Image Processing on the GPU

IkkJin Ahn ikkjin@gmail.com, Michael Lehr michael.lehr@gmail.com,, Paul Turner turnerpd@seas.upenn.edu
University of Pennsylvania
GPU Programming and Architecture
February 27, 2005

Abstract: We propose to create an Image Processing Toolkit that will exploit the GPU architecture to achieve real time or near real time image processing. The main focus of this toolkit is to provide base Image Processing tools, and in doing so will also provide some matrix multiplication tools, and matrix solvers. The main contribution of this effort is to implement an image segmentation routine which is based in isoperimetric graph partitioning. We will implement this on a Windows platform using NVidia FX video cards as opposed to the Linux-based OpenVIDIA project.

1. Background:

Image processing is a prime topic for acceleration on the GPU. Many image processing techniques have sections which consist of a common computation over many pixels. This fact makes image processing in general a prime topic for acceleration on the GPU.

Computer vision and image processing are very related in that image processing techniques can be applied to achieve computer vision or within computer vision algorithms. The essence of computer vision is to bring more meaning to raw image data. This can include detecting edges, regions, textures, objects, face recognition etc. In all of the cases you can either work with still images or to video sequences. Processing video sequences in real time using advanced computer vision algorithms is a major area of interest.

The OpenVIDIA project has accelerated many image processing algorithms on Linux systems, and have been able to successfully work with video in real time. They have accelerated, Canny edge filters, Canny 'corner' filter, optical flow, feature tracker, 3D registration, and can create output for 3D Studio Max. All of the examples for image processing are based on video streams, but can also be applied to still images.

2. Overall Library Design

We will provide two general modes for the library routines: *method-call* and *state-machine*. The *state-machine* will be more efficient sequencing multiple operations on the same image, while the *method-call* may be more convenient for certain types of applications. All calls will take in single image data, and some will take in a sequence of image data (where sequential data is applicable). Depending on the library call, they will output another image, a modified image, or a structure containing the resulting computation.

3. Topics:

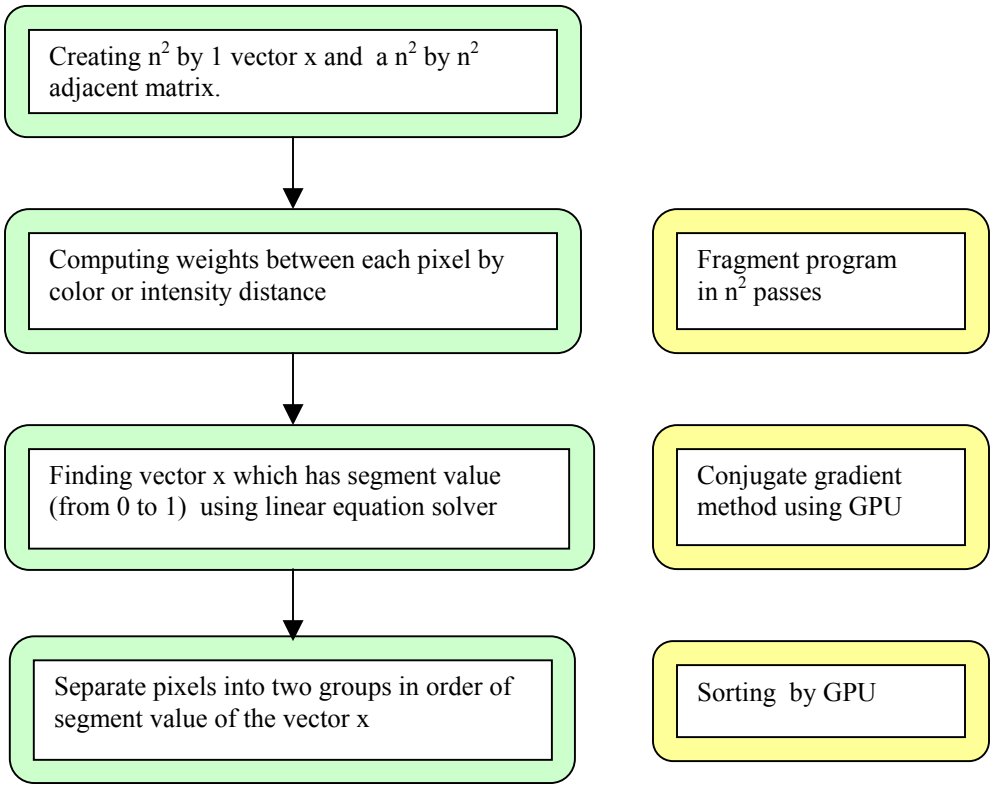
We would like to provide the following functionality, and will deliver the parts with an asterisk.

- Computer Vision Tools
  - Image Segmentation
    - Isoperimetric graph partition *
    - Ncut
    - Active Contour *
  - Image Enhancement
    - Histogram Equalization
    - Noise Reduction
  - Convolution
    - Blur *
    - Sharpen *
    - Gabor Filters *
    - Any Kernel *
  - FFT
    - Refocus
  - Edge Detection
    - Canny's *
    - Sobel, Prewitt

        - Roberts
        - Marr-Hildreth (Laplacian of Gaussian) *
    - Feature Point Detection
        - Harris Corner Detector *
    - Pixel Motion
        - Optical Flow *
        - Layer extraction
- Matrix Tools
    - Matrix multiplication, add, and sum
        - Sparse *
        - Dense
    - Linear Equation Solver
        - Gaussian Elimination
        - Conjugate Gradients *
    - Eigen Vector Solver

3.1. Isoperimetric Graph Partitioning [1]

Spectral methods of graph partitioning have been shown to provide a powerful approach to the image segmentation problem. Isoperimetric graph partitioning is one of the recent spectral methods. 'Isoperimetric Graph Partitioning for Data Clustering and Image Segmentation', Grady and Schwartz (2003) The method has improved the speed and stability of the segmentation but is still too slow to apply to each frame of real time video sequences. Hopefully, each step of isoperimetric method can be improved by GPGPU techniques. The following diagram shows steps of isoperimetric method and correspondent GPU techniques for each step.



The graph partitioning problem is to choose subsets of the vertex set such that the sets share a minimal number of spanning edges while satisfying a specified cardinality constraint. In isoperimetric method, we define a isoparametric constant h as

$$h_G = \inf_S \frac{|\partial S|}{Vol_S}$$

and try to find a set S (i.e., a segment) which minimize h_G. $|\partial S|$ denotes the boundary of a set, S, i.e., the sum of the weights from inside of set S to outside of set S. Vol_S can be expressed as $Vol_S = \sum(di \ \forall \ vi \in S)$. Intuitively, h_G means the ratio of the connectivity between a segment and backgroud divided by the size of the segment. To find a segment which manimize h_G, we can make a laplacian matrix and then can solve the laplacian matrix. The laplacian matrix is defined as this.

$$L_{ij} = \begin{cases} d_i & \text{if } i = j, \\ -w(e_{ij}) & \text{if } e_{ij} \in E, \\ 0 & \text{otherwise.} \end{cases}$$

By definition of L,

$$|\partial S| = x^T L x$$
$$Vol = x^T d$$

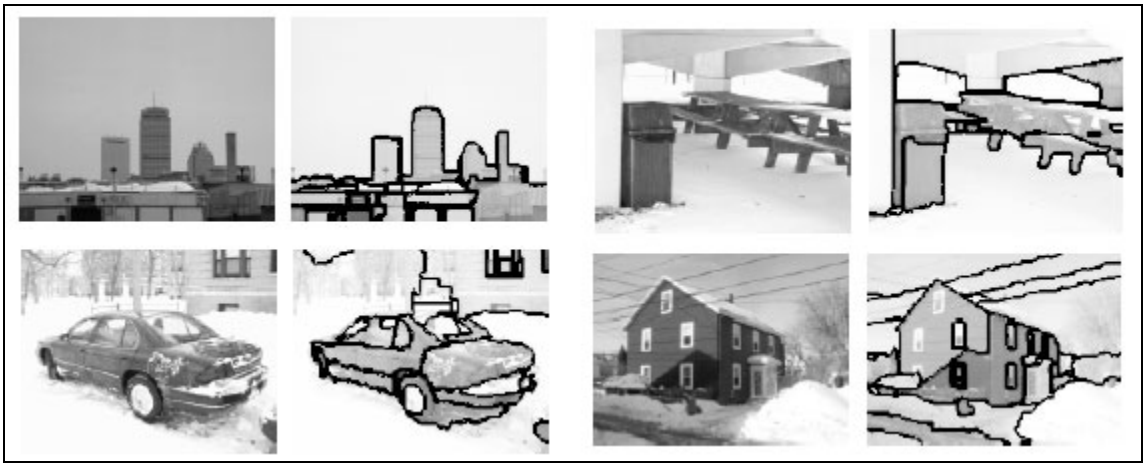
To minimize h_G, we can solve a lagrange minimizer. The resulting linear equation is

$$2Lx = \Lambda d.$$

where Λ is a scalar. Unfortunately, the matrix L is singular. Therefore we arbitrarily designate a node, v_g, to include in S (i.e., fix $x_g = 0$). This is reflected by removing the g^{th} row and column of L, denoted by L_0, and the g^{th} row of x and d, denoted by x_0 and d_0. Then we will $L_0 x_0 = d_0$, which can be solved by gaussian elimination or conjugate method.

Solving equation for x_0 yields a real-valued solution that may be converted into a partition by setting a threshold.

Here are some results from the algorithm



3.2. Active Contour

The active contour (snakes) of an image is curves over an image used in order to find object boundaries. It is based off of a cost function that tries to maximize the gradient across the curve, while minimizing the angle of the curve. It can be

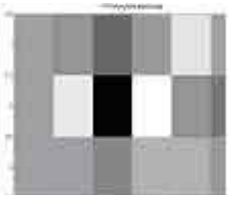
accomplished using dynamic programming. We envision an iterative approach of cost O(n) for an n x n size image. During computation all data can be stored on the GPU.

3.3. Convolutions (Blur, Sharpen, All Kernels)

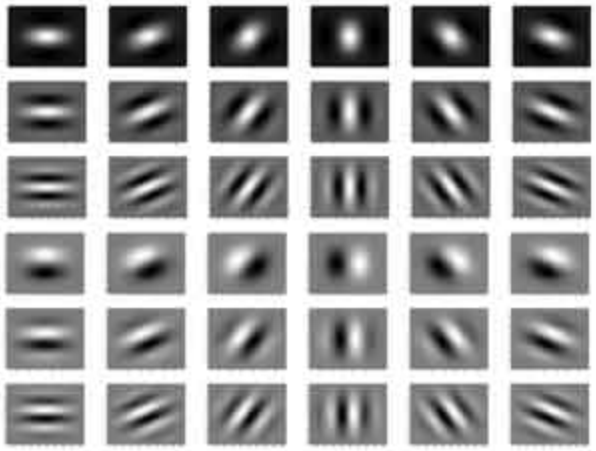
Running a convolution across an image is an important primitive in computer vision. We will allow the user to use some pre-defined convolutions for Blur and Sharpen (and others listed below). All convolution functions will return an image with the results from running the convolution.

3.4. Gabor Filters

Many computer vision techniques first will convolve an image using a set of gabor filters first in order to extract information. Gabor filters are odd or even filters (top/bottom of right) using a sign/cosine wave under a gaussian. They can be used for edge detection, extraction of texture features (at differing frequencies), and in some ways they simulate simple and complex cells in the retina and the V1 of the visual cortex. The user will be able to run a gabor filter, set of gabor filters, or adaptive set of gabor filters across an image. The user will also be able to extract the resulting images, or the average filter response(s).



Filter Responses



Different Gabor Filters

3.5. Canny's Edge Detector

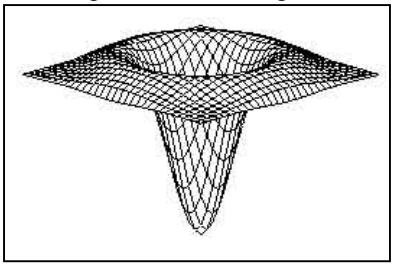
Canny's edge detector will first smooth the image using a Gaussian, and then a 2D second derivative function is applied to the image. Then we compute the magnitude of the gradient from the filter and detect the maximum. Then two thresholds are chosen, HIGH and LOW and edges are followed from HIGH until a pixel with a magnitude lower than LOW is found.

3.6. Marr-Hildresh (Laplacian of Gaussian)

This is a convolution of an image based off of the 'Mexican Hat Filter' or Laplacian. The Laplacian of an image is defined as:

$$L(x,y) = \frac{\partial^2 I}{\partial x^2} + \frac{\partial^2 I}{\partial y^2}$$

First the image is smoothed using a Gaussian filter, and then this filter is applied to the smoothed image. Since we are doing two filters they can actually be combined into one so that only one convolution is necessary.



3.7. Harris Corner Detector

Harris corner are relatively fast on the CPU and yield good results. We would like to implement the regular corner detector as well as a multi-scaled harris coner detector which will mip-map the image and detect corners at each level. This is useful for obtaining hidden edge detection because an edge may only be able to be detected at a higher level.

3.8. Optical Flow

Optical flow is widely used to calculate pixel movement between frames. This is done by optimizing pixel variation through a sequence of images. Large fast movements may result in incorrect flow. Optical flow can be used in order to obtain rough estimates of where objects are by using the fact that larger movements in the visual field are usually from objects that are closer to the camera. This function will result in either n-1 images, or one image displaying the flow of a set of specified pixels throughout the sequence.

3.9. ***Sparse Matrix and Conjugate Gradients [2]***

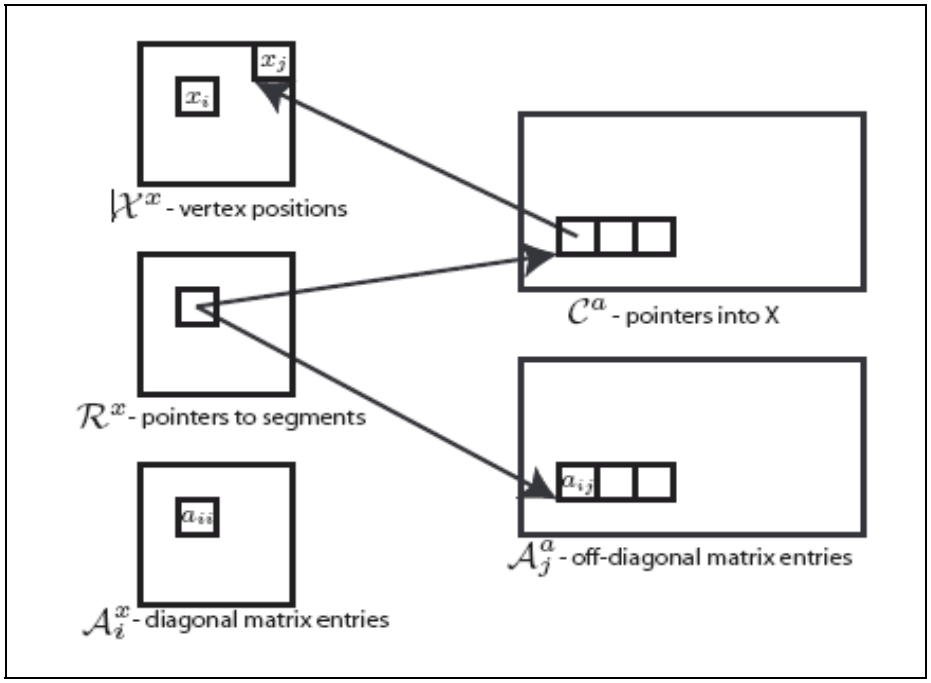
Linear Algebra is heavily used for our image segmentation algorithm. There are two fundamental areas that require attention. Basic operations (multiplication and addition) and solving Linear systems.

Basic operations such as dense matrix multiplication and adding has already been well studied for the course and do not need a review here.

At the core of the algorithm lies the deceptively simple equation ($L_0x_0 = d_0$). And to solve this requires solving "a large sparse system of symmetric equations where the number of nonzero entries in L will equal 2m." Using a dense matrix solution algorithm would be very costly in this case and a sparse algorithm has the potential to greatly increase the efficiency of the implementation as a whole. While dense matrix GPU multipliers are relatively simple to create, they do not possess enough redeeming qualities to justify their use in this case

Since the algorithm is to run on the GPU, an iterative conjugate gradient method lends itself best to the operating environment. This method gives us the option of sacrificing accuracy for speed and visa versa. This algorithm is also more stable than eigenvector approximation which can converge on inappropriate results in some circumstances, or fail to converge at all.

On the GPU, the conjugate gradient solver requires two non-trivial functions: sparse matrix and vector multiplication and vector inner product. The matrix-vector multiplication data representation must be handled carefully. Matrix A's elements are stored in a very interesting manner. The diagonal elements (aii) are held in one texture so they match directly with the vector for each i (X_i). Off-diagonal, non-zero elements are stored in a second texture in sequence with yet another texture (R) providing a series of pointers to the beginning of each segment. To ensure a very complicated and impressive looking design, the pointer texture also applies to the coordinates in another indirection texture (C) that in turn provides pointers into the unknown vector. Note that this means that the layouts of A and C are the same, but R is still required as an indicator of the boarder between segments. As a high level understanding, what this all does is allow a tight mapping between the matrix and the unknowns vector, stripping away large unneeded portions and taking advantage of the sparseness.



Graphical representation of textures involved

It should be expressly noted that these textures should be created at the same time the original matrix is being initialized so as to save time and not require a searching re-traversal of the space. Using:

$$
\begin{aligned}
j &= \mathcal{R}^x[i] \\
\mathcal{Y}^x[i] &= \mathcal{A}_i^x[i] * \mathcal{X}^x[i] + \sum_{c=0}^{k_i-1} \mathcal{A}_j^a[j+c] * \mathcal{X}^x[\mathcal{C}^a[j+c]],
\end{aligned}
$$

The matrix vector-multiplication can be completed through a series of passes into y with appropriate upper bounds on the summation.

This method takes GPU abilities up to the time of the GeForce FX into account. While this may be too modern for many users, it is here considered an appropriate level to be operating at.

The most well known method for solving a linear system of equations is Gaussian Elimination. Simply put, this is a series of row based operations, such as multiplying, adding, and swapping rows. This is done in an attempt to simplify equations down so that the variable's values may be found one at a time. In the opinion given here, this is not a good algorithm to implement on the GPU as it requires too much algorithmic awareness of the data content. This awareness would have to be maintained on the CPU and the BUS hit of reading that data back is too great, especially when taking the computations done by the CPU to determine the next step. This quickly becomes little more than a novelty. The concept may be further investigated, but little is expected from this branch of inquiry.

These two fundamental parts of linear algebra provide a tool-base to allow a much grander realm, only touched upon in our work here.

4. Summary

In summary we are going to create a library with two modes, *method-call* and *state-machine*, with the following functions:

- Computer Vision Tools
  - Image Segmentation
    - Isoperimetric graph partition *
    - Active Contour *
  - Convolution
    - Blur *
    - Sharpen *
    - Gabor Filters *
    - Any Kernel *
  - Edge Detection
    - Canny's *
    - Marr-Hildreth (Laplacian of Gaussian) *
  - Feature Point Detection
    - Harris Corner Detector *
  - Pixel Motion
    - Optical Flow *
- Matrix Tools
  - Matrix multiplication, add, and sum
    - Sparse *
  - Linear Equation Solver
    - Conjugate Gradients *

We will be implementing the algorithms above using both the CPU and GPU. We will be testing on an AMD Athlon 2Ghz with a NVidia Quadro FX 3400 PCIe.

[1] Grady, L., Schwartz E., “Isoperimetric Graph Partitioning for Data Clustering and Image Segmentation” (2003)
[2] Bolz et al. “Sparse Matrix Solvers on the GPU: Conjugate Gradients and Multigrid”, ACM Transactions on Graphics, Volume 22 , Issue 3 (July 2003)