12-2010

A Self Organization-Based Optical Flow Estimator with GPU Implementation

Manish Shiralkar
Clemson University, manishshiralkar@yahoo.com

A Self Organization-Based Optical Flow Estimator with GPU Implementation

---

A Dissertation
Presented to
the Graduate School of
Clemson University

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy
Computer Engineering

---

by
Manish Shiralkar
December 2010

Accepted by:
Dr. Robert Schalkoff, Committee Chair
Dr. Walter Ligon
Dr. Stanley Birchfield
Dr. Robert Geist

Abstract

This work describes a parallelizable optical flow estimator that uses a modified batch version of the Self Organizing Map (SOM). This gradient-based estimator handles the ill-posedness in motion estimation via a novel combination of regression and a self organization strategy. The aperture problem is explicitly modeled using an algebraic framework that partitions motion estimates obtained from regression into two sets, one (set H_c) with estimates with high confidence and another (set H_p) with low confidence estimates. The self organization step uses a uniquely designed pair of training set ($Q = H_c$) and the initial weights set ($W = H_c \cup H_p$). It is shown that with this specific choice of training and initial weights sets, the interpolation of flow vectors is achieved primarily due to the regularization property of SOM. Moreover, the computationally involved step of finding the winner unit in SOM simplifies to indexing into a 2D array making the algorithm parallelizable and highly scalable. To preserve flow discontinuities at occlusion boundaries, we have designed anisotropic neighborhood function for SOM that uses a novel OFCE residual-based distance measure. A multi-resolution or pyramidal approach is used to estimate large motion. As the algorithm is scalable, with sufficient number of computing cores (for example on a GPU), the implementation of the estimator can be made real-time. With the available true motion from Middlebury database, error metrics are computed.

Dedication

I dedicate this work to my parents.

Acknowledgments

First I would like to thank my advisor Dr. Robert Schalkoff for guiding me throughout my time in graduate school. Also, I would like to thank Dr. Robert Geist for providing access to NVIDIA CUDA GPUs. I am sure that the learning that happened during Computer Vision course conducted by Dr. Stanley Birchfield will always be useful throughout my career and it is highly appreciated. I would also like to thank all the instructors from whom I have learned during my stay at Clemson.

Table of Contents

List of Figures

Chapter 1

Introduction

Optical flow computation is an important component of the early vision problem. The computation results in a 2 dimensional (2D) vector field over the image pixel locations $\underline{x}$ at time t. The vectors $\underline{v}(\underline{x}, t)$ are referred to as flow vectors and the 2D vector field is called the optical flow field. The optical flow model is based upon the illumination changes between a small number of (typically 2) consecutive frames (images) of a video sequence. The flow field represents the motion of objects within the sequence.

This work describes a new optical flow estimator that uses a self organization strategy and is applicable to image sequences containing a priori unknown number of motion classes along with occlusion boundaries. It is well known that the problem of estimating motion is ill-posed with regression or regularization being typically used in gradient-based estimators to overcome the ill-posedness. The optical flow estimator developed in this work handles the ill-posedness via a novel combination of regression and a self organization strategy (specifically, self organization using unsupervised learning). The aperture problem is explicitly modeled using an algebraic framework that partitions motion estimates obtained from regression into two sets,

one (set H_c) with estimates with high confidence and another set (H_p) with low confidence estimates. The choice of the training set ($Q = H_c$) and the initial weights set ($W = H_c \cup H_p$) used during unsupervised learning is novel and leads to interpolation/correction of flow vectors. Using Heskes [Hes01] interpretation of Self Organizing Map (SOM) we show that the interpolation of flow vectors is achieved primarily due to the regularization property of SOM. Anisotropic neighborhoods designed for SOM using a novel residual-based distance measure preserve flow discontinuities at occlusion boundaries.

A multi-resolution or pyramidal approach is used to estimate large motion. As self organization-based motion estimation is computationally intense, parallel processing on Graphics Processing Units (GPU) is used for speedup. With sufficient number of GPU computing cores the implementation of the estimator can be made real-time. With the available true motion from Middlebury database, error metrics like Angular Error (AE) and End Point Error (EE) are computed.

1.1 2D Motion Field and Optical Flow Field

Assume that some objects are moving in 3D space and are being imaged. The 3D motion of the objects can be represented using 3D motion vectors. These 3D motion vectors when projected on the camera's 2D image plane lead to 2D motion field. Optical flow field, on the other hand, is a 2D vector field which is estimated from the image intensity variations and does not always correspond to the 2D motion field. For example, consider a rotating white textureless sphere with Lambertian reflective surface. The 3D rotation motion generates a 2D motion field when projected on a 2D plane, but when imaged, the sphere appears with same image intensity across frames and hence with zero magnitude optical flow vectors. Optical flow is thus defined as

the apparent motion of brightness patterns in the image frames. The goal of optical flow estimators is to output an optical flow field that is as close to 2D motion field as possible. Hence forth, the terms optical flow and 2D motion are used interchangeably.

Applications that use motion information vary based on the density of the flow field. For example, motion segmentation of images requires dense flow fields where as tracking applications work with sparse flow fields. Optical flow has applications in fields such as surveillance, object-based video compression and recovery of 3D shape of objects.

Quantitative evaluation of various optical flow estimators can be found at Middlebury website (http://vision.middlebury.edu/flow). The Middlebury database is a set of test cases that is used to benchmark optical flow estimators. It is a good place to get an idea of the state-of-the-art in this field. Broadly, there are three categories [BB95a] of estimators for optical flow, namely, the frequency-domain estimators, the gradient-based estimators and the feature-based estimators.

The estimator developed in this work is gradient-based. The spatial and temporal gradients of the image intensity are used to determine motion. Gradient-based (differential) methods can further be classified into regularization-based global methods such as the Horn-Schunck [HS81] approach and into regression-based local methods such as the Lucas-Kanade [LK81] technique. Local methods determine estimates by optimizing some local energy-like expression whereas global methods try to minimize a global energy functional. The local methods are known to be robust to noise compared to global methods. On the other hand, global methods generate denser flow fields than the local methods. The estimator developed in this work starts as a local method and then uses a self-organization strategy (specifically, unsupervised learning) to interpolate flow field. This interpolation corrects the estimates and increases the flow field density as well. This self organization-based optical flow estimator can

be positioned in between local and global methods with respect to the density of flow field.

Differential methods rely on the Brightness Constancy Assumption (BCA) which assumes that all changes in brightness in the image sequence are attributed to motion. BCA is valid most of the time but violations have to be mitigated to get acceptable flow fields. A major source of BCA violation is occlusion. With multiple objects moving in different direction, the 2D motion field may have motion boundaries with different flows on either side of the boundary. This happens when an object occludes another. At these motion boundaries, brightness pattern previously unseen in a frame appears (dissoccludes) in the next frame. Similarly, brightness pattern in a frame might disappear (occlude) in the next frame. In these cases, there is no preservation of brightness pattern at the motion boundaries and BCA fails. These motion boundaries are referred to as occlusion boundaries. Other sources of BCA violation like specular reflections, transparent or translucent surfaces, ambient lighting variations and self illumination are ignored in this work. Estimation of multiple object flow fields with mitigation of occlusion boundaries is the focus of this work.

1.2 Gradient-based Estimation Theory

Gradient-based methods start explicitly with the Brightness Constancy Assumption (BCA). Suppose an image point $\underline{x} = [x\ y]^T$ at time t is moved to $[x + d_x\ y + d_y]^T$ at time $t + d_t$. Under a constant brightness assumption, the images of the same object point at different times have the same intensity value. Therefore

$$I(x + d_x, y + d_y, t + d_t) = I(x, y, t) \tag{1.1}$$

The Taylor series expansion of the left side term with only the linear terms, yields:

$$I(x + d_x, y + d_y, t + d_t) = I(x, y, t) + I_x d_x + I_y d_y + I_t d_t$$

where $I_x = \frac{\partial I}{\partial x}$, $I_y = \frac{\partial I}{\partial y}$ and $I_t = \frac{\partial I}{\partial t}$. Applying BCA (Eq 1.1) we get:

$$I_x d_x + I_y d_y + I_t d_t = 0 \tag{1.2}$$

Optical Flow Constraint Equation (OFCE): (Eq 1.2) can be written in terms of the flow vectors by dividing it by dt as:

$$I_x u + I_y v + I_t = 0 \qquad or \qquad \nabla I_{\underline{x}}^T \underline{v} = -I_t \tag{1.3}$$

The OFCE (Eq 1.3) is a relation between the spatial gradients $\nabla I_{\underline{x}}^T = [I_x \ I_y]$ and the temporal gradient I_t of image intensity $I(\underline{x}, t)$, with the velocity $\underline{v} = [u \ v]^T$ at pixel location $\underline{x}$.

Spatial Coherence: OFCE (Eq 1.3) when applied at a single pixel location is underconstrained, as there are two unknowns u and v in a single equation. Thus the problem of estimating motion using just single OFCE is ill-posed. Figure 1.1 shows the motion components u and v constrained by line $\nabla I_{\underline{x}}^T \underline{v} = -I_t$. Velocity $\underline{v} = [u \ v]^T$ can lie anywhere on the line and the exact position cannot be determined without adding additional constraints.

To overcome the ill-posedness, two techniques are typically used, namely, regression and regularization. In the regression-based methods, it is assumed that two (or more) adjacent pixels correspond to points of the same object and thus have same motion. This kind of spatial coherence assumption is characterized as "piecewise

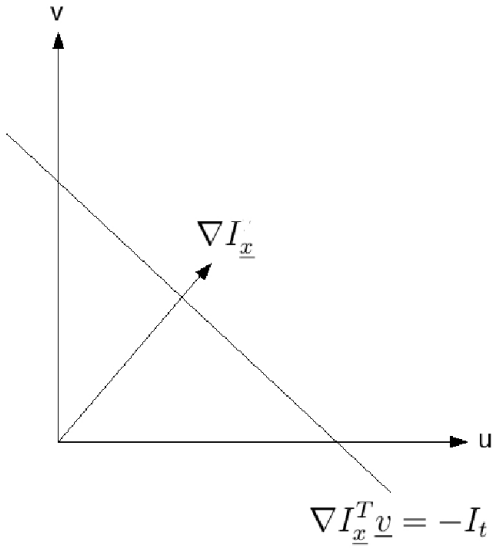


Figure 1.1: Optical Flow Constraint Line.

constant" motion assumption. On the other hand, the regularization-based methods assumes that flow within a neighborhood varies gradually since it is caused by single motion. Thus the spatial coherence assumption for regularization-based methods is characterized as "piecewise smooth" motion assumption. Regression-based methods are local in approach, in the sense that, the solution (motion estimate) obtained at a pixel has no influence on the solution obtained at nearby pixels. This is in contrast with global methods where solution (motion estimate) obtained at a pixel location is influenced by solutions at pixel locations in the neighborhood.

1.2.1 Regression-based Methods

With the assumption that motion is "piecewise constant", the regression-based methods pool OFCEs within a neighborhood (window or aperture) and simultaneously solve them to compute single motion vector. The motion is assigned to the pixel at the center of the aperture. This procedure is repeated by forming a neighborhood around each pixel.

For example, Lucas-Kanade [LK81] use the weighted least squares for obtaining the regression solution. They minimize the following error term (using L2 norm).

$$E_{LK} = \sum_{R} W^2 \left(I_x u + I_y v + I_t\right)^2 \tag{1.4}$$

where R is the region-of-support of pixels forming the aperture and W is a window function giving less weight to residuals further from the center from R. The confidence in the motion estimates can be ascertained by eigen analysis of the least squares matrix. The confidence of solution is typically low in low textured areas of the image.

Aperture Problem: In sparse texture area of the image, the spatio-temporal gradients in (Eq 1.4) may not have enough information to provide an unique solution (motion estimate). This problem is referred to as the Aperture Problem. Figure 1.2 shows motion of a rectangular region. The window labeled Aperture 1, has gradient information in both spatial directions and motion can be estimated reliably.

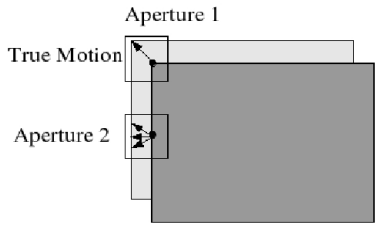


Figure 1.2: Aperture Problem

In an aperture with texture in only one dimension (Aperture 2 in Figure 1.2), motion can be estimated only in direction parallel to intensity gradient. Motion perpendicular to this direction cannot be uniquely identified due to lack of information. In an aperture with no texture, I_x, I_y and I_t will all be zero, hence no motion estimate can be made. Assuming single motion, the estimation ability and accuracy of the estimate with larger aperture is likely to be better than with smaller aperture

due to pooling of more information. But increasing the aperture size increases the chances of pooling information from pixels with different motions and hence leading to grossly wrong motion estimate. Thus aperture size cannot be made arbitrarily large and this problem is referred to as the Generalized Aperture Problem. Black and Anandan [BA96] try to overcome issues at occlusion boundaries by using robust statistics to estimate the dominant motion by ignoring some OFCEs (outliers).

The regression-based methods purge motion estimates with low confidence measures (mostly with aperture issues) thus leading to sparse but reliable flow field.

1.2.2 Regularization-based Methods

As seen earlier, the Brightness Constancy Assumption (Eq 1.1) leads to Optical Flow Constraint Equation (Eq 1.3) which is ill-posed due to presence of two unknowns u and v in a single equation. The regularization-based techniques try to overcome this ill-posedness by assuming "piecewise smoothness" of motion (Spatial Coherence). Regularization methods try to solve the problem by minimizing a global energy functional which incorporates both the Brightness Constancy Assumption (BCA) and the Spatial Coherence Assumption. The typical form of the global energy term is given [BSL+07] as:

$$E_{Global} = E_{Data} + \lambda E_{Prior} \tag{1.5}$$

where the Data Term E_{Data} is based on OFCE errors or residuals, and where the Prior Term E_{Prior} is based on smoothness assumption favoring certain flow fields over others. λ controls the relative importance of E_{Data} and E_{Prior}. With larger value of λ, more smooth flow fields are obtained.

The Data Term is obtained by aggregating the error (residual) per pixel of the OFCEs

over the image using some norm. For example, the Horn-Schunck algorithm [HS81] uses L2 norm to get:

$$E_{Data} = \sum_{x,y} \left[I_x u + I_y v + I_t \right]^2 \tag{1.6}$$

The Prior Term is derived by assuming that neighboring points in an image will move in a similar manner, constraining the optical flow to change smoothly over the image. For example, the Horn-Schunck algorithm [HS81] uses Laplacian of the optical flow to enforce smoothness with:

$$E_{Prior} = \sum_{x,y} \left[u_x^2 + u_y^2 + v_x^2 + v_y^2 \right] \tag{1.7}$$

where $u_x = \frac{\partial u}{\partial x}$, $u_y = \frac{\partial u}{\partial y}$, $v_x = \frac{\partial v}{\partial x}$ and $v_y = \frac{\partial v}{\partial y}$.

Robust Statistics: The use of L2 norm assumes that the gradients of the flow field are Gaussian and IID [BSL+07]. These assumptions are not true especially near occlusion boundaries. Black and Anandan [BA96] use robust penalty functions for both the Data Term and the Prior Term to handle violations of assumptions used in gradient-based techniques. While Black and Anandan [BA96] use Lorenzian penalty function, the recent algorithms typically use L1 norm [BSL+07]. There are various algorithms that refine the regularization-based approach using temporal smoothness, spatial weighting functions, photometrically invariant features and some other techniques, which are described in [BSL+07].

The solution (motion estimates) in regularization-based methods is obtained by minimizing the global energy term (Eq 1.5) typically using either Gradient Descent algorithms or by using Variational approaches.

1.2.3 Self Organization Approach

This work uses clustering techniques to overcome ill-posedness of motion estimation while generating dense flow fields. The idea is to feed motion data as input to a clustering algorithm to identify clusters of motion vectors. The motion data should have both the high and low confidence motion estimates obtained from a regression-based technique. Once the cluster membership is known it is possible to refine low confidence estimates to make them similar to high confidence estimates. This refinement is justified using the spatial coherence assumption, either "piecewise constant" or "piecewise smooth". The motion estimates with low confidence measures are the ones that are affected due to aperture issue. The refinement of these vectors thus helps mitigate the aperture problem.

Self Organization: Not all clustering algorithms are suitable for motion refinement. Some clustering algorithms require that the number of clusters to be formed be specified before hand. For a general flow field it is not possible to know the number of clusters a-priori and such algorithms cannot be used for general flow field estimation. A class of clustering algorithms follow self organization approach and do not require number of clusters as input. These self organizing algorithms use competitive learning with soft-max adaptation to identify clusters. Self organizing algorithms like Self Organizing Map (SOM) [Koh90] and Neural Gas (NG) [MBS93] are used in this work to show feasibility of using self organization methods in motion estimation/refinement.

1.3 Large Motion using Pyramids

The OFCE (Eq 1.3) is a differential motion model based on spatial and temporal gradients of image intensity. The gradients are computed using image intensity values within a small neighborhood. If the motion magnitude is larger than this neighborhood, then the motion model that we use is no longer valid. That is, OFCE assumes small motion. To handle large motion, multi-resolution or pyramidal approach is required. If the resolution of images is reduced sufficiently, then use of the differential motion model (OFCE) becomes valid.

The pyramidal approach typically involves construction of pyramids of spatially filtered (low-pass) and sub-sampled images. The differential motion model must be valid at the coarsest-resolution of the pyramid. Motion estimation starts at the coarsest resolution and moves through remaining pyramid levels. Motion computed at a coarser resolution is used to estimate motion at the next finer resolution in the pyramid. To make sure that the OFCEs are valid at the finer resolution, the motion estimate from coarser resolution is scaled and used to warp one of the finer resolution images (say image1) towards another (say image2). The remaining (delta) motion between the warped image and image2 is small and can be computed. The process is repeated till the highest resolution in the pyramid. The delta motion computed at each level is scaled and accumulated to generate the final motion estimate.

In this work, the pyramid is used along with the self organization-based motion estimation technique to handle large motion. Another side-effect of multi-resolution processing is the increase in density of flow estimates.

1.4 Parallel Computing on GPU

The self organization-based motion estimation is computationally intense. The good thing is that, all stages for computing optical flow using this technique can be made data-parallel (rather datum-parallel) and can be speeded up using Single Instruction Multiple Data (SIMD) type machines. To effectively leverage SIMD hardware capabilities, it is important that the processing of each data element be independent of the processing of other data elements.

In this work, NVIDIA GPUs are used as SIMD machines for general purpose computing. GPUs are well suited for data-parallel computations as they have multiple computing cores driven by very high memory bandwidth. NVIDIA has Compute Unified Device Architecture (CUDA) which is suitable for "general purpose computing on GPU" (GPGPU). Apart from NVIDIA specific CUDA, there is OpenCL (Open Computing Language) specification which can be used for GPGPU. We use NVIDIA CUDA in this work.

1.5 Middlebury Database

Evaluation of optical flow estimators is based on applications. For tracking applications, sparse flow estimators are good as they provide reliability measures along with the flow. For applications that require dense optical flow, the quantitative evaluation of estimators is based on error metrics. To compute error metrics, true motion needs to be known for the image sequence. This is where standard test databases have a role to play. Middlebury database (http://vision.middlebury.edu/flow/data/) has test sequences with complex scenes and serves as a effective benchmark for dense optical flow estimators. With the available true motion, error metrics like Angular

Error (AE) and End Point Error (ER) are computed.

1.6 Outline

The work described here in is a new optical flow field estimator that uses self organization techniques and is applicable to image sequences containing a priori unknown number of motion classes along with occlusion boundaries. The rest of the chapters go into details of the estimator and its evaluation. Chapter 2 explains the theory behind self organization and analyzes various algorithms and ascertains suitability for motion estimation. Chapter 3 is about the application of self organization techniques to motion estimation. It goes into details of gradient estimation, local (regression-based) motion estimation and use of SOM and Neural Gas algorithms for motion refinement or interpolation of flow. Online and batch versions of these algorithms are also discussed. Chapter 4 is about large motion and the use of multi-resolution or pyramidal framework. This chapter thus provides the end-to-end details of the optical flow estimator. The quantitative evaluation of the algorithm also can be found in this chapter. Chapter 5 shows how the estimator developed in Chapter 4 is mapped onto GPU. Conclusion is stated and follow on work is discussed in Chapter 6.

Chapter 2

Self Organization Approaches and Algorithms

Self organizing algorithms use competitive learning with soft-max adaptation to identify clusters. Self organizing algorithms like Self Organizing Map (SOM) [Koh90] and Neural Gas (NG) [MBS93] are used in this work. This chapter goes into details of clustering algorithms but leaves their application to motion estimation to the next chapter.

Clustering: The goal of clustering is to group data such that data points with certain similarity are assigned to the same class. Clustering methods differ in various aspects including the assignment of data points to classes which might be crisp or fuzzy, the arrangement of clusters which might be flat or hierarchical, or the representation of clusters which might be represented by explicit cluster identification or by few prototypical vectors (exemplars) [CHHV06].
For our optical flow estimator, explicit cluster membership information is not required. The algorithms that are considered here deal with crisp assignments and

representation of clusters by exemplars. Going forward, various neural clustering algorithms are detailed and the applicability for optical flow is accessed.

Neural Networks: In this work, we deal with neural networks and leverage their ability to learn. The network architectures can roughly be divided into three categories, each based on a different philosophy [Koh90], namely, the feedforward networks, the feedback networks and the competitive or self-organizing networks. The feedforward networks are made up of layers of neurons, typically, with an input layer, a hidden layer and an output layer. There are weights associated with the connections between layers and information in the net always passes in one direction, that is from the input layer towards the output layer. Back propagation is a widely used technique for training the feedforward network weights. Feedforward neural networks with back propagation learning are quite popular for pattern mapping, especially classification [Sch97]. Feedback neural networks, on the other hand, have directed cycles in the network connections. These are also called recurrent networks and the fully-recurrent ones are formed by neurons that have directed connections with every other neuron in network. Recurrent networks form autoassociative memory and the stored pattern correspond to the stable states of a nonlinear system [Sch97]. The third type of networks, called the self-organizing networks are based on competitive learning. The neighboring neurons in these neural networks compete in their activities by means of mutual lateral interactions, and develop adaptively into detectors of specific signal patterns [Koh90].

Another way to categorize neural networks is according to the way they are trained. Categorization by type of training leads to two types of networks, namely, supervised networks and unsupervised networks. In supervised learning, the network weights are trained using a training set consisting of input and output training

patterns. In unsupervised learning, on the other hand, there are no known output patterns. Unsupervised learning uses just the inputs to train the weights. The network weights adapt or learn based on the experiences collected through the input patterns. Some measure of pattern associativity or similarity is used to guide the learning process, which usually leads to some form of network correlation, clustering, or competitive behavior [Sch97]. The feedforward networks and some feedback networks use supervised learning while self-organizing networks use unsupervised learning.

For the optical flow estimator, the clustering ability of the self-organizing networks is of importance and this chapter focuses on unsupervised learning and self-organizing networks. Specifically, we discuss k-Means, the Self Organizing Map (SOM) and Neural Gas (NG). All the algorithms have both online version as well as batch version. It will be seen that Batch SOM plays an important role in optical flow estimation.

Quantization of Vector Space: Cluster representation using exemplars leads to quantization of vector space. Vector quantization allows modeling of vector spaces (specifically, their density functions) by a set of prototype vectors. Vector quantization has applications in density estimation and data compression. Assume data points $\underline{x} \in \mathbb{R}^m$ are distributed according to an underlying distribution P. The goal of vector quantization is to find prototype locations $\underline{w}_i \in \mathbb{R}^m, i = 1, \ldots, N$, such that these prototypes represent the distribution P as accurately as possible. A data vector $\underline{x}$ is described by the best-matching or "winning" reference vector $\underline{w}_{i(\underline{x})}$ for which the distortion error $d(\underline{x}, \underline{w}_{i(\underline{x})})$ (for example, squared Euclidean distance) is minimal [MBS93]. These reference vectors are also referred to as codebook vectors. The optimal choice of reference vectors $\underline{w}_i$ varies based on the cost or error function being minimized.

The neural clustering algorithms that we are interested in, represent clusters by neurons or rather by the corresponding weight vectors. This essentially is vector space quantization and the weight vectors approximate the underlying (unknown) distribution of the vector space. Each of these algorithms minimize a different cost function, hence the reference vectors obtained by different algorithms need not be same. The following discussion closely follows [CHHV06].

2.1 k-Means

k-Means optimizes the standard quantization error [CHHV06]

$$E_{kmeans}(\underline{w}) \sim \sum_{i=1}^{N} \int \chi_{I(\underline{x})}(i)\ d(\underline{x}, \underline{w}_i)\ P(d\underline{x}) \tag{2.1}$$

where $I(\underline{x})$ denotes the index of the closest prototype (one of the $\underline{w}$'s), the winner for $\underline{x}$, and $\chi_{I(\underline{x})}(i)$ is the indicator function or characteristic function (from the set theory). More specifically, $\chi_{I(\underline{x})}(i) = 1$ if $\underline{w}_i$ is the winner unit weight, otherwise $\chi_{I(\underline{x})}(i) = 0$. $\chi_{I(\underline{x})}(i)$ restricts the integration to only part of domain of $\underline{x}$ for which $\underline{w}_i$ is the winner.

k-Means online version: The online version of the algorithm is used when the distribution P is not known a priori, but instead a stochastic sequence of sample data points $x_j(t)$ which is governed by P is available. The learning rule consists of a stochastic gradient descent, yielding

$$\Delta\underline{w}_i = \epsilon\ \delta_{ij(\underline{x}_j)}\ (\underline{x}_j - \underline{w}_i) \tag{2.2}$$

for all prototypes $\underline{w}_i$ given a data point $\underline{x}_j$. ϵ is the step size and δ_{ij} is the Kronecker delta.

k-Means batch version: If the discrete training data $\underline{x}_1, \ldots, \underline{x}_M$ are known a priori, then fast batch version can be used. Batch k-Means optimizes the same cost function as the online variant [BB95b]. Starting from random positions of the prototypes, batch learning performs the following two steps until convergence [CHHV06]

1. Determine the winner index $I(\underline{x}_j)$ for each data point $\underline{x}_j$ using some distance measure.

2. Determine the new prototypes as

$$\underline{w}_i = \frac{\sum\limits_{j|I(\underline{x}_j)=i} x_j}{|\{j|I(\underline{x}_j) = i\}|} \tag{2.3}$$

k-Means is very sensitive to initialization of the prototypes since it adapts the prototypes only locally according to their nearest data point. There is no guarantee that it will converge to the global minimum and the results depend on the initialization of weights $\underline{w}_i$. As the algorithm is fast, k-Means can be run multiple times with different starting conditions to mitigate the dependence on initialization.

For simple optical flow cases, k-Means gives good results when the number of motion classes are known before hand. In general, the number of flow classes is unknown and the choice of k is a challenging issue. k-Means is therefore not useful for general optical flow estimation.

2.2 Self Organizing Map (SOM)

SOM implements a form of local competitive learning. SOM is a neural learning structure involving networks that perform dimensionality reduction through conversion of feature space (input data space) to yield topologically ordered similarity graphs or maps or clustering diagrams [Koh90].

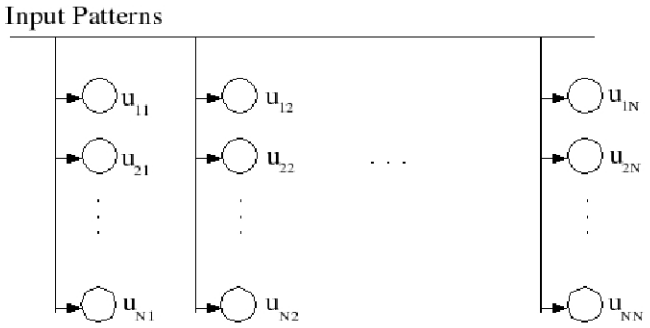


Figure 2.1: SOM Topology

SOM is a sheet-like neural network (Figure 2.1) with weights associated with each unit. The ability of SOM to provide visualization of clusters present in high dimensional space by projection onto a 2D map is very useful. An important aspect of the SOM network is the lateral connections between the units. The lateral connections help in determining the neighborhood during training of the net and are responsible for maintaining the topology of input space during the projection onto the lower dimensional map.

SOM uses soft-max adaptation as part of its update strategy, that is, it not only updates the winner unit weights, but also updates the weights of the neighboring units (Figure 2.2). The k-Means algorithm in contrast follows winner-takes-all approach and is thus very local. The soft-max update helps SOM avoid local minima during training of weights.

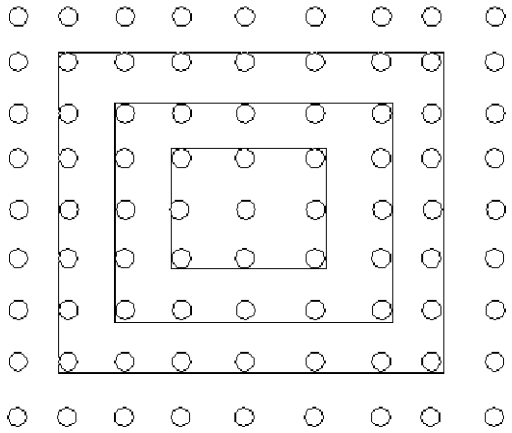


Figure 2.2: SOM Neighborhood

SOM online version: The learning rule for the online version of SOM consists of a stochastic gradient descent, yielding

$$\Delta \underline{w}_i = \epsilon \; h_\lambda(nd(I(\underline{x}_j), i)) \; (\underline{x}_j - \underline{w}_i) \tag{2.4}$$

for all prototypes $\underline{w}_i$ given a data point $\underline{x}_j$. $h_\lambda(t) = exp(-t/\lambda)$ is decaying exponential with $\lambda > 0$. nd is a two-dimensional neighborhood around the winning unit given by index $I(\underline{x}_j)$.

SOM [Koh90] does not posses a cost function in the the continuous case and its mathematical investigation is difficult [CFP98]. However, if the winner is chosen as the unit i with minimum averaged distance

$$d_{ij} = \sum_{l=1}^{N} h_\lambda(nd(i,l)) \; d(\underline{x}_j - \underline{w}_l) \tag{2.5}$$

as shown by [Heskes2001], then it optimizes the cost [CHHV06]

$$E_{SOM}(\underline{w}) \sim \sum_{i=1}^{N} \int \chi_{I^*(\underline{x})}(i) \; \sum_{l=1}^{N} h_\lambda(nd(i,l)) \; d(\underline{x}_j - \underline{w}_l) \; P(d\underline{x}) \tag{2.6}$$

where $I^*(\underline{x})$ denotes the winner index (according to Eq 2.5) of the closest prototype (one of the $\underline{w}$'s), and where $\chi_{I^*(\underline{x})}(i)$ is the indicator function. More specifically, $\chi_{I^*(\underline{x})}(i) = 1$ if w_i is the winner unit weight, otherwise $\chi_{I^*(\underline{x})}(i) = 0$.

SOM batch version: Batch SOM optimizes the same cost function as the online variant [Che97]. For the discrete training data $\underline{x}_1, \ldots, \underline{x}_M$, starting from random positions of the prototypes, batch learning performs the following two steps until convergence [CHHV06]

1. Determine the winner index $I^*(\underline{x}_j)$ for each data point $\underline{x}_j$ using distance measure given by Equation 2.5.

2. Determine the new prototypes as

$$\underline{w}_i = \frac{\sum_{j=1}^{M} h_\lambda(nd(I^*(\underline{x}_j), i))\ \underline{x}_j}{\sum_{j=1}^{M} h_\lambda(nd(I^*(\underline{x}_j), i))} \tag{2.7}$$

SOM is not as sensitive to initialization of the prototypes as k-means, since it adapts the prototypes using soft-max approach as against winner-takes-all approach.

2.3 SOM, VQ and Mixture Models

Mixture Models is a probabilistic framework for representing clusters of data. Each cluster is represented by a parametric distribution, eg. Gaussian distribution. The entire dataset, that is, all the clusters are together modeled by mixture of the individual distributions. If all the clusters are represented by Gaussian distributions, then the entire dataset is modeled as a mixture of Gaussians. The parameters of the distributions and the mixture values (fractions) are estimated by the maximum likelihood criteria using Expectation Maximization (EM) algorithm [DLR77].

Heskes [Heskes2001] explores the links between SOM, Vector Quantization(VQ) and Mixture Models and we follow that material closely here. In the next chapter we utilize these results to derive cost function for motion estimation. SOM can be interpreted as Vector Quantization with lateral interactions. With this interpretation, the error function or cost in discrete domain minimized by SOM is given as:

$$F_{quant}(P,W) = \sum_{j=1}^{M}\sum_{i=1}^{N} p_{ij} \sum_{l=1}^{N} h_{il} D(\underline{x}_j, \underline{w}_l) \tag{2.8}$$

where M is the cardinality of inputs/training set and N is the cardinality of weights set. p_{ij} denotes the probability that the input $\underline{x}_j$ is assigned to node with weight $\underline{w}_i$ and where $D(\underline{x}_j, \underline{w}_l)$ is some distance measure (for example, it could be Euclidean distance measure). Even if we assign input $\underline{x}_j$ to weight $\underline{w}_i$, there is a confusion probability h_{il} that the input $\underline{x}_j$ is instead quantized by the weight vector $\underline{w}_l$ in the neighborhood of $\underline{w}_i$. h_{il} corresponds to the lateral interaction strength and typically $h_{il} = exp[-d_{il}/2\sigma^2]$ where d_{il} refers to the node distance on a two dimensional grid. The annealed version of SOM uses an entropy term of the form

$$F_{entropy}(P) = \sum_{j=1}^{M}\sum_{i=1}^{N} p_{ij}\ log\left(\frac{p_{ij}}{g_i}\right) \tag{2.9}$$

where g_i is interpreted as prior probability assignments. The usual choice of g_i is $1/N$. The final energy functional is given as:

$$F(P,W) = \lambda F_{quant}(P,W) + F_{entropy}(P) \tag{2.10}$$

Expectation Maximization: Heskes [Heskes2001] derives the Expectation Maximization (EM) algorithm for SOM using the energy functional (Eq 2.10). The ex-

pectation step takes the form

$$p_{ij}(W) = \frac{g_i \; exp[-\beta \sum_{t=1}^{N} h_{it} D(\underline{x}_j, \underline{w}_t)]}{\sum_{l=1}^{N} g_l \; exp[-\beta \sum_{t=1}^{N} h_{lt} D(\underline{x}_j, \underline{w}_t)]} \tag{2.11}$$

Minimizing $F(P, W)$ with respect to the parameters W, given the current set of assignment P gives the maximization step

$$\underline{w}_l(P) = \frac{\sum_{j=1}^{M} \sum_{i=1}^{N} p_{ij} h_{il} x_j}{\sum_{j=1}^{M} \sum_{i=1}^{N} p_{ij} h_{il}} \tag{2.12}$$

Mixture Model Interpretation: From [Hes01] we also have interpretation of SOM as mixture model plus additional regularization. The energy functional being optimized in this case is given as

$$E(W) = -L(W) + E_{regular}(W) \tag{2.13}$$

$$L(W) = \sum_{j=1}^{M} log \; P(\underline{x}_j | W) \tag{2.14}$$

$$E_{regular}(W) = -\sum_{j=1}^{M} log \sum_{i=1}^{N} g_i \; e^{-\beta V_i(W)} \tag{2.15}$$

where $L(W)$ is the optimization criterion corresponding to a maximum likelihood procedure for mixture of Gaussians when no lateral interactions are considered, similar to that of Vector Quantization. In Vector Quantization case, the mixture of Gaussians is given as $P(\underline{x}|W) = \sum_{i=1}^{N} g_i \; G(\underline{x}|\underline{w}_i)$.

Whereas, for SOM, [Hes01] shows that $E(W)$ is obtained by the following mixture of Gaussians

$$P(\underline{x}|W) = \sum_{i=1}^{N} \tilde{g}_i(W) G(\underline{x}|\underline{\tilde{w}}_i) \tag{2.16}$$

with

$$\tilde{g_i}(W) \equiv \frac{g_i \; e^{-\beta V_i(W)}}{\sum_{l=1}^{N} g_l \; e^{-\beta V_l(W)}}$$

$E_{regular}(W)$ uses the variance term $V_i(W)$ that when minimized leads to regularization. Lateral interactions that are characteristic of SOM are responsible for the variance term and in turn responsible for regularization. The variance term is obtained by decomposition of error $\sum_{l=1}^{N} h_{il} D(\underline{x}, \underline{w}_l)$ into a bias term $D(\underline{x}, \underline{\tilde{w}}_i)$ and a variance term $V_i(W) = \sum_{l=1}^{N} h_{il} D(\underline{\tilde{w}}_i, \underline{w}_l)$ with $\underline{\tilde{w}}_i = \sum_{l=1}^{N} h_{il} \underline{w}_l$. The essence here is that the average error can be decomposed into an error of an average weight $\underline{\tilde{w}}_i$ and a variance term independent of the input x.

SOM can be applied to motion estimation as it does not need to know the number of motion classes a priori. Moreover, we will see in the next chapter that using the interpretation of SOM as mixture model plus additional regularization, how a special case of SOM leads to non-linear interpolation of optical flow.

2.4 Neural Gas (NG)

Neural Gas (NG) [MBS93] also uses the soft-max adaptation rule like SOM, but the neurons that are updated along with the best matching (winner) unit are selected differently. It selects the neighbors for update using a neighborhood ranking scheme. A set $D_{\underline{x}_j} = \{d(\underline{x}_j, \underline{w}_i), i = 1, 2, \ldots, N\}$ is calculated and is used to generate a rank index for each of the units (weight vectors). The unit (weight vector) with least distance from the training vector has the highest rank. This rank is used to determine the neighborhood of the winner unit.

The name Neural Gas reflects the fact that the neighborhood used to update networks is determined by the relative distances between the neural units within the unit

weight space and not determined by relative distances between neural units within a topologically prestructured lattice. Neighborhood-ranking requires explicit ordering of the rank of each unit. Explicit ranking is computationally costly and can be replaced with an implicit ordering metric [AS98].

Neural Gas (NG) optimizes the following cost function [MBS93]

$$E_{NG}(\underline{w}) \sim \frac{1}{2C(\lambda)} \sum_{i=1}^{N} \int h_\lambda(k_i(\underline{x}, \underline{w}))\ d(\underline{x}, \underline{w}_i)\ P(d\underline{x}) \tag{2.17}$$

where $k_i(\underline{x}, \underline{w}_i) = |\{\underline{w}_j | d(\underline{x}, \underline{w}_j) < d(\underline{x}, \underline{w}_i)\}|$ is the rank of the prototypes sorted according to the distances, and $C(\lambda)$ is the constant $\sum_{i=1}^{N} h_\lambda(k_i)$. $h_\lambda(k_i)$ is typically a decaying exponential.

NG online version: The learning rule consists of a stochastic gradient descent, yielding

$$\Delta\underline{w}_i = \epsilon\ h_\lambda(k_i(\underline{x}_j, \underline{w}))\ (\underline{x}_j - \underline{w}_i) \tag{2.18}$$

for all prototypes $\underline{w}_i$ given a data point $\underline{x}_j$.

NG batch version: For the discrete training data $\underline{x}_1, \ldots, \underline{x}_M$, starting from random positions of the prototypes, batch learning performs the following two steps until convergence [CHHV06]

1. Determine $k_{ij} = k_i(\underline{x}_j, \underline{w}) = |\{\underline{w}_l | d(\underline{x}, \underline{w}_l) < d(\underline{x}, \underline{w}_i)\}|$ as the rank of prototype $\underline{w}_i$.

2. Determine the new prototypes based on the hidden variables k_{ij} as

$$\underline{w}_i = \frac{\sum_{j=1}^{M} h_\lambda(k_{ij})\ x_j}{\sum_{j=1}^{M} h_\lambda(k_{ij})} \tag{2.19}$$

The Neural Gas algorithm does not require the knowledge of the number of clusters, and hence can be used for motion estimation.

Chapter 3

Self Organization-based Optical Flow

The work described here is a new optical flow field estimation technique that is applicable in image sequences containing a priori unknown number of motion classes along with occlusion boundaries. It is well-known that the determination of motion parameters for even a single motion class using an optical flow formulation poses several well-known challenges, including an inherently locally ill-posed estimation problem with the possibility of the aperture problem.

This work can be categorized as an intensity-based differential (gradient-based) method. The algorithm described here tries to overcome the aperture problem by formulating an algebraic framework involving rank, condition numbers and min-norm solution and then by using non-linear interpolation. Unsupervised learning techniques are used for non-linear interpolation that also help in cases with multiple motion classes with occlusion.

In regularization techniques (seen in Chapter 1), the iterations propagate flow to neighboring areas and can cause problem at occlusion boundaries. At the occlusion

boundaries two different motion fields/classes are neighbors. Each motion class tries to propagate its values leading to averaging of the flow field. This issue at occlusion boundaries in regularization methods has been addressed by [BA96] using robust statistical techniques. Even for unsupervised learning-based techniques (specifically for SOM), the occlusion boundaries need to be handled and we do so by using anisotropic neighborhood functions. We use Neural Gas inspired anisotropic functions in SOM to make sure that only one of the competing training motion vectors wins. This avoids the averaging of motion of two competing training motion vectors. In effect, smearing of motion at occlusion boundaries is minimized.

The steps of our motion estimation technique are exemplified by processing the two images of a synthetic image sequence as shown in Figure 3.1. The images have eight square objects moving in eight different directions, therefore we have eight motion classes to be detected. The true motion (u, v) for each object is shown in Figure 3.2. The origin of the coordinate system is at the top left corner of the image. The temporal difference image is given in Figure 3.2.

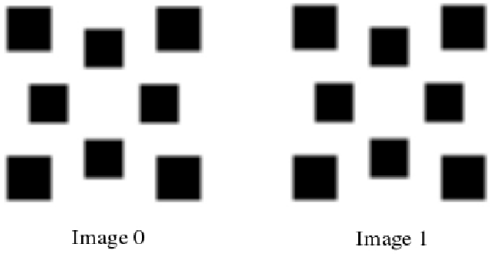


Figure 3.1: Binary Image Sequence.

Our motion estimation algorithm can be split into following three consecutive stages.

1. Gradient Estimation

2. Local Motion Estimation

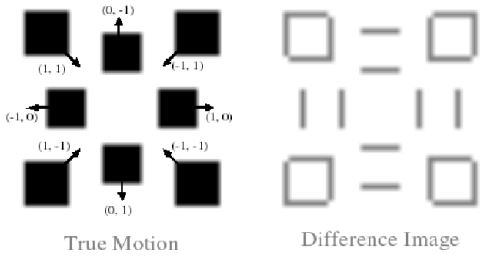


Figure 3.2: True Motion and Temporal Difference Image.

3. Non-Linear Interpolation

3.1 Gradient Estimation

As we use intensity gradients, we assume that the image intensity is differentiable. This assumption generally holds for natural images, as the image formation process involves integration. On the other hand, the differentiability assumption may not hold for synthetically generated images. When images are binary (eg. intensity of 0 or 255 only), they are not differentiable. In order to calculate gradients, the images need to be smoothed and made differentiable. We use the approximate differentiation as given in [HS81]. Figure 3.3 shows the spatial and temporal gradients computed for the sample image sequence.

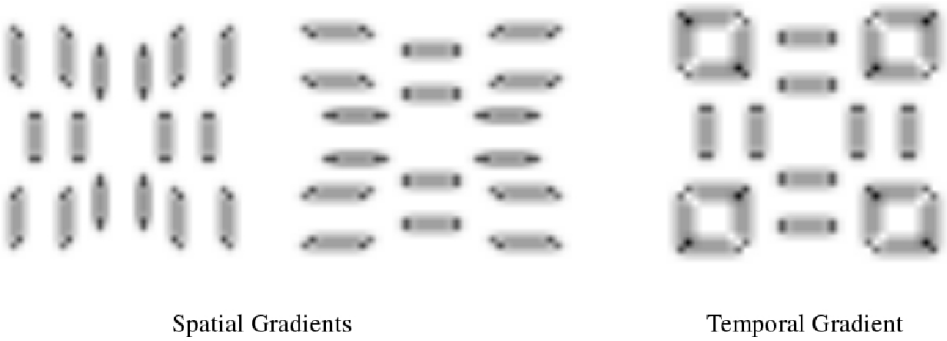


Figure 3.3: Gradients I_x, I_y and I_t

3.2 Local Motion Estimation

In chapter 1, we saw the Optical Flow Constraint Equation (OFCE) (Eq 1.3)

$$I_x u + I_y v + I_t = 0$$
$$\nabla I(\underline{x}, t)^T \underline{v} = -I_t$$

derived using the Brightness Constancy Assumption (BCA) (Eq 1.1). OFCE establishes a relationship between intensity gradients I_x, I_y, I_t and the motion vector components u and v. As noted earlier, OFCE at single pixel location is ill-posed as there are two unknowns (u, v) and just a single equation. To overcome ill-posedness, spatial coherence is assumed, which states that nearby pixels are likely to correspond to the same object and thus would have same or similar motion. In local motion computation, we assume piecewise constant motion and solve multiple OFCEs simultaneously to get the motion estimate. For example, an aperture or window of size 2×2 yields four equations and can be formulated as:

$$\begin{bmatrix} I_{x1} & I_{y1} \\ I_{x2} & I_{y2} \\ I_{x3} & I_{y3} \\ I_{x4} & I_{y4} \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix} = - \begin{bmatrix} I_{t1} \\ I_{t2} \\ I_{t3} \\ I_{t4} \end{bmatrix}$$
$$D \qquad \underline{v} \quad = - \quad \underline{f}_t \tag{3.1}$$

The regression-based motion estimate can then be obtained using Moore-Penrose inverse as

$$\underline{v} = -(D)^{\dagger} \underline{f}_t \tag{3.2}$$

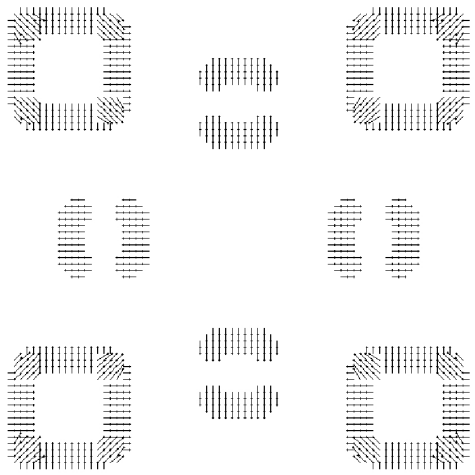


Figure 3.4: Local Motion Estimation with Aperture Problem.

3.2.1 Aperture Problem - An Algebraic Viewpoint

Typically, a square neighborhood around a pixel is considered for OFCE pooling to form an aperture with more regional support. With more pixels being pooled, the matrix D and vector f_t grow more taller. For example, in a 3×3 aperture, nine pixels contribute their OFCEs making D a 9×2 matrix, while making f_t a 9×1 sized vector. The solution is still given by (Eq 3.2). The solution depends on the gradient values and is correlated with the amount of texture within the aperture. We have developed an algebraic viewpoint of the aperture problem that involves the rank of matrix D (Eq 3.1). It has to be noted that the rank of matrix D will be at most two (2), independent of the number of pixels used in the spatial coherence augmentation. There are always the following three cases:

1. $\text{rank}(D) = 0$. This case corresponds to a location with no spatial texture or where motion is not discernible. (Interior regions of the squares in Figure 3.4).

2. rank(D) = 1. This image location suffers from the aperture problem. Motion vector component parallel to the spatial gradient can only be estimated. The vector component normal to the spatial gradient could be of any magnitude and is indeterminate. Let $\underline{v} = \underline{v}_{partEst} + \underline{v}_a$, where $\underline{v}_{partEst}$ is the observable component of motion and where $\underline{v}_a$ is the component that cannot be determined due to the aperture problem. $\underline{v}_a$ satisfies $D\underline{v}_a = \underline{0}$ or $\underline{v}_a \in nullspace(D)$. The Moore-Penrose inverse of D is used to find minimum norm solution for this rank deficient case. (Sides of the squares in Figure 3.4).

3. rank(D) = 2. This case leads to a complete and acceptable motion estimate if the condition number of D is reasonable. (Corners of the squares Figure 3.4).

The algebraic viewpoint provides a formal, systematic technique for the recovery of the missing local motion component $\underline{v}_a$ using modified versions of unsupervised learning algorithms.

Handling Occlusion: A larger aperture size with the system of the form of (Eq 3.1) leads to more rank(D) = 2 systems which, in turn, leads to more algebraically complete local motion estimates. But as the aperture size is increased, the chance of pooling information from pixel locations from different motion classes increases. This is very often the case near occlusion boundaries; the algebraic system can be numerically well-conditioned, but the solution (motion estimates) could be wrong. This is referred to as the *generalized aperture problem* [BA96]. The effects of outliers can be minimized by using anisotropic neighborhood functions. The anisotropic weights give more importance to the pixels with dominant motion in the neighborhood. The optimal weights, however, vary spatially and need to be estimated. Black and Anandan [BA96] implicitly calculate these weights by applying redescending influence functions

to residual values.

Approximating Weighted LS: Given a 3×3 aperture, in this work, we approximate an anisotropic neighborhood using the four weight functions shown in Figure 3.5. The 3×3 aperture is split into partly overlapping 2×2 apertures using weights W_k, $k = 1, 2, 3, 4$. Each 2×2 aperture leads to an overdetermined system and the assumption is that atleast one of these systems would remain unaffected by occlusion and is best suited for motion estimation. Instead of solving one system of equations for 3×3 aperture, we now need to solve four systems corresponding to four different orientations within the encompassing 3×3 aperture. We estimate motion and the residual error for each orientation by solving the overdetermined systems:

$$D_k \, \underline{v} = -\underline{f}_t \quad k = 1, 2, 3, 4 \tag{3.3}$$

Only one of the four systems is eventually retained. We choose the orientation that gives the least residual value and use motion estimate obtained at the chosen orientation as the best local estimate for the whole 3×3 aperture, thus approximating locally weighted least squares.

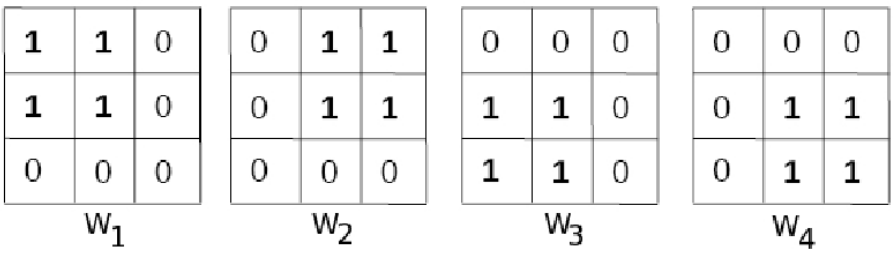


Figure 3.5: Opportunistic Anisotropic Neighborhood Weights.

3.2.2 Efficient 2-Pass Evaluation

From (Eq 3.3), at each pixel location, 4 least squares estimates need to be produced. This is computationally demanding. The computational load can be reduced considerably by exploiting the redundancies in evaluation of the systems. Consider a block of pixels shown in Figure 3.6. The anisotropic weights W_ks are centered and superimposed around two pixel locations p7 and p8. The circles show the pixel locations that are used to form a system (Eq 3.1) by pooling the optical flow constraints for each of the W_ks. The system formed at p7 using W_2 is same as the system formed at p8 using W_1. Similarly, the system formed at p7 using W_4 is same as the system formed at p8 using W_3. Instead of solving the same systems at multiple pixel locations, we can save computations by utilizing this redundancy using a two pass approach.

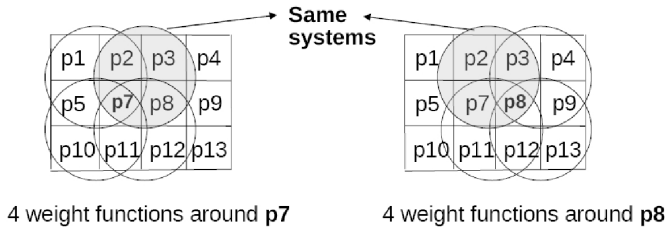
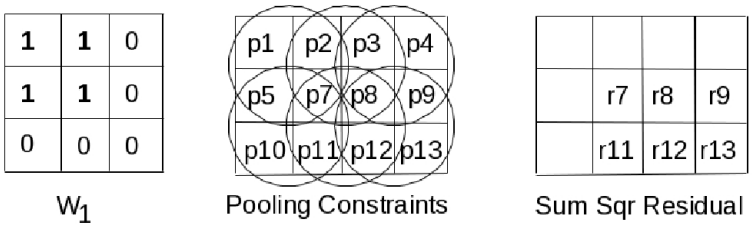


Figure 3.6: Opportunistic Anisotropic Weights - Redundant Systems.

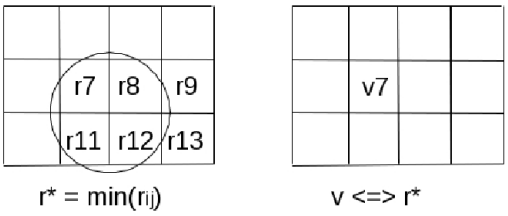
Pass I: At each pixel location, use weights W_1 and solve least squares system (Eq 3.1). Calculate the corresponding residuals and store them as shown in Figure 3.7(a). For example, $r7$ is the residual of the system formed using $p1$, $p2$, $p5$ and $p7$ while $r8$ is the residual of the system formed using $p2$, $p3$, $p7$ and $p8$. Weights W_2, W_3 and W_4 do not play any role in this 2-pass approach.

Pass II: At each pixel location determine the minimum residual amongst the four neighbors (Figure 3.7(b)). This step is equivalent to choosing one of the four orientations within a 3×3 aperture. For example, at location of pixel p7 in the residues array, four neighboring values $r7$, $r8$, $r11$ and $r12$ are compared. These four residues correspond to four systems around $p7$ formed as if using weights W_1, W_2, W_3 and W_4 respectively. Choosing the minimum of these four residue values gives the best orientation to compute motion that is least likely to be affected by occlusion. The motion corresponding to the minimum residual is chosen as the 'best' motion estimate.

By using the two pass approach the number of systems that need to be solved for an $r \times c$ block of pixels is reduced from $4rc$ to $(r+1)(c+1)$ which approximately saves computational load of $3rc$ systems.



(a) Pass I



(b) Pass II

Figure 3.7: Opportunistic Motion Estimation - 2 Pass

3.2.3 Matrix Rank-based Partitioning

The algebraic viewpoint helps identify the motion estimates that suffer from aperture problem and have missing motion components. Unsupervised learning techniques are later employed in the recovery of these missing local motion components. We partition the motion estimates into two sets, set H_c which is a set of motion vectors with complete solution and set H_p which is a set of motion vectors with partial solution. The motion vectors are augmented with two more components specifying their pixel locations leading to 4 dimensional feature vectors. Thus the vectors in sets H_c and H_p are of the form:

$$\underline{v} = (x, y, u, v)^T \tag{3.4}$$

where u and v are motion components at pixel location (x, y). If the rank of matrix D is either 0 or 1 (Aperture Problem) or if the residual is high (violation of spatial coherence), we put the motion estimate in set H_p, otherwise, the motion estimate is put in set H_c.

3.3 Unsupervised Learning

Once we have partitioned the local motion estimates into sets H_c (complete motion estimates) and H_p (partial motion estimates), it is time to interpolate the information and come up with global motion estimate. The non-linear interpolation is done using unsupervised learning techniques. We evaluate the use of Neural Gas (NG) and Self Organizing Map (SOM) for motion interpolation. Some modifications to the standard algorithms are required, and so different names are used to distinguish between various version. We discuss three algorithms, namely, NGFlow which uses modified online version of NG, SOMFlow which uses modified online version of

SOM, and BatchSOMFlow which uses modified and simplified version of Batch SOM algorithm.

3.3.1 NGFlow

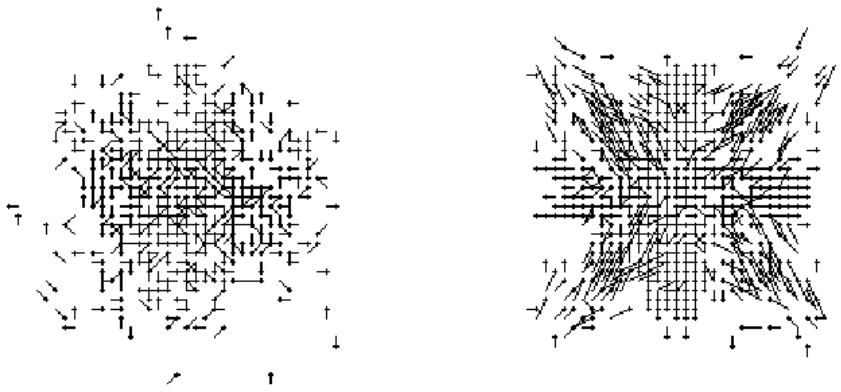
Neural Gas (NG) [MBS93], as seen in Chapter 2, uses soft-max adaptation to train a set of weights $W = \{\underline{w}_1, \underline{w}_2, \ldots, \underline{w}_N\}$ using training vectors $Q = \{\underline{q}_1, \underline{q}_2, \ldots, \underline{q}_M\}$. Before training starts the weights W need to be initialized. Typical initialization scheme involves use of random numbers. For motion interpolation, if we initialize the NG weights W using a Gaussian distribution and train using local motion output such that $Q = H_c \cup H_p$ then for the example case, the output of NG is as shown in Figure 3.8. However, a more opportunistic scheme for weight initialization exists. We set $W = H_c \cup H_p$ as initial weights and train them using NG with training set $Q = H_c$ and it leads to better interpolation. As shown in (Eq 3.4), we have 4 dimensional feature vectors and the NG weight update needs modification so that only the motion components u and v get updated while the pixel location components x and y remain untouched.

Neural Gas [MBS93, AS98] algorithm has been modified for motion interpolation to use following steps [SS09]:

1. Initialize unit weights as $W = H_c \cup H_p$ and use set $Q = H_c$ as the training set. For the example case, Figure 3.9 show initial W and Figure 3.10 shows the training set Q.

2. Present the input training vector $\underline{q}_j$ and compute the distances set $D_{\underline{q}_j} = \{d(\underline{q}_j, \underline{w}_i), i = 1, 2, \ldots, N\}$. Use a weighted distance metric given by

$$d(\underline{q}_j, \underline{w}_i) = ||\underline{q}_j - \underline{w}_i||_R = \sqrt{(\underline{q}_j - \underline{w}_i)^T \; R \; (\underline{q}_j - \underline{w}_i)}$$



(a) Initial Weights (b) Weights after 10 iterations

Figure 3.8: Neural Gas Clustering (Gaussian Initialization).

where diagonal matrix R is used to compensate scale differences between the motion components and the pixel locations. Due to small motion model, motion components have values of order 10^0 where as pixel location components can have values of order of 10^2. Setting diagonal of R to $[1, 1, 100, 100]$ leads to scaling up of motion components and makes sure that $d(\underline{q}_j, \underline{w}_i)$ is sensitive to changes in motion components as well.

3. Calculate d_{max} and d_{min} from the distances set $D_{\underline{q}_j}$.

4. Adapt weight vectors according to

$$\Delta \underline{w}_i = \epsilon(t)\, h_{\lambda'}(m_{ij}(\underline{q}_j, W))\, F\,(\underline{q}_j - \underline{w}_i) \tag{3.5}$$

where $\epsilon(t)$ is the learning rate, $m_{ij}(\underline{q}_j, W)$ is the implicit ordering of W defined as $m_{ij} = \frac{d_{ij} - d_{min}}{d_{max} - d_{min}}$. d_{min}, d_{max} being the minimum and maximum distance between $\underline{q}_j$ and all $\underline{w}_i$s, $h_{\lambda'}(m_{ij}) = exp(-m_{ij}\lambda'(t))$, $\lambda'(t) = \lambda(t)/(N-1)$ and

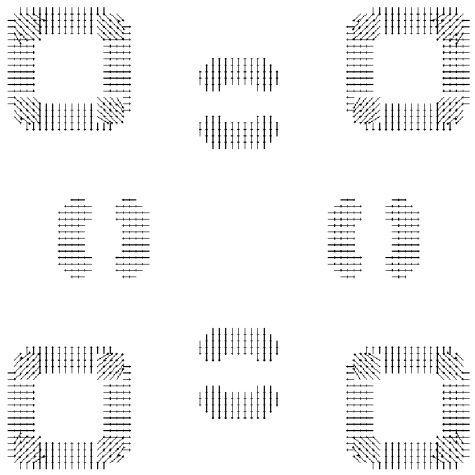


Figure 3.9: Initial Weights in $W = H_c \cup H_p$

where

$$F = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

only corrects the last 2 elements of $\underline{w}_i$ (i.e., the flow). Pixel locations are persistent. It should be noted that weights $\underline{w}_i$s closer to $\underline{q}_j$ both in terms of pixel locations and motion values get corrected the most. $\underline{w}_i$s corresponding to pixel locations far away from $\underline{q}_j$ hardly get corrected. Similarly, $\underline{w}_i$s corresponding to motion much different from $\underline{q}_j$ also hardly get corrected.

5. Repeat steps 2, 3 and 4 for each $\underline{q}_j \in Q$ to form one iteration.

6. Multiple iterations are required for flow interpolation.

$h_{\lambda'}(m_{ij})$ determines the neighborhood function around the winner weight. As

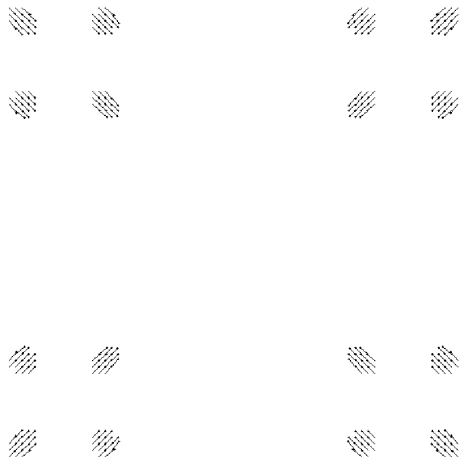


Figure 3.10: Training Vectors in H_c

training vectors are presented one after another, weights in the corresponding neighborhoods get their motion corrected leading to non-linear interpolation. In effect, vectors affected by aperture issue get their missing motion components filled in during the training. The outcome of interpolation using NGFlow for the eight squares case is shown in Figure 3.11.

Step 2 of NGFlow algorithm is computationally intense. For each training vector $\underline{q}_j$, the distances set $D_{\underline{q}_j}$ needs to be computed and it depends on the cardinality of set W which it typically large in motion estimation due its construction involving sets H_c and H_p. The weights set W grows with image size, leading to high computation time.

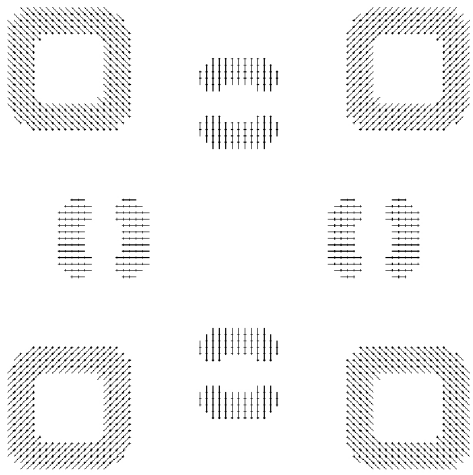


Figure 3.11: Weights after NGFlow Interpolation

3.3.2 SOMFlow

The SOM [Koh90] algorithm, as seen in Chapter 2, can be used instead of Neural Gas algorithm to significantly increase the speed of interpolation by leveraging the topology of the SOM network. But the neighborhood function used in SOM has to be modified to be effectively used for flow interpolation. SOM's neighborhood function is tied down to the topology of the network and is not a function of weight space as in NG. This leads to isotropic neighborhood and causes issues near occlusion boundaries in optical flow. We mitigate this by modifying the SOM neighborhood to be anisotropic in form. Similar to NGFlow, the set H_c is used as the training set and $W = H_c \cup H_p$ is used as the weight set. We choose to use 2D topology for SOM mirroring the image dimensions.

Fast Indexing: As topology dimensions are equal to 2-D image dimensions, we have one-to-one correspondence between SOM weights and image pixels. This mapping

helps speed up the training process by orders of magnitude. The usual training process involves finding the winner weight corresponding to the training vector under consideration. This step involves finding distance of the training vector from all the weight vectors and is followed by finding the weight vector ($\underline{w}_c$) with the minimum distance. This is computationally involved process and can be by-passed by utilizing the one-to-one mapping. Recall that weights are initialized using $W = H_c \cup H_p$ and each vector is 4-dimensional with the form (x, y, u, v). The training vectors from H_c are also of the same form. As all training vectors are also present in W, we already know the location $\underline{r}_c \in \mathbb{R}^2$ of winner weight. The first two components, that is, (x, y) are same for training vector and the winner weight. So the process of finding the winner weight simplifies to indexing in a 2D array and is independent of the cardinality of weights set W.

Anisotropic Neighborhood: The typical SOM [Koh90] weight update is given by

$$\Delta \underline{w}_i = \epsilon(t)\ h_{ic}(t)\ (\underline{q}_j - \underline{w}_i) \tag{3.6}$$

The neighborhood function $h_{ic}(t)$ used in the SOM algorithm can be specifically chosen to better handle motion at occlusion boundaries. Typically, two choices for $h_{ic}(t)$ occur [Koh90].

The simpler of them (step function) refers to a neighborhood set of array points around node c (Figure 3.12). Let the neighborhood be denoted $N_c(t)$, whereby $h_{ic}(t) = 1$ if $\underline{r}_i \in N_c(t)$ and $h_{ic}(t) = 0$ if $\underline{r}_i \notin N_c(t)$. The radius of $N_c(t)$ monotonically decreases with time. Another widely applied, smooth neighborhood kernel

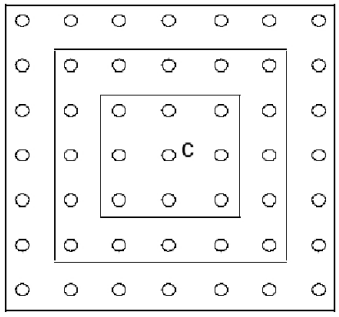


Figure 3.12: Neighborhood around the winner weight

has Gaussian form given by

$$h_{ic} = exp\left(-\frac{\| \underline{r}_c - \underline{r}_i \|^2}{2\sigma^2(t)}\right)$$

Both of the above specified neighborhoods are isotropic and can lead to unwanted averaging of motion at occlusion boundaries. Recall that at the occlusion boundaries, two different motion classes exist. Each motion class could have training vectors spatially close to one another. Weight vectors belonging to a motion class should be modified only by the training vectors of same motion class. Otherwise the weight vectors will be competitively trained for two different motions leading to incorrect interpolation.

We design an anisotropic neighborhood function $h_{\lambda' ic}$ to better handle interpolation near occlusion boundaries and the design is inspired by NG neighborhood. Figure 3.13a shows an example with training vectors from different motion classes around occlusion boundaries and Figure 3.13b shows an example of anisotropic neighborhood function around a winner weight vector (In our case, its also a training vector). The anisotropic neighborhood function should restrict influence of a training vector to

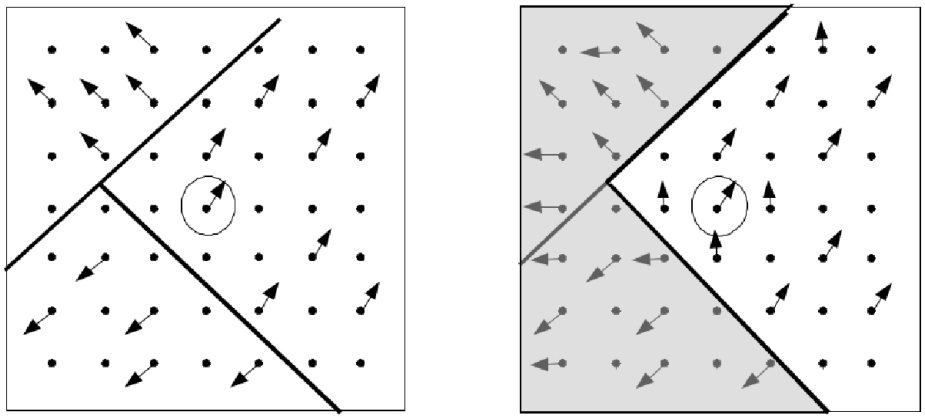


Figure 3.13: Anisotropic Neighborhood around the winner weight

only the weight vectors that belong to the same motion class. The weight update using training vector $\underline{q}_j \in H_c = \{\underline{q}_1, \underline{q}_2, \ldots, \underline{q}_M\}$ is defined by:

$$\Delta \underline{w}_i = \epsilon(t)\ F\ h_{\lambda' ic}(t)\ (\underline{q}_j - \underline{w}_i) \tag{3.7}$$

where $h_{\lambda' ic} = h_{\lambda'}(m_{ij}(\underline{q}_j, W_{nbr}))$. $W_{nbr} = \{\underline{w}_i \mid \parallel \underline{r}_c - \underline{r}_i \parallel < radius\}$ with r_c being the location (x, y) of winner unit $\underline{w}_c$ in SOM 2D topology. $m_{ij}(\underline{q}_j, W_{nbr})$ is the implicit ordering of W_{nbr} defined as $m_{ij} = \frac{d_{ij} - d_{min}}{d_{max} - d_{min}}$. d_{ij} is as given in (Equation 3.8). d_{min}, d_{max} being the minimum and maximum distance between $\underline{q}_j$ and $\underline{w}_i$s, $h_{\lambda'}(m_{ij}) = exp(-m_{ij}\lambda'(t))$, $\lambda'(t) = \lambda(t)/(N_{nbr} - 1)$. Conceptually, the $\underline{w}_i \in W_{nbr}$ around $\underline{q}_j$ are weighted according to how close they are in the weight space (4D of the form (x, y, u, v)). $m_{ij}(\underline{q}_j, W_{nbr})$ orders the $\underline{w}_i \in W_{nbr}$ with m_{ij} smaller for $\underline{w}_i$'s closer to $\underline{q}_j$ in 4D weight space. $h_{\lambda'}(m_{ij})$ is the weight associated with $\underline{w}_i$ and is high for smaller m_{ij} that is when $\underline{w}_i$ is closer to $\underline{q}_j$ in 4D weight space. In terms of flow, it means that, the motion vectors (weights) similar to training motion vectors get higher weight values than other weights in W_{nbr}. Thus, anisotropic neighborhood is formed.

Residual Distance Measure: Another novelty of our approach is the use of a non-obvious distance measure. A typical weighted distance metric is given by

$$d_{ij} = d(\underline{q}_j, \underline{w}_i) = ||\underline{q}_j - \underline{w}_i||_R = \sqrt{(\underline{q}_j - \underline{w}_i)^T \; R \; (\underline{q}_j - \underline{w}_i)} \tag{3.8}$$

where R is the diagonal matrix consisting of estimated inverse variance values of the vector components.

In our experience, this distance measure (Equation 3.8) does not result in required anisotropic behavior. We explore two alternative distance measures that involve residue value of the OFCE (Equation 1.3). First OFCE residue-based distance measure uses

$$d_{ij} = d(\underline{q}_j, \underline{w}_i) = (I_{xi}u_j + I_{yi}v_j + I_{ti})^2 \tag{3.9}$$

where, I_{xi}, I_{yi} and I_{ti} are the gradients corresponding to $\underline{w}_i$ at pixel location r_i. u_j and v_j are the velocity components of training vector $\underline{q}_j$. d_{ij}'s are then used in m_{ij} computation as shown earlier. Second alternative does way with m_{ij} computation and instead uses

$$d_{ij} = \sum_{k=1}^{n \times n} |I_{xk}u_j + I_{yk}v_j + I_{tk}| \tag{3.10}$$

where, I_{xk}, I_{yk} and I_{tk} are the gradients corresponding to $\underline{w}_k$ in small $n \times n$ window around $\underline{w}_i$. u_j and v_j are the velocity components of training vector $\underline{q}_j$. $h_{\lambda' ic}$ is now defined as $h_{\lambda' ic} = exp(d_{ij}(\underline{q}_j, W_{nbr})/\alpha)$. For 3×3 window around $\underline{w}_i$, it was empirically determined that α with value 250 works well. This second alternative (Equation 3.10) better defines anisotropic neighborhood. Moreover, the memory storage requirement for d_{ij}'s is also eliminated which helps in GPU implementation as will be seen later.

Implementation with 2 Dimensional Vectors: For SOM the vector dimensions can be reduced to 2 by removing the pixel position information from each vector. As the SOM units are in a 2D topology mirroring the image dimensions, the location of unit in the 2D topology gives the pixel location. Storing pixel location (x, y) is redundant and can be obtained by looking at the index of the SOM unit. For the following algorithm steps, the vectors in H_c and H_p are assumed to be 2 dimensional with just (u, v) components.

The algorithm we hereafter refer to as SOMFlow (to distinguish it from Kohonen's SOM) is as follows:

1. Initialize unit(neuron) weights as $W = H_c \cup H_p$ and use set H_c as the training set Q.

2. For input $\underline{q}_j$ select $\underline{w}_c$ as the winner weight by indexing into SOM 2D topology using $\underline{r}_c \in \mathbb{R}^2$ as index. $\underline{r}_c$ is the location (x, y) of training vector $\underline{q}_i$.

3. Update weights of winning unit and weights in its neighborhood. The updating strategy with anisotropic neighborhood function is given by

$$\Delta \underline{w}_i = \epsilon(t)\ h_{\lambda' ic}(t)\ (\underline{q}_j - \underline{w}_i) \tag{3.11}$$

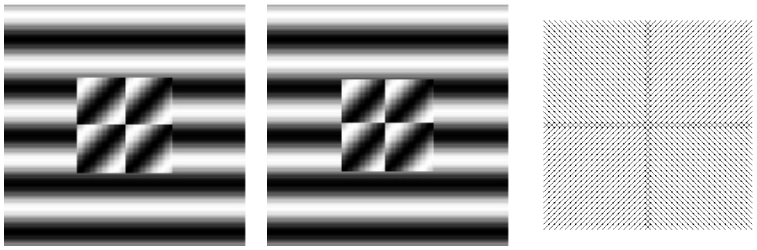
   Compared with Equation 3.7, note the absence of matrix F in the update equation. Use residual distance measure (Equation 3.10) for computing $h_{\lambda' ic}(t)$. Use $h_{\lambda' ij} = exp(-d_{ij}/\alpha)$.

4. Repeat steps 2 and 3 for each $\underline{q}_j \in Q$ to form one iteration.

5. Multiple iterations are required for flow interpolation.

As training vectors are presented one after another, weights in the corresponding neighborhoods get their motion corrected leading to non-linear interpolation. Similar to as in NGFlow, the vectors affected by aperture issue get their missing motion components filled in during the training. The outcome of interpolation using SOMFlow for the eight squares case is similar to the output of NGFlow (Figure 3.11).

Example - Isotropic Vs Anisotropic: Consider the image sequence given in Figure 3.14. The images were generated using code developed by Kamitani [Kam]. With the four squares moving diagonally towards the center of the image, there are

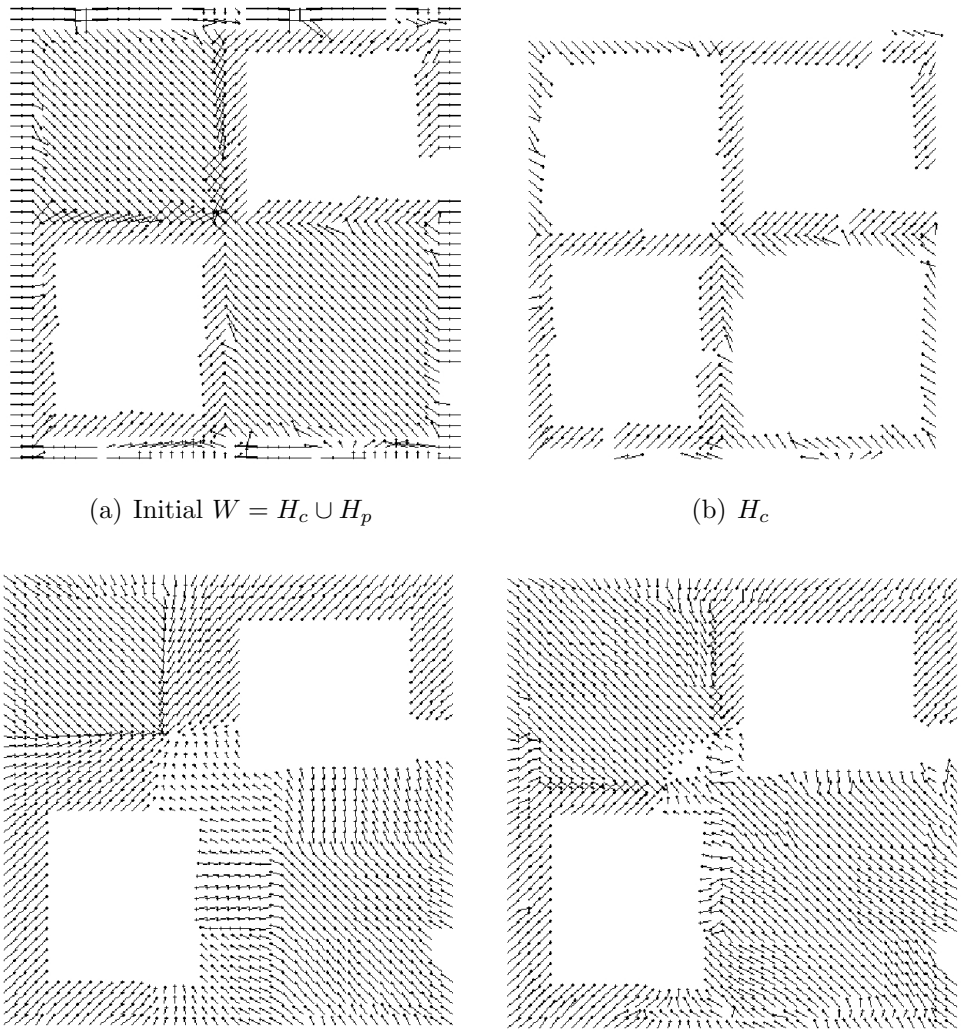


(a) 1^{st} frame (b) 2^{nd} frame (c) True Motion - 4 Squares only (Enlarged)

Figure 3.14: Sine wave sequence - Occlusion Example

4 motion classes and 4 occlusion boundaries. The use of isotropic neighborhood functions, as specified in (Eq 3.6), leads to incorrect flow vectors at the occlusion boundaries (Figure 3.15(c)).

The SOMFlow update using (Eq 3.11) gives flow with motion discontinuities preserved at occlusion boundaries as seen in Figure 3.15(d). Thus SOMFlow helps solve the generalized aperture problem as it tries to overcome aperture problem while preserving motion discontinuities.



(a) Initial $W = H_c \cup H_p$

(b) H_c

(c) Final W with isotropic neighborhood

(d) Final W with anisotropic neighborhood

Figure 3.15: Sinewave sequence - SOMFlow correction

3.3.3 BatchSOMFlow

The execution time for SOMFlow can be reduced by parallel processing. We use NVIDIA CUDA GPUs as a hardware platform for parallel implementation. GPUs are well suited for data-parallel computations as they have multiple computing cores driven by very high bandwidth. To be data-parallel, each weight vector should be independently and parallely processable. The SOM algorithm as-is (online version) is not data-parallel, but the batch version of SOM is data-parallel. Instead of looking at neighborhood of weight vectors around each training vector, the batch version [CHHV06] looks at neighborhood of training vectors around each weight vector. *The cost function optimized by batch-SOM is equivalent to the function optimized by online-SOM* [Che97].

With initial weights as $W = H_c \cup H_p$ and the training set $Q = H_c$, the weight update for BatchSOMFlow (with isotropic neighborhood) is defined by:

$$\underline{w}_i = \frac{\sum_{j=1}^{M} h_{ij} \underline{q}_j}{\sum_{j=1}^{M} h_{ij}} \tag{3.12}$$

where, $h_{ij} = 1$ if $\underline{q}_j \in Q_{nbr}$, otherwise $h_{ij} = 0$. $Q_{nbr} = \{\underline{q}_j \mid \parallel \underline{r}_j - \underline{r}_i \parallel < radius\}$ where $\underline{r}_j \in \mathbb{R}^2$ and $\underline{r}_i \in \mathbb{R}^2$ are the location vectors of training vector $\underline{q}_j$ and weight vector $\underline{w}_i$ respectively.

To handle occlusion, we use anisotropic neighborhood function $h_{\lambda' ij}$ and then the weight update for BatchSOMFlow is given as:

$$\underline{w}_i = \frac{\sum_{j=1}^{M} h_{\lambda' ij} \underline{q}_j}{\sum_{j=1}^{M} h_{\lambda' ij}} \tag{3.13}$$

where, $h_{\lambda' ij} = exp(d_{ij}(Q_{nbr}, \underline{w}_i)/\alpha)$. The distance d_{ij} uses the residual distance (Eq 3.10) for better anisotropic form.

The above update function can be understood if we look at location of a weight vector as the center of a neighborhood. At occlusion boundary, the neighborhood of a weight has training vectors with different motion values. We apply residual-based distance measure to training vectors around the weight to get an anisotropic neighborhood function. This weights the training vectors according to their distance (OFCE residues) from the weight vector. The idea is to train the weight vector by only the training vectors that satisfy OFCE (Equation 1.3) using spatio-temporal gradients at the weight vector location. Training vectors that lead to large OFCE residue at weight vector location should not affect the weight vector, even if they are spatially close.

The above specified update (Eq 3.13) is data-parallel (or rather datum-parallel) as each weight can be independently and parallely updated. Moreover, *only one iteration is required* as can be seen (in Section 3.4) by analyzing the effect of incorporating $W = H_c \cup H_p$ and $Q = H_c$ into SOM cost function given by [Hes01]. Also, the later section (Section 3.5) shows that, with this particular use of weights and training set, we achieve interpolation of flow vectors primarily due to the regularization property of SOM.

3.4 BatchSOMFlow Quantization Error

Given a set of input vectors Q and weights W, let p_{ij} denote the probability that the input $\underline{q}_j$ is assigned to node with weight $\underline{w}_i$. Even if we assign input $\underline{q}_j$ to weight $\underline{w}_i$, there is a confusion probability h_{il} that the input $\underline{q}_j$ is instead quantized by the weight vector $\underline{w}_l$. The quantization error [Hes01], as was seen in Chapter 2, is

given as:

$$F_{quant}(P,W) = \sum_{j=1}^{M}\sum_{i=1}^{N} p_{ij} \sum_{l=1}^{N} h_{il} D(\underline{q}_j, \underline{w}_l) \tag{3.14}$$

where $M = |Q|$ and $N = |W|$. h_{il} corresponds to the lateral interaction strength and typically $h_{il} = exp[-d_{il}/2\sigma^2]$ where d_{il} refers to the node distance on a two dimensional grid between $\underline{w}_i$ and $\underline{w}_l$. As seen earlier, anisotropic lateral interaction can be designed using $h_{\lambda' il}(d_{il}) = exp(-d_{il}/\alpha)$ with the distance d_{il} based upon the residual distance (Eq 3.10). But for the following analysis we do not use anisotropic interaction strengths so that we can keep the equations similar to those in [Hes01]. As $h_{\lambda' il} >= 0$, we can always replace h_{il} with $h_{\lambda' il}$.

With $W = H_c \cup H_p$ and $Q = H_c$, the term $\sum_{i=1}^{N}$ in Equation 3.14 can be written in two parts. Let the cardinality of set H_p be K. As $H_c \cap H_p = \emptyset$, $N = M + K$. Therefore $\sum_{i=1}^{N} = \sum_{i=1}^{M+K}$ leads to

$$F_{quant}(P,W) = \sum_{j=1}^{M}\sum_{c=1}^{M} p_{cj} \sum_{l=1}^{N} h_{cl} D(\underline{q}_j, \underline{w}_l) + \sum_{j=1}^{M}\sum_{k=1}^{K} p_{kj} \sum_{l=1}^{N} h_{kl} D(\underline{q}_j, \underline{w}_l) \tag{3.15}$$

Recall that p_{cj} denotes the probability that the input $\underline{q}_j$ is assigned to node with weight $\underline{w}_c \in H_c$. As $H_c \subset W$ by design, $\underline{q}_j \in W$ as one of the $\underline{w}_c$s and therefore $p_{cj} = 1$. As there is only one $w \in W$ that satisfies $\underline{w}_c = \underline{q}_j$ and $p_{cj} = 1$, it implies that the probability that the input $\underline{q}_j$ is assigned to any other node is 0.

p_{kj} denotes the probability that the input $\underline{q}_j$ is assigned to node with weight $\underline{w}_k \in H_p$. $\underline{q}_j$'s will never be assigned to any weight in H_p as they always find a match in H_c with probability equal to 1. Hence all the p_{kj} are zero and the second summation term evaluates to zero. Therefore we have

$$F_{quant}(P,W) = \sum_{j=1}^{M}\sum_{c=1}^{M} p_{cj} \sum_{l=1}^{N} h_{cl} D(\underline{q}_j, \underline{w}_l) \tag{3.16}$$

It can be seen that $p_{cj} = 1$ when $\underline{w}_c = \underline{q}_j$ and $p_{cj} = 0$ for $\underline{w}_c \neq \underline{q}_j$. This further simplifies the quantization term to

$$F_{quant}(W) = \sum_{j=1}^{M} \sum_{l=1}^{N} h_{cl} D(\underline{q}_j, \underline{w}_l) \tag{3.17}$$

where index c of h_{cl} corresponds to $\underline{w}_c$ such that $\underline{w}_c = \underline{q}_j$ which can be written as

$$F_{quant}(W) = \sum_{j=1}^{M} \sum_{l=1}^{N} h_{jl} D(\underline{q}_j, \underline{w}_l) \tag{3.18}$$

As seen in Chapter 2, Heskes EM algorithm for SOM includes the maximization step

$$\underline{w}_l(P) = \frac{\sum_{j=1}^{M} \sum_{i=1}^{N} p_{ij} h_{il} \underline{q}_j}{\sum_{j=1}^{M} \sum_{i=1}^{N} p_{ij} h_{il}}$$

which simplifies for $W = H_c \cup H_p$ and $Q = H_c$ as

$$\underline{w}_i = \frac{\sum_{j=1}^{M} h_{ji} \underline{q}_j}{\sum_{j=1}^{M} h_{ji}} \tag{3.19}$$

Moreover, as probability values are known and do not change, we do not require the expectation step. We get the weights in one step without iterations.

3.5 Regularization due to BatchSOMFlow

As seen in Chapter 2, [Hes01] shows SOM as mixture model plus additional regularization. The energy functional being optimized in this case is given as

$$E(W) = -L(W) + E_{regular}(W)$$
$$L(W) = \sum_{j=1}^{M} log\ P(\underline{q}_j|W)$$
$$E_{regular}(W) = -\sum_{j=1}^{M} log \sum_{i=1}^{N} g_i\ e^{-\beta V_i(W)}$$

where $L(W)$ is the optimization criterion corresponding to a maximum likelihood procedure for mixture of Gaussians when no lateral interactions are considered, similar to that of Vector Quantization. In Vector Quantization case, the mixture of Gaussians is given as $P(\underline{q}|W) = \sum_{i=1}^{N} g_i\ G(\underline{q}|\underline{w}_i)$. Whereas for SOM, the mixture of Gaussians is given as

$$P(\underline{x}|W) = \sum_{i=1}^{N} \tilde{g}_i(W) G(\underline{x}|\underline{\tilde{w}}_i)$$

with $\tilde{g}_i(W) \equiv \frac{g_i\ e^{-\beta V_i(W)}}{\sum_{l=1}^{N} g_l\ e^{-\beta V_l(W)}}$.

Minimizing $E(W)$ minimizes $E_{regular}(W)$ which in turn minimizes the variance term $V_i(W)$ and this leads to regularization.

With $W = H_c \cup H_p$ and $Q = H_c$, the probability of finding $\underline{q} \in H_c$ given W is 1. $P(q|W)$ evaluates to 1 and therefore $L(W) = 0$ and is independent of W. Therefore we have just regularization term left.

$$E(W) = E_{regular}(W) \tag{3.20}$$

The expectation-maximization (EM) algorithm developed by [Hes01] for SOM tries

to optimize

$$E(W) = -\sum_{j=1}^{M} log \sum_{i=1}^{N} g_i exp(-\beta \sum_{l=1}^{N} h_{il} D(\underline{q}_j, \underline{w}_l))$$

To show SOM as mixture model with added regularization, [Hes01] splits the term $\sum_{l=1}^{N} h_{il} D(\underline{q}_j, \underline{w}_l)$ into bias term $D(\underline{q}_j, \underline{\tilde{w}}_i)$ and a variance term $V_i(W) = \sum_{l=1}^{N} h_{il} D(\underline{\tilde{w}}_i, \underline{w}_l)$ with $\underline{\tilde{w}}_i = \sum_{l=1}^{N} h_{il} \underline{w}_l$.

Simplifying for $W = H_c \cup H_p$ and $Q = H_c$, we see that $\underline{\tilde{w}}_i = \underline{q}_j = \underline{w}_c$ with $\underline{w}_c \in H_c$. Therefore, $D(\underline{q}_j, \underline{\tilde{w}}_i) = D(\underline{q}_j, \underline{q}_j) = 0$. $V_i(W)$ simplifies to $V_c(W) = \sum_{l=1}^{N} h_{cl} D(\underline{w}_c, \underline{w}_l)$ and the regularization term becomes

$$E_{regular}(W) = -\sum_{j=1}^{M} log \sum_{c=1}^{N} g_c e^{-\beta V_c(W)} \tag{3.21}$$

As the algorithm minimizes $E_{regular}(W)$, the variance $V_c(W)$ is minimized. Minimizing $V_c(W)$ leads to training of $\underline{w}_l$ towards $\underline{w}_c$. As $\underline{w}_c \in H_c$, we see that motion vectors $\underline{w}_l$ get their missing motion components from $\underline{w}_c$ and thus SOMFlow regularizes the motion estimate solution.

Chapter 4

Pyramid BatchSOMFlow - pyrSOMFlow

This chapter provides the end-to-end details of the optical flow estimator. We refer to the estimator as pyrSOMFlow. Its quantitative evaluation is also provided in this chapter. All the steps required to estimate optical flow field (given two frames in a image sequence) are listed. For the self-organization part, the BatchSOMFlow algorithm described in Chapter 3 is used. Moreover, to handle large motion, the algorithm is embedded in a multi-resolution framework. The OFCE (Eq 1.3) is based on spatial and temporal gradients of image intensity that are computed using image intensity values within a small neighborhood. If the motion magnitude is larger than this neighborhood, then the motion model that we use is no longer valid. That is, OFCE assumes differential motion. If the resolution of images is reduced sufficiently enough, then use of the differential motion model (OFCE) becomes valid.

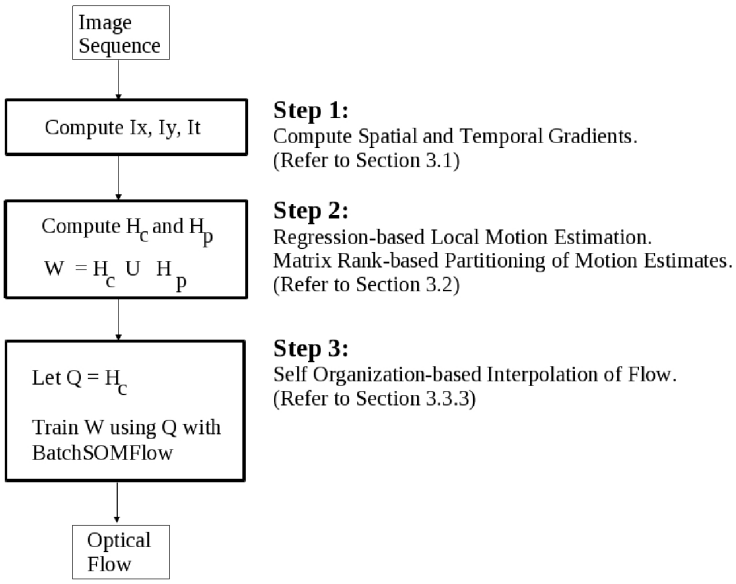


Figure 4.1: Flow Chart - Estimating Small Motion

4.1 Small Motion with BatchSOMFlow

At each resolution or level of the pyramid, the small motion model developed in Chapter 3 is applied. Before going into pyramidal processing we gather here in one place all the required steps from Chapter 3. Figure 4.1 shows the steps required to get the optical flow if the motion is small enough for differential model (Eq 1.3) to be valid.

The motion estimation algorithm can be split into following three consecutive stages.

1. *Gradient Estimation:* We use the approximate differentiation as given in [HS81].

2. *Local Motion Estimation:* We use the efficient 2-Pass evaluation (Section 3.2.2) to compute local motion estimates using the following weighted Least Squares formulation.

$$E = \sum_{3\times 3} W^2 \left(I_x u + I_y v + I_t\right)^2$$

Sub-systems of the following form are formed during the efficient 2-Pass evaluation.

$$\begin{bmatrix} I_{x1} & I_{y1} \\ I_{x2} & I_{y2} \\ I_{x3} & I_{y3} \\ I_{x4} & I_{y4} \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix} = - \begin{bmatrix} I_{t1} \\ I_{t2} \\ I_{t3} \\ I_{t4} \end{bmatrix}$$

$$D \qquad \underline{v} \quad = - \quad \underline{f}_t$$

and are solved using Moore-Penrose inverse to get regression-based motion estimate

$$\underline{v} = -(D)^{\dagger} \underline{f}_t$$

Matrix Rank-based Partitioning: If the rank of matrix D is either 0 or 1 (Aperture Problem) or if the residual is high (violation of spatial coherence), we put it in set H_p, otherwise, the motion estimate is put in set H_c. Storing pixel location (x, y) is redundant and can be obtained by looking at the index of the SOM unit. The vectors in H_c and H_p are assumed to be 2 dimensional with just (u, v) components.

3. *Self Organization-based Interpolation of Flow:* We initialize the SOM weights using $W = H_c \cup H_p$ and train the weights with $Q = H_c$. BatchSOMFlow is used to train the weights and the weight update equation is given as

$$\underline{w}_i = \frac{\sum_{j=1}^{M} h_{\lambda' ij} \underline{q}_j}{\sum_{j=1}^{M} h_{\lambda' ij}}$$

where, $h_{\lambda' ij} = exp(d_{ij}(Q_{nbr}, \underline{w}_i)/\alpha)$. Q_{nbr} typically uses a window size of 15×15.

The distance d_{ij} uses the residual distance given by

$$d_{ij} = \sum_{k=1}^{n \times n} |I_{xk} u_j + I_{yk} v_j + I_{tk}|$$

where, I_{xk}, I_{yk} and I_{tk} are the gradients corresponding to $\underline{w}_k$ in small $n \times n$ window around $\underline{w}_i$. u_j and v_j are the velocity components of training vector $\underline{q}_j$. For 3×3 window around $\underline{w}_i$, α with value 250 works well.

4.2 Multiresolution or Pyramidal Approach

The pyramidal approach typically involves construction of pyramids of spatially filtered (low-pass) and sub-sampled images. The differential motion model must be valid at the coarsest-resolution and that determines the required number of levels of the pyramid. Motion estimation starts at the coarsest resolution and moves through remaining pyramid levels. Motion computed at a coarser resolution is used to estimate motion at the next finer resolution in the pyramid. To make sure that the OFCEs are valid at the finer resolution, the motion estimate from coarser resolution is scaled and used to warp one of the finer resolution images (say image1) towards another (say image0). The remaining (delta) motion between the warped image and image0 is small and can be computed. The process is repeated till the highest resolution in the pyramid. The above specified algorithm is used to compute delta motion as the small motion constraint is satisfied. The delta motion computed at each level is accumulated and scaled to generate the final motion estimate. Median filtering of delta motion can optionally be done to suppress noise. Median filter does not disturb the flow discontinuities.

Figure 4.2 shows example of pyramidal processing. The dotted lines show the

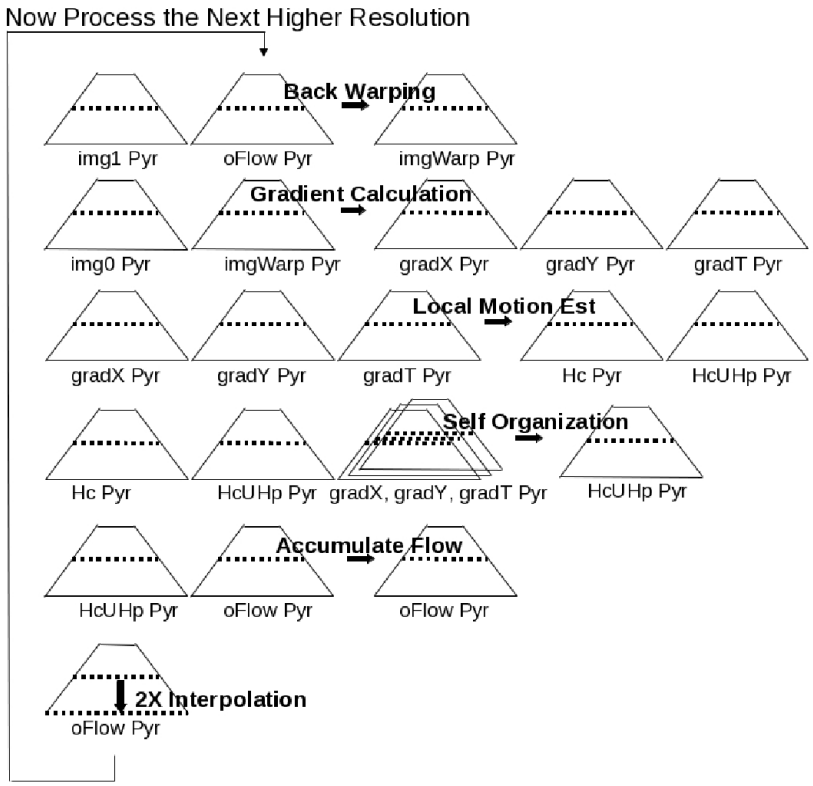


Figure 4.2: Multiresolution or Pyramidal Approach

pyramid level that is being processed. The pyramids on the left of the dark arrow are inputs to the processing step (for example, **Back Warping**) and the pyramids on the right are the output of the processing step. Two steps, namely, **Back Warping** and **2X Interpolation** use bicubic convolution interpolation [Rob81]. Consider two images, img0 and img1. The processing starts with construction of data pyramids img0Pyr and img1Pyr by low-pass filtering and sub-sampling images img0 and img1 respectively. Other pyramids shown in Figure 4.2 are allocated and initialized with zero values. Following description uses suffix (k) along with the pyramid name to identify the k^{th} level of the pyramid. For example, oFlowPyr(k) refers to the k^{th} level of pyramid oFlowPyr, oFlowPyr(0) refers to the base level and oFlowPyr(L-1) refers to the top or coarsest-resolution of the pyramid with L number of levels. The first

step (**Back Warping**) requires use of optical flow information from oFlowPyr(k) to backwarp img1Pyr(k). The resultant backwarped image is stored at imgWarpPyr(k). Note that when processing starts at the coarsest-resolution (level L-1), oFlowPyr(L-1) has zero values. Therefore at level L-1, imgWarpPyr(L-1) is same as img1Pyr(L-1). At other levels, this step leads to cancellation of coarse motion (accumulated so far) between img0Pyr(k) and img1Pyr(k). The remaining (delta) motion that may remain can be computed using small motion model (Eq 1.3) with img0Pyr(k) and imgWarpPyr(k). The second step (**Gradient calculation**) computes the spatial gradients (gradXPyr(k), gradYPyr(k)) and temporal gradients (gradTPyr(k)) using img0Pyr(k) and imgWarpPyr(k). The third step (**Local Motion Est**) uses regression and matrix rank to generate HcPyr(k) and HcUHpPyr(k). The (**Self Organization**) in the fourth step then trains HcUHpPyr(k) to get the corrected motion at level k. The fifth step (**Accumulate Flow**) is for keeping track of the flows estimated at different levels. Adding the flow HcUHpPyr(k) to oFlowPyr(k) gives the optical flow estimate for the k^{th} level of pyramid. If k is equal to zero, that means that the base or highest resolution is reached and oFlowPyr(0) is the optical flow estimate from the technique. The sixth step (**2X Interpolation**) is applicable only if k is greater than zero. In that case, oFlowPyr(k) is interpolated and scaled up by 2 and stored in higher resolution oFlowPyr(k-1). The steps are repeated for level k-1 and so on.

4.3 Predictor-Corrector Approach

Predictor-Corrector Approach [SW82] can be used to iteratively refine motion estimates. Figure 4.3 shows the use of Predictor-Corrector Approach in use in the multi-resolution framework. Adding the flow HcUHpPyr(k) to oFlowPyr(k) gives the optical flow estimate for the k^{th} level of pyramid. This estimate/prediction can be

used to refine/correct the estimates by going back to warping stage but staying at the same resolution. For Self Organization-based motion estimation, the Predictor-Corrector Approach works best at the highest resolution.

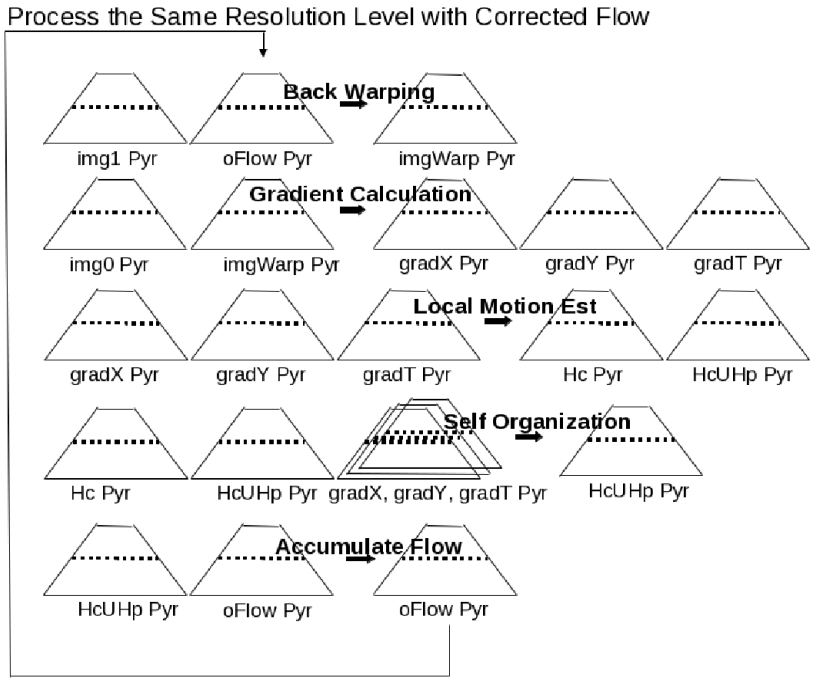


Figure 4.3: Predictor-Corrector Approach

4.4 Evaluation using Middlebury Database

The Middlebury database is a set of test cases that is used to benchmark optical flow estimators. Quantitative evaluation of various optical flow estimators can be found at Middlebury website (http://vision.middlebury.edu/flow). Figure 4.4 shows the Average Angular Error (AAE) and Average Endpoint Error(AEE) for the testcases from Middlebury database. True flow is known for the eight testcases and error metrics can be computed. The charts show the error values with varying SOM neighborhood extent. Due to the use of anisotropic distance measure, the errors

values do not change drastically even with increasing SOM neighborhood windows. The AAE and AEE values from Figure 4.4 suggest that pyrSomFlow estimator gives reasonable results and is comparable to other techniques listed on Middlebury website.

AAE	Dimetrodon	Grove2	Grove3	Hydrangea	RubberWhale	Urban2	Urban3	Venus
15x15	3.55	3.17	8.06	2.66	5.89	4.94	7.53	5.8
17x17	3.49	3.19	8.06	2.67	5.92	5.02	7.61	5.73
19x19	3.44	3.28	8.12	2.68	6.06	5.13	7.63	5.69
21x21	3.43	3.38	8.32	2.69	6.11	5.25	7.77	5.81
23x23	3.42	3.46	8.33	2.71	6.29	5.41	7.82	6.01

AEE	Dimetrodon	Grove2	Grove3	Hydrangea	RubberWhale	Urban2	Urban3	Venus
15x15	0.06	0.06	0.14	0.05	0.1	0.09	0.13	0.1
17x17	0.06	0.06	0.14	0.05	0.1	0.09	0.13	0.1
19x19	0.06	0.06	0.14	0.05	0.11	0.09	0.13	0.1
21x21	0.06	0.06	0.15	0.05	0.11	0.09	0.14	0.1
23x23	0.06	0.06	0.15	0.05	0.11	0.09	0.14	0.1

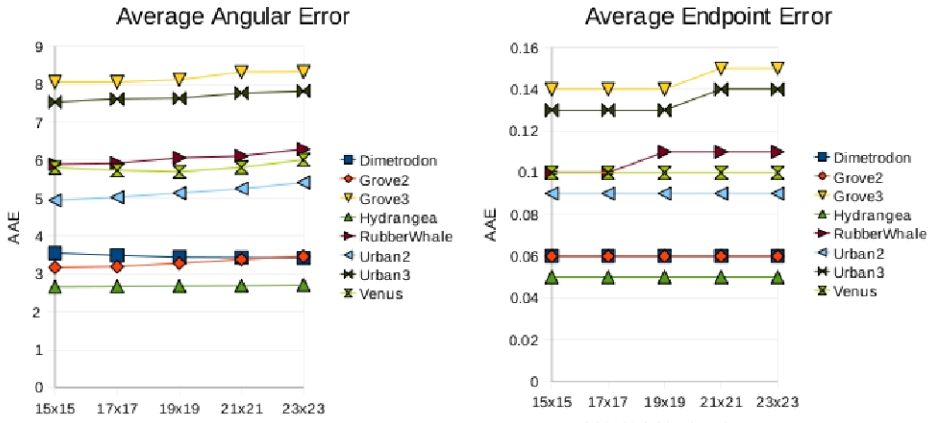
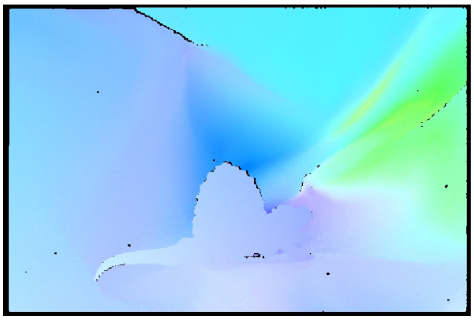


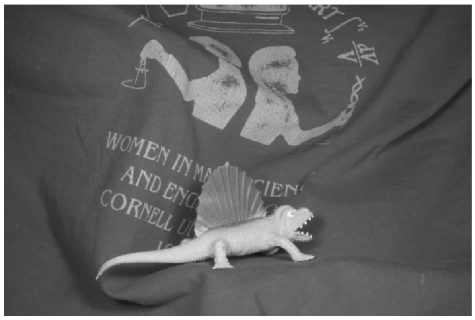
Figure 4.4: AAE and AEE

The Figures 4.5 to 4.12, show the true flow and the estimated flow for the Middlebury image sequences.

Comparison with other Optical Flow Estimators: Figure 4.13 shows the AAE metric comparison of pyrSOMFlow with other three optical flow estimators. The implementation of these other estimators can be obtained at (http://www.cs.brown.edu/ dqsun/research/software.html). Classic+NL [DSB10] is one of the leading regularization-based estimators with approximate computation time of 16 minutes [DSB10] for "Ur-

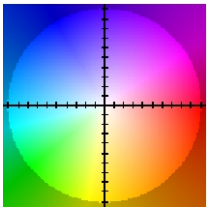


(a) True Flow. Max Motion: 4.6700; u = -4.329 .. -0.268; v = -2.647 .. 2.232



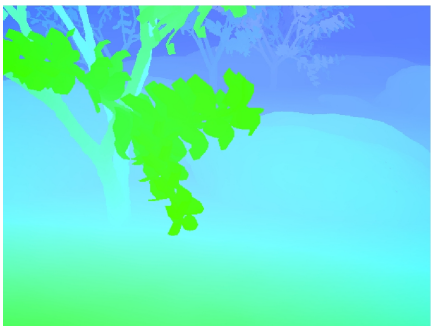
(b) Frame 10 (584 × 388)

(c) pyrSOMFlow. Max Motion: 4.6860; u = -4.268 .. -0.287; v = -2.001 .. 2.280

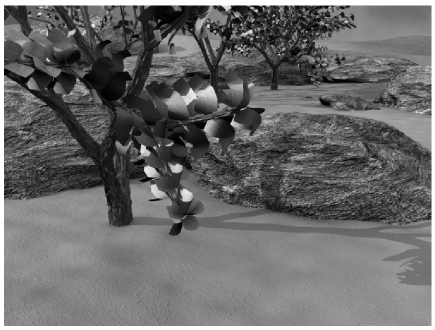


(d) Flow Colors

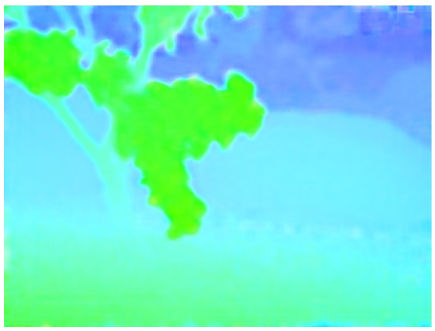
Figure 4.5: Dimetrodon Sequence. AAE = 3.55 and AEE = 0.06



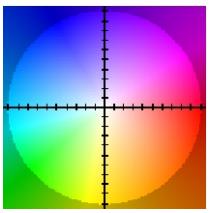
(a) True Flow. Max Motion: 5.0313; u = -3.313 .. -1.999; v = -2.261 .. 4.012



(b) Frame 10 (640 × 480)



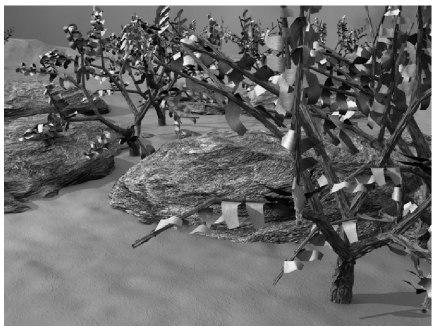
(c) pyrSOMFlow. Max Motion: 5.1651; u = -4.836 .. -0.932; v = -2.629 .. 4.164



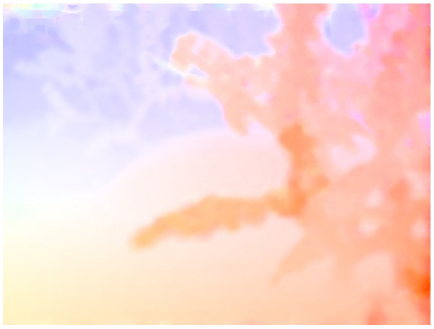
(d) Flow Colors

Figure 4.6: Grove2 Sequence. AAE = 3.17 and AEE = 0.06

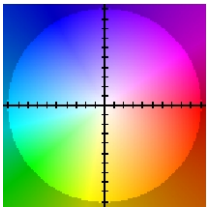
(a) True Flow. Max Motion: 18.6087; u = -2.320 .. 14.335; v = -4.091 .. 11.893



(b) Frame 10 (640 × 480)

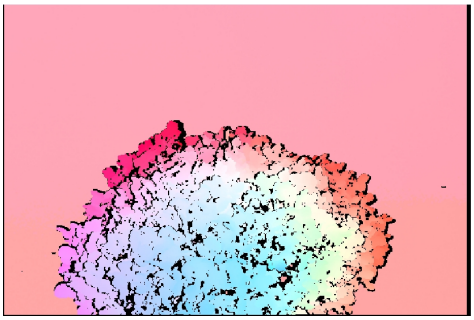


(c) pyrSOMFlow. Max Motion: 19.8309; u = -3.687 .. 16.961; v = -5.228 .. 11.523



(d) Flow Colors

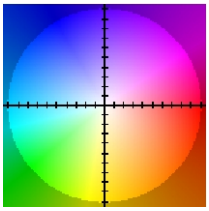
Figure 4.7: Grove3 Sequence. AAE = 8.06 and AEE = 0.14



(a) True Flow. Max Motion: 11.1214; u = -7.021 .. 11.014; v = -3.199 .. 2.351

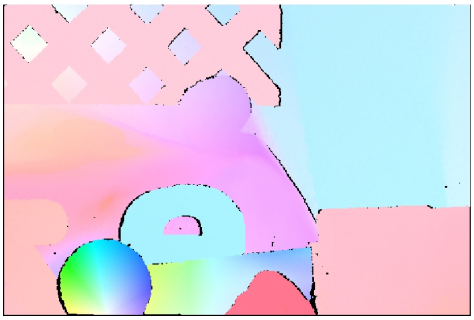
(b) Frame 10 (584 × 388)

(c) pyrSOMFlow. Max Motion: 10.8979; u = -6.410 .. 10.835; v = -3.457 .. 2.282



(d) Flow Colors

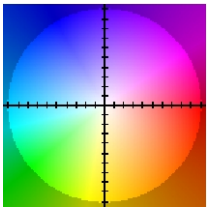
Figure 4.8: Hydrangea Sequence. AAE = 2.66 and AEE = 0.05



(a) True Flow. Max Motion: 4.6157; u = -4.576 .. 2.575; v = -2.575 .. 2.919

(b) Frame 10 (584 × 388)

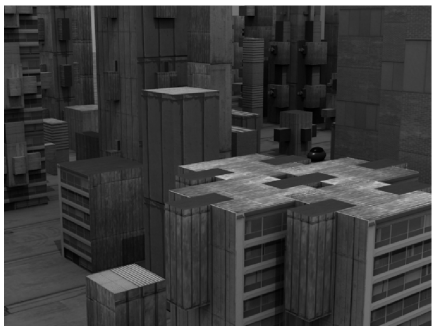
(c) pyrSOMFlow. Max Motion: 4.3020; u = -4.302 .. 2.698; v = -2.310 .. 2.471



(d) Flow Colors

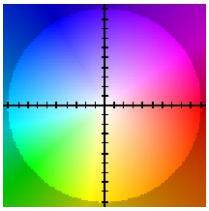
Figure 4.9: RubberWhale Sequence. AAE = 5.89 and AEE = 0.1

(a) True Flow. Max Motion: 22.1906; u = -21.322 .. 6.261; v = -0.872 .. 8.512



(b) Frame 10 (640 × 480)

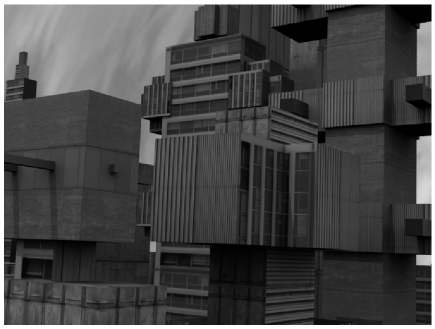
(c) pyrSOMFlow. Max Motion: 22.3448; u = -21.356 .. 4.492; v = -1.453 .. 12.630



(d) Flow Colors

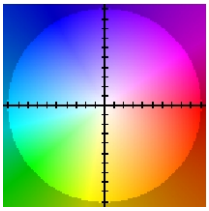
Figure 4.10: Urban2 Sequence. AAE = 4.94 and AEE = 0.09

(a) True Flow. Max Motion: 17.6161; u = -4.181 .. 2.351; v = -3.335 .. 17.334



(b) Frame 10 (640 × 480)

(c) pyrSOMFlow. Max Motion: 23.9820; u = -5.608 .. 5.167; v = -5.180 .. 23.738

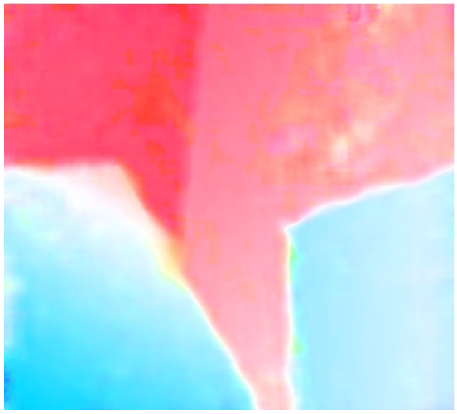


(d) Flow Colors

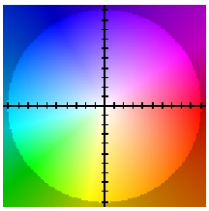
Figure 4.11: Urban3 Sequence. AAE = 7.53 and AEE = 0.13

(a) True Flow. Max Motion: 9.3750; u = -9.375 .. 7.000; v = 0.000 .. 0.000

(b) Frame 10 (420 × 380)



(c) pyrSOMFlow. Max Motion: 8.8874; u = -8.887 .. 7.151; v = -3.262 .. 2.938



(d) Flow Colors

Figure 4.12: Venus Sequence. AAE = 5.80 and AEE = 0.1

ban" test sequence on Matlab. BA refers to Black and Anandan's [BA96] flow estimator while HS refers to Horn and Schunck's [HS81] flow estimator. pyrSomFlow is tuned more for computational efficiency than for keeping error metrics low. The next Chapter talks about the parallelization of pyrSomFlow and the real-time possibilities on GPU.

AAE	Dimetrodon	Grove2	Grove3	Hydrangea	RubberWhale	Urban2	Urban3	Venus
Classic+NL	2.28	1.41	4.93	1.82	2.4	2.03	2.03	3.29
pyrSOMFlow	3.55	3.17	8.06	2.66	5.89	4.94	7.53	5.8
BA	3.38	2.73	6.95	2.64	5.92	4.25	11.08	7.75
HS	3.69	4	8.4	2.99	7.27	6.21	16.32	9.4

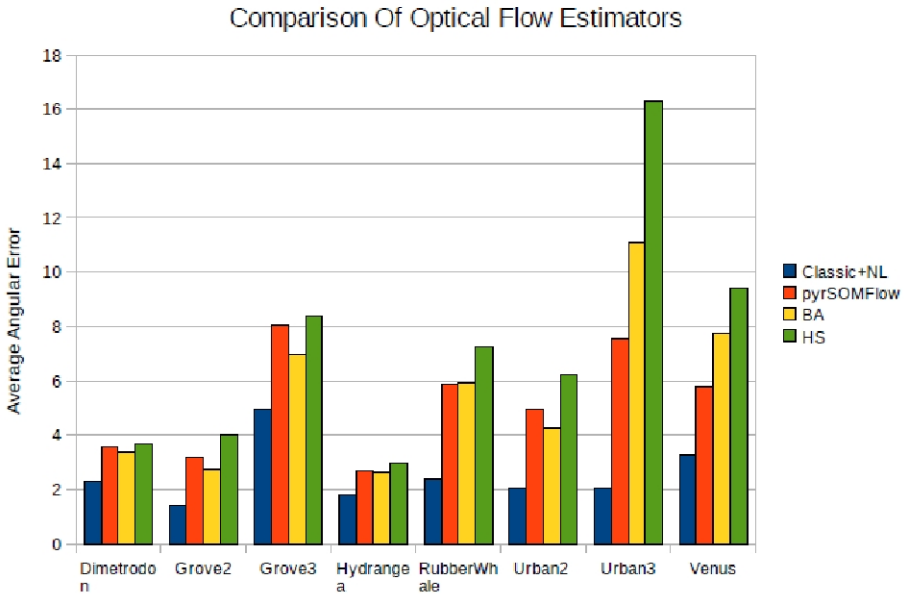


Figure 4.13: AAE Comparison

Chapter 5

Parallel Implementation on GPU

For non-linear interpolation of flow we use unsupervised techniques which are computationally intense. The good thing is that, all stages for computing optical flow using these techniques can be made data-parallel and can be speeded up using SIMD type machines.

5.1 GPGPU - GPU Computing

GPGPU refers to General Purpose computing on Graphics Processing Unit also known as GPU Computing [OHL+08].

Hardware Architecture: The GPUs have evolved from being just 3D graphics accelerators to being general purpose parallel processors with graphics capabilities. GPU used to be fixed-function processor built to handle massive parallelism inherent in 3D graphics. 3D Graphics involves description of objects using geometric primitives (mostly triangles) and processing of these primitives to be displayed on the screen. The graphics pipeline used to involve processing of input with steps implemented in

hardware using fixed-function units. Two main steps in the graphics pipeline involve processing vertices and subsequent processing of fragments made out of these vertices. The GPU architecture first evolved away from fixed-function towards programmable vertex and fragment processors. With this evolution GPUs supported non-graphics programs as well but had to be written such that non-graphics data needed to be masqueraded as vertices and fragments. The latest GPUs have unified processors for both vertex and fragment processing and allow non-graphics programs to be written without graphics context [OHL+08].

Software Development: Programming can be done with high-level languages like C. Graphics processing is highly parallelizable and GPUs take advantage by replicating the hardware units to process more data at a time. For GPGPU, the Single Program Multiple Data (SPMD) programming model is followed. There are GPU vendor specific programming interfaces and languages like NVIDIA's CUDA and AMD's Stream. Where as OpenCL is vendor independent specification that is designed to provide a unified API for heterogeneous computing on several kinds of parallel devices, including GPUs, multicore CPUs and the Cell Broadband Engine (www.gpgpu.org). In this work we use NVIDIA GPUs as SIMD machines for general purpose computing. Figure 5.1 shows the difference between the typical layouts of a CPU and a GPU. GPUs are well suited for data-parallel computations as they have multiple computing cores driven by very high memory bandwidth.

5.2 NVIDIA CUDA

The NVIDIA CUDA (Compute Unified Device Architecture) facilitates access to the computational power of GPU for non-graphics applications. CUDA is a

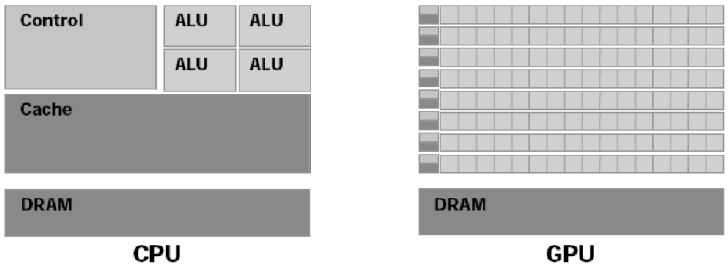


Figure 5.1: CPU vs GPU (CUDA Programming Guide)

software and hardware architecture for issuing and managing computations on GPU as a data-parallel computing device without the need of mapping them to a graphics API. The CUDA API comprises an extension to the C programming language. When programmed through CUDA, the GPU can be considered as a highly multi-threaded co-processor to host CPU. Both the host (CPU) and the device (GPU) maintain their own DRAM, referred to as host memory and device memory, respectively. One can copy data from one DRAM to the other using DMA.

Hardware: The device is implemented as a set of multithreaded Streaming Multiprocessors (SMs) as illustrated in Figure 5.2. A multiprocessor consists of eight Scalar Processor (SP) cores along with other supporting functional units. Each multiprocessor has a SIMD architecture: At any given clock cycle, each processor of the multiprocessor executes the same instruction, but operates on different data. Each multiprocessor has on-chip memory of the four following types:

1. One set of local 32-bit registers per processor,

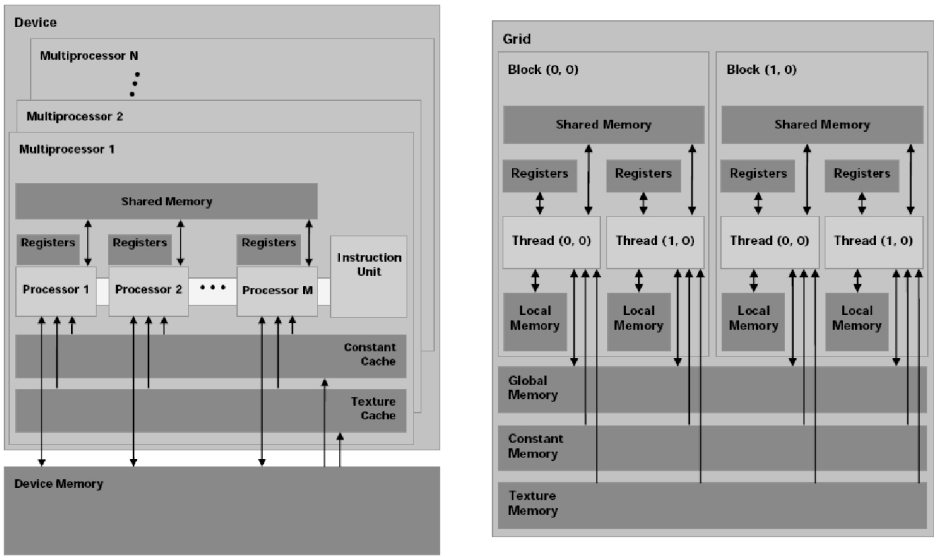
2. A parallel data cache or shared memory that is shared by all the processors and implements the shared memory space,

3. A read-only constant cache that is shared by all the processors and speeds up reads from the constant memory space, which is implemented as a read-only

region of device memory,

4. A read-only texture cache that is shared by all the processors and speeds up reads from the texture memory space, which is implemented as a read-only region of device memory.

The local and global memory spaces are implemented as read-write regions of device memory and are not cached. Each multiprocessor accesses the texture cache via a texture unit.



(a) CUDA Hardware Model (b) CUDA Memory Model

Figure 5.2: CUDA Architecture (CUDA Programming Guide)

Programming Model: As per NVIDIA CUDA Programming Guide: "CUDA is a parallel programming model and software environment designed to overcome the challenge of developing applications that transparently scale their parallelism to manycore GPUs with widely varying number of cores". At its core are three key abstractions - a hierarchy of thread groups, shared memories and barrier synchronization - that

are simply exposed to the programmer as a minimal set of extensions to C. These abstractions allow the programmer to partition the problem into coarse sub-problems which can be solved independently in parallel. The sub-problems can be solved with fine grain parallelism with cooperation allowed among the finer pieces.

Data-parallel portions of applications can be executed using multiple threads by downloading a program (called a kernel) on the GPU. Data-parallel processing maps data elements to parallel processing threads. Not all threads may run on the device at the same time though. The threads are batched into thread-blocks and these thread-blocks are in turn grouped into block-grid. Each thread is identified by its thread ID, which is the thread number within the block and each block is identified by its block ID, which is the block number within the grid. This allows the total number of threads that can be launched in a single kernel invocation to be much larger than number of processors on the device. But there are some restrictions that are placed on communication between threads due to thread-batching. Only threads within a thread-block can communicate where as threads across thread-blocks cannot communicate and synchronize with each other. A block is processed by only one multiprocessor, so that the shared memory space resides in the on-chip shared memory leading to very fast memory accesses. The multiprocessors registers are allocated among the threads of the block.

5.3 Mapping pyrSOMFlow onto CUDA

We use eleven different kernels for the eight stages of the CUDA-based pyramidal optical flow computation. The pyramidal approach starts with a data pyramid creation stage for the input images and involves low-pass filtering and subsampling. This is one-time activity before the pyrSOMFlow algorithm starts. pyrSOMFlow at

each pyramid level goes through pipeline of seven stages involving ten kernels.

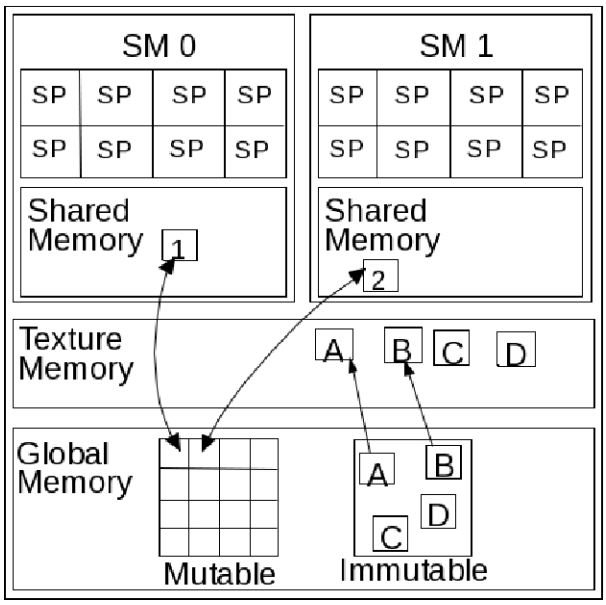


Figure 5.3: Mapping Optical Flow Computation on CUDA

Each kernel follows a similar pattern of implementation. Figure 5.3 shows the typical mapping of memory elements. The inputs of each stage are 2 dimensional arrays and the contents of the arrays do not change, that is they are immutable during processing of that stage. The input arrays are held in device global memory but are mapped as textures leading to use of texture cache for their access. The output arrays are written (hence are mutable) during the processing and are held in device global memory but are not mapped as texture, since textures need to be read-only. The data is not directly written to output arrays in global memory as it has very high latency. The data is written in two stages making use of low latency shared memory. First, the shared memory is used as a scratch-pad and written to and read from by threads. Later, once computation of a thread-block is done, the data from shared memory is transferred to global memory. Simultaneous writing by all threads to global memory improves performance by leveraging memory coalescing.

The one-time data pyramid creation stage requires one kernel and the seven stages with the ten kernels of pyrSOMFlow as:

1. *Bicubic Backwarp*: A single kernel is used where in optical flow values are used to backwarp an image. Bicubic convolution interpolation is used during the backwarp process.

2. *Gradients Estimation*: A single kernel is used to compute all three gradients (I_x, I_y, I_t).

3. *Local Motion Estimation*: Two kernels are required for this stage. One kernel per pass of the efficient 2-Pass evaluation of weighted least squares (Section 3.2.2). At the end of Pass2, we get the sets H_c and H_p.

4. *BatchSOMFlow*: A single kernel is used to train weights $W = H_c \cup H_p$ using $Q = H_c$.

5. *Median Filtering*: The trained weights W are filtered using a single kernel that implements median filtering. Median Filtering in optional.

6. *Accumulate Flow*: A single kernel is needed.

7. *Interpolate Flow to Higher Resolution*: Bicubic convolution interpolation is used during the process with 2X change in resolution and requires three kernels. The bicubic convolution interpolation is separable and can be implemented with separate vertical and and horizontal cubic interpolation passes. The two cubic passes work when the data in available in checkerboard pattern. So first kernel fills data in checkerboard pattern and the second and third kernels implement the vertical and horizontal cubic interpolation passes respectively.

5.4 Speedup using NVIDIA CUDA

Figure 5.4 shows the speedup obtained on CUDA for processing a pair of frames in Middlebury database image sequences to compute the optical flow. It can be seen that the speedup increases with increasing number of GPU cores. This is due to the embarassingly parallel structure of pyrSomFlow. Real-time performance can be achieved by further increasing the number of GPU cores.

Timing in milli sec	Dimetrodon	Grove2	Grove3	Hydrangea	RubberWhale	Urban2	Urban3	Venus
Intel i7 (serial code)	5020.43	6891.63	6785.03	5207.69	5220.54	6645.02	6268.82	3402.77
NVIDIA NVS 140M	1647.93	2237.4	2208.45	1649.43	1643.33	2231.99	2194.06	1158.12
NVIDIA FX5600	128.54	169.74	167.8	128.71	127.48	170.09	167.25	90.89
NVIDIA Tesla T10	95.41	128.25	127.02	95.86	95.65	129.29	127.3	66.9
NVIDIA GTX 480	35.99	48.99	48.49	35.98	35.88	48.95	48.44	25.96

Speedup	Dimetrodon	Grove2	Grove3	Hydrangea	RubberWhale	Urban2	Urban3	Venus
NVS 140M (16 core)	3.05	3.08	3.07	3.16	3.18	2.98	2.86	2.94
FX5600 (128 core)	39.06	40.6	40.43	40.46	40.95	39.07	37.48	37.44
Tesla T10 (240 core)	52.62	53.74	53.42	54.33	54.58	51.4	49.24	50.86
GTX 480 (480 core)	139.5	140.66	139.93	144.75	145.48	135.76	129.41	131.07

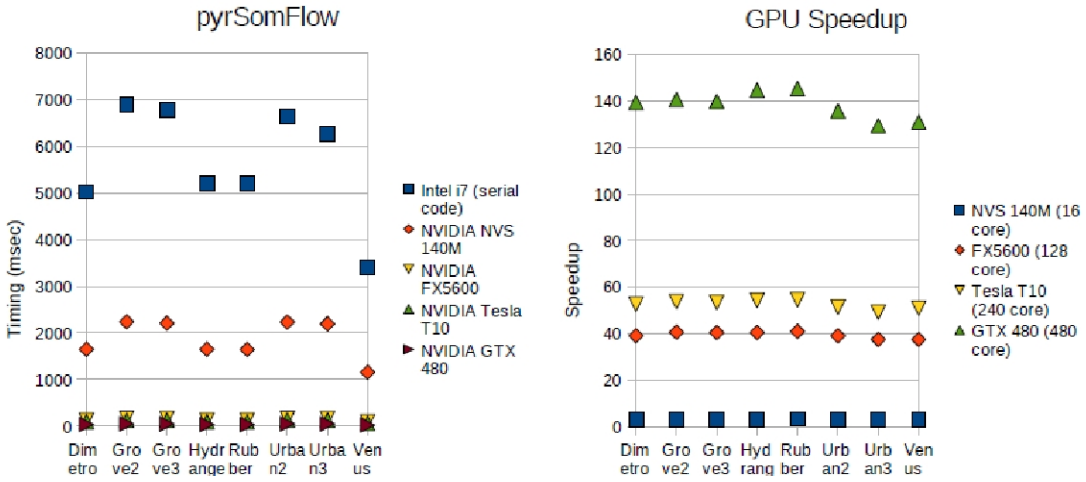


Figure 5.4: Speedup with CUDA

Figure 5.5 shows the timing of various stages for serial code and code executed with CUDA. These timings are for the case when Predictor-Corrector (Section 4.3) iteration is applied once at the highest resolution. Median Filtering of optical flow with a 5×5 window is also done to improve the flow fields [DSB10]. The time for

each stage shows the accumulated time over all levels of the pyramid. The testcase (Rubber Whale) has image size is 584×388.

```
Intel i7 3.33GHz (Serial Code)
====================================

RubberWhale
------------
Data Pyramid Time   =    2.263000 (ms)
Backward Warp Time  =   21.504002 (ms)
Gradient Calc Time  =    5.648000 (ms)
LocalMotion Time    =   32.080002 (ms)
Batch SOM Time      = 4936.356934 (ms)
Median Filter Time  =  219.104996 (ms)
Flow Add Time       =    1.397000 (ms)
BiCubic Interp Time =    2.189000 (ms)
Data Copy Back Time =    0.000000 (ms)
------------------------------------
Total Time          = 5220.542969 (ms)
```

```
NVIDIA GeForce GTX 480 1.40 GHz
==================================

RubberWhale
------------
Data Pyramid Time   =  0.909000 (ms)
Backward Warp Time  =  0.941000 (ms)
Gradient Calc Time  =  0.323000 (ms)
LocalMotion Time    =  0.913000 (ms)
Batch SOM Time      = 26.265999 (ms)
Median Filter Time  =  4.829000 (ms)
Flow Add Time       =  0.223000 (ms)
BiCubic Interp Time =  0.424000 (ms)
Data Copy Back Time =  1.056000 (ms)
------------------------------------
Total Time          = 35.883999 (ms)
```

Figure 5.5: Timing Details for Rubber Whale Testcase

Chapter 6

Conclusion

This work describes a new optical flow estimator that uses a self organization-based strategy. The determination of motion parameters for a single motion class using an optical flow formulation poses several well-known challenges, including an inherently locally ill-posed estimation problem and the possibility of the aperture problem. An a priori unknown number of motion classes with possible occlusion boundaries further compounds the problem. The self organization-based optical flow estimator handles image sequences containing a priori unknown number of motion classes along with occlusion boundaries.

The conceptual core of the self organization-based optical flow estimation approach consists of two sequential phases:

1. Local (regression-based) estimation of image motion parameters.

2. Global pass for determination of refined motion estimates using a modified SOM. Initialization of SOM weights and selection of training vectors is key to the applicability of self organization approach for optical flow estimation.

We cast the self organization-based optical flow estimation approach in a pyra-

midal framework to handle large motion vectors. We use modified version of batch SOM to make the estimator datum-parallel. The GPU (CUDA) implementation significantly speeds up the computation and due to its embarrassingly parallel nature, shows increasing speedup with increasing number of computing cores.

We evaluate the algorithm by comparing flow with known true estimates of Middlebury database image sequences. The attributes that we looked for optical flow were density of optical flow field along with preservation of occlusion boundaries and the ability to handle large-motion. Average Angular Error (AAE) and Average Endpoint Error (AEE) metrics were computed with the help of ground truth flow from Middlebury database.

6.1 Possible Extensions

The optical flow estimator is based on Brightness Constancy Assumption. In image areas where the assumption doesn't hold true, for example due to shadows, the estimator output is incorrect. The estimator could be extended to handle these kind of BCA violations in a more robust manner.

6.2 Contribution

Following is a list of contributions that have been made towards optical flow estimation knowledge.

1. We have shown the applicability of self organization-based techniques for estimating optical flow. The error metrics obtained for Middlebury database show that the quality of estimates is close to the leading techniques if not the best in class. Most existing dense optical flow estimation algorithms cast optical flow

estimation as a regularization problem and optimize a global energy function involving data term and a smoothness term. The self organization-based optical flow estimation on the other hand is two step process with an initial regression-based motion estimate that is later refined/interpolated by self-organization techniques.

2. During regression-based motion estimation, we employ an efficient 2-pass approximation strategy to compute locally weighted least squares. This makes the initial estimates less susceptible to occlusion boundary issues.

3. Moreover, we explicitly cast the aperture problem in an algebraic framework and use matrix-rank to identify motion estimates that are reliable (set H_c) and the ones that suffer from aperture issue (set H_p).

4. Self organization work well due to our novel formulation of weight initialization set ($W = H_c \cup H_p$) and training set (H_c). Using Heskes [Hes01] interpretation of SOM as mixture model with added regularization, we show why the vectors in set H_p get their motion corrected/refined during SOM training.

5. During self organization phase (for modified SOM), anisotropic neighborhoods are formulated. The distance norm which provides good anisotropic behavior is novel and uses residuals of the optical flow constraint equation. As SOM propagates motion information spatially, the motion discontinuities at occlusion boundaries are preserved due to the use of these anisotropic neighborhood functions.

6. Another contribution is with respect to making the whole estimator parallelizable. The parallelism of the algorithm can be termed embarrassingly parallel and it is shows good speedup on SIMD type machines. We have shown using

NVIDIA CUDA, the speedup obtained using GPUs with varying number of cores.

7. Due to the inherent independence, the optical flow can be estimated with a mask to avoid computation. If flow is required only in certain areas or edges, the interpolation can be turned off at other places.

Bibliography

[AS98] A. S. Atukorale and P. N. Suganthan. An efficient neural gas network for classification. In *5th Int. Conf. on Automation, Robotics, Control and Vision, Singapore*, 1998.

[BA96] M.J. Black and P. Anandan. The robust estimation of multiple motions: Parametric and piecewise-smooth flow-fields. *CVIU*, 63(1):75–104, January 1996.

[BB95a] S.S. Beauchemin and J.L. Barron. The computation of optical flow. *CS*, 27:433–467, 1995.

[BB95b] Lon Bottou and Yoshua Bengio. Convergence properties of the k-means algorithms. In *Advances in Neural Information Processing Systems 7*, pages 585–592. MIT Press, 1995.

[BSL+07] Simon Baker, Daniel Scharstein, J. P. Lewis, Stefan Roth, Michael J. Black, and Richard Szeliski. A database and evaluation methodology for optical flow. In *In Proceedings of the IEEE International Conference on Computer Vision*, 2007.

[CFP98] M. Cottrell, J. C. Fort, and G. Pags. Theoretical aspects of the som algorithm. *Neurocomputing*, 21(1-3):119 – 138, 1998.

[Che97] Yizong Cheng. Convergence and ordering of kohonen's batch map. *Neural Comput.*, 9(8):1667–1676, 1997.

[CHHV06] M. Cottrell, B. Hammer, A. Hasenfuss, and T. Vilmann. Batch and median neural gas. *Neural Networks*, 19:762–771, 2006.

[DLR77] A. P. Dempster, N. M. Laird, and D. B. Rubin. Maximum likelihood from incomplete data via the em algorithm. *JOURNAL OF THE ROYAL STATISTICAL SOCIETY, SERIES B*, 39(1):1–38, 1977.

[DSB10] S. Roth D. Sun and M. Black. Secrets of optical flow estimation and their principles. In *CVPR*, 2010.

[Hes01] Tom Heskes. Self-organizing maps, vector quantization, and mixture modeling. *IEEE Transactions on Neural Networks*, 12(6):1299–1305, November 2001.

[HS81] Berthold Horn and Brian G. Schunck. Determining optical flow. *AI*, 17:185–203, 1981.

[Kam] Yukiyasu Kamitani. Image generation with matlab. http://www.cns.atr.jp/ kmtn/imagematlab/index.html.

[Koh90] Teuvo Kohonen. The self-organizing map. *Proceedings of the IEEE*, 78(9):1464–1480, 1990.

[LK81] B.D. Lucas and T. Kanade. An iterative image registration technique with an application to stereo vision. In *DARPA81*, pages 121–130, 1981.

[MBS93] T. M. Martinetz, S. G. Berkovich, and K. J. Schulten. Neural-gas network for vector quantization and its application to time-series prediction. *IEEE Trans. on Neural Networks*, 4(1):558–569, 1993.

[OHL+08] J.D. Owens, M. Houston, D. Luebke, S. Green, J.E. Stone, and J.C. Phillips. Gpu computing. *Proceedings of the IEEE*, 96(5):879 –899, may 2008.

[Rob81] Keys Robert. Cubic convolution interpolation for digital image processing. *IEEE Transactions on Acoustics, Speech and Signal Processing*, 29:1153–1160, 1981.

[Sch97] Robert J. Schalkoff. *Artificial Neural Networks*. McGraw-Hill, 1997.

[SS09] M.P. Shiralkar and R.J. Schalkoff. Multiple-class spatiotemporal flow estimation using a modified neural gas algorithm. *Optical Engineering*, 48(1), 2009.

[SW82] R.J. Schalkoff and C.S. Warnekar. A predictor-corrector approach to tracking 3-d objects using perspective-projected images. *Proc. IEEE Southeastcon Ft. Walton Beach, FL*, pages 371–374, 1982.