とマスクデータをテクスチャとしてビ

デオメモリに格納．

iv. 入力画像をバイリニア変換で縮小：幅と高さを概ね $\frac{1}{1.2}$ 倍にして 4 の倍数となるものを縮小画像とする．本研究では 10 段階の画像を用いた．

v. ニューラルネットに入力し結果画像を得る：畳み込みおよびサブサンプリングの繰り返し．

vi. 結果画像の点を対応する顔領域に変換：ポジティブな点（0.8 以上）の右 6 ピクセル，上 7 ピクセルの矩形領域に 1 を加算．

vii. 結果画像を入力画像の大きさにバイリニア変換で拡大：vi で塗りつぶされた矩形領域が，実際の顔の位置と大きさにあうように変換する．

viii. 上記結果画像を最終結果画像に加算：様々な大きさの入力画像の結果をすべて足しあわせる．

ix. false positive の排除：最終結果画像のなかから，ある閾値以下の点を排除する．

この中で GPU で計算するのは, iv, v, vi, vii,viii である．ix を行うまでは，比較的遅いとされるビデオメモリからのデータの読み出しを行わない．すなわち，時間のかかるビデオメモリからメインメモリへのデータ移動の回数は最小限になっている．

これらのプロセスで使われている関数を分類すると，畳み込み，サブサンプリング，バイリニア変換，矩形の塗りつぶし，画像の加算に分けられる．画像の加算は単なるテクスチャの加算なので，それ以外の関数について説明する．

3.2 畳み込み

畳み込みとは，あるマスクを入力画像に重ねて，対応するマスクと画像のピクセルの値の積の総和を出力画像のピクセルの値にする処理である．CFF の場合，総和にニューラルネットのバイアスを加えたものをシグモイド関数 ($sigmoid(x) = \tanh(0.5x)$) に通す作業が追加される．図 3 を見るとわかるように，現在出力の対象となっているピクセル（黒塗りされているピクセル）の値を計算するには，入力画像およびマスクがテクスチャになっていればよい．

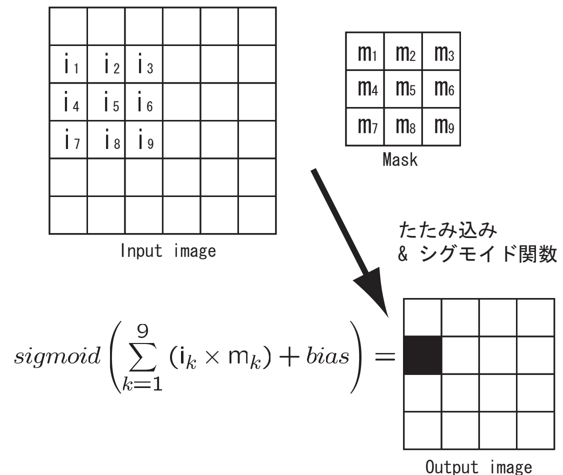

図 3　畳み込み＆シグモイド関数

たとえば，マスクが 3×3 の場合，必要な座標値は 9 個であるが，入力画像のある基準座標が指定されたとき，その座標値をプラスマイナスすることで，9 個の座標値が表せる．そこで，ある基準座標値から他の 8 ピクセルへの差分を dif1..., dif8 として相対座標値をすべて表す．たとえば基準座標値を左下のピクセルとすると，

```
float2 dif1 = float2(1.0,0);
float2 dif2 = float2(2.0,0);
..................
float2 dif8 = float2(2.0,2.0);
```

のような定数をシェーダの中で定義してやる．基準座標値に対して，相対座標値を適用することでテクスチャサンプリングを行い，畳み込みの値を計算する．

3.3 サブサンプリング

CFF で用いられるサブサンプリング処理は，画像の幅と高さをそれぞれ半分にする縮小変換である．この処理も畳み込みと同様の方法で実行可能であるが，畳み込みよりも単純である．

なぜなら，4 点の平均の値に重みを掛けてバイアスを加えたものをシグモイド関数に通せばよいだけだからである．

3.4 バイリニア変換

HLSL を用いると，C 言語とほとんど同じようにバイリニア変換のアルゴリズム [3] を記述できる．

3.5 矩形の塗りつぶし

この処理では対象としているピクセルの値が 0.8 以上ならば，周りのピクセルに値を書き込むことになるが，これは scatter 処理であり，GPU では行えない．よって，この scatter 処理を gather 処理に変換する必要がある．変換方法は簡単で，対象となるピクセルの座標から，左に 6 ピクセル，下に 7 ピクセルの範囲をすべて検索して，その中に何個 0.8 以上の値を持つピクセルがあるかどうかを調べ，その個数を対象となるピクセルの値とすればよい．

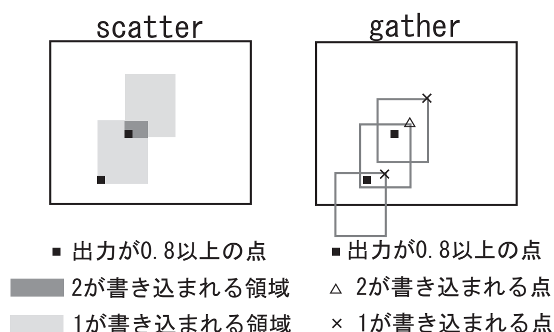

図 4 scatter から gather への変換

アルゴリズムの違いを図示すると図 4 の様になる．一つの点あたりの計算量は gather の方がかなり多くなるが，総合的にみると GPU で処理する方が速く計算できると期待できる．

4 結果

まずは実験環境を以下に示す．

- CPU:Intel Pentium4 640(3.2GHz)
- GPU:NVIDIA Geforce7800GTX
- OS:Microsoft WindowsXP SP2
- Compiler:Intel C++ Compiler 9.0

実験に用いた画像は 6 枚で，その解像度の一覧は以下の通りである，

表 1 画像解像度一覧

	解像度（ピクセル数）
入力画像 1	1920 × 1280 (2457600)
入力画像 2	1440 × 960 (1382400)
入力画像 3	1080 × 720 (777600)
入力画像 4	780 × 520 (405600)
入力画像 5	640 × 480 (307200)
入力画像 6	320 × 240 (76800)

入力画像 1 をもとに計算した結果を図 5 に示す．

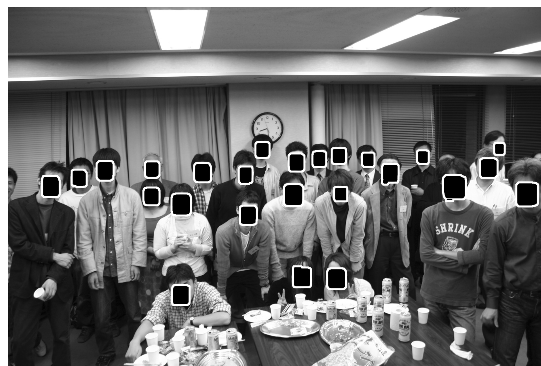

図 5 入力画像 1 と GPU の結果画像の合成画像

4.1 全体の速度の比較

プログラム全体の計算時間を，大きさの異なる画像を入力して比較する．同じ画像でも処理時間にばらつきがあったので, 計算時間は, 2000 回実行した平均をとった．処理時間の実測値を表 2 に，それをグラフにプロットしたものを図 6 に示す．

表 2 および図 6 より，GPU によって高速化されていることが確認できる．ピクセル数が少ないときに，処理時間の比（CPU/GPU）が比較的小さいのは，ピクセル数に関係なく一定の時間を消費する処理があることを示唆する．

図 7 に，テクスチャ管理（テクスチャ生成，テ

表 2　プログラム全体の速度の比較

ピクセル数	GPU(秒)	CPU(秒)	CPU/GPU
76800	0.02490	0.1589	6.379
307200	0.04816	0.6241	12.96
405600	0.06072	0.8344	13.74
777600	0.1135	1.596	14.06
1382400	0.1968	2.893	14.70
2457600	0.3559	5.158	14.50

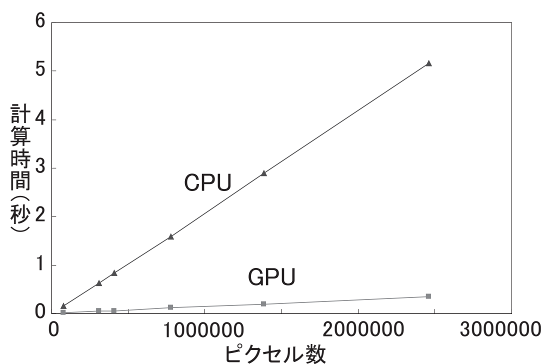

図 6　プログラム全体の速度の比較

クスチャ削除など）にかかる時間と，それ以外の処理にかかる時間をプロットした．これを見ると，ピクセル数が 0 の時，テクスチャ管理にかかる時間は 0 になりそうにない．よって，テクスチャ管理にはピクセル数に関係ない処理があるといえる．全てのテクスチャデータをメモリに載せることができれば，2 回目以降の計算ではこの処理は不要だが，この実験で使用した入力画像の中には，256MB よりも大きな作業領域を必要するものがあるので，不要になったテクスチャを適時削除する処理を行った．小さな画像なら，最初にテクスチャを全て生成して，それを削除せずに使い回すことで，さらに高速化できる．

5　おわりに

本研究は，顔画像検出法である CFF を，GPU を用いて高速に計算することを目的とした．GPU で計算させるために，DirectX+HLSL でプログラムを書き，CPU で計算するよりも高速（14 倍程度）に計算できることを確認した．

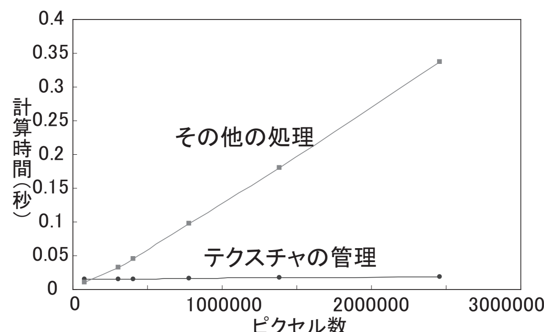

図 7　テクスチャ管理にかかる時間

CPU と GPU のやりとりを最小化したが，これも有効に働いたと考えられる．また，テクスチャの管理に一定のオーバーヘッドがあることがわかった．

本研究では，ニューラルネットの学習は CPU で計算したが，学習を GPU で行わせることも可能と考えている．

参考文献

[1] C. Garcia and M. Delakis. Convolutional face finder: A neural architecture for fast and robust face detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. 26, No. 11, November 2004.

[2] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, November 1998.

[3] ディジタル画像処理編集委員会. ディジタル画像処理. CG-ARTS 協会, 2004.