Optimization of Algebraic Multigrid Through GPU Computing

Steven R. Hussung
Mercer University
steven.r.hussung@live.mercer.edu

Motivation

Our goal is to solve a sparse linear system. The standard method for solving a linear system is Gaussian Elimination. This method is inefficient for matrices with many zeros (sparse matrices). Because other matrix operations are fast when applied to sparse matrices, we can use other methods to solve these sparse systems quickly.

Error-Smoothing, Sparse, Iterative Solvers

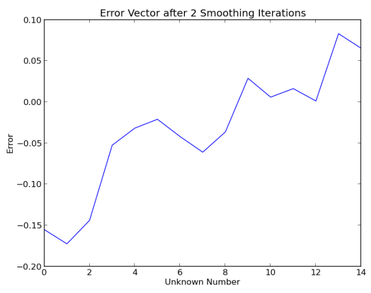
Iterative solvers work by improving an initial guess until it is close enough to the actual answer. Many iterative solvers, such as Weighted Jacobi and Gauss-Siedel, smooth the error vector as they iterate on specific kinds of systems. Mathematically,

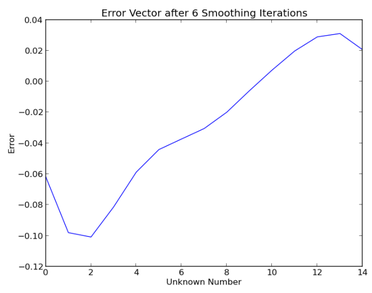
$$e_{i+1} = G\, e_i$$

(where G is a smoothing operator). To find the final solution to our system, we will continue applying G, resulting in better and better answers. Finding G is certainly non-trivial, but if we allow the assumption of an error-smoothing operator we can continue.

Error Smoothing

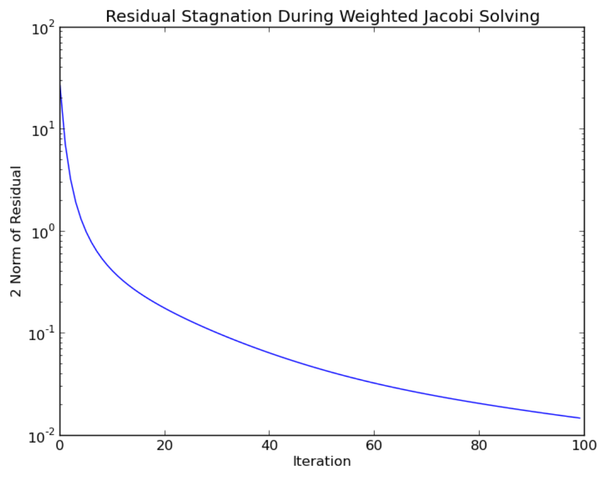
The following pictures illustrate the error smoothing process, beginning with a random x vector, and corresponding random error vector. The system being solved is a 1-dimensional Poisson problem.





Smoothing will bring the unknowns of the error vector closer together, and closer to zero. As this method continues for a high number of iterations the x vector will (hopefully) become a reasonable answer.

The 1-dimensional Poisson problem is a very well-behaved case, so the norm of its error vector will decrease quickly as the method continues. However, many larger, more complicated systems will cause both the error and residual to stagnate. Although the norm of the residual may continue to decrease, it will do so at a much slower rate.



Two-Grid Iterative Method for Solving Linear Systems

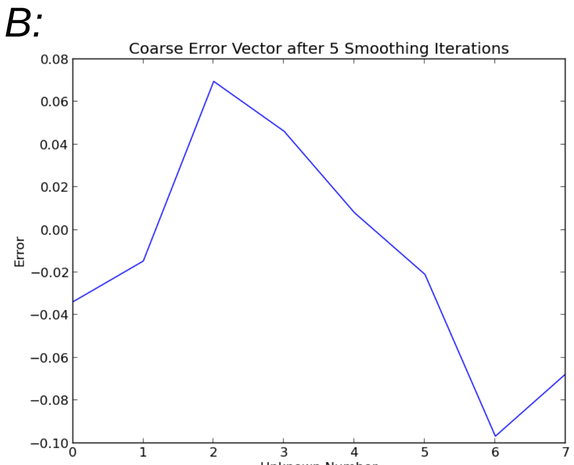
The two-grid coarse correction iterative method attempts to solve this problem by constructing a “coarse” system, and smoothing the error in that system in an attempt to eliminate low-frequency error.

For example: say we have been smoothing a small system using a simple Weighted Jacobi method for several iterations and we have the error in Figure 3-A. The error in Figure 3-A is too smooth for Weighted Jacobi to work efficiently. If we were to sample the error vector at only a few points, we would see the error in Figure 3-B. The graph in Figure 3-B is *less smooth*. Using an error-smoothing method on this modified system would have a greater effect than using it on the original. This improved solution could then be used to find a more accurate answer to the original problem.

Figure 3

A:

B:

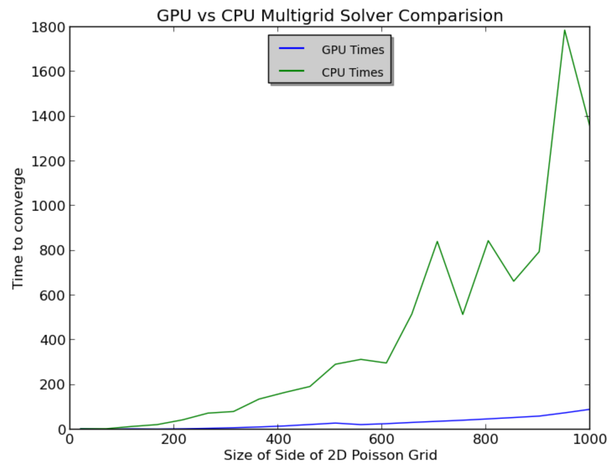


Multigrid Iterative Method for Solving Sparse Linear Systems

The multigrid method is an expansion of the two-grid method. Once a coarse system has been made, we simply have another linear algebra problem. We can make another coarse system to solve it, and then repeat this process until the system is trivially easy to solve. Finally, we move back through all of the intermediate levels to arrive at a closer answer to the original system. This entire process is called “cycling”.

Optimization through GPU Computing

In an attempt to speed up the multigrid process we decided to try executing the main multigrid algorithm on the GPU. The GPU is able to execute the main bottleneck in multigrid, a sparse matrix-vector multiplication, 2.5 to 10 times as fast as a CPU (Bell and Garland). The coarse systems we used were from the PyAMG package, and the code used to execute the cycling was a modified CUSP program. The GPU version of the code ran *much* faster than the CPU version.

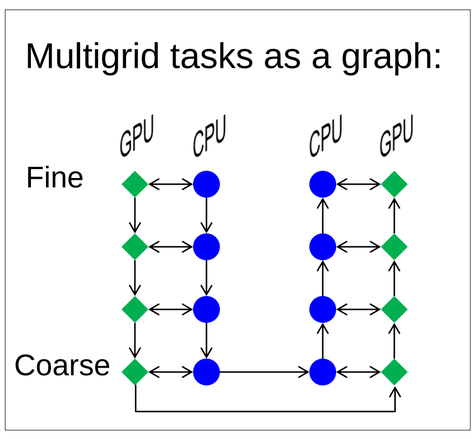


Multigrid with LU Factorization

An additional possibility is to use the CPU to find a sparse Lower-Upper (LU) Factorization of the coarsest system’s *A* matrix while the GPU cycles. Once the factorization is complete, the coarsest system could be solved much more quickly. The CPU would then find the LU Fact. of the second coarsest *A* matrix, and so on.

Hybrid Cycling

In hybrid-cycling, the CPU and GPU would perform different parts of the cycling process. If the execution times for individual tasks (smoothing, restricting, interpolating, solving) could be estimated, then the path of lowest time-cost could be found using Dijkstra’s Shortest Path Algorithm.



Acknowledgements:
Thanks to Dr. Luke Olson, Abby Gregory, Passionate-on-Parallel REU Staff and Participants, UIUC, and NSF.

Citations:
Bell and Garland: “Efficient Sparse Matrix-Vector Multiplication on CUDA”

Passionate on Parallel 2012 REU Program

This REU is co-funded by the ASSURE program of the Department of Defense in partnership with the National Science Foundation REU Site program under Award No. 1004311.