1. Total runtime was measured, not just calculation time
   1. Bunny:
      1. OpenMP: 39 Seconds
      2. CUDA (Not optimized): 39 Seconds
      3. Thrust: \*
   2. Blade:
      1. OpenMP: ~133 Seconds
      2. CUDA: 43 Seconds
      3. Thrust: \*
   3. UNC Power Plant
      1. Could not access file, link was broken on parallel.ecse.rpi.edu and file is not available on geoxeon.
2. The CUDA implementation seems to scale the best of the two I was able to implement. Both programs have quite a long period dedicated to simply calculating the aggregated points from the 100x100x100 grid (this requires ~ 1,000,000,000,000 inverse square roots to be calculated no matter how many points are in the input set. After this, CUDA scales better than OpenMP as can be seen in the runtimes above.
3. Between the OpenMP and CUDA versions of this algorithm, the differences were in the second significant figure, and are likely related to the issue described below in the (\*) section describing my issues with the calculateNearby kernel.
4. The CPU version runs noticeably better on large data sets than the brute force method. The GPU brute force method was faster on both the bunny and blade files, but scaled worse. The difference between the blade and bunny files was 5.2 seconds for the brute force method and 4 seconds for the new method. On the CPU, the blade file was able to be calculated using the new algorithm, but the program running the brute force algorithm had to be stopped as it appeared it was not going to finish for quite some time. Given a larger data set, the difference on the graphics card would likely have become more obvious. The newer method has a higher upfront time penalty, but requires less additional time as the data set increases in size. Both of the new programs reported noticeably less than the brute force method. They both underreport by approximately 20%, but come out to approximately the same answer.

\*I did not have time to implement this algorithm using thrust. My CUDA implementation was buggy and I ran out of time while trying to correct it. As the code currently stands, the value of a variable only updated by an atomicAdd function call appears to be nondeterministic and varies from run to run for a reason I could not determine. Additionally, adds and atomicAdds in this kernel (calculateNearby) seem to fail as the result is always 0. I’m not sure why this is, but trying to debug it kept me from having the opportunity to attempt a Thrust implementation.