Final Project Report

Austin McDowell

For my final project I identified time-consuming functions in the radiative transfer code `BoxFit` and attempted to vectorize them. For my PhD I am doing research on gamma-ray burst (GRB) afterglows. A GRB afterglow is produced when a relativistic jet launched from a GRB accelerates the surrounding medium and produces synchrotron radiation at the contact between the jet and the medium. Synchrotron radiation is produced at a wide range of frequencies and afterglows are observed in the radio, x-ray, and infrared bands. Telescope observations collect data to create an afterglow light curve which is the observed flux density as a function of time. To study afterglows we use a parameterized model to rapidly generate artificial light curves which can be used in Markov chain Monte Carlo (MCMC) analysis to fit the observed light curves and sample the posterior distributions of our model parameters. To speed up the generation of the artificial light curves we create a table from the output of radiative transfer calculations done with `BoxFit`. The `BoxFit` code takes fluid states from a 2D, relativistic, moving-mesh hydro simulation and calculates a spectrum (flux as a function of frequency) assuming a synchrotron emission mechanism. In order to fully cover our parameter space we must generate over one million spectra.

In order to reduce the compute time I wanted to vectorize the time-consuming functions in the `BoxFit` code. To locate the time-consuming functions I used VTune, a profiling software developed by Intel. VTune can be used to find inefficient sections of code, locate parts of the code that would benefit from threaded performance, and identify hardware related issues. I ran VTune on `BoxFit` with the NASA Pleiades cluster to generate code reports. I then set up a Virtual Network Computing (VNC) session on Pleiades and created an SSH tunnel so that I could view the VTune GUI on my local machine. By looking at the code in a top-down tree I was able to identify a time-consuming function: `sphericalfromcartesian`. This function, as its name suggests, takes a cell on the domain in cartesian coordinates and does a transformation to spherical coordinates. This function was slowed down by its usage of special functions such as `cos`, `acos`, `atan`.

Before I could vectorize `sphericalfromcartesian` I wanted to understand where the function was called in `BoxFit`. Normally I would use VTune to do this, however, the Pleiades cluster went down for maintenance before I was able to look at the results again. Instead I tracked the function by hand and found that `sphericalfromcartesian` was called by a function that calculates the absorption and emission coefficients that are needed to find the intensity in a cell on the domain. I also found that `BoxFit` works on a single cell at a time and passes the cell through several functions defined in classes before finally calculating the total flux. Unfortunately, this structure made it difficult to vectorize the code since vectorizing relies on executing the same instruction for multiple pieces of data (SIMD). To incorporate my vectorization of `sphericalfromcartesian` I would have to restructure how `BoxFit` handles individual cells.

Instead of rewriting `BoxFit`, I decided to extract the `sphericalfromcartesian` function and vectorize it separately from the main code. This would allow me to focus on the vectorization. I created a "fake" domain for the function to loop over and apply the spherical transformation and added extra pointers to arrays in a user-defined `struct` called `cor` which contained the coordinate information of the cells currently being worked on. I then added preprocessor directives to check

which vectorization was native to the processor, either Advanced Vector Extensions (AVX) or Streaming SIMD Extensions (SSE). Next, I re-wrote the `sphericaltocartesian` function with the vectorized functions. I loaded the 4 points of data into the `__m128d` vector and used the special functions `__mm_sqrt_pd`, `__mm_acos_pd`, and `__mm_atan_pd` to perform the spherical transformation. Finally, I stored the vectors to the the pointer array that I had added to `cor` with the `__mm_store_pd` function. One thing I learned while doing all of this was that the vectorized trigonometric functions that are used in the transformation are only available to Intel compilers. Finally, I ran the non-vectorized function and the vectorized function on various grid sizes for 10000 repetitions to compare the timings. The results are presented below with a dashed line showing $t \propto n$. I found that the vectorized function was faster than the non-vectorized by about 15% and both scale with the grid size n. I think it makes sense for both versions to have the same scaling since I did not change the underlying algorithm but the number of points that it was applied to. I also think it makes sense for the computation time to be $O(n)$ since the coordinate transformation is performed with simple operations.

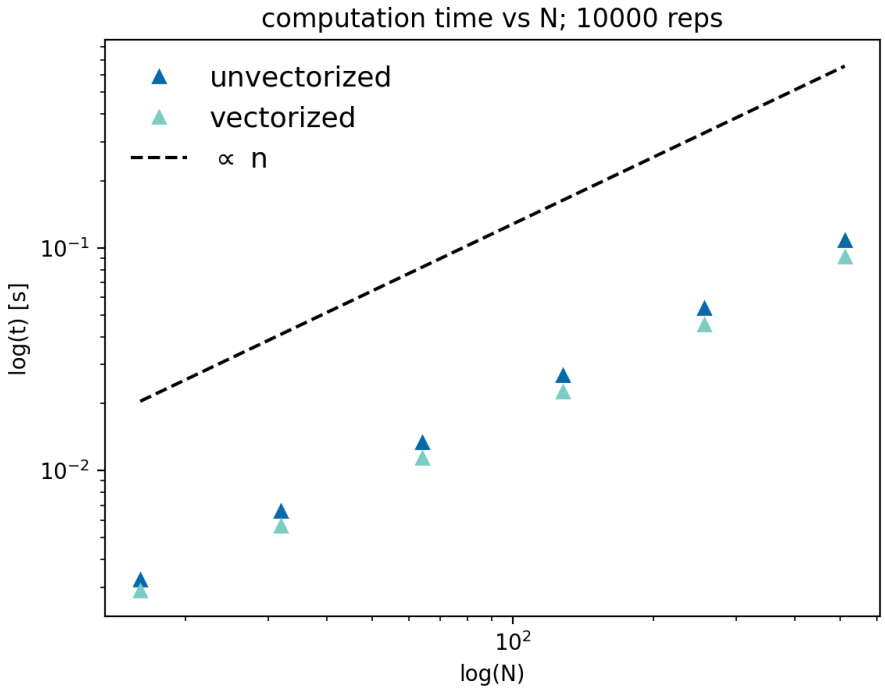

Figure 1: compute time vs grid size for vectorized and unvectorized spherical coordinate transformation

There are other ways I could have tried to speed up the computation time. I could check the VTune results again and see if there are any other functions that would better lend themselves to vectorization given the `BoxFit` structure. With more time I would have tried to parallelize the individual cell calculations so that several could be done at the same time with neighboring cells communicating the necessary information. `BoxFit` works on one annulus at a time by fixing r and looping over ϕ. This makes me think that I could also improve performance by parallelizing the flux calculations from each annulus. By doing this project I learned about VTune, `BoxFit`, and AVX and SSE vectorization.