Rectification:

For the rectification portion of the lab, the code was run on a 2013 MacBook Pro with a 2.6 Ghz dual-core processor. The portion of the code that was parallelized was the double for loop that iterated over each pixel in the original image and rectified the colours for the new image.

The speedup plot looks as follows:

As shown above, there is a slight speedup when using 2 threads of about

(Told / Tnew) = (20.78/18.82) =~ 110%. However, after 2 threads, the overhead cost of

multithreading slows down rectification significantly. For 4 threads, this is around

(Told / Tnew) = (20.78/34.62) =~ 60%, and the number is similar for 8, 16, and 32 threads. This could be because the laptop had only a dual-core processor; at 2 threads, it could run 1 thread on each core. However, for numbers greater than 2 threads, parallelism is still capped at two cores and the threads are running concurrently (i.e. 2 threads sharing one core’s resources). This results in only additional co-ordination overhead without the benefits of parallelism, thus slowing down the overall runtime.

Unique Rectified Image:



Pooling:

For the pooling portion of the lab, the code was once again run on a 2013 MacBook Pro with a 2.6 Ghz dual-core processor. The portion of the code that was parallelized was the double for loop that iterated over each 2x2 square in the original image and pooled the values for the new image.

The speedup plot looks as follows:

As shown above, there is a slight speedup when using 2 threads of about

(Told / Tnew) = (15.1/14.15) =~ 106%. However, after 2 threads, the overhead once again slows down the pooling process significantly. For 4 threads, this is around

(Told / Tnew) = (15.1/22.8) =~ 66%, and this number is similar for 8, 16, and 32 threads. As mentioned above, this slowdown is due to co-ordination overhead costs ramping up while there are no actual improvements in parallelization.

Unique Pooled Image:

