Project 2 Report – CIS 677

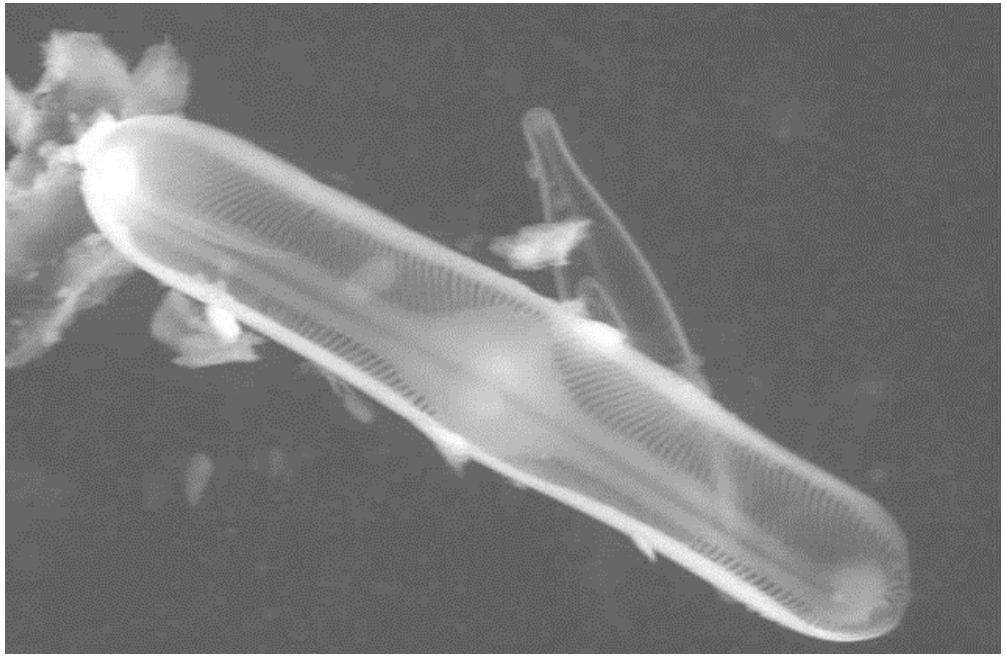
Matthew Lueder

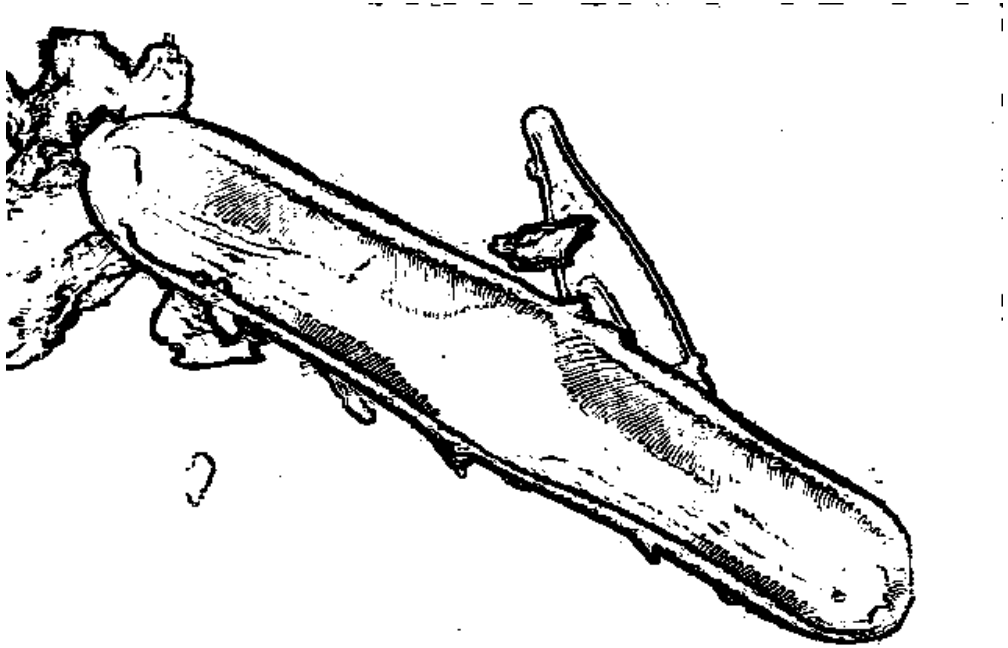
Outline

- The purpose of this program is to test the affects of various degrees of multi-threading on different platforms. We use the task of edge detection to do this.
- Written in C++
- Uses std::thread multi-threading library (in C++11 standard library)
- std::mutex used in some instances to test the effects of only allowing one thread to access shared resource at a time
- Boost::timer used for recording elapsed time on Windows, Gettimeofday() used for recording elapsed time on Unix
- The program was tested on two platforms:
  - A Lenovo Y-50 laptop with an intel i7-4710hq processor.
    - Windows 10
    - 4 physical cores, 8 logical cores (hyper-threading)
    - 2.50 Ghz
    - 16 GB RAM
  - "Okami" – GVSU HPC resource
    - Fedora 24 w/gcc 6.1.1
    - Two 8-core AMD Opterons (16 total cores)
    - 32 GB RAM
- Threshold level can be adjusted easily through a static global variable.
- Only the multithreaded portion of the program is timed. The time it takes to read and write the image is not considered

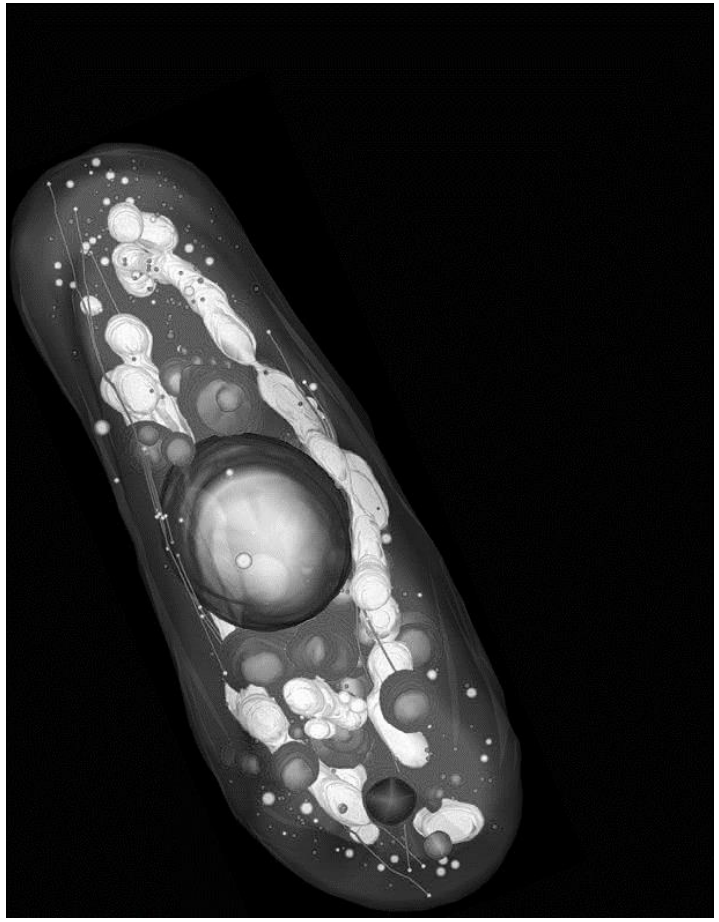
1. Takes path to black and white bmp file that the user would like to apply a filter to and the output path as command-line arguments
2. Loads bmp file into c++ structure, with pixel values stored in a 2D vector.
3. A 1-pixel border is added to the picture as padding, to allow the Sobel operator to be applied to all pixels. The pixel values for the border are based on the values of the adjacent pixels.
4. The number of threads to divide up the work of applying the Sobel operator is varied in a loop. This allows us to compare the time it takes to apply the filter as the number of threads is increased.
5. Every time a certain number for the amount of threads to use is chosen, the filter is applied 3 times in a loop. The time it takes to apply the filter is recorded each time and the number that gets reported is the average. This gives us more accurate estimates.
6. All created threads are given a certain range of rows in the pixel matrix to work on. The range is based on the number of threads (the image is divided up evenly, with the main thread getting the remainder).
7. Each thread goes through every assigned pixel, applies the Sobel operator with thresholding, then puts the output into a shared output image.
8. A mutex that can be switched on and off is included to allow us to see how making it so only one thread can access the shared output image at a time effects performance.

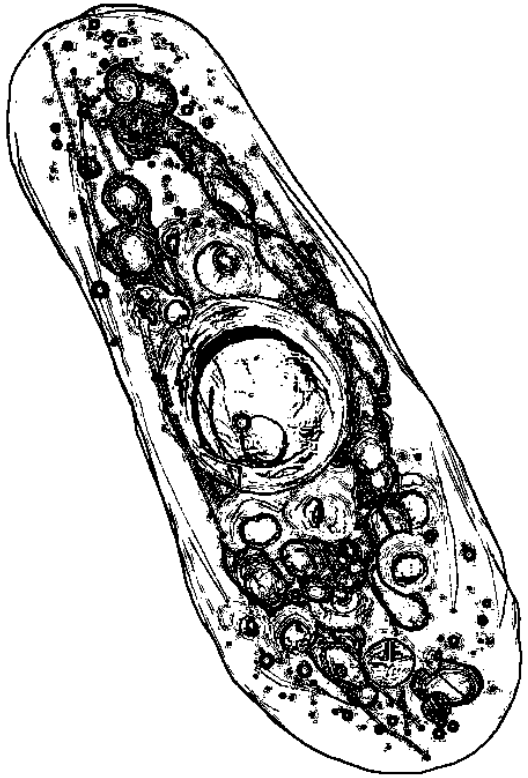
Results - Images



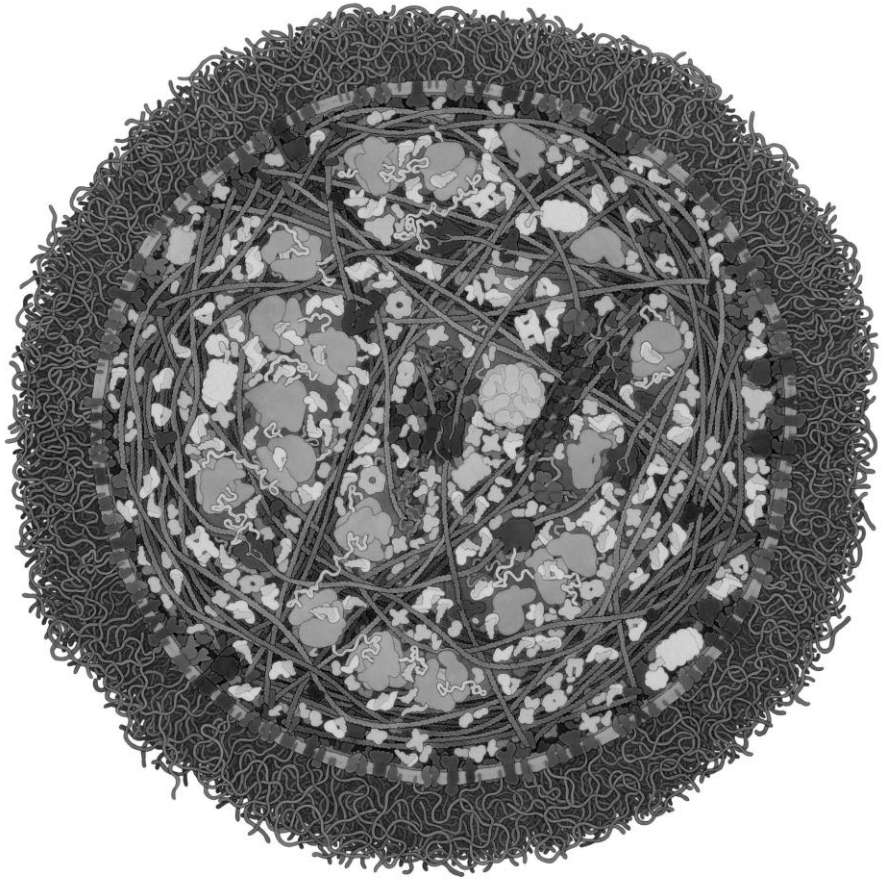


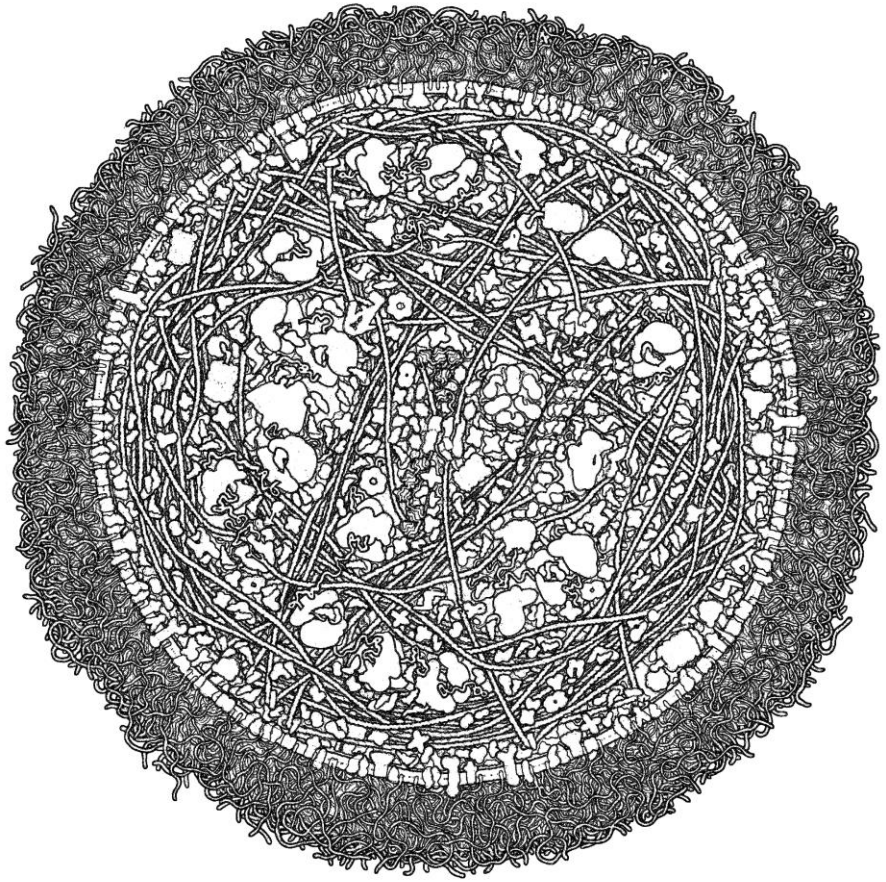
Results - Images



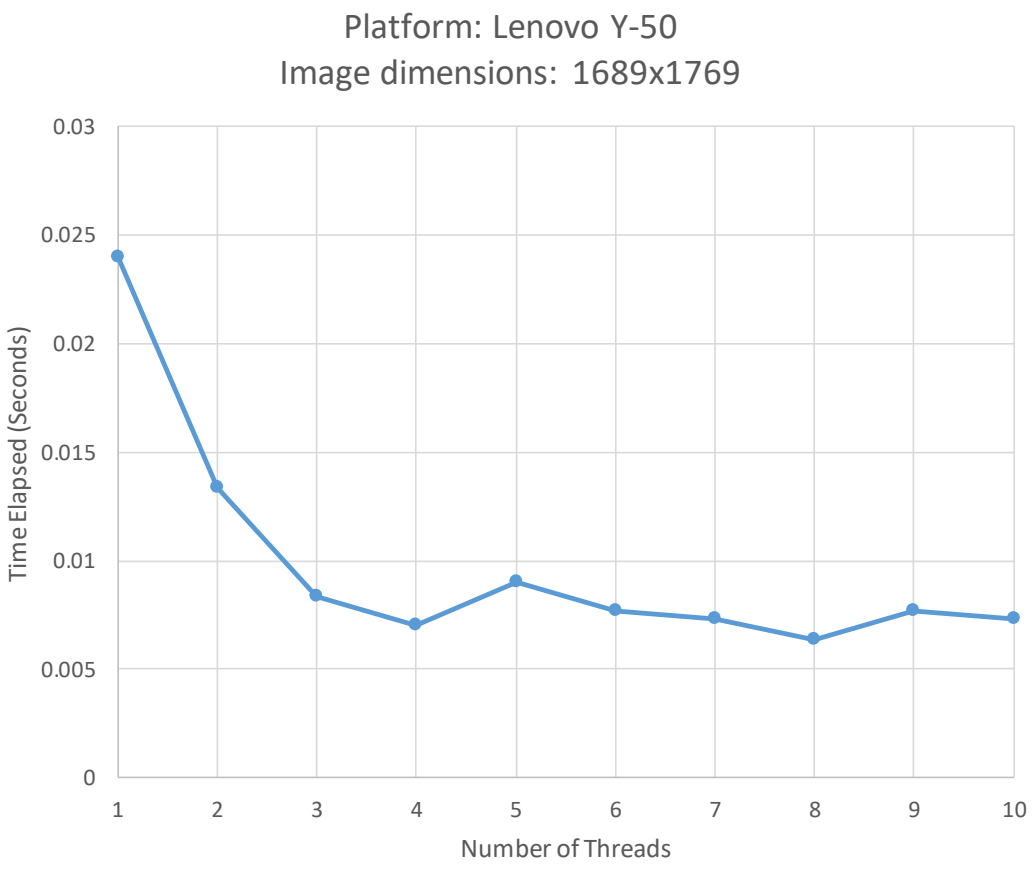


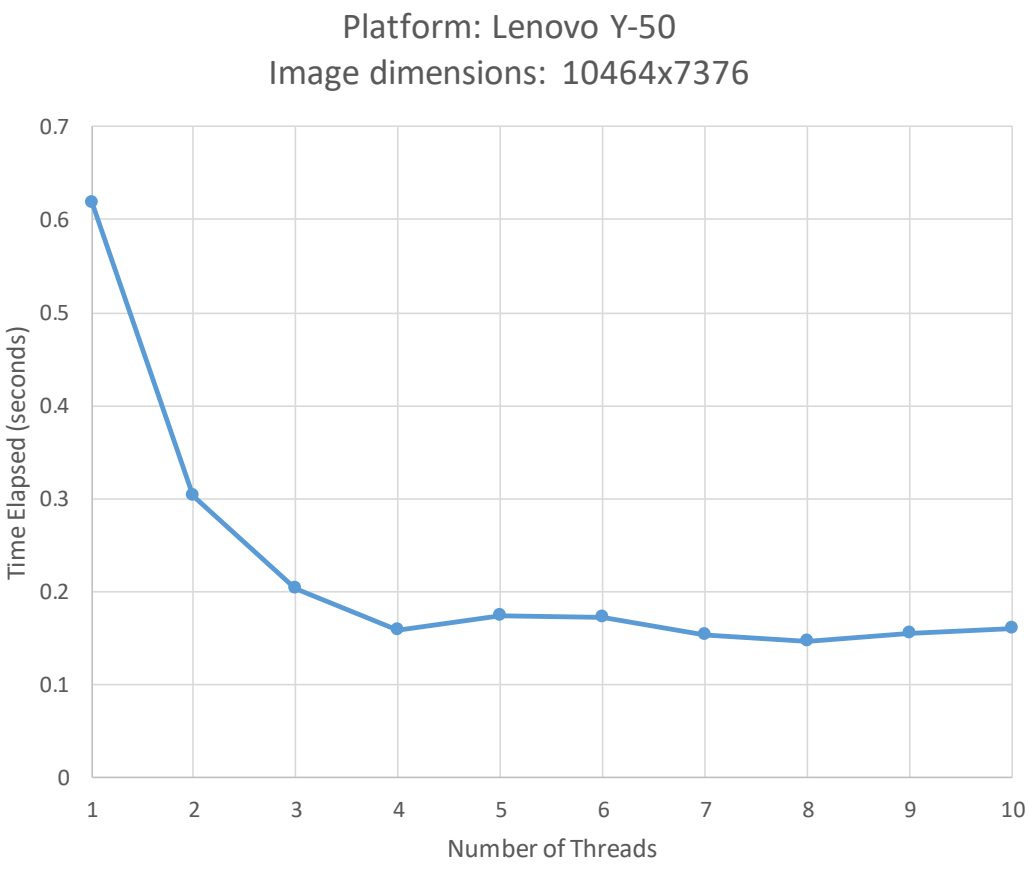
Result - Images



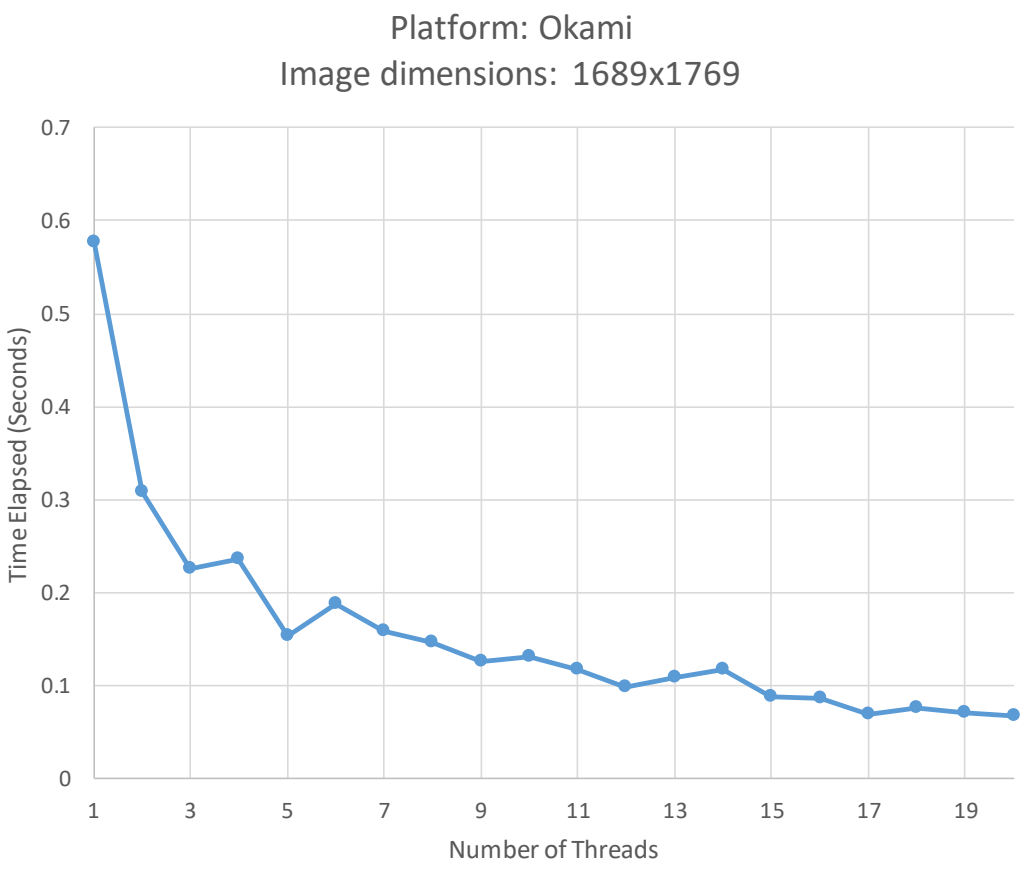


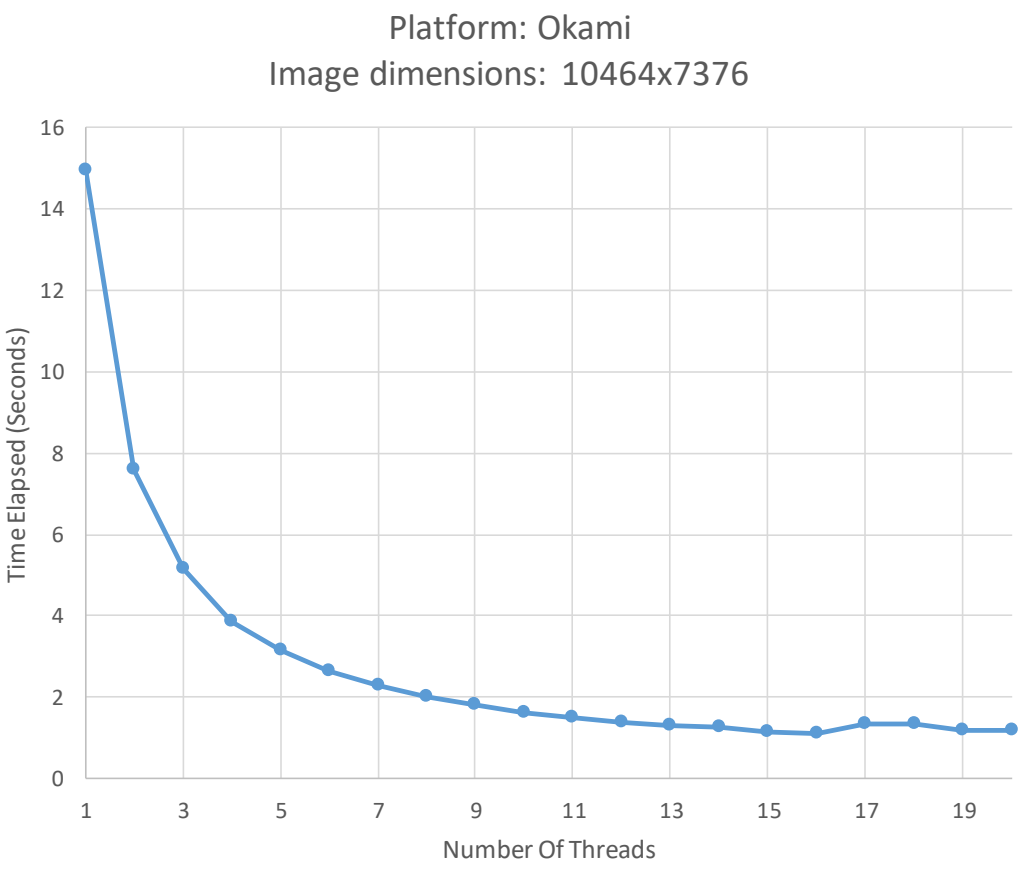
Results – Lenovo Y-50



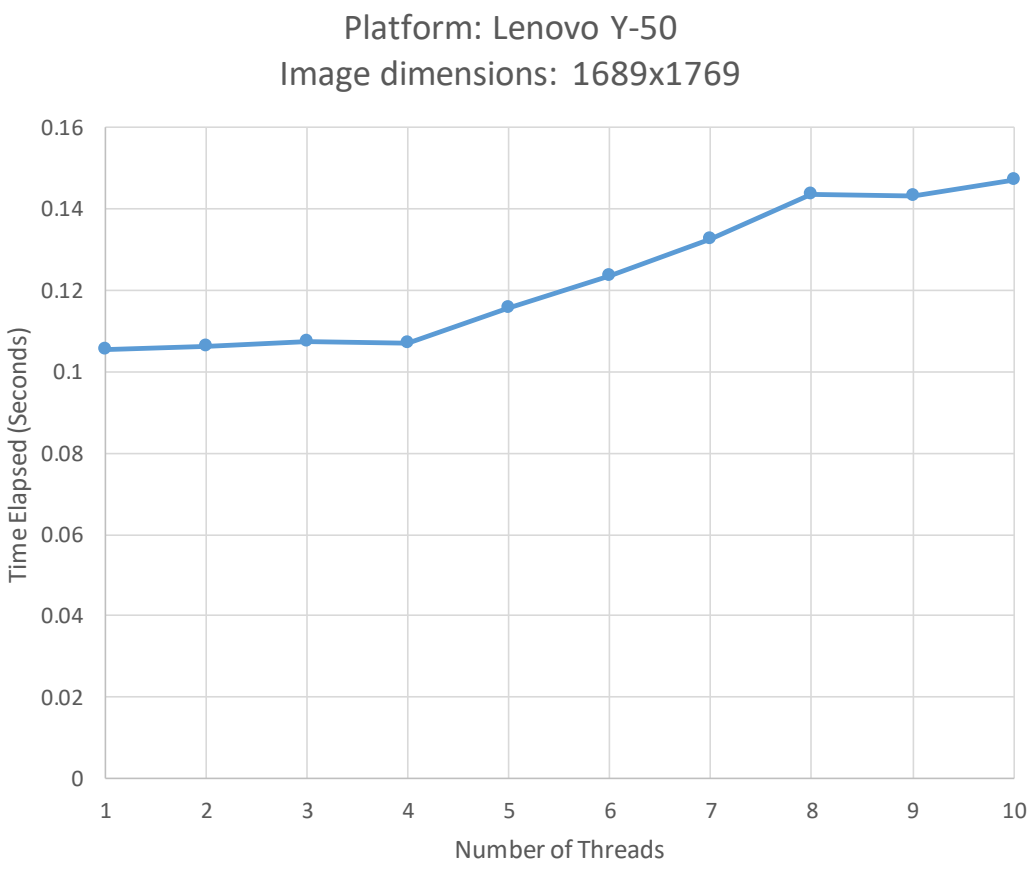


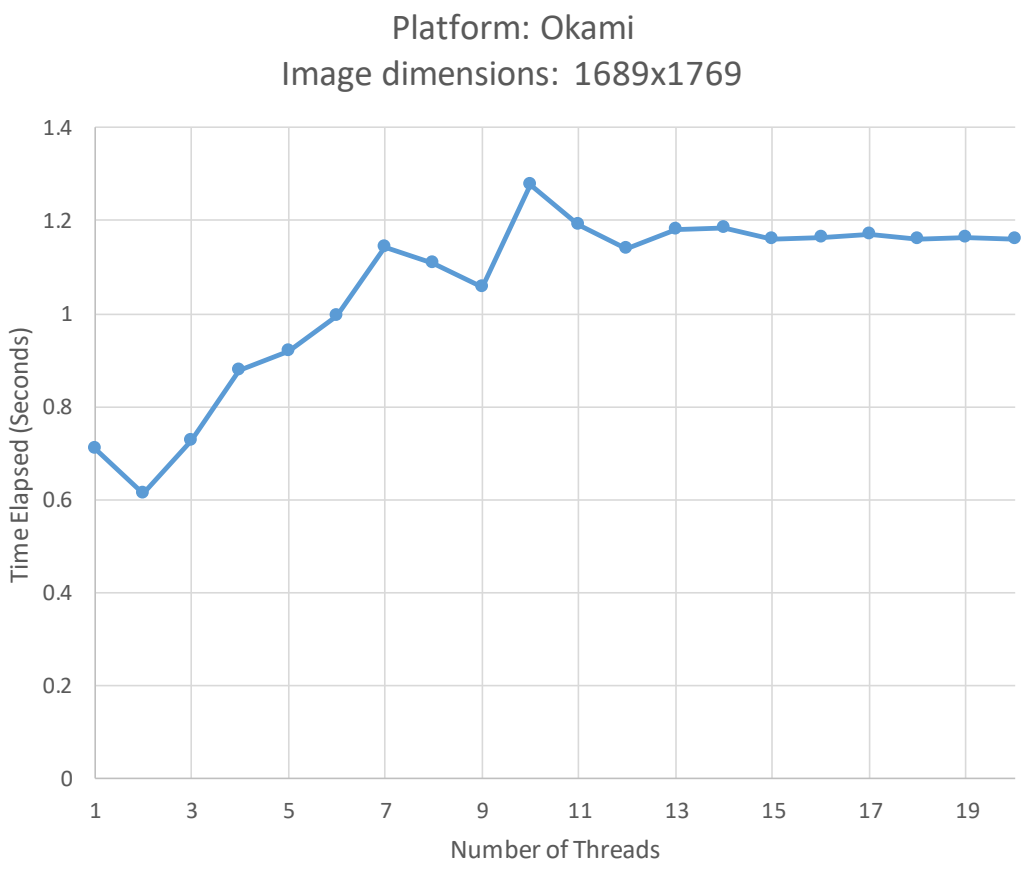
Results - Okami





Results – With Mutex





Observations

- The performance gained by adding another core seems to decrease exponentially with each thread added.
  - Going from one to two threads resulted in the largest speedup (around 2x)
  - Going from two to three threads resulted in an approximately 1.5x speedup
  - Going from three to two threads resulted in an approximately 1.25x speedup
- Different file sizes seemed to yield similar results however larger file sizes resulted in a smoother curve.
- On the Okami platform performance increased with an increased amount of threads until the number of threads exceeded the number of cores (16). On the Lenovo, the same thing occurred in relation to the number of physical cores (4). This is interesting because the Lenovo is hyper threaded and has 8 logical cores, yet going over 4 threads results in decreased performance.
- When the shared output image that each thread writes to is protected by a mutex so only one thread can write to it at a time, performance decreases with an increased number of threads. This occurs even with large image sizes because each thread attempts to write to the output after calculating the value for a single pixel. The performance of the program with the mutex would be greatly increased if each thread stored the pixel values locally and only attempted to write to the output image once. The mutex in this case is unnecessary because each thread writes to a different element in the output image. This was only done to see how it would affect performance.
- There is a significant performance hit when applying the mutex to the single-threaded application. This means that there is a high cost associated with locking and unlocking the mutex.
- My mid-tier laptop seemed to run much faster than GVSU's high-performance computer? I am not sure why this is.