Eileen Evans
Phoebe DeVries

Using the code

Each of the codes creates a matrix G. The first code, `JapanStrains_serial.py`, runs the code in serial. The second code, `JapanStrains_1.py`, runs the code at the lowest level of parallelism. The third code, `JapanStrains_2.py`, parallelizes the code for each triangle edge, and the last code, `JapanStrains_3.py,` parallelizes the code for each triangle. To run these codes, just execute them at the command line (i.e. `python JapanStrains_1.py`). Note that all of these codes call on some version of `CalcG`, the function that calculates the matrix G, which in turn calls on `CalcTriStrains`, the function that assembles the vectors to be sent to the CUDA kernel.

Running the forward model and creating the visualizations was done in Matlab.

The README file is below, for more details.

The README

We have made four sets of codes: one serial, one for step 1, one for step 2, and one for step 3. In each of the three steps, the computations are successively more parallelized. For a detailed description of the differences between the steps please see our website; this README just explains how to run the four sets of codes.

Each of the four code sets creates the matrix G (the computationally intensive part of the problem) and then calculates a strain vector Gs = e, where s is an input slip distribution of the Tohoki-Oki earthquake.

The serial code set, which can be run by executing the command `python JapanStrains_serial.py`, runs all of the computations in serial. `JapanStrains_serial.py` loads the observation and triangle mesh data files, and calls the function `CalcG_serial` to calculate the matrix *G*. `CalcG_serial`, in turn, calls the function `CalcTriStrains_serial`, which assembles vectors to be sent to the function `advs_serial`. The function `advs_serial` runs the computations of the Green's functions in serial. Thus, in total, three functions are called when you run `JapanStrains_serial.py`.

To run the set of codes for step 1, just execute the command `python JapanStrains_1.py`. (Please note that you will have to be logged into a node on resonance to run the parallel code sets, because we use PyCUDA and therefore require a GPU.) Analogous to the serial code set, `JapanStrains_1.py` will call the function `CalcG_1`, which will call the function `CalcTriStrains_1`, which will in turn call the function `advs_parallel_1`. The function `advs_parallel_1` is where all the computations are done; now, this function contains a CUDA kernel.

Similarly, execute the command `python JapanStrains_2.py` to run set of codes for step 2, and execute the command `python JapanStrains_3` to run the set of codes for step 3.

When these codes terminate, the total run time, the total time spent executing the CUDA kernel, and the total time spent executing the kernel including all setup and data transfer are printed to the screen. In addition, the output matrix G and the output stains will be saved to text files, and the names of these text files are printed to the screen.

Verification

We have included two scripts, `advsserial_verify.py` and `advsparallel_verify.py`, in our submitted folder. These scripts calculate the matrix *G* using the serial code and using the basic CUDA kernel for some dummy input values. When run, they output the dummy matrix G; it is clear that the CUDA kernel calculates the same values as the serial function. (We also verified our python `advs_serial` function against the original Matlab code written by Brendan Meade for a range of inputs.)

Specifying the input data

For benchmarking purposes, we have made three sets of data inputs, a tiny set (10 observation points and 37 triangles), a small set (400 observation points and 37 triangles), and a medium set (50000 observation points and 110 triangles), and a final set (50,000 observation points and 2621 triangles). Each set of data inputs are stored in three text files made in Matlab (c.txt, v.txt, and Obs.txt). For the tiny input set, the output matrix *G* will be of size [60 x 111]; for the small input set, *G* will be of size [2400 x 111]; and finally for the medium input set, *G* will be of size [300000 x 330], and for the final input set *G* will be of size [300000 x 7863].

To specify which size data input you wish to use, please open the JapanStrains script --`JapanStrains_serial.py, JapanStrains_1.py, JapanStrains_2.py, or JapanStrains_3.py`, depending on which code set you would like to test. The first line of all of the JapanStrains scripts is: PROBLEM_SIZE = "TINY". To run the codes with the larger data input, just change this line to: PROBLEM_SIZE = "SMALL", PROBLEM_SIZE = "MEDIUM", or PROBLEM_SIZE = "FINAL".

Visualization

Calculating the entries of the matrix G efficiently was our goal -- and this is done in all three parallel versions of our code. (Our maximum speedup was about 800.)

The data inputs we used for these visualizations are the "FINAL" inputs (PROBLEM_SIZE = "FINAL"). Running JapanStrains_3 with these inputs, the output strain values we obtain are the strains at all observation points due to Tohoki-Oki Earthquake. By multiplying these strains by a Young's Modulus for the upper crust (typically assumed to be ~30 GPa), we obtain stress changes at each observation point due to the earthquake. Visualizations of these stress changes throughout the Earth's crust can be seen in the Visualization section.