Implementing AMR Grids on a GPU for Massive Parallelization

Stacey Allison
(North Georgia College & State University, Dahlonega, GA, 30597)

Dr. Dianna Spence
(North Georgia College & State University, Dahlonega, GA, 30597)

Dr. Bobby Philip
(Oak Ridge National Laboratory, Oak Ridge, TN, 37830)

Adaptive Mesh Refinement (AMR) is a technique used to discretize the domain of a partial differential equation (PDE), allowing the PDE to be represented as systems of equations. These systems then can be solved using various numerical methods, such as the Jacobi method or the Gauss-Seidel method. One such solver, known as the Red-Black Gauss-Seidel (RBGS) method, was the focus of this research effort. RBGS is very similar to the Gauss-Seidel method, in that it constantly updates the input used, resulting in a faster convergence to the solution. Each node of the AMR grid is color-coded, alternately red and black, such that each red node is connected only to black nodes, and vice versa. Each iteration of the method first updates each red node using the un-updated black node values. Then, all black nodes are updated using the updated red node values. Our main objective for this project was to start with the current algorithm for RBGS, written in FORTRAN (a programming language), and convert it to CUDA (a programming language), so that it could run on a graphics processing unit (GPU). The big advantage of converting to CUDA is that, although most modern central processing units (CPUs) have the capability of running operations in parallel, the GPU can do so much more efficiently. After the RBGS algorithm was converted to CUDA, we tested both the CUDA and FORTRAN versions of the algorithm to determine how each performed when increasing the number of objects passed to them and when increasing the size of the objects passed. We found that, although the processing time for both codes increased linearly with increasing the size of the input, the increase in the CUDA code's processing time per unit increase in dataset size was much smaller than that of the FORTRAN version. When passing in a large-enough chunk of data, the CUDA code version performed significantly better than the FORTRAN code version. Other tests revealed very similar results when using different solver methods. Thus, we conclude that the CUDA codes were much better than the FORTRAN codes in these run-time tests, but only when passing data sets containing more than one million cells to compensate for the computational "overhead" resulting from copying all the data to the GPU.