Final Project

EEE5640

Omar Riaz

**Proposal**

My final project will be an evaluation of various image processing algorithms using CUDA. I will be focusing on filtering algorithms, comparing both performance and results. The goal of the project will be to answer the question of which filtering algorithms to choose for a CUDA application, and how from-scratch implementations compare to the NVIDIA NPP library, specifically NPPIF, which contains filtering and computer vision functions.

**Workloads**

1. Median Filter

The median filter, which replaces each pixel with the median of its neighboring pixels, is an efficient and flexible filter for removing noise or image graininess.

1. Gaussian Filter

This filter like the median filter is a convolutional operator that instead of taking a median, applies a gaussian kernel. It is used to blur images or remove noise.

1. Laplacian Filter

An approximation of the second order derivative of the image is used to calculate the Laplacian of the image in this filter, in order to highlight sharp details or edges. The Laplacian filter is often used for edge-detection.

1. Sobel Filter

Also known as the Sobel-Feldman filter, this filter is used for edge-detection like the Laplacian filter. It uses two kernels convolved to approximate the horizontal and vertical derivatives of the image.

Additional filters like high-pass or low-pass can also be evaluated as an extra credit opportunity.

**Dataset**

Images from the CIFAR=100 dataset will be used for input to the filters. This dataset has only photo-realistic images that are commonly used for image recognition and convolutional neural network training since the images are labelled. Because these filters are commonly used along with computer vision applications, it makes sense to use a dataset of images also commonly used in CV.

**Platform**

For this project I will use the Discovery cluster to run my CUDA programs, attempting to make use of one NVIDIA GPU node. For an extra credit opportunity, I can also try running my program on two different NVIDIA GPU models. I will use both the latest versions of CUDA and GCC available on Discovery and compile my programs with nvcc.

**Results to Collect**

I will generate images and record execution time for each filtering algorithm evaluated. These results will be compared in the final project document. Below is an overview of the expected grade based on the results collected.

A: Implement and evaluate 3 filters, using 3 input images, compare results to respective NPP functions, provide low-level analysis on why NPP is faster, revise and improve my implementations after review

A-: Implement and evaluate 3 filters, using 3 input images, compare results to respective NPP functions, and provide low-level analysis on why NPP is faster

B+: Implement and evaluate 3 filters, using 3 input images, and compare results to respective NPP functions, provide some basic analysis on why NPP is faster

B: Implement and evaluate 2 filters, using 3 input images and compare results to respective NPP functions, provide some basic analysis on why NPP is faster

C: Implement and evaluate 2 filters, using 3 input images

Extra credit: Run on multiple different GPU models, try more filters like low-pass and high-pass

**Final Project**

**Methods**

There were several changes made to the project to improve the results and analysis. Firstly, the sample images used were taken from Google Images and were all the same resolution of 128x128. This was because the CIFAR dataset images had a very small resolution, making the execution time of the filters very low and more difficult to compare. An image of a printed circuit board, an ape, and a pathway with some trees

Secondly, instead of comparing my implementations to Nvidia’s NPP functions, they were compared to a repository of implementations created by Khosro Bahrami, a former machine learning and computer vision postdoc researcher at University of North Carolina and current computer vision engineer at Apple. The reasons for this are twofold. Firstly, the NPP samples were only written for Windows usage and while the library is maintained, the documentation is not as strong as other NVIDIA libraries and many CV engineering teams prefer their own libraries, as discussed on the NVIDIA forums. Secondly, the library isn’t available on Discovery and the aforementioned downsides made an alternative (with code that would be more easily analyzed) preferable. The codes for Bahrami’s implementations can be found in the following repository:

<https://github.com/KhosroBahrami/ImageFiltering_CUDA>

All the filters function off of the same core code for convolution. The kernel is written such that each thread is given the entire input image and computes the result value of one pixel of the output image. The kernel for the soble and sharpen filter, which was the same in both implementations of the filters, are shown below.

Sobel Filter

|  |  |  |
| --- | --- | --- |
| -1 | 0 | -1 |
| -2 | 0 | 2 |
| -1 | 0 | 1 |

|  |  |  |
| --- | --- | --- |
| 1 | 2 | 1 |
| 0 | 0 | 0 |
| -1 | -2 | -1 |

Sharpen Filter

|  |  |  |
| --- | --- | --- |
| -1 | -1 | -1 |
| -1 | 9 | -1 |
| -1 | -1 | -1 |

For both my implementations and Bahrami’s, the block size was 16. All of the filters for each of the implementations were run on the same GPU node. One was a Tesla K40m running CUDA 3.5 with 11441 Mbytes global memory and 2880 CUDA cores. The CPU’s specifications were the following:

Architecture: x86\_64

CPU op-mode(s): 32-bit, 64-bit

Byte Order: Little Endian

CPU(s): 48

On-line CPU(s) list: 0-47

Thread(s) per core: 2

Core(s) per socket: 12

Socket(s): 2

NUMA node(s): 2

Vendor ID: GenuineIntel

CPU family: 6

Model: 63

Model name: Intel(R) Xeon(R) CPU E5-2690 v3 @ 2.60GHz

Stepping: 2

CPU MHz: 1597.387

CPU max MHz: 3500.0000

CPU min MHz: 1200.0000

BogoMIPS: 5199.97

Virtualization: VT-x

L1d cache: 32K

L1i cache: 32K

L2 cache: 256K

L3 cache: 30720K

NUMA node0 CPU(s): 0-11,24-35

NUMA node1 CPU(s): 12-23,36-47

**Results**

The input images, compared to the output images after each filter are shown below.

A close up of a gorilla

Description automatically generated with medium confidence A close up of a monkey

Description automatically generated with medium confidence

Original Image Median Filter

A close up of a gorilla

Description automatically generated with medium confidence 

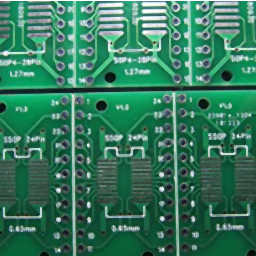
Original Image Sharpen Filter

A close up of a gorilla

Description automatically generated with medium confidence 

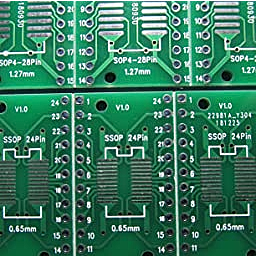
Original Image Sobel Filter

A close-up of a circuit board

Description automatically generated with medium confidence 

Original Image Median Filter

A close-up of a circuit board

Description automatically generated with medium confidence 

Original Image Sharpen Filter

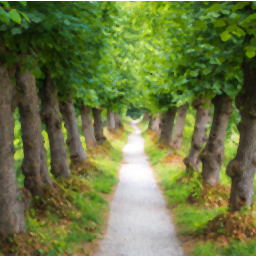
A close-up of a circuit board

Description automatically generated with medium confidence A picture containing diagram

Description automatically generated

Original Image Sobel Filter

**A picture containing tree, grass, outdoor, path

Description automatically generated **

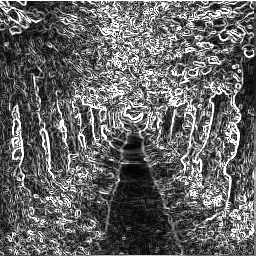
Original Image Median Filter

**A picture containing tree, grass, outdoor, path

Description automatically generated **

Original Image Sharpen Filter

**A picture containing tree, grass, outdoor, path

Description automatically generated **

Original Image Sobel Filter

The execution times for all of these, executed on each of the CPU and GPU were measured and are reported below.

|  |  |  |  |
| --- | --- | --- | --- |
| Filter | CPU | **GPU** | **x Speedup** |
| **Median** | **12.0 ms** | **5.2 ms** | **2.3** |
| **Sharpen** | **13.0 ms** | **5.5 ms** | **2.4** |
| **Sobel** | **10.6 ms** | **4.7 ms** | **2.3** |

Then, Bahrami’s GPU implementations for the median filter, sharpening filter and sobel filter were tested on the Discovery cluster. The execution times of these filters on each of the CPU and GPU are shown below.

|  |  |  |  |
| --- | --- | --- | --- |
| Filter | CPU | **GPU** | **% speedup** |
| **Median** | **2.9 ms** | **0.9 ms** | **3.22** |
| **Sharpen** | **3.0 ms** | **1.1 ms** | **2.72** |
| **Sobel** | **2.3 ms** | **0.7 ms** | **3.28** |

Convolving with the Sobel filter was slower, since the convolution was done on a greyscale image, rather than an RGB one. For the RGB images, each convolution was done separately, essentially being 3 three times as much work. However with the Sobel, conversion to greyscale was necessary. The sharpening filter had the longest execution times across the board. It was almost identically in number of steps to the median filter but required an extra rescaling step. On average, my CUDA implementations of the convolutional filters were still 5.72 times slower than Bahrami’s.

**Conclusion**

Initially, my convolution algorithm involved a sliding window that ran across each image, which was highly inefficient. Computing each pixel directly from the neighboring pixels of the original image was far more efficient, and was what sped up my execution time to be comparable to other standard implementations. Since Bahrami used OpenCV for much of his computation, including the convolution, as well as using datatypes from OpenCV like Mat, which may be more optimized for the image resolution that he was using, meaning extra bits aren’t being transferred between the device and host. A strong next step to improving the performance of my implementation would be to use an FFT-based convolution, which are known to be less computationally complex.