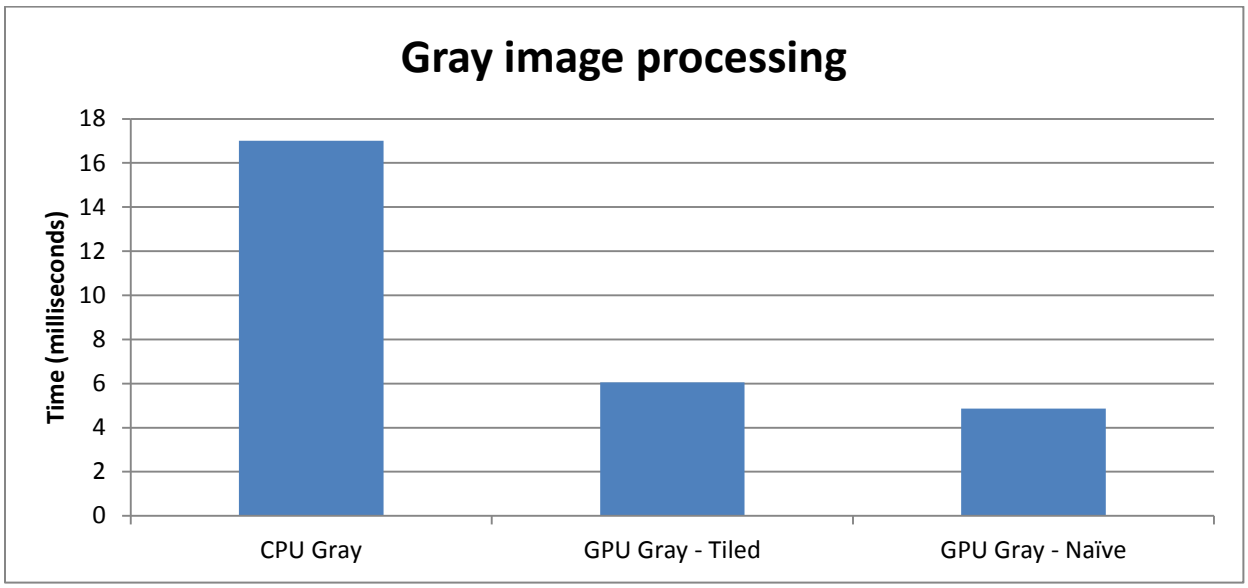
Assignment 3 Report

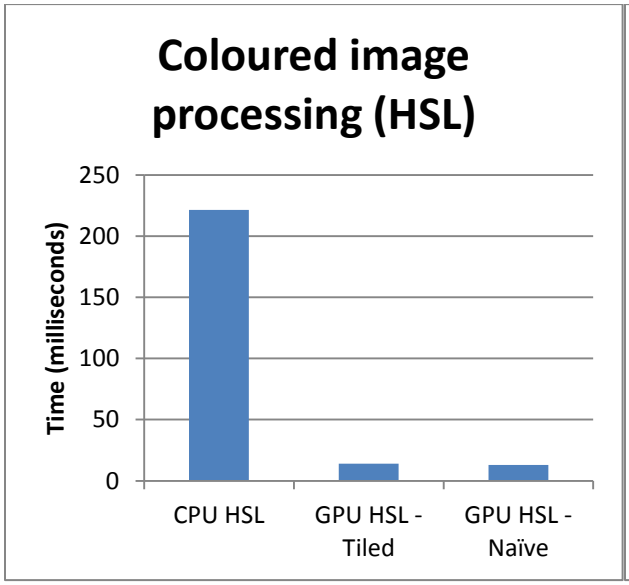
CMPT – 431

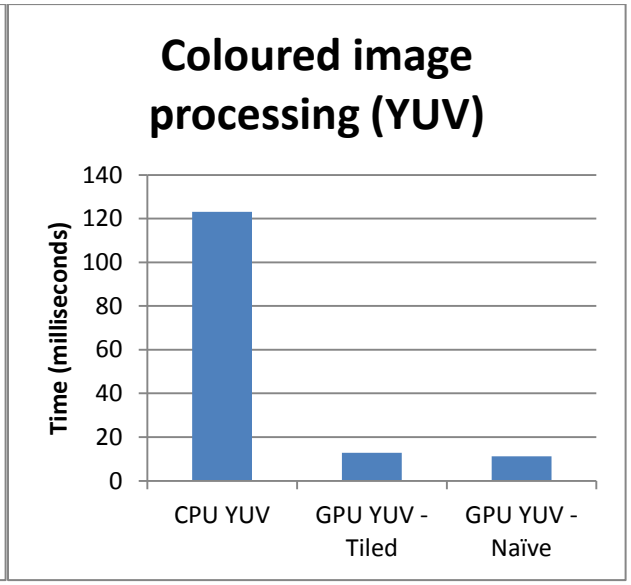
Joshua Campbell - jkcampbe@sfu.ca

Adam Penner – adpenner@sfu.ca

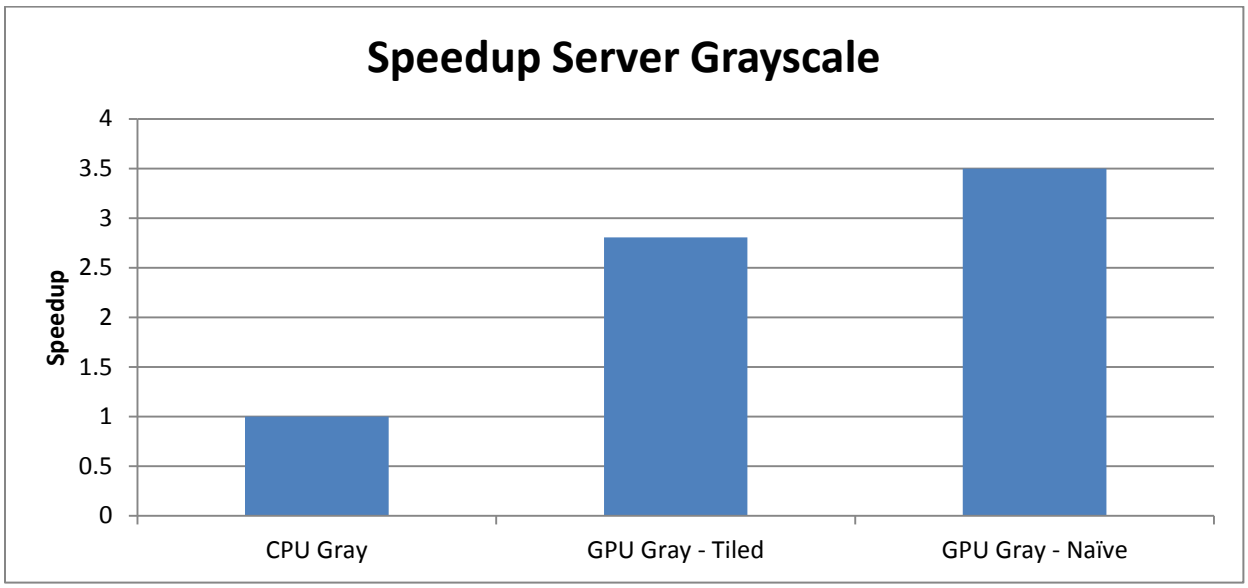
Server Time Graphs:

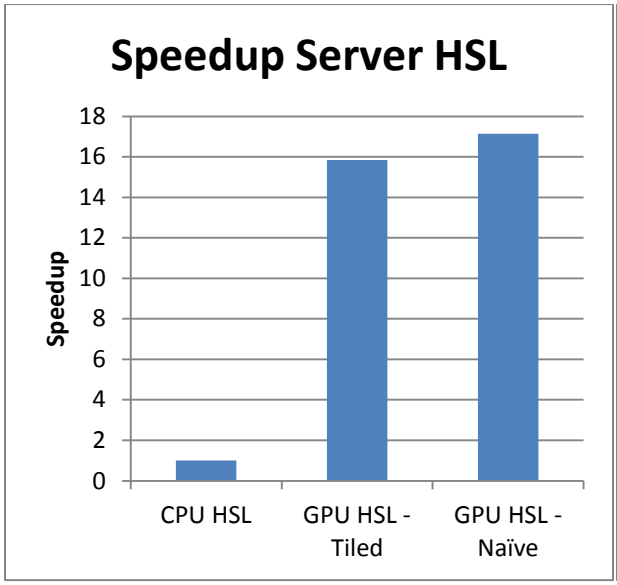


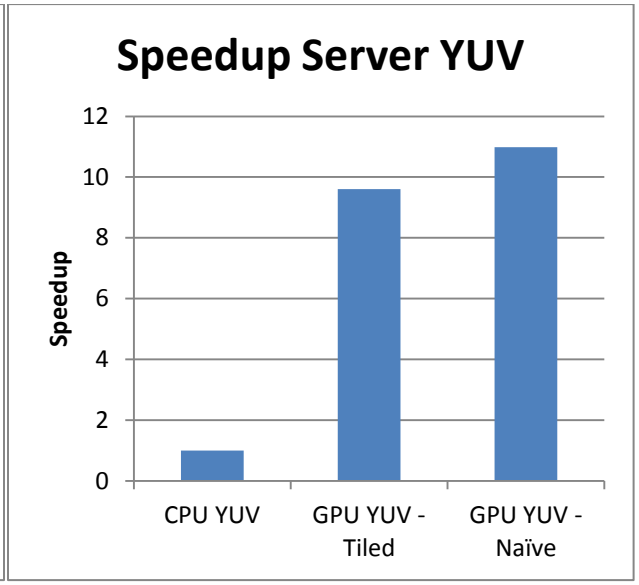




Server Speedup Graphs:





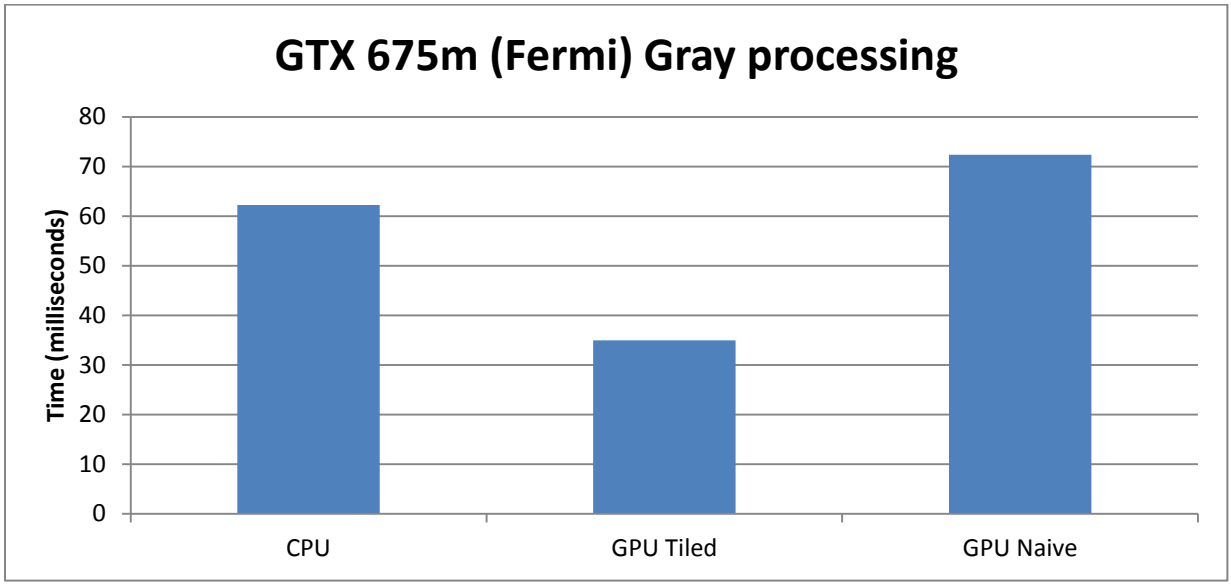


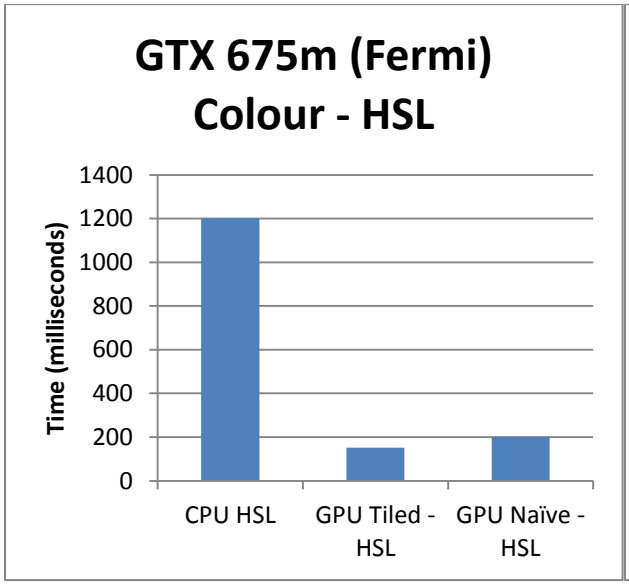
Analysis:

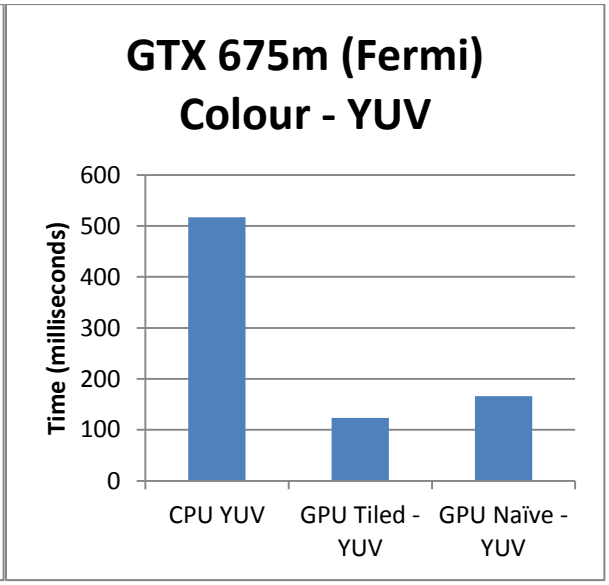
On the server (which appears to be running a GTX 760 (Kepler architecture), we can see that both the Tiled and Naïve implementations of the program are faster than the serial CPU version of the code for all of the image processing techniques. However, the naïve code tends to be slightly faster than the tiled implementation of the code (around a 4-5 millisecond difference). The main area where these two implementations occur is in the calculation of the histogram (the gpu_histogram cuda kernel function in histogram-equalization.cu). This is the only place

that we could find where the structure of the program would possibly allow for a benefit from using shared memory (tiling). However, due to the nature of how the histogram is calculated, you must use atomicAdds in order to prevent race conditions between the threads if they happen to be modifying the same address at the same time in order to get a correct histogram (and eventually a correctly processed image). As we can see from these charts, this use of shared memory provided no performance gains on the server. However, when the same code was run on a GTX 675m (NVIDIA's Fermi architecture), the tiled code executed quite a bit faster (around twice as fast when doing a grayscale image processing) than the naïve implementation (diagrams can be found below). After doing some research into the matter, the NVIDIA Kepler Architecture Whitepaper described some improvements that were made from the Fermi to the Kepler architecture that could explain this huge difference in performance gains. One of the improvements in Kepler described in the whitepaper is that atomic operations such as atomicAdd have received a 9 times performance increase from the previous Fermi architecture (page 12 in the included NVIDIA document). Due to these architectural differences, the improvements seen in Fermi when using shared memory seem to have been optimized out in Kepler through the use of low latency global atomic operations.

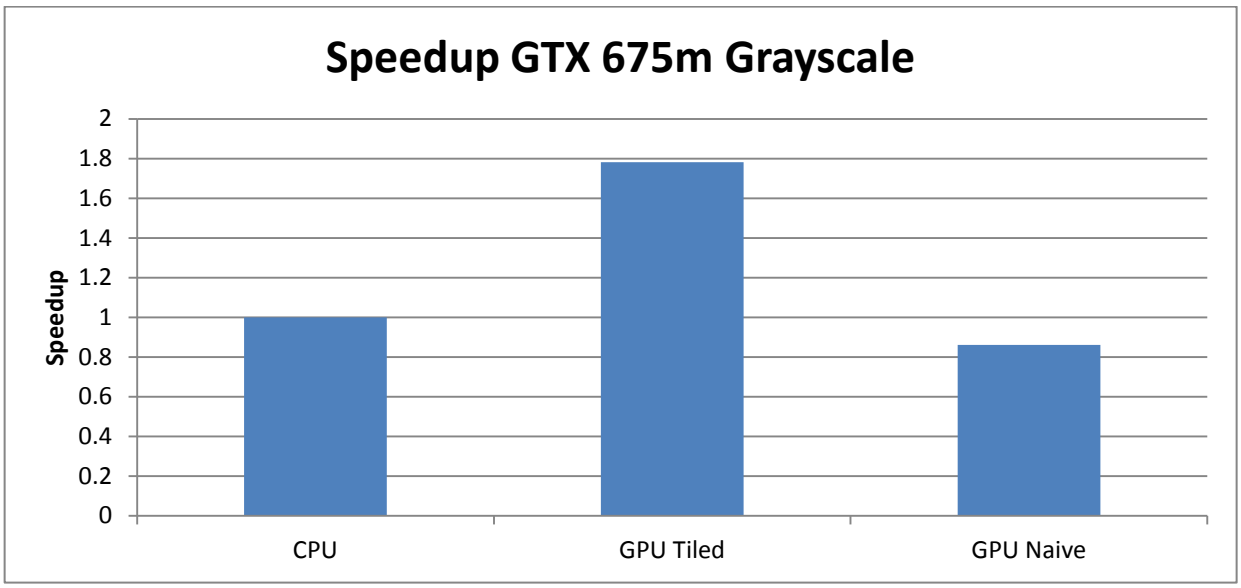
Fermi (GTX 675m Time Graphs):

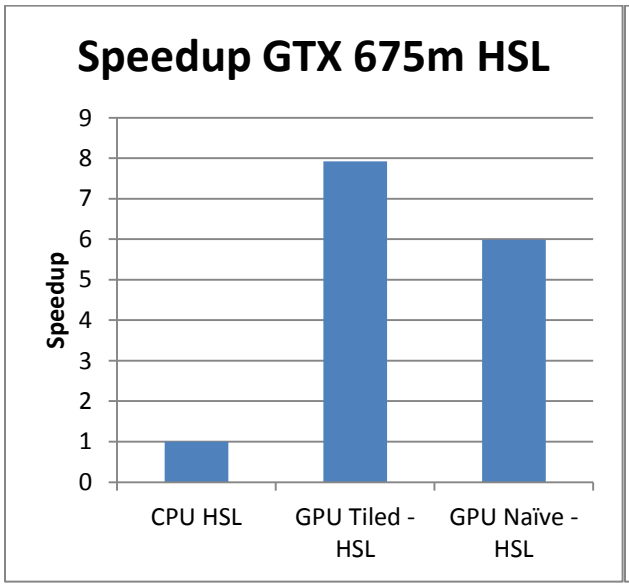


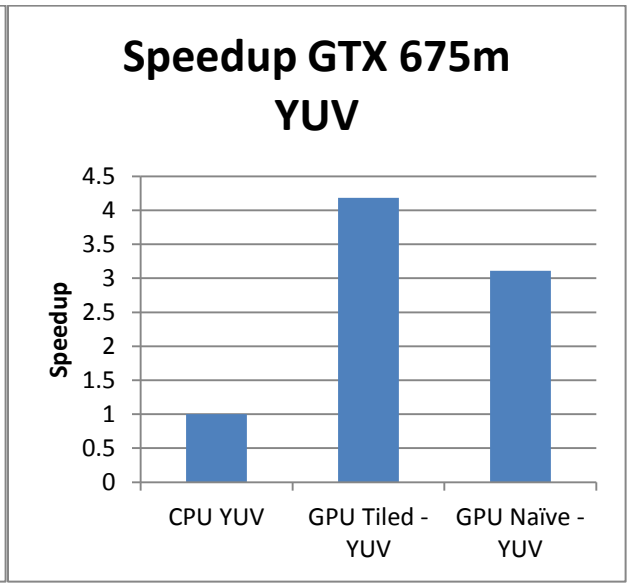




Fermi (GTX 675m Speedup Graphs):







Link to NVIDIA Kepler Whitepaper document:

http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf

Other test machine:

CPU: Intel i7-3630QM @ 2.4 GHz (turboboost up to 3.4 GHz)

GPU: NVIDIA GTX 675m 2GB GDDR5 (Fermi architecture)

HDD: 750 GB 7200 RPM

RAM: 16 GB DDR3 @ 1333 MHz

cs-synar-06 GPU SpeedUp

	gpu_rgb2hsl	gpu_hsl2rgb	gpu_histogram (tiled)	gpu_histogram (naive)
GPU Time	1.719424	0.584064	2.226656	0.996512
CPU Time	117.010109	87.19043	7.257536	7.257536
SpeedUp	**68.0519226206**	**149.2823218004**	3.2593880689	7.2829388909

	gpu_rgb2yuv	gpu_yuv2rgb	gpu_lutcalc	gpu_imgoutcalc
GPU Time	1.151296	0.75088	0.012576	0.265904
CPU Time	58.99968	48.385376	0.001125	9.526176
SpeedUp	**51.2463171938**	**64.4382271468**	0.0894561069	35.8256212769

Originally going into the assignment it seemed obvious to convert as many possible portions into gpu functions, starting with the most commonly used functions among the tests. This included calculating the histogram, lutcalc, and the final image from the equalization. These functions while common only minimally helped speed up the color based tests, while the gray test time was cut almost in half. As we found out though not everything could be converted well or should be as in the case of calculating the CDF and the lutcalc. Calculating the cdf via gpu would almost negate any good the gpu could do as we need the previous indexed value to calculate the current one. Another case in which converting a function to run on the GPU that didn't work out as well was the lutcalc. This had a minor decrease in overall performance by 0.01 ms on average. Tackling the color conversion functions next, this is where we gained incredible speedup and drastically reducing the overall times. By adding timers around individual sections of code and around various gpu functions, we were further able to locate time consuming hotspots and improve code performance.