Bulletin of Networking, Computing, Systems, and Software – www.bncss.org, ISSN 2186–5140
Volume 2, Number 1, pages 42–46, January 2013

An Implementation of Hough Transform on the GPU

Tomagou Norihiro, Koji Nakano, Yasuaki Ito
Department of Information Engineering
Hiroshima University
Kagamiyama 1-4-1, Higashi-Hiroshima, 739-8527, JAPAN

***Abstract*—Recent Graphics Processing Units (GPUs) have many processing cores. To use the powerful computing ability, GPUs are wildly utilized for general purpose processing. The main contribution of this paper is to present a new implementation of Hough transform using the GPU. The Hough transform identifies straight lines in a binary edge image. In our GPU implementation, the voting process in the Hough transform is performed for each degree in parallel. Also, the shared memory is used with bank conflict-free access. We have implemented our parallel algorithm with NVIDIA GeForce GTX680. The experimental results show that the Hough transform for a 512×512 image with 33232 edge points can be done in only 0.638ms, while a conventional CPU implementation runs in 43.388ms. Thus, our GPU implementation attains a speed-up factor of 68.**

***Keywords*-Image processing, Line detection, Hough transform, GPGPU, CUDA**

I. はじめに

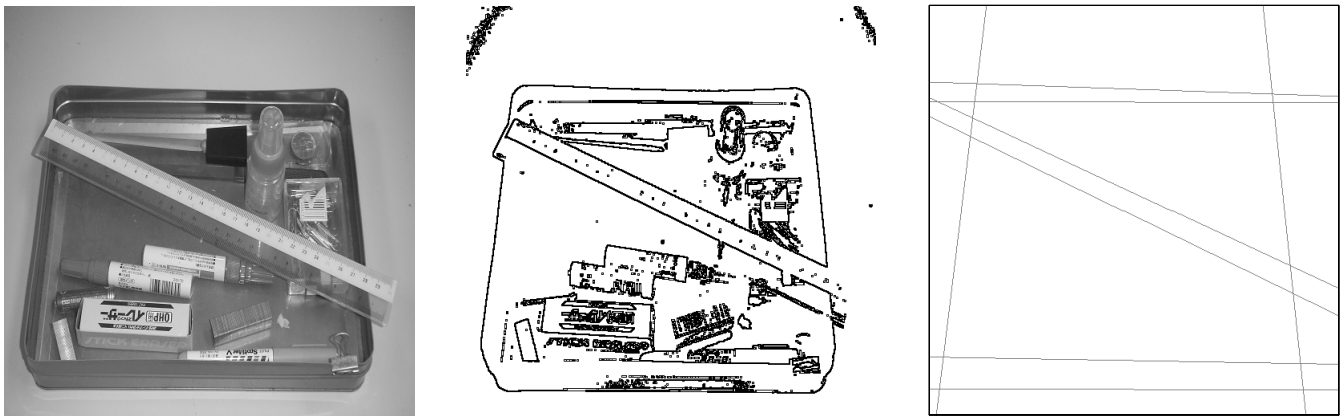
複数のコアで同時に 1 つのタスクを行う並列計算は，処理のスループット向上のための有効な手段である．プロセッサに搭載されるコアの数は年々増加しており，これに伴い並列計算の効果も増してきている．その中でも，グラフィックス処理用の LSI である GPU(Graphics Processing Unit) を汎用計算に使用した GPGPU(General-Purpose computation on GPUs) が注目されており，様々な研究開発が行われている [1], [2], [3], [4], [5], [6], [7], [8], [9], [10]．Nvidia 社が提供している CUDA [11] は，同社の GPU 向けの統合開発環境であり，C 言語で記述することで並列処理を実現できる．GPU には CUDA コアが数百から数千個程度搭載されており，これらに実行単位であるスレッドを割り当てることで並列計算を行う．スレッドの集まりであるブロックは，ストリーミングマルチプロセッサと呼ばれるコアの集合に割り当てられて実行される．それぞれのストリーミングマルチプロセッサは，アクセスが非常に高速なシェアードメモリをそれぞれ持っており，同一ブロック内のスレッドはシェアードメモリを共有している．異なるブロックに属するスレッドはお互いのシェアードメモリにアクセスすることができないため，効率的にシェアードメモリを利用するにはブロックの分割を良く考慮する必要がある．また，シェアードメモリは 32 個のバンクによって構成されており，あるバンクに対して同時に複数のメモリアクセスが発生した場合，1 つずつ順番に処理されるため時間がかかってしまう．これをバンクコンフリクトと呼ぶ．バンクコンフリクトはメモリアクセスを工夫することによって回避することが可能な場合がある．

ハフ変換は，デジタル画像から特徴量 (主に直線) を検出するために用いられる一般的な手法である [12]．直線検出の為のハフ変換は，エッジ抽出を行った画像に対して，任意のエッジ点が多く通る直線を決定するために，その点が通る直線のパラメータ (ρ, θ) に対して，離散化された対応空間に投票を行う．この投票処理の為の変換には $\rho = x\cos\theta + y\sin\theta$ を用いる．この式の θ を $0°$ から $180°$ の範囲で少しずつ増加させながらそれぞれ ρ の値を計算し，投票空間の (ρ, θ) に対応する値をインクリメントすることによって，複数回投票がある点の値は大きくなってゆく．投票数が閾値以上であれば，その時の (ρ, θ) が直線であると判断できる．例としてこれらの処理の過程を図 1 に示す．図 1(a) はカメラで撮影した元画像であり，これにソーベルフィルタ処理を適用し，二値化を行うことで図 1(b) のエッジ画像が得られる．このエッジ画像に対してハフ変換を行い，投票空間で閾値以上の投票数を持つ点を逆変換することで，図 1(c) に示す直線が得られる．この時得られる投票空間を次の図 2 に示す．暗い点ほど投票が多い点で，いくつかの点に投票が集中しており，これらの点が直線をあらわす．投票が多い点を逆変換することで直線を検出する．

GPU を用いたハフ変換に関する関連研究として，Fung らは，OpenGL を用いて直線のハフ変換を実装した [13]．この実装は，CUDA といった GPGPU 環境が利用可能になる以前に行われ，プログラム可能なピクセルシェーダを利用し，グラフィック処理としてハフ変換を実行した．Braak らは，GPU を用いた画像に依存する実装と依存しない実装を提案し，そのトレードオフを考察した [14].

本研究では GPU を利用した高速なハフ変換の手法を提案する．本手法の主なアイデアは，角度ごとに並列に実行することと，シェアードメモリを利用することで，メモリアクセスの効率化を図ることである．さらに，シェアードメモリのバンクコンフリクトが発生しないように改善し，さらなるパフォーマンスの向上を達成した．

本論文の後の構成は次のとおりである．II 節では，直線検出の為のハフ変換の手法について説明し，III 節では，提案手法である，GPU を利用したハフ変換の実装方法について述べる．IV 節で実験結果を示し，最後の V 節ではまとめと今後の課題について述べる．



(a) 入力画像　(b) ソーベルフィルタによる二値エッジ画像　(c) ハフ変換による直線検出結果

Figure 1.　ハフ変換を用いた直線検出の例

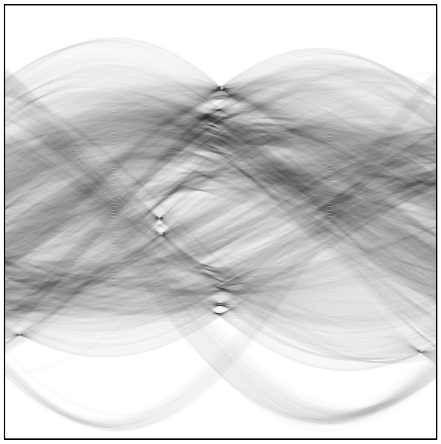


Figure 2.　パラメータ空間 ($\theta\rho$ 空間)

II. Hough Transform

本節では，直線検出のハフ変換のアルゴリズムについて説明する．以降，入力データとなる 2 値エッジ画像のサイズは $n \times n$ を想定する．図 3 のように，入力されたエッジ画像の中心が原点となるように xy 空間を設定すると，エッジ点 (x, y) の範囲は，x, y ともに区間 $\left[-\frac{n}{2}+1, \frac{n}{2}\right]$ となる．xy 空間のエッジ点 (x, y) は，次式で与えられるハフ変換によって $\theta\rho$ 空間へと変換できる．

$$\rho = x\cos\theta + y\sin\theta \quad (0 \leq \theta < 180) \tag{1}$$

このとき，ρ は不等式 $-\frac{n}{\sqrt{2}} < \rho \leq \frac{n}{\sqrt{2}}$ を満たす．また，θ と ρ の値は，幾何的に求めることができる．図 3 のように，原点から角度 θ の直線を描くことを考える．値 ρ は，原点からその直線への距離をあらわしている．つまり，$\theta\rho$ 空間の点 (θ, ρ) は，xy 空間の直線に対応する．ハフ変換のアイデアは，xy 空間におけるすべての点に対して，$\theta\rho$ 空間に投票することである．点 $(x_0, y_0), (x_1, y_1), \ldots, (x_{k-1}, y_{k-1})$ を xy 空間に k 個のエッジ点とするとハフ変換は次のようになる．

[ハフ変換アルゴリズム]

for $i \leftarrow 0$ to $k-1$
　for $\theta \leftarrow 0$ to 179
　　begin
　　$\rho \leftarrow x_k \cos\theta + y_k \sin\theta$
　　$v[\theta][\rho] \leftarrow v[\theta][\rho] + 1$
　　end

このアルゴリズムでは，それぞれの θ ごとに投票処理を行う．したがって，$\sin\theta$ と $\cos\theta$ の値は初めに 1 回だけ計算すればよい．また，それぞれの θ で並列に ρ の値を計算し投票することができるため，GPU を利用することで高速に処理することが可能である．

III. ハフ変換の GPU 実装

本節では GPU を用いてハフ変換を行う手法として，以下の 3 つの手法について説明する．

(A)　グローバルメモリのみを用いた手法
(B)　シェアードメモリを利用した手法
(C)　バンクコンフリクトが発生しないように (B) を改良した手法

(A) グローバルメモリを用いた手法

まず，シェアードメモリを利用せず，グローバルメモリのみを使ってハフ変換を行う手法を説明する．

処理の流れは以下のようになる．

1) ホストからデバイスのグローバルメモリにエッジリストを転送
2) 並列にハフ変換を行い，グローバルメモリに投票
3) グローバルメモリからメインメモリにハフ空間を転送

ブロックの割り当ては，1 ブロックにつき 1 度とする．投票処理では，θ の要素数を 180，ρ の要素数を 192 の 180×192

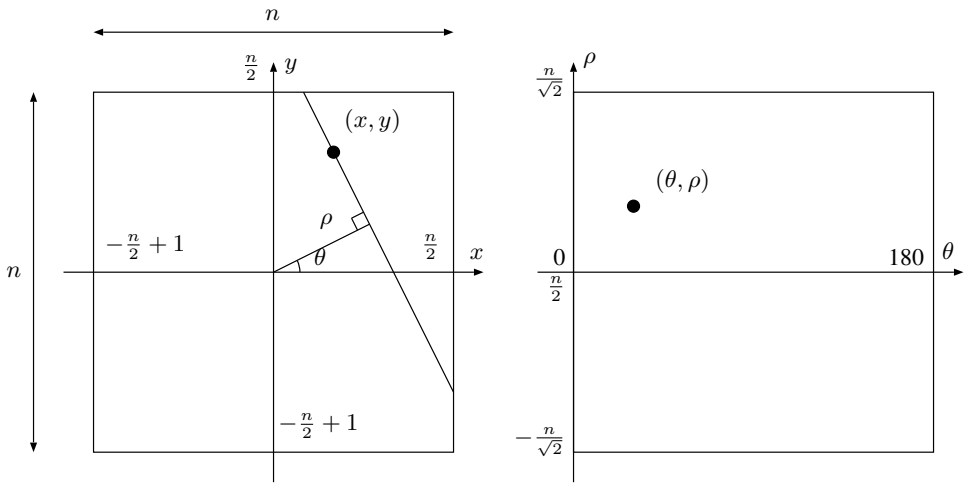


Figure 3. ハフ変換で用いる 2 次元 xy 空間と $\theta\rho$ 空間

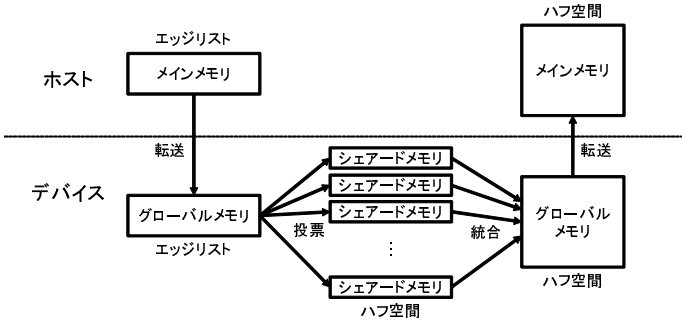


Figure 4. 処理の流れ

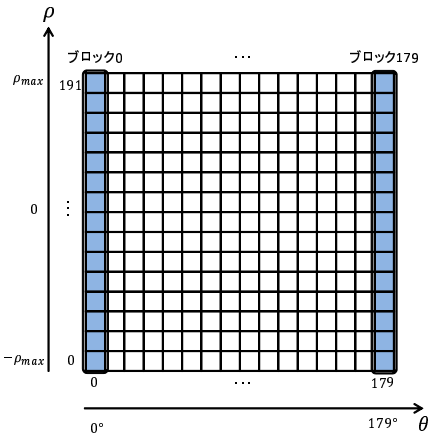


Figure 5. ブロックの割り当て

として投票空間のサイズを固定し，グローバルメモリ上に int 型の配列として確保する．rho の要素数を 192 に固定するため，変換式 $\rho = x\cos\theta + y\sin\theta$ によって得られた ρ の値を 0 から 191 の値に正規化する必要がある．画像サイズを $n \times n$ としたときの ρ の最大値は $\rho_{max} = \frac{n}{\sqrt{2}}$，最小値は $-\rho_{max}$ であるので，変換後の値を ρ' をすると，変換式は次のようになる．

$$\rho' = 96\left(1 + \frac{\rho}{\rho_{max}}\right) \tag{2}$$

この式で得られた ρ' を整数化し，対応する要素に対して投票処理を行う．各ブロックでの投票処理は，エッジ点ごとにスレッドを割り当て，1024 スレッド毎に並列に行う．複数のスレッドが同時に同じ要素に対して加算を行う可能性があるので，データの整合性を保証するために排他的に加算をする atomicadd() 関数を利用する必要がある．投票処理が終了したら，得られた投票空間をホストのメインメモリに転送する．

(B) シェアードメモリを利用した手法

グローバルメモリと比較してアクセスが高速なシェアードメモリを用いた実装方法について説明する．

この方法では前述の実装方法と同様に，投票空間の θ1 度に対して 1 ブロックを割り当てるが，それぞれのブロックは 192 個の要素を持つ int 型配列をシェアードメモリに確保し，そこに投票処理を行う．この時，(A) の手法と同様に (2) 式を用いて正規化を行い，atomicadd() 関数を用いる．投票が終了したら，各ブロックのシェアードメモリの内容をグローバルメモリへと統合する．

処理の流れをまとめると図 4 のようになる．また，図 5 にブロックごとのシェアードメモリの割り当て図を示す．

この手法では，複数のスレッドがシェアードメモリの同一のバンクに同時にアクセスする可能性がある．したがって，バンクコンフリクトが発生し，処理速度が低下してしまう．これを改良した，バンクコンフリクトが発生し

ない高速な実装手法を次に提案する．

(C) バンクコンフリクトを回避した手法

GPU での各ブロックでの処理は実際にはワープ単位 (32 スレッド毎) で行われる．したがって，投票先のシェアードメモリを ρ の要素数の 32 倍だけ確保し，同時に実行されるスレッドの投票先をそれぞれのバンクにすることで，バンクコンフリクトは回避できる．エッジ点は 32 個ごとにハフ変換され，互いに異なるバンクへと投票される．

投票が終了したら，それぞれのバンクの投票値の合計処理を 32 スレッドずつ並列に行う．ただし，合計処理の開始位置が各スレッドで同じだとバンクコンフリクトが発生してしまう為，図 6 に示すように開始位置を一つずつずらしながら，各スレッドが横方向に加算処理を行う．加算処理の途中でバンク 31 からバンク 0 へと循環するようにメモリアクセスを行う．スレッド id を idx とすると，加算用カウンタ $i = 0, 1, ...31$ に対してバンク番号 b は次のようにビット論理積による下位 32 ビットを抜き出すマスクをかけることで得ることができる．

$$b = (idx + i)\&0\text{x}1\text{F} \tag{3}$$

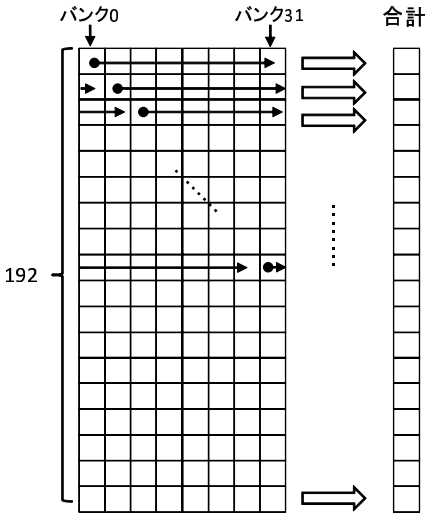


Figure 6. バンクコンフリクトの発生しない加算

シェアードメモリ確保後の初期化処理も，これと同様の方法で行うことでバンクコンフリクトを回避することができる．

最終的に得られた合計値を，ブロック全体で統合する．シェアードメモリの初期化・投票・統合のいずれの処理でもバンクコンフリクトは発生しないため，高速にハフ変換を行うことが可能である．

IV. 性能評価

本研究では GPU として NVIDIA 社の GeForce GTX 680 を利用した．GTX 680 のコア数は 1536，コアの動作周波数は 1006MHz，グローバルメモリの容量は 4GB である．CUDA のプログラムのコンパイルは nvcc コンパイラのバージョン 4.2 を使用し，最適化オプションとして-O2 を利用している．実行する際のブロック数は 180, スレッド数は 1024 とした．逐次処理と比較するため CPU ハフ変換を実装し，Intel 社の Xeon X7460 プロセッサ (動作周波数 2.66GHz) を単一コア/単一スレッドで実行した．また，ハフ空間のサイズは 180×192 である．

前章で説明した，GPU を用いてグローバルメモリのみを利用してハフ変換を行う手法を手法 (A)，シェアードメモリを利用した手法を手法 (B)，手法 (B) をバンクコンフリクトが発生しないように改善した手法を手法 (C) として，それぞれの手法での処理時間を CPU のものと比較した．

まず，入力画像としてカメラで撮影した写真 (図 1(a)) に対してソーベルフィルタをかけ，閾値処理で二値化を行って得られるエッジ画像 (図 1(b)) を使用した．画像サイズは 512×512 であり，画像に含まれるエッジ点の個数は 33232 個である．この画像に対して，CPU 及び GPU でハフ変換処理を行った時の処理時間を表 I に示す．尚，実行時間はハフ変換にかかった時間のみを計測しており，ホストとデバイス間でのメモリの転送時間は含まれていない．この結果から，CPU の処理時間と比較して，グローバルメモリのみを利用した手法では 58 倍，バンクコンフリクト未回避の手法では 29 倍，バンクコンフリクトを回避した手法では 68 倍の高速化を達成した．また，手法 (B) よりも手法 (A) が高速なのは GPU の内部キャッシュを効率的に利用している為だと考えられるが，手法 (C) はそれよりも高速に処理を完了することができた．

Table I
図 1 のエッジ画像 (エッジ点数:33232) に対するハフ変換の実行時間

	実行時間 [ms]	高速化率
CPU	43.388	—
手法 (A)	0.751	58
手法 (B)	1.519	29
手法 (C)	0.638	68

さらに，ランダムに生成した 1000 から 10000000 のエッジ点を含む 4096×4096 ピクセルの画像に対して同様に処理を行い，エッジ点数の変化による処理時間の違いを調査した．表 II に，実行時間と CPU の実行時間に対する高速化率を示す．画像サイズは固定なので，エッジ点数が多くなると画像中のエッジ密度は増加する．隣接エッジは同一のアドレスに投票されバンクコンフリクトが発生する確率が高いため，エッジ密度が増加するにつれて手法 B は高速化率が低下している．一方で，バンクコンフリクトを回避した手法 C は高速化率の低下が無く，バンクコンフリクト未回避の手法 (B) と比較して明らかに高速に処理を完了できている．また，特にエッジ点が多い場合について，手法 (A) と比較して手法 (C) は高速となっており，最大では CPU 比で 90 倍程度の高速化を達成することができた．

V. まとめ

本論文では，GPU を利用した並列処理環境における高速なハフ変換の手法を提案した．実験結果から，GPU(GeForce GTX 680) で共有メモリのバンクコンフリク

Table II
エッジ点数を変化させたときのハフ変換の実行時間と CPU に対する高速化率

		手法 (A)		手法 (B)		手法 (C)	
エッジ点数	CPU	時間 [ms]	高速化率	時間 [ms]	高速化率	時間 [ms]	高速化率
1000	1.408	0.166	9	0.186	8	0.181	8
10000	13.619	0.321	42	0.407	34	0.298	46
100000	136.907	2.030	68	2.948	46	1.817	75
1000000	1377.754	18.005	77	35.095	39	16.331	84
10000000	14073.099	237.730	59	723.411	20	162.232	87

トを回避実装により，エッジ点数 33232 点を含む 512×512 ピクセルのエッジ画像のハフ変換に要した時間は 0.638ms であり，CPU(Xeon X7460) の単一スレッドでの実行時間の 43.388ms と比較して約 68 倍の高速化を達成することができた．また，エッジ点数が多い場合は最大で約 90 倍の高速化が可能であることを示した．

REFERENCES

[1] R. Farivar, D. Rebolledo, E. Chan, and R. H. Campbell, "A parallel implementation of k-means clustering on GPUs," in *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, July 2008, pp. 340–345.

[2] P. Harish and P. J. Narayanan, "Accelerating large graph algorithms on the GPU using CUDA," in *Proceedings of the 14th International Conference on High Performance Computing*, 2007, pp. 197–208.

[3] Y. Ito, K. Ogawa, and K. Nakano, "Fast ellipse detection algorithm using Hough transform on the GPU," in *Proceedings of International Workshop on Challenges on Massively Parallel Processors*, 2011, pp. 313–319.

[4] D. Man, K. Uda, H. Ueyama, Y. Ito, and K. Nakano, "Implementations of a parallel algorithm for computing Euclidean distance map in multicore processors and GPUs," *International Journal of Networking and Computing*, vol. 1, no. 2, pp. 260–276, 2011.

[5] K. Nishida, Y. Ito, and K. Nakano, "Accelerating the dynamic programming for the matrix chain product on the GPU," in *Proceedings of International Workshop on Challenges on Massively Parallel Processors*, 2011, pp. 320–326.

[6] K. Ogawa, Y. Ito, and K. Nakano, "Efficient Canny edge detection using a GPU," in *Proceedings of International Workshop on Advances in Networking and Computing*, 2010, pp. 279–280.

[7] A. Uchida, Y. Ito, and K. Nakano, "Fast and accurate template matching using pixel rearrangement on the GPU," in *Proceedings of International Conference on Networking and Computing*, 2011, pp. 153–159.

[8] S. Wang, S. Cheng, and Q. Wu, "A parallel decoding algorithm of LDPC codes using CUDA," in *Proceedings of Asilomar Conference on Signals, Systems, and Computers*, October 2008, pp. 171–175.

[9] Z. Wei and J. JaJa, "Optimization of linked list prefix computations on multithreaded GPUs using CUDA," in *Proceedings of International Parallel and Distributed Processing Symposium*, 2010.

[10] K. Nishida, Y. Ito, and K. Nakano, "Accelerating dynamic programming for the optimal polygon triangulation on the GPU," in *International Conference on Algorithms and Architectures for Parallel Processing*, 2012, pp. 1–15.

[11] NVIDIA Corp., "CUDA ZONE," http://developer.nvidia.com/category/zone/cuda-zone.

[12] R. O. Duda and P. E. Hart, "Use of the Hough transformation to detect lines and curves in pictures," *Communications of the ACM*, vol. 15, no. 1, pp. 11–15, 1972.

[13] J. Fung, S. Mann, and C. Aimone, "Openvidia: Parallel gpu computer vison," in *Proc. of the ACM Multimedia 2005*, November 2005, pp. 849–852.

[14] G.-J. van den Braak, C. Nugteren, B. Mesman, and H. Corporaal, "Fast hough transform on gpus: Exploration of algorithm trade-offs," in *Proc. of Advanced Concepts for Inteligent Vison Systems*, 2011, pp. 611–622.