Parallel Computing
Final Project Report
Due: 12/06/17

Benjamin Horn
Chris Klammer

Uncanny - Image Edge Detection in Parallel

Introduction

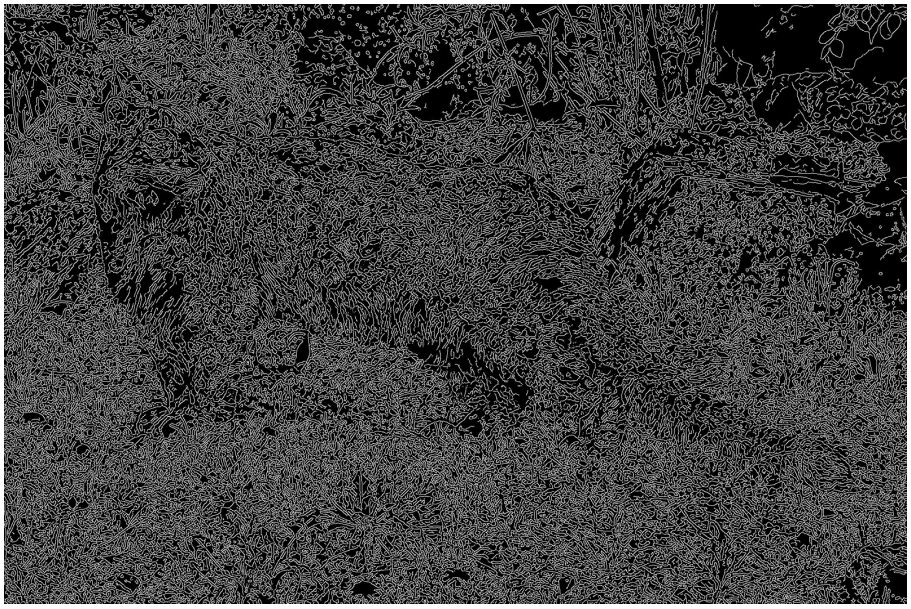
The goal of the Uncanny project is to parallelize the canny edge detection algorithm. Through this, we can compare the run times and efficiency of both the serial and parallel executions to show the potential performance improvement when running in parallel. After a semester of studying and implementing parallel computing via Cuda, a parallel computing platform, we had the tools to further explore image processing in parallel. Upcoming coops also fueled our interest in image processing, since image processing along with machine vision will become a core focus.

The Canny edge detection algorithm is an often used algorithm for edge detection, image processing, machine vision, object detection and many more applications. However, the algorithm is not as simple as detecting changes in photo brightness and marking them as an edge. The process for the Canny method is broken into the following main steps.

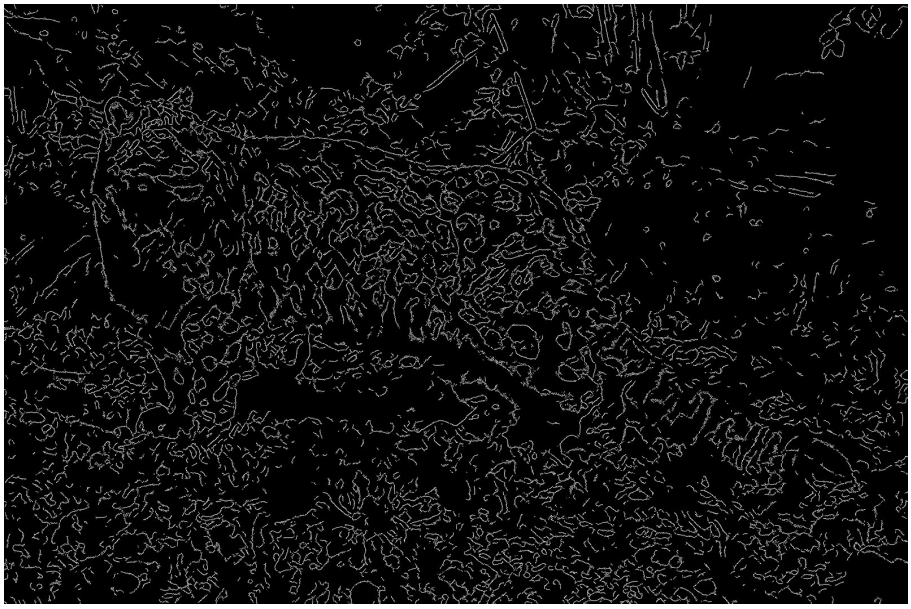
1) **Image smoothing:** Before the processing can start, image noise must be removed. Image noise is the large amount of sporadic changes in brightness within the photo that do not identify an edge. This could be a background object like a tree or the sky, something you would not want identified in your edge detection output. A common filter used for smoothing is the Gaussian Filter, or the Gaussian Blur.
    a) The removal of this image noise greatly impacts the final product.
        i) Original Image: (https://en.wikipedia.org/wiki/Snow_leopard)

(1) 

ii) Image Edge Detected with 1 Gaussian Blur Iteration

(1) 

iii) Image Edge Detected with 10 Gaussian Blur Iteration

(1) 

iv) Image Edge Detected with 30 Gaussian Blur Iteration

(1) 

The image becomes more and more clear as relative noise in the image is removed.

2) **Edge Thinning:** Removing the unwanted false edges from the image. This is done by a process called Non-maximum suppression. During this step edges going in the same direction are compared. Edges that do not have the absolute local maximum, determined from the sharp changes in brightness of the pixels, will be suppressed and set to '0'. The strongest edge in a group going in the same direction will be kept and used in the next step.

3) **Edge Strength Determination:** In this step the image is closer to the final product, however there are still many false edges. The goal of this step is to identify the weak edges and suppress them from the image. In order to do this a high and low threshold value are determined from the image. Then images can lie in the following three categories.

    (1) **Strong Edge:** Greater than the high threshold.
    (2) **Weak Edge:** Less than the high threshold but greater than the weak edge threshold.
    (3) **Suppressed edge:** If an edge falls below the weak threshold value, it will be suppressed and removed the edge detected image.

4) **Final Analysis:** The last step in the process is to determine if the weak edge pixel should be kept. To do this the pixel in question has it's 8 surrounding neighbor pixels examined, if at least 1 of the 8 are a strong edge pixel, it is kept, if not - it is suppressed and removed.
    a) **Example:** A Weak Pixel that would be suppressed. Where the 0 values are blank space, not identified by a weak or strong edge.
        i) Each block represents a pixel.

0	0	0
0	Weak	0

0	0	0

b) **Example:** A Weak Pixel that would be preserved.

i) Each block represents a pixel.

0	Strong	0
0	Weak	0
0	0	0

After understanding the above process of the edge detection process, you can further analyze the compute complexity. The edge detection algorithm is best looked at side by side comparing the CPU and GPU implementation time and space complexity.

Assuming an (m) x (n) image.

Step	Serial Complexity	Serial Dependencies	Parallel Complexity	Parallel Dependencies
Gaussian Blur	$O(k^{2}*m*n)$	Kernel Size - k Pixels - m * n	$O(k^{2})$	1 pixel per thread
Non-maximum suppression	O(m*n*c)	Comparisons - c Pixels - m * n	O(c)	1 pixel per thread
Double Threshold	O(m*n)	Pixels - m*n	O(1)	1 pixel per thread
Hysteresis	O(m*n*8) = O(m*N)	Comparisons - 8 Pixels - m*n	O(8) = O(1)	1 pixel per thread
Load Image into Memory	1	Step Not Needed	O(ImageSize/Tile Size) (shared)	Overhead for paralllel
Memory Transfer	1	Step Not Needed	O(ImageSize/Tile Size) (shared)	Overhead for paralllel
Total Complexity	$O((m*n)^{4*}k^{2}*c$ = $O((imageSize)^{4*}k^{2}*c$	Not Great	$O(k^{2}*c*(ImageSize/TileSize)^{2})$	Huge Gain

Although canny edge detection was developed in 1986, the algorithm and edge detection have come a long way. Before beginning the project, research was done on current solutions and ongoing work concerning image processing in parallel, specifically canny edge detection and the current state of the GPU.

Research done on canny edge detection in parallel at University of Maryland, College Park opened up about the improvement and growth in performance by the GPU over the last few years. Showing that the CPU was generally stayed steady in its growth (Luo & Duraiswami, 1, Figure 1).

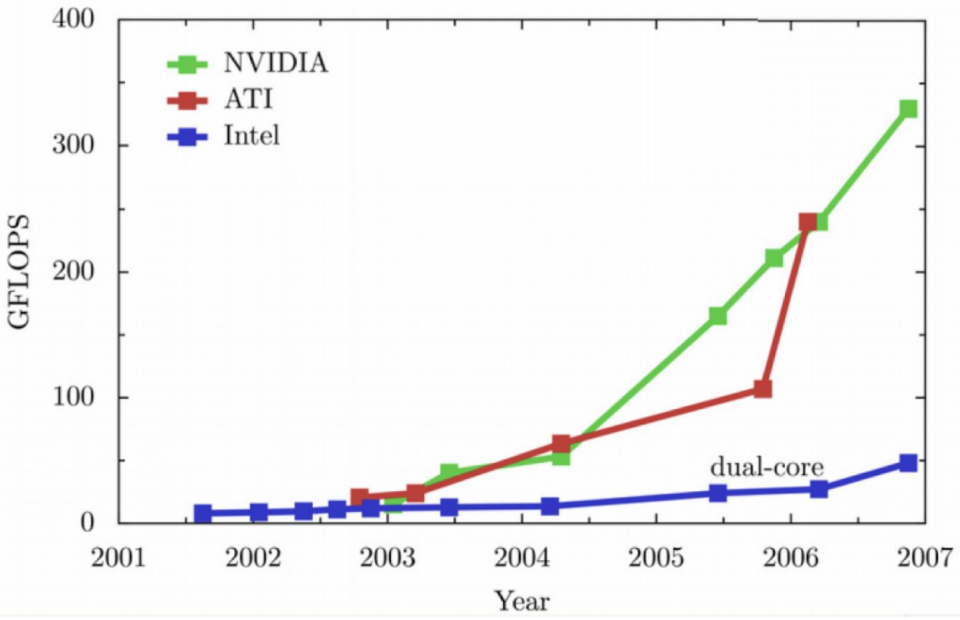


Figure 1. GPU and CPU growth in speed over the last 6 years (from [1]).

On top of this deviation in performance gain over the years, the GPU prioritizes throughput over latency, making it a prime candidate for parallel applications.

A similar study published in 2009, showed a similar approach at image processing using relatively recent Nvidia GPUs and Cuda to solve the problem. This study demonstrated the use of 3 different GPU units, in comparison to the serial implementation using the following hardware. (Jackson 26)

1) GeForce 8400GS – 1 Multi Processor(s) - 8 Cores
2) GeForce 8500GT – 2 Multi Processor(s) – 16 Cores
3) GeForce 9800GTX – 8 Multiprocessor(s) – 128 Cores

Results showed a significant increase in performance with the most powerful GPU having over 6x speed increase over the serial implementation.

Single Pixel per Thread (Jackson, 28)

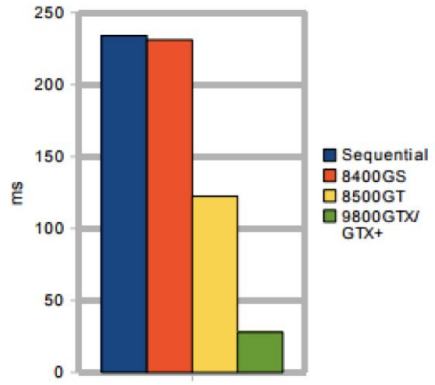


Figure 4.3: Sequential time vs one pixel per thread GPU algorithm; 32 threads.

Multi Pixel per Thread (Jackson, 29)

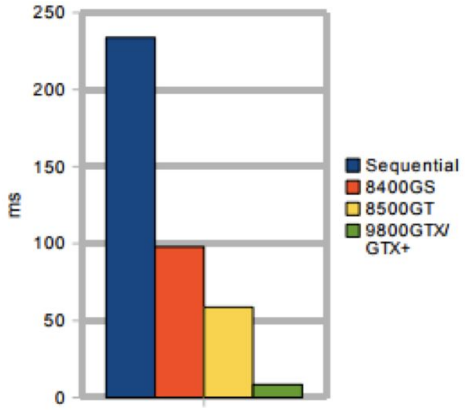


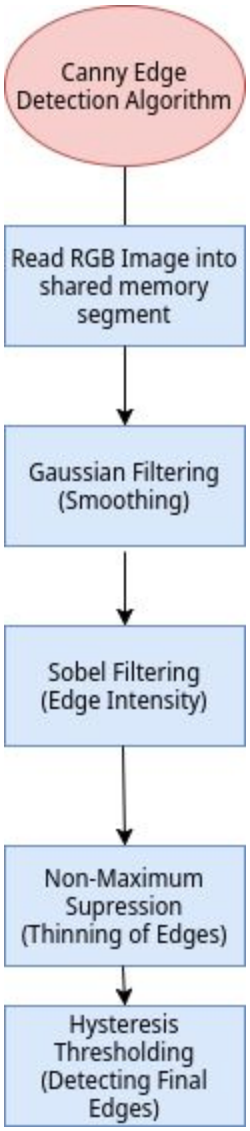
Figure 4.4: Sequential time vs multiple pixels per thread GPU algorithm; 32 threads.

When this same test was ran using multiple pixels per thread instead of just one, the increase in performance went up significantly again, over 10x improvement (Jackson 29).

Both studies showed different implementation and different hardware usage but yielded considerable performance gains when implemented in parallel.
Through this project, we have accomplished many of the goals we set. Ultimately, a better understanding of parallel computing both on paper and in practice. We have created a successful canny edge detection algorithm in parallel, that takes an input image, and outputs the image edge

detected. For comparison, a serial implementation of the edge detection algorithm was created to compare to the output image of the GPU implementation, to confirm they are the same. Through both the CPU and GPU implementation, the time to compute is recorded as well, to show performance gain.

Design and Optimization Approach



All of the portions of the code were parallelized. The biggest issue was implementing shared memory/(attempting) at textured memory for the gaussian filtering. The Gaussian Filtering effectively used shared memory, stencil operation, and tiling in order to improve efficiency. The last 4 steps used just a global Idx which corresponded to the threadIdx, blockDim, and blockIdx.

The last 4 steps were very easily parallelizable as the global Idx could perform all the operations with their neighbors also being located in the same thread block. If I had time, I was planning to attempt to use shared memory on the final four steps as well to add complexity and improve efficiency. Unfortunately, we ran out of time to fully implement this and will have to do this at a later date.

Intensity Gradient:

- Take Current Pixel(Global Idx)
- Derive R, G, and B in x and y direction (if applicable)
- Write dx and dy[globalIdx] into global memory

Non-Max Supression:

- Calculate Gradient Angle
- Find neighboring pixels from gradient angle and interpolate into two magnitudes
- Check if any of the magnitudes are greater than the current pixel
- If greater, suppress pixel, otherwise, leave it
- nms[globalIdx] = result

Hysteresis

- High Pass
    - Check if current pixel is above threshold, if it is park as a strong edge
- Low Pass
    - Check if current pixel is strong
    - If strong, check if neighbors are above low threshold if applicable
    - Write results to global memory

Application Performance Analysis and Project Results

Edge detection run time is averaged from 5 runs to reduce the effect of outliers.

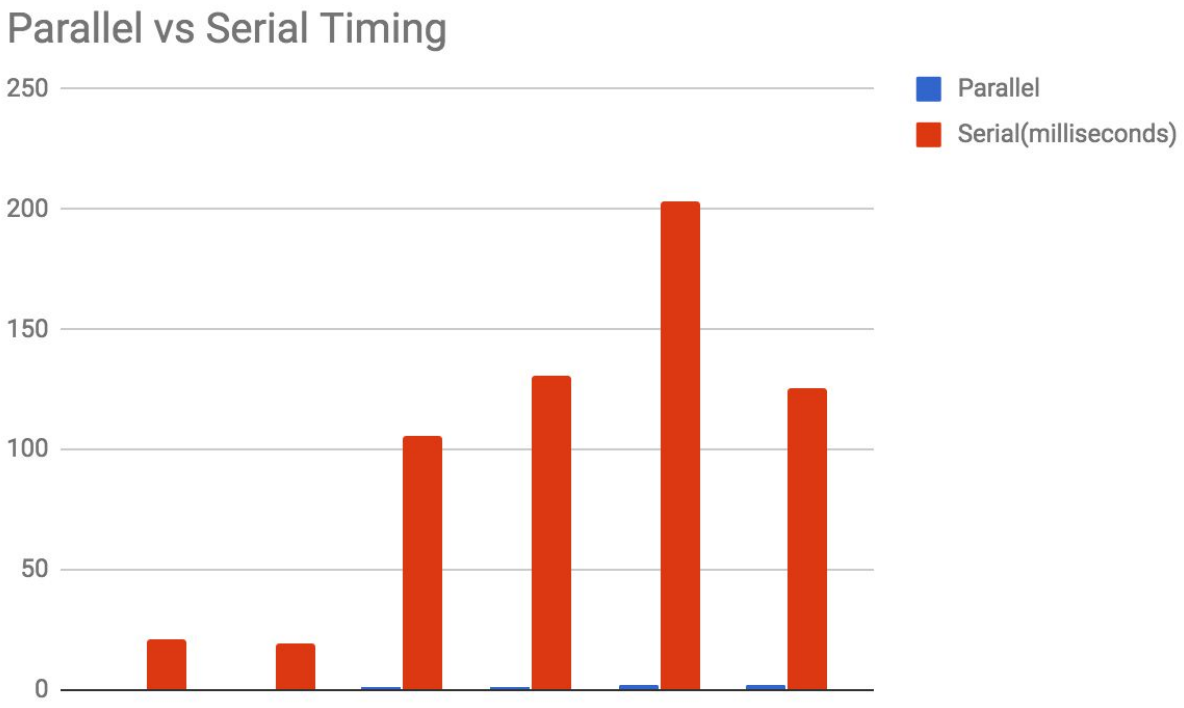
$$\text{Performance Gain} = \frac{(Parallel - Serial)}{Parallel} * 100\,\%$$

Run	Dimensions	Pixels	Size	Parallel	Serial(milliseconds)	Performance Gain
1	400x400	160,000	8 KB	0.197792	20.9	**10466.65588**
2	100x100	10,000	172 bytes	0.109664	19.2	**17408.02451**
3	1920x1080	2,073,600	884 KB	1.4932	105.5	**6965.362979**
4	1920x1080	2,073,600	679 KB	1.516992	130.7	**8515.734295**

5	2231 × 1487	3,317,497	3.5 MB	2.2111	203.3	**9094.518565**
6	2880 × 995	2,865,600	370 KB	1.89334	125.6	**6533.779459**
7	11184 × 16144	180,554,496	98 MB	101.610596	9835.43	**9579.531847**
8	7787 × 11601	90,336,987	103 MB	54.244385	5252.8	**9583.582918**

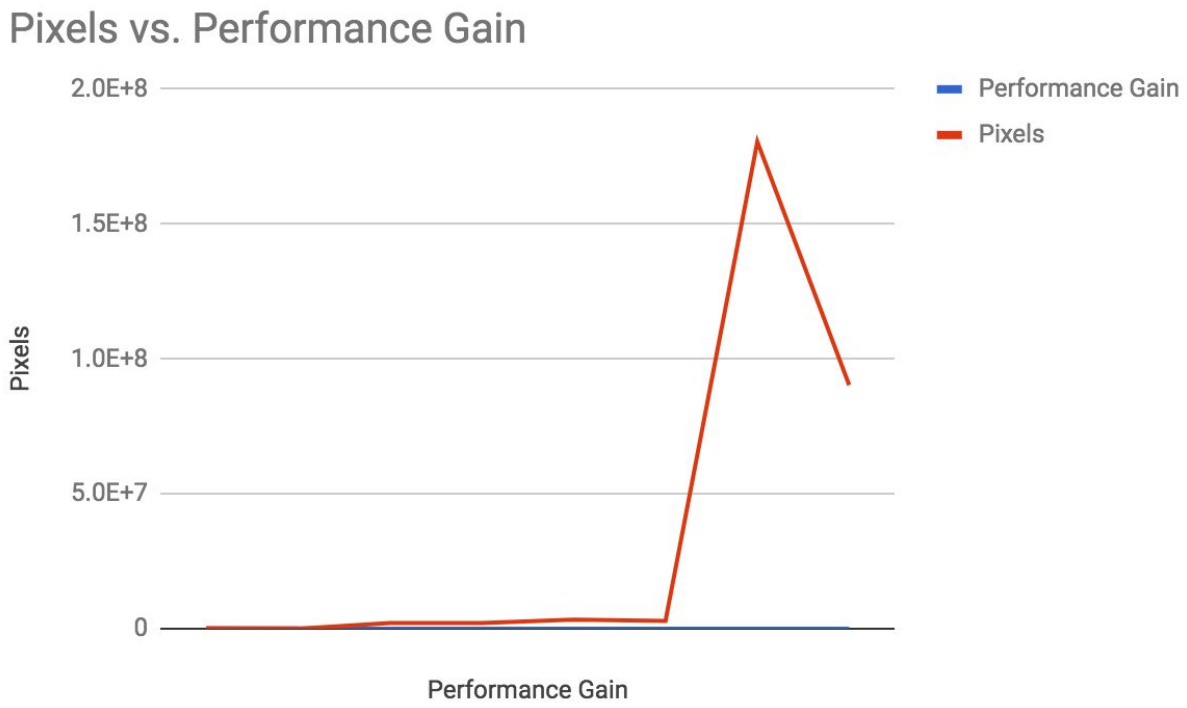
Results:

The results showed a large performance gain when implementing the edge detection solution in parallel. It was expected that small images of size 100x100 pixels would actually be faster on the CPU due to the increased overhead of a parallel process. However, this was not the case as the parallel implementation showed significant performance gain at each stage, for each image size and complexity.



Above shows the results of the parallel vs serial when the parallel results are far under the serial results. (Leaving off the extremely large image data runs due to skewing visuals with large scales on the charts)

As the number of Pixels go up, the relative performance gain does not. It stayed around the same factor of gain throughout each image.



Potential Improvements:

Potential improvements on Uncanny would be to use more modern smoothing and filtering techniques. Some have shown considerable performance gain over the Gaussian Filter for the smoothing step of the algorithm, with the ability to remove just as much noise from the photo. Another potential improvement would be to further build out the image comparison program to detect smaller changes in images but still show their similarities. This could also be parallelized and built out and better optimized for machine vision and movement detection by noticing small changes between before and after images.

Image Comparison Explained:

The basic functionality of the comparison algorithm is first to load the two images being compared to image arrays, provided by OpenCV libraries(OpenCV Docs, Arrays Section). From there we can obtain the size of each image using their height and width components. The first check is if pictures are the same size if not the comparison returns false. However, if they are the same size, the images will be compared using an equals function outputted into an output image. Where if each corresponding pixel is equivalent, the resultant pixel will be 255 (White). If each pixel is the same, i.e., the images are identical; the resultant output image will be entirely white. From here, we

can iterate through the entire image and check the value of each pixel and calculate the percentage of corresponding pixels.

```
Compare(inputImageCPU, inputImageGPU, outputImage, =);
```

The shortcomings of the comparison function is the following.

1) An image shifted 1 pixel off from itself could return 0% alike.
2) Zooming in on an image and comparing it to the original image could return 0% alike. Does not account for “picture in picture”.
3) Tilting/ rotating an image and comparing it to the original image could return 0% alike.

Therefore the comparison functions could be expanded on to handle all cases, but for our edge detection algorithms, comparing the two output images for percent similarity will suffice.

Division of Work + Contribution Essays

Benjamin Horn

- Serial implementation of the canny edge detection algorithm
- Project Documentation
- Performance comparison between serial and parallel runs
- Implement image comparison to check CPU vs GPU output images

Chris Klammer - Responsibilities

- CUDA implementation for Canny Edge Detection algorithm\
- Review Project Documentation
- Performance comparison between different edge detection algorithms (if needed)

Benjamin Horn - Contribution Essay

Throughout the project, we have both made contributions to the final product logistically, technically, and through the implementation. Logistically, my responsibilities have been to create and start the powerpoint class presentation, create and begin the final report, create charts and comparison graphs for the last outputs, and to create and format the GitHub repository. The project's technical aspect had burdensome requirements as well. I researched the canny edge detection algorithm to gain a better understanding of its steps and the complexity of each. Research was also done on current implementation and ongoing research concerning edge detection algorithms and edge detection algorithms in parallel. Lastly, my responsibilities for project implementation were to create a comparison script to compare the output images from the GPU implementation and the CPU implementation. Also, to create the serial implementation, with could take in an image and output the image without using parallel and Cuda utilities.

Chris Klammer - Contribution Essay

Throughout the project, we have both made contributions to the final product logistically, technically, and through the implementation. Logistically, my responsibilities have been to finish and submit the powerpoint class presentation, finish and submit the final report, complete charts and comparison graphs for the last outputs, and to finalize and format the GitHub repository. The project's technical aspect had burdensome requirements as well. I researched the canny edge detection algorithm to gain a better understanding of its steps and the complexity of each. Research was also done on current implementation and ongoing research concerning edge detection algorithms and edge detection algorithms in parallel. Lastly, my responsibilities for project implementation were to create the parallel CUDA implementation of the canny algorithm. This meant taking each of the steps the edge detection algorithm goes through and parallelizing them on the GPU.

Bibliography

"Edge Detection." MATLAB & Simulink, www.mathworks.com/discovery/edge-detection.html.

"Canny edge detector." Wikipedia, Wikimedia Foundation, 28 Nov. 2017, en.wikipedia.org/wiki/Canny_edge_detector

Jackson, Alexander Lee: "A PARALLEL ALGORITHM FOR FAST EDGE DETECTION ON THE GRAPHICS PROCESSING UNIT". Honors Thesis to Washington Lee University, https://pdfs.semanticscholar.org/9c1e/43ca873edf3193a22f63f915e5918b0de8db.pdf

Yuancheng "Mike" Luo and Ramani Duraiswami, Canny Edge Detection on NVIDIA CUDA, 2008, Computer Science & UMIACS, University of Maryland, College Park, http://www.umiacs.umd.edu/~ramani/pubs/luo_gpu_canny_fin_2008.pdf

CUDA Programming Guide. NVIDIA Corporation, Santa Clara, CA, 2009

"OpenCV Documentation Index." OpenCV Documentation Index, docs.opencv.org/.

"CUDA Programming." Sobel Filter implementation in C, cuda-programming.blogspot.com/2013/01/sobel-filter-implementation-in-c.html

Useful Large Photo Library: https://commons.wikimedia.org/w/index.php?title=Category:Large_images#mw-category-media

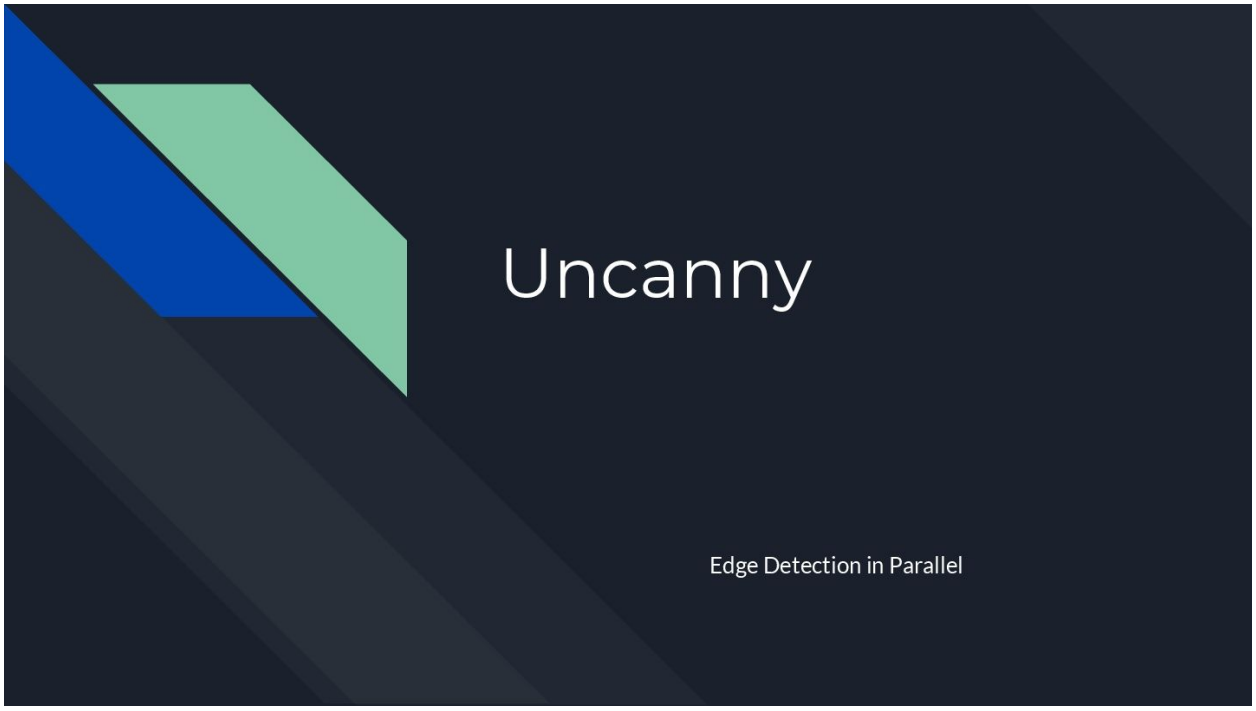
"Snow Leopard." Wikipedia, Wikimedia Foundation, 1 Dec. 2017, en.wikipedia.org/wiki/Snow_leopard

Example of code in CUDA: https://github.com/kinap/canny-edge-detector

Code Appendix - this section is complete, we just need to add code to this repo

https://github.com/hornbd96/uncanny

Presentation Slides



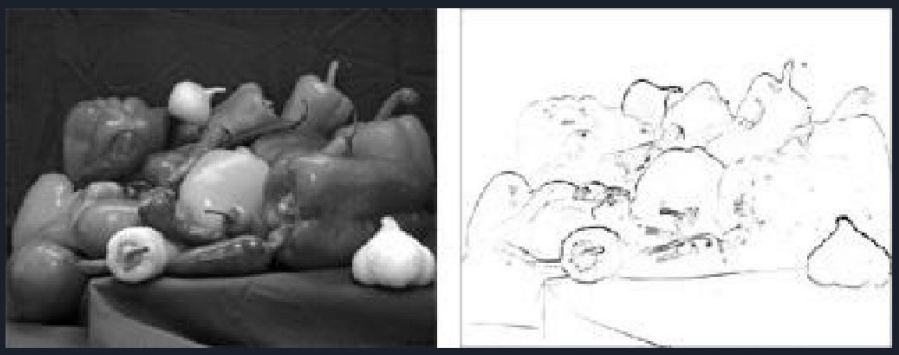
What is Edge Detection?

Determining the edges within an image by Identifying sharp changes in brightness.

Sobel Method

Irwin Sobel and Gary Feldman

Fuzzy Logic Method



Canny Method (Gaussian)

Canny Method Continued

Developed by John F. Canny in 1986.

- Apply Gaussian filter for image smoothing
- Find the intensity gradients (Horizontal / Vertical / etc)
- Thin the detected edges with non-maximum suppression
- Remove values without a strong gradient value
- Suppress all the other edges that are weak and not connected to strong edges.

?	?	?
?	Weak	?
?	?	?

Why is Edge Detection Important?

- Image Processing
- Machine/ Computer Vision
  - Identifying Movement
- Filtering out unneeded object from images

Uncanny - Parallel Edge Detection

- Serial Implementation
  - Output: Edge detection image and time the process took
- Parallel Implementation
  - Output: Edge detection image and time the process took
- Comparison Script
  - How accurate the 2 images are between the serial and parallel
- Documentation and Visuals of Outputs
  - Performance gain is dependent on image size/ complexity

Uncanny - Parallel Implementation

- Read RGB Image into shared memory space
- Perform Gaussian Filtering
  - Utilizes tiling, shared memory, and stencil operation
  - Works on smoothing the image
- Intensity Gradient
  - Derivative of color contrast with respect to x and y in adjacent images
  - Example: (where idx = blockIdx.x * blockDim.x + threadIdx.x)
  - deltaX_channel[idx] = (int16_t)(0.2989 * deltaXred + 0.5870 * deltaXgreen + 0.1140 * deltaXblue)
- Non-Maximal Suppression
  - Center pixel suppressed if it is not the max of its neighbors (in the gradient direction)
- Hysteresis Thresholding
  - One thread per pixel according to a globalIdx
  - First Pass: Check if each pixel is above a “high” threshold, if it is, considered a “strong” pixel
  - Second Pass: If the pixel was “strong” on the first pass, we check its 8 neighbors and if they are above a low threshold, we mark them as “strong” connected through a “strong” edge

Sobel Implementation (Alternate Operator)

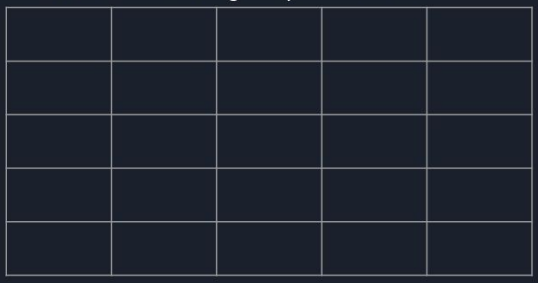
- Uses two 3x3 kernels, one in the x direction one in the y direction
- Calculates some gradient value finding the magnitude from our components
- Efficient but is not a high accuracy edge detector

$$\mathbf{G}_x = \begin{bmatrix} +1 & 0 & -1 \\ +2 & 0 & -2 \\ +1 & 0 & -1 \end{bmatrix} * \mathbf{A} \quad \text{and} \quad \mathbf{G}_y = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} * \mathbf{A}$$

Source: http://www.egr.msu.edu/classes/ece480/capstone/fall13/group04/

Potential Bottlenecks

- Keeping the entirety of the image in global memory
  - Includes the copying from shared memory to global memory
- Edge pixel candidates must be stored in global memory for Non-Maximum Suppression
- How we tile the image for each thread block to work on
- Issues with synchronization between thread blocks when connecting components in the final step



Stretch Goals

Performance Comparison between Multiple Algorithms

Deeper comparison of performance based on image size/ complexity

	Canny	Fuzzy	Sobel
Parallel	3s	4s	2s
Serial	4s	8s	10s