# CUDA Implementation of Morphological Laplacian

## Introduction

This document is a brief overview of the Morphological Laplacian sample task provided to myself, Brad Suchoski, as a sample problem to be completed in C++ and CUDA, OpenCL or OpenGL. I chose CUDA because it is what I have the most experience with and with which I am most familiar. Much of the task could have most likely been done through API calls to NVidia's thrust or NPP libraries. However, I assumed that the point of this task was to test my knowledge and ability to write CUDA kernels so I decided to write out the full kernels instead.

## Task Description and Preliminary Steps

The task to complete was to take an image of my choice and separetly apply a Morphological Laplacian operator to the three color components. Being a Penn State Alumni, I chose this picture of the PSU lion statue.

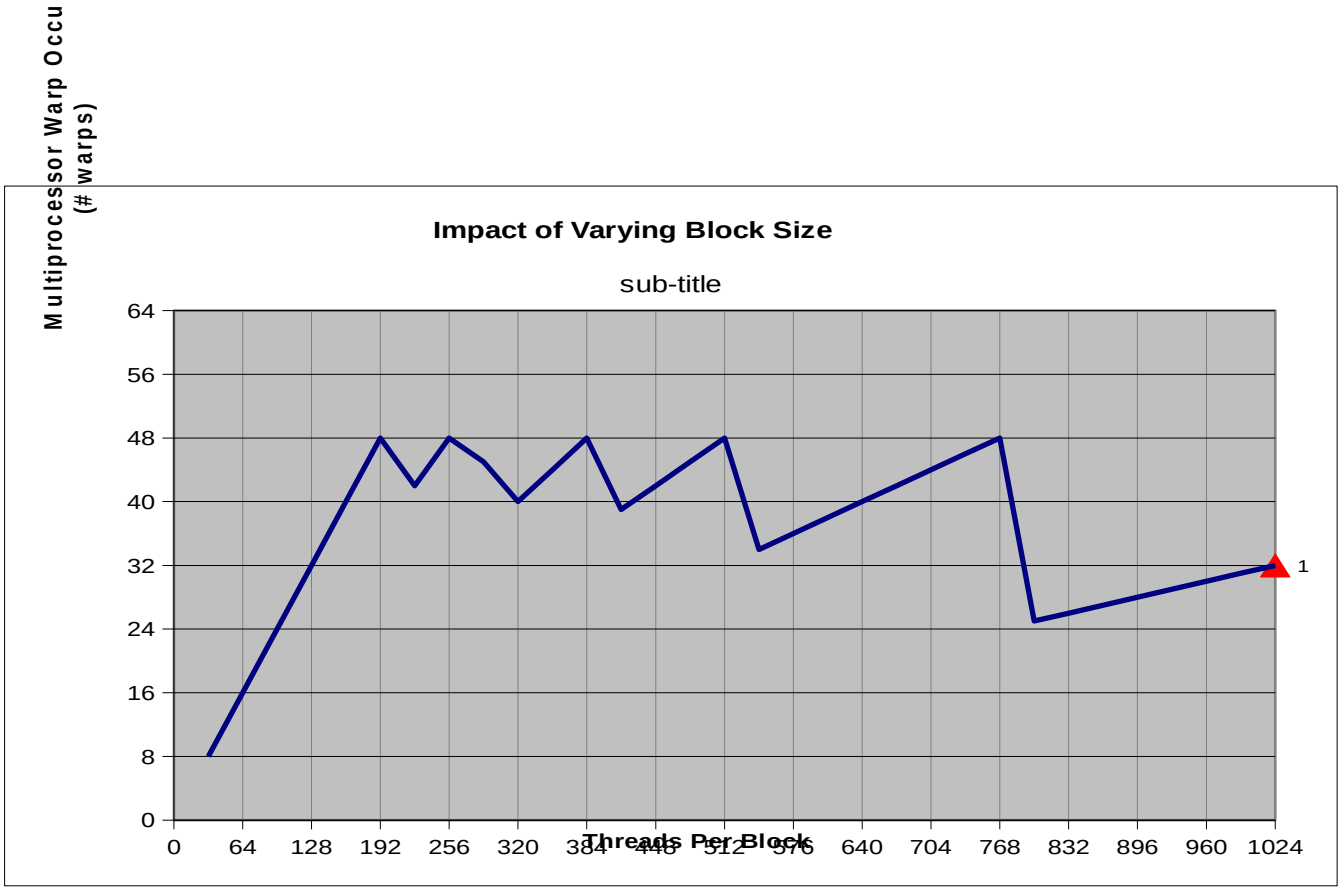
The input image for the task was to be in YUV420P format. The source image above is a jpg file taken from Google. So the first step was to convert it to YUV420P format using the following ffmpeg command

```
ffmpeg -i psulion.jpg -f rawvideo -pix_fmt yuv420p psulion.yuv
```

This converted the source jpg image to the required planar YUV format needed for the program.

# Implementation Details

As stated before, the implementation of the project was done in C++/CUDA. There are three main files. `main.cpp` obviously contains the int `main(...)` function, and is responsible for doing all of the I/O and memory allocation. `yuv2rgb.cu` contains the kernel that convers the input image from YUV420P to planar RGB format and `morph.cu` contains the kernel that applies the Morphological Laplacian operator. Kernel block sizes were determined using the CUDA Ocupancy Calculator




spreadsheet provided with the CUDA installation. The graph above shows the block size vs warp occupancy as calculated by the spreadsheet for the `laplace_morph_kernel`. 384 threads per block was chosen because it is a maximum point near the middle of the block size range which should balance ability to mask memory latency with context switching overhead. Similar optimization was done for the `yuv2rgb_kernel`.

# Test Environment and Hardware

All of the testing was done on my home computer running Arch Linux with an Intel Xeon E3-1230 V2 processor and an old GeForce GT 610 GPU of compute capability 2.1 that I have installed. This simple application would not have benefited much from improvements such as the L1 and L2 cache and new intrisic functions added in later generations of GPU's. However, it obviously would have benefited from the increase in number of SM cores and memory bandwidth. The GPU is so outdated that it was

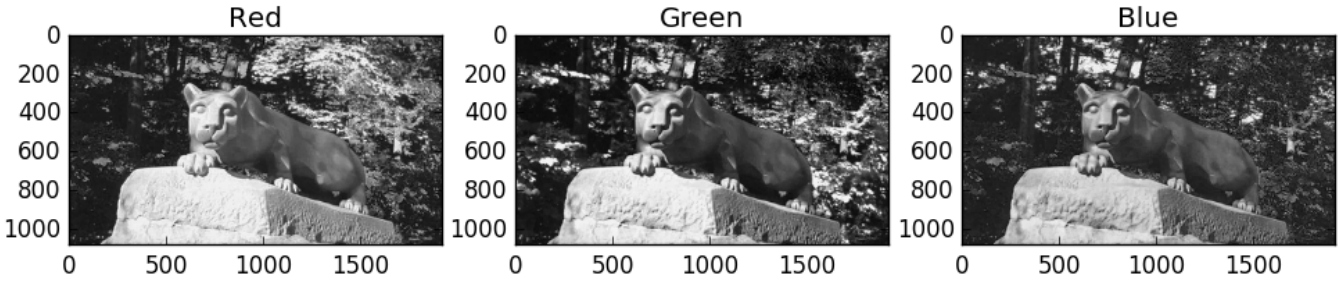
only able to run the v367.27 Nvidia driver which only suppors up to CUDA 6.5. I attempted to run the program on an Amazon AWS GPU instance, but due to time constraints I was not able to get that working.

The device details for the GPU used for testing are as follows

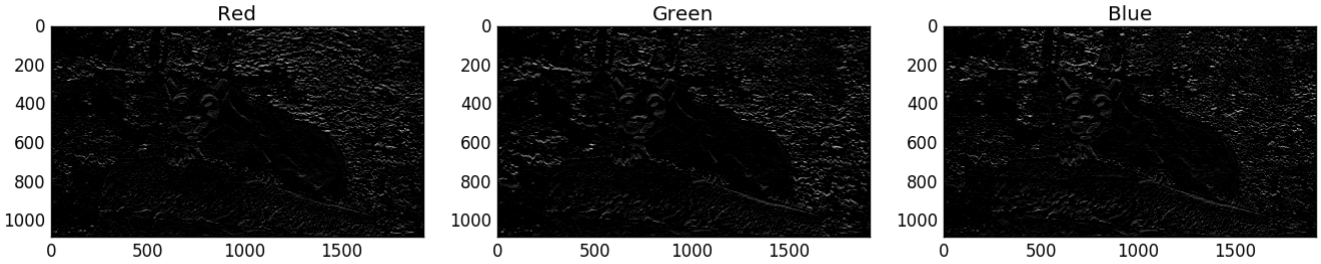
```
/deviceQuery Starting...

 CUDA Device Query (Runtime API) version (CUDART static linking)

Detected 1 CUDA Capable device(s)

Device 0: "GeForce GT 610"
  CUDA Driver Version / Runtime Version          8.0 / 6.5
  CUDA Capability Major/Minor version number:    2.1
  Total amount of global memory:                 957 MBytes (1003945984 bytes)
  ( 1) Multiprocessors, ( 48) CUDA Cores/MP:     48 CUDA Cores
  GPU Clock rate:                                1620 MHz (1.62 GHz)
  Memory Clock rate:                             500 Mhz
  Memory Bus Width:                              64-bit
  L2 Cache Size:                                 65536 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(65536), 2D=(65536, 65535),
  Maximum Layered 1D Texture Size, (num) layers  1D=(16384), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(16384, 16384), 2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 32768
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  1536
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (65535, 65535, 65535)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 1 copy engine(s)
  Run time limit on kernels:                     Yes
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Disabled
  Device supports Unified Addressing (UVA):      Yes
  Device PCI Bus ID / PCI location ID:           1 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice() with device
simultaneously) >

deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 8.0, CUDA Runtime Version
= 6.5, NumDevs = 1, Device0 = GeForce GT 610
Result = PASS
```

# Results and Performance

The intermediate and final results of the project were plotted using python's matplotlib. The scripts for generating them are in the wo python files included. The following two figures respectively show the results of separating the image into it's three color components




then applying the Morphological Laplacian operator to each component




The recombined image which is shown below is difficult to see in this document, but is included as `FinalResult.jpg`.

The elapsed running time to convert the image from YUV420P to RGB and apply the Morphological Laplacian operator to the three color bands was 53.66ms, which equates to roughly 38,642 MPixels/s or 18.64 1920x1080 frames per second. This is not quite real-time performace for standard 24 frame/s HD video, but this test was done on very outdated hardware.