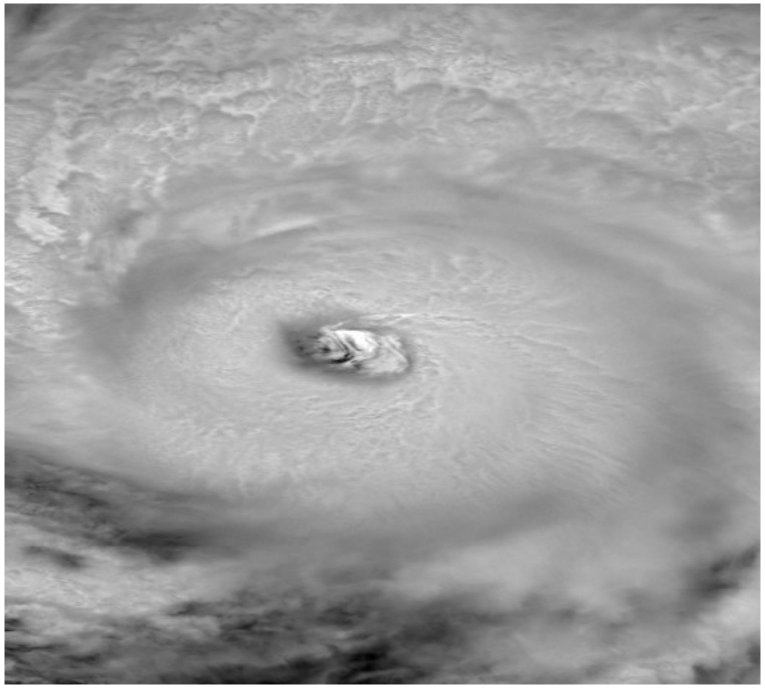
Implementation of Belief Propagation on the GPU using CUDA with application towards Cloud Tracking and Reconstruction

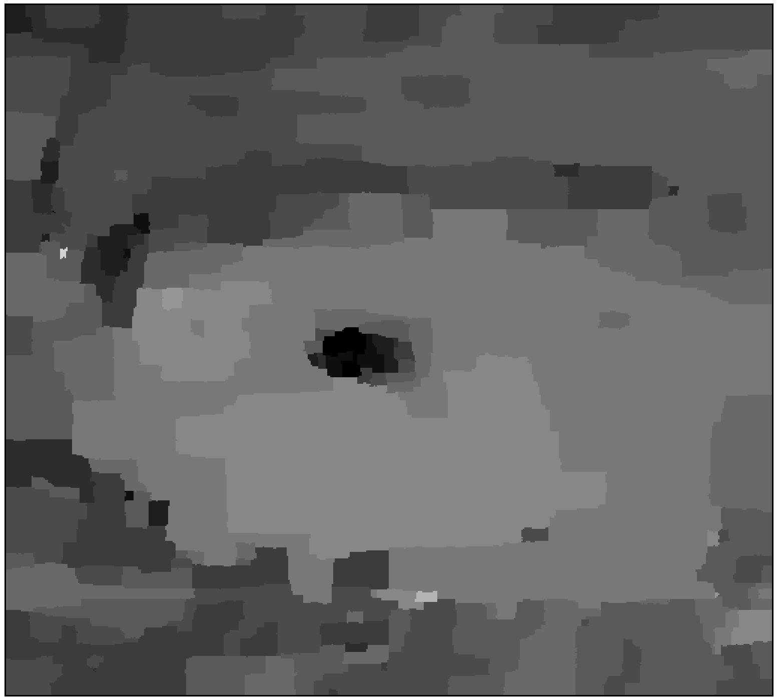
Scott Grauer-Gray
and
Chandra Kambhamettu

University of Delaware
Department of Computer and Information Sciences

Kannappan Palaniappan

University of Missouri
Department of Computer Science





Motivation and Goals

Common problems in computer vision:

Generation of accurate disparity map from a pair of stereo images

and

Generation of accurate motion vectors from a set of sequential images

Want the results as quickly as possible...possibly for use in real-time applications

Local methods (SAD, SSD, NCC, other variations):

Often fast, but not too accurate

Global methods

Use Markov random field to generate NP-hard energy minimization problem

Use belief propagation/graph cuts to generate approximate solution

Often generates better results than local methods, but takes longer to retrieve them

Motivation and Goals

Goal: To generate an accurate disparity map/motion vectors as quickly as possible

Local methods fast, but not too accurate; global methods more accurate, but not as fast...

Possible solution: Speed up global method without losing accuracy

How: Take advantage of the parallel processing power and abilities of the current generation of Graphics Processing Units (GPUs)

Paper describes implementation of the global belief propagation (BP) algorithm on the GPU

The use of the GPU significantly sped up the BP implementation versus the same implementation on the CPU with no loss of accuracy

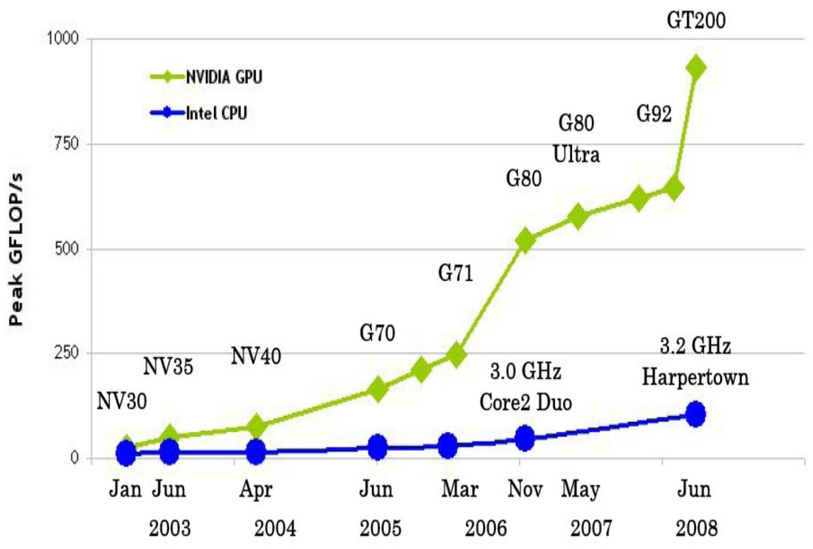
General-Purpose Processing on the GPU (GPGPU)

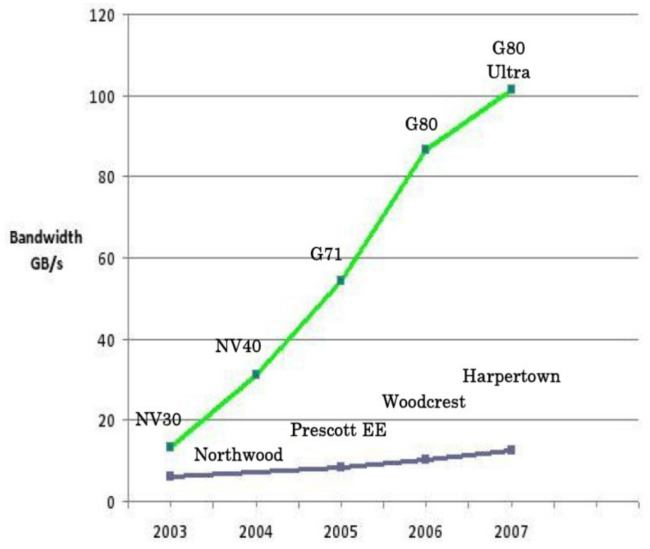
Graphic processing unit

Evolved from specialized hardware to a many-core parallel processor that can be used for general-purpose programming.

More powerful than the most powerful CPUs when measured in GFLOPs

Maps well to algorithms where the same computations are performed on many elements in parallel





GT200 = GeForce GTX 280	G71 = GeForce 7900 GTX	NV35 = GeForce FX 5950 Ultra
G92 = GeForce 9800 GTX	G70 = GeForce 7800 GTX	NV30 = GeForce FX 5800
G80 = GeForce 8800 GTX	NV40 = GeForce 6800 Ultra	

General-Purpose Processing on the GPU (GPGPU)

- **GPGPU initially required the use of a graphics API and the use of vertex/fragment shaders**
  - Essentially had to “trick” the GPU into performing general-purpose computation rather than graphics processing
  - Unable to scatter data to anywhere on DRAM on GPU
  - Even with limitations, GPGPU still did take off as a research area
  - GPU implementations of algorithms as varied as foreground-background segmentation in computer vision, N-body simulations in physics, and the three-dimensional Euler solver in math
- **Two previous publications describe implementations of belief propagation on the GPU**
  - “Belief Propagation on the GPU for Stereo Vision” by Brunton, etc. and “Real-time Global Stereo Matching Using Hierarchical Belief Propagation” by Yang, etc.
  - Both took the step of mapping the algorithm to the graphics API
  - Both publications reported that their implementation was able to achieve faster results versus the sequential CPU implementation with no loss of accuracy

General-Purpose Processing on the GPU (GPGPU)

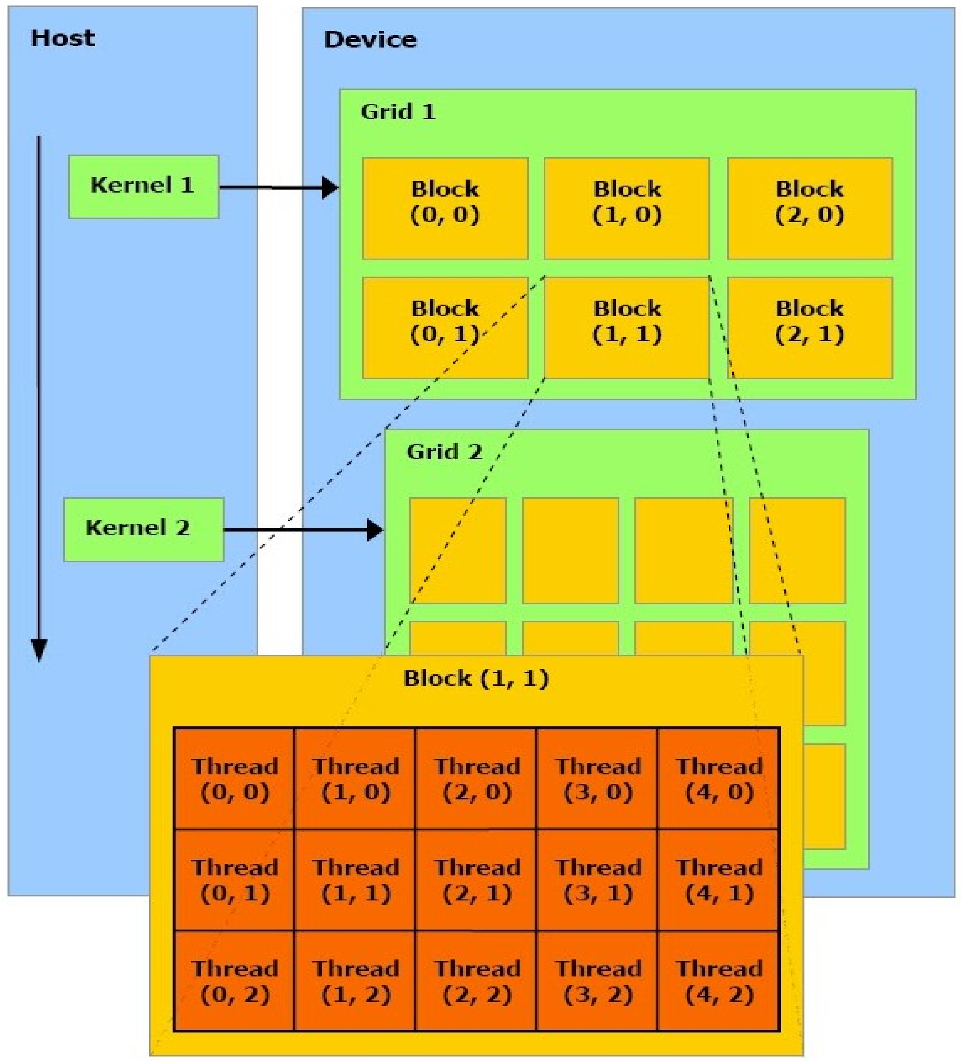
CUDA – Compute Unified Device Architecture

- Developed by nVidia and available on the current generation of nVidia graphics cards
- Provides the capability to perform general-purpose computations on the GPU without the use of a graphics API
- Provides the ability to scatter data to anywhere on GPU
- Marks a turning point in GPGPU

CUDA processing on the GPU

- Multiple threads executed in parallel
  - Each thread runs the same instructions as defined in the current kernel
  - Each thread has a unique ID – can be organized into a 1D, 2D, or 3D array
  - Threads are placed together in a thread block that can share data via shared memory and synchronize memory accesses
  - Limited number of threads in a single block
  - Multiple thread blocks can be organized into a 1D or 2D grid

CUDA threads, blocks, and grids



The host issues a succession of kernel invocations to the device. Each kernel is executed as a batch of threads organized as a grid of thread blocks

Energy Minimization for stereo vision

Use Markov random field to generate NP-hard energy minimization problem

Data cost

In stereo vision -> defined as cost of assigning a disparity D for pixel P

Calculated using brightness consistency assumption: data cost increases as difference in intensity between corresponding pixels increases

Discontinuity Cost

In stereo vision -> represents the cost of assigning disparities d_p and d_q to neighboring pixels p and q

Cost increases as the difference between d_p and d_q increases

Total Energy -> sum of the data costs $D_p(d)$ for each pixel p and the discontinuity costs $V(d_p, d_q)$ for each pair (p, q) of neighboring pixels

Goal is to find the disparity map that minimizes the total energy

Belief Propagation for stereo vision

Energy minimization problem is NP-hard...

Problem is equivalent to finding the maximum a posteriori (MAP) estimator of a MRF

Belief propagation can be used to find approximate solution

Belief Propagation

Iterative algorithm

In each iteration, messages are computed and sent from each pixel to its neighbors

Values in the received messages and the data costs are used to compute the messages to send in the next iteration

Each message can be viewed as a vector containing a value for each possible disparity value

When all the iterations are complete, the values for each label in the messages as well as the data costs are used to retrieve the estimated disparity at each pixel

Message values converge after “enough” iterations

Belief Propagation for stereo vision

“Naive” belief propagation takes lots of space and many iterations are required for convergence

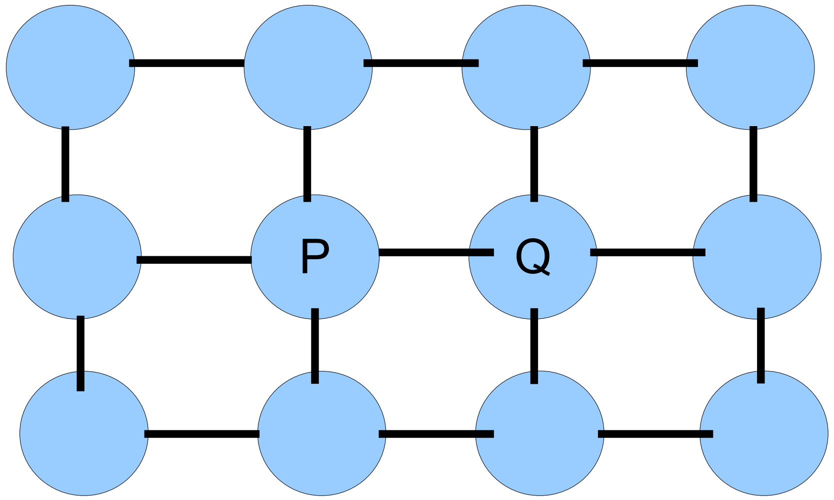
Felzenszwalb presents three speed-ups to BP in “Efficient Belief Propagation for Early Vision” (CVPR 2004)

Hierarchical scheme: the output messages from a coarser scale are used to initialize the messages at a finer scale -> messages converge in fewer iterations

Pixels divided into 2 sets `checkerboard' manner; each pixel in one set passes messages to its neighbors in other set -> Workload in each iteration is cut in half by alternating between updating the messages in the two sets

Running time to generate each message is reduced from $O(n^2)$ to $O(n)$ for common discontinuity models by using a two-pass algorithm over the disparity space

Message Updates in belief propagation



Message update from pixel p to neighbor q:

$$M^{t}_{p\to q}(d_q) = \min_{d\_p}(V(d_p, d_q) + D(d_p) + \sum_{s \in N(p)\backslash q} (M^{t-1}_{s\to p}(d_p)))$$

Message value is low if d_q is a good “label” for pixel q

Standard way to compute message: find d_p that minimizes message value for each choice of d_q
-> Takes $O(n^2)$ time

Possible to reduce time complexity to O(n) using certain discontinuity models...

Message Updates in O(n) time – truncated linear model

O(n) time message updates using truncated linear model

Discontinuity cost $V(d_p, d_q)$ between neighboring pixels p and q is equal to absolute difference between d_p and d_q capped at a constant T_{disc} to account for outliers/occlusions/noise

1. Initialize the value m_d at each disparity d by aggregating previous message and data cost values corresponding to d.
2. Set m_{max} to the sum of the minimum m_d and T_{disc}.
3. Update m in two passes that are performed sequentially and 'in place', so each update influences future ones

   (a) $m_d = \min(m_d, m_{d-1} + 1)$ from d = 1 to n-1

   (b) $m_d = \min(m_d, m_{d+1} + 1, m_{max})$ from d = n-2 to 0

Ability to scatter data in GPU memory needed to perform message updates in one call to GPU...cannot be done using vertex/fragment shaders and graphics API...

Belief Propagation on the GPU

CUDA implementation of algorithm uses speed-ups presented by Felzenszwalb

<u>Algorithm consists of the following steps:</u>

1. Smooth stereo set of images with Gaussian filter

2. Calculate the data cost $D_p(d)$ for pixel p at each point (x, y) at the bottom level 0.

3. Iteratively calculate the data cost for each pixel at each succeeding level by aggregating the data cost of the corresponding 2 X 2 block of pixels at the preceding level.

4. For level L-1 down to 0

    A. For iteration I-1 to 0

        i. For each pixel in the "current" set, compute the messages to send to neighboring pixels in "other" set using the current message values and data costs

        ii. Swap "current" set and "other" set

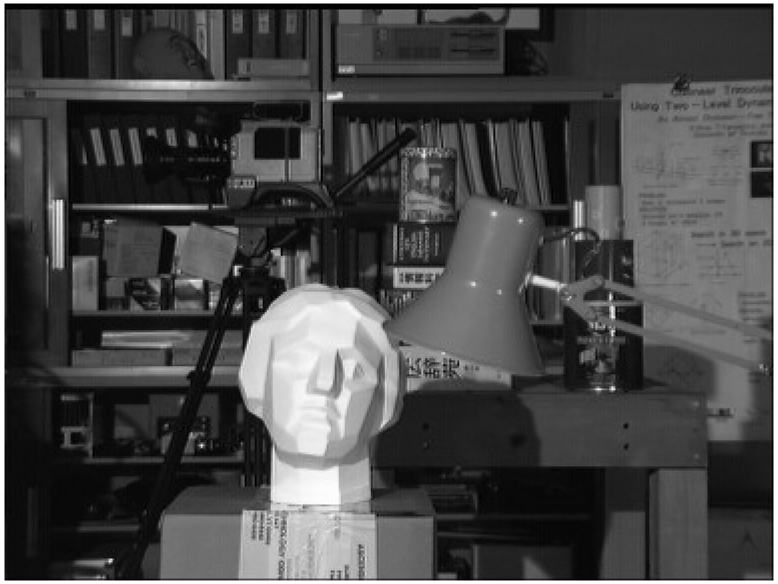
    B. If not at final level, copy the message values at each pixel to the corresponding 2 X 2 block of pixels at the next level down.

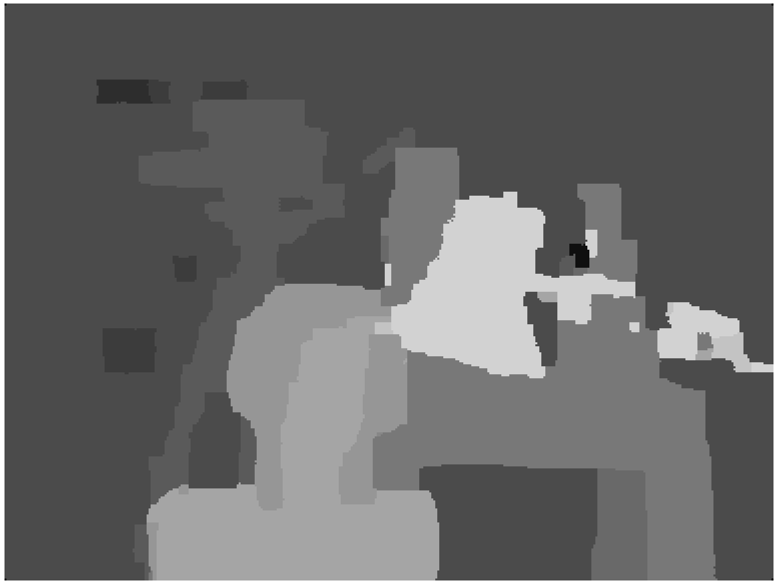
5. Retrieve the estimated disparity map by finding, for each pixel, the disparity that minimizes the sum of the data cost and message values.

Each step/substep is mapped to the SIMD model of the CUDA kernal -> same operations performed independently for every pixel

Results

Algorithm implemented is same as Felzenszwalb used in "Efficient Belief Propagation for Early Vision" (CVPR 2004) -> resulting disparity maps are nearly identical





Parameters for result: sigma for smoothing = 1.0

Number of levels = 5

Number of iterations = 10

T_{data} = 15.0 (truncated linear model used)

T_{disc} = 1.7 (truncated linear model used)

weight of data cost= .07

Results

Implementation benchmarked on two different computers

1. Laptop: Intel Core 2 Duo CPU running at 2.00 GHz and a nVidia 8600M GT GPU with two multiprocessors
2. Desktop: Intel Quad-core Xeon CPU running at 2.00GHz and a nVidia 8800 GT GPU with 14 multiprocessors

Parallel CUDA implementation as compared with sequential CPU implementation on each system

Parameters used for all results shown

$T_{data} = 15.0$

$T_{disc} = 1.7$

weight of data cost= .07

Sigma used for smoothing and number of levels/iterations in BP implementation are specified for each result.

Results

Benchmarking on 384 X 288 Tsukuba stereo set with BP parameters of 5 levels and 6 iterations/level (no smoothing) – same parameters used by Brunton in graphics API BP implementation

Computer 1:

CPU Implementation: 2.2 seconds

CUDA Implementation: .7 seconds (transfer time of image data/disparity values between GPU and CPU included)

.47 seconds (transfer time not included)

Computer 2:

CPU Implementation: .35 seconds

CUDA Implementation: .33 seconds (transfer time of image data/disparity values between GPU and CPU included)

.086 seconds (transfer time not included)

Brunton's graphics API implementation:

System: 3.4 GHz Pentium 4 and a nVidia GeForce 6800 GT GPU

CPU Implementation: 1.189 seconds

GPU Implementation using graphics API: .610 seconds (not given if transfer time included)

Results

Benchmarking on 512 X 512 set of stereo satellite images with Gaussian smoothing using sigma = 1.0 and BP parameters of 5 levels and 10 iterations/level

Computer 1:

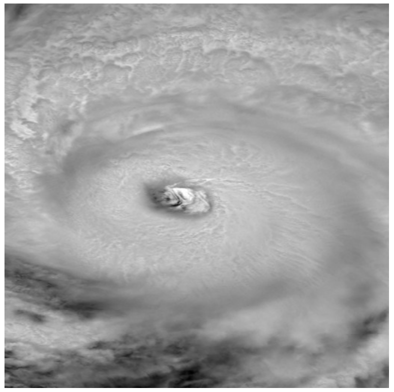
CPU Implementation: 10 seconds

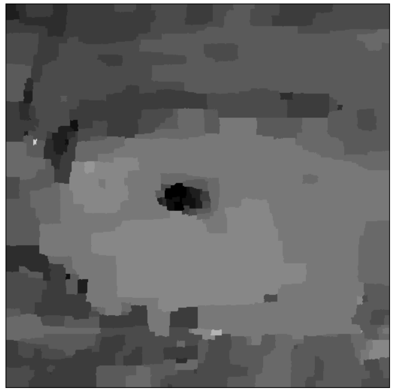
CUDA Implementation: 1.6 seconds (transfer time of image data/disparity values between GPU and CPU included)

Computer 2:

CPU Implementation: 1.3 seconds

CUDA Implementation: .5 seconds (transfer time of image data/disparity values between GPU and CPU included)





Motion Estimation using BP/CUDA

Given: 12 hour sequence of 512 X 512 satellite images taken at one-minute intervals of Hurricane Luis (taken by satellite NOAA GOES-9 on September 6, 1995)

Goal: generate accurate motion vectors showing cloud movement in these images as quickly as possible

Solution: Use CUDA implementation of belief propagation!

General structure of a BP implementation for motion is the same as for stereo

Main difference is that the labels in BP represent 2D motion vectors rather than 1D disparity values

Limited storage on the GPU

- Not enough DRAM available to store data costs and messages to run motion estimation BP on 512 X 512 satellite images

- **Solution**: divide the images into multiple blocks, run BP on the blocks, and combine the results

  - Method presented in "Low Memory Cost Block-Based Belief Propagation for Stereo Correspondence" -> increase in error rate is minimal

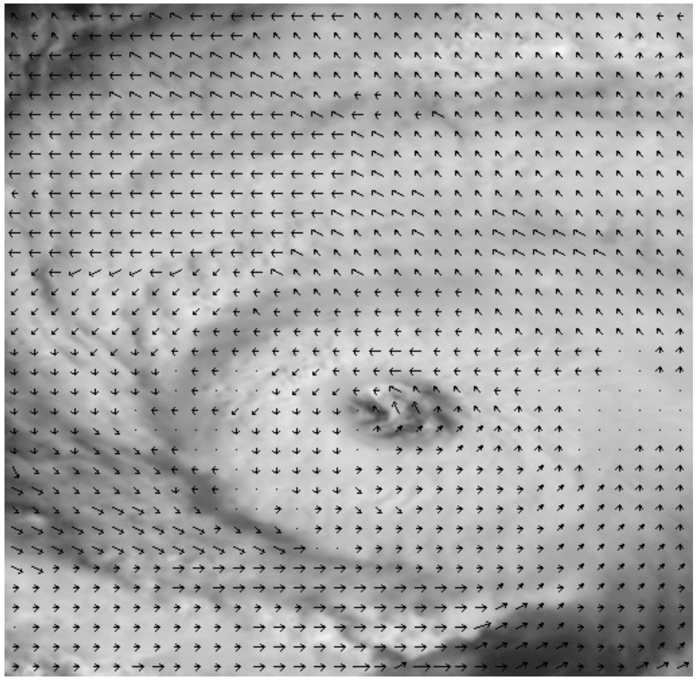
Results for Motion Estimation

Range of motion in the satellite images is [-5, 5] in the x and y directions -> 121 possible motions

Each 512 X 512 image is divided into four 256 X 256 images

Each image then processed using 4 levels with 10 iterations per level

Results combined to generate a full set of motion vectors



Benchmarking on Computer 1:

CPU Implementation: 64 seconds (5 BP levels - image division not necessary/performed)
CUDA Implementation: 15 seconds (transfer time of image data/disparity values between GPU and CPU included)

Benchmarking on Computer 2:

CPU Implementation: 16 seconds (5 BP levels - image division not necessary/performed)
CUDA Implementation: 1.5 seconds (transfer time of image data/disparity values between GPU and CPU included)

Future Work

Using this implementation for real-time cloud analysis

Exploring the possibility of temporal smoothing of the output disparities/motion vectors over a sequence of images

Thank you

Chandra Kambhamettu

Kannappan Palaniappan

NASA

ICPR

PRRS Workshop

You all for listening

References

NVIDIA Corporation: NVIDIA CUDA compute unified device architecture programming guide. NVIDIA Corporation, Jan 2007.

A. Brunton, C. Shu, and G. Roth. Belief propagation on the GPU for stereo vision. In Proc. 3rd Canadian Conf. Computer and Robot Vision, page 76, 2006.

P. Felzenszwalb and D. Huttenlocher. Efficient belief propagation for early vision. In IEEE Int. Conf. Computer Vision and Pattern Recognition (CVPR'04), pages 261–268, 2004.

Y. Tseng, N. Chang, and T. Chang. Low memory cost block-based belief propagation for stereo correspondence. In IEEE Int. Conf. Multimedia and Expo, pages 1415–1418, 2007.

Q. Yang, L. Wang, R. Yang, S. Wang, M. Liao, and D. Nist´er. Real-time global stereo matching using hierarchical belief propagation. In British Machine Vision Conf., pages 989–998, 2006.