A FAST EIKONAL EQUATION SOLVER FOR PARALLEL SYSTEMS

WON-KI JEONG AND ROSS T. WHITAKER*

Abstract. This paper presents a novel solver for the eikonal equation that is designed to run efficiently on massively parallel systems. The proposed method manages a list of active nodes and iteratively updates the solutions on those grid points until they converge. The management of the list does not entail the extra burden of ordered data structures. The proposed method has suboptimal worst-case performance, but in practice, on real and synthetic datasets, it performs fewer computations per node than optimal alternatives. Furthermore, the proposed method uses local, synchronous updates and therefore has better cache coherency, is simple to implement, and scales efficiently on parallel architectures, e.g., multiprocessor systems or GPUs (graphics processing units). The paper describes the method, the implementation on the GPU, and performance analysis that compares against the state-of-the-art eikonal solvers.

1. Introduction. The eikonal equation, which is a nonlinear Hamilton-Jacobi partial differential equation (PDE), is given by

$$H(\mathbf{x}, \nabla \mathbf{x}) = |\nabla \phi(\mathbf{x})|^2 - \frac{1}{f^2(\mathbf{x})} = 0, \ \ \forall \mathbf{x} \in \Omega \tag{1.1}$$

where Ω is a domain in R^n, $\phi(\mathbf{x})$ is the travel time or distance from the source, and $f(\mathbf{x})$ is a positive speed function defined on Ω. The application of the solutions of eikonal equation is numerous, e.g., computer vision [2, 9, 13], image processing [5, 11], and geoscience [8, 12, 6].

A number of different numerical strategies have been proposed to efficiently solve the eikonal equation. These include iterative schemes [9], expanding box schemes [3, 14, 17], expanding wavefront schemes [7, 10, 16], and sweeping schemes [15, 18]. Among these strategies, the most efficient eikonal equation solvers are based on adaptive updating schemes and ordered data structures. For instance, the Fast Marching Method (FMM) [10], which is optimal in the worst case, uses a heap data structure, which includes nodes from the entire wavefront. During each iteration, the heap determines the as yet unsolved grid value that is guaranteed to depend only on neighbors whose values are solved. The heap must be updated with each updated grid value ($O(\log N)$) and each grid point is solved only once. For this algorithm, the heap becomes a bottleneck especially on higher dimensional datasets, and does not readily allow for parallel solutions, such as those available with *single instruction multiple datastream* (SIMD) or multicore processors, which are currently the most cost effective way of organizing parallel compute power. Furthermore, grid points must be updated one at a time, in a way that does not guarantee locality, opportunities for coherency in cache or local memory.

Even though there exists theoretically optimal approaches to solve eikonal equations, they are not fast enough on many applications in practice. For example, the 3D prestack seismic migration process [6] should compute 5D traveltime tables, which consist of point-to-point traveltimes between surface points (2D) to all volume grid points (3D), and requires running the eikonal solver a huge number of times. Tracking the multiple phases of seismic wave reflections also requires multiple solutions of eikonal equations with various boundary conditions [8]. Interactive computation of skeletons from dynamic 3D models is another application of fast eikonal solvers [13]. In such applications, faster solvers can greatly reduce the running time and allow interactivity, and therefore the need for fast *parallel* algorithms still remains.

In this paper we propose a novel numerical algorithm, which we call the Fast Iterative Method (FIM), to solve eikonal equations on massively parallel architectures. The proposed method relies on a modification of an iterative solution of the upwind discretization of the equation. Unlike the traditional iterative schemes, the proposed method selectively updates the points chosen by a convergence measure and avoids using expensive ordered data structures or special updating orders. The method is simple to implement and faster than the existing methods on a wide range of speed functions in two and three dimensions. In addition, the proposed algorithm can be easily ported to parallel architectures, including multicore processors, cluster systems, or graphics processing units. In the following sections, we show the proposed algorithm, the implementation details on the GPU, and the performance comparison with an existing, optimal solver.

*Scientific Computing and Imaging institute, School of Computing, University of Utah, Salt Lake City, Utah 84112 (`wkjeong,whitaker@cs.utah.edu`).

2. Algorithm. FIM is developed based on the observations from the two well-known numerical eikonal solvers. One is the iterative method introduced by Rouy et al. in [9], which updates the solution for every point iteratively until it converges using a Godunov discretization of Hamiltonian. This method does not rely on the causality principle. It is simple to implement, but it is inefficient, because every grid point must be visited (and a quadratic evaluated) until the entire grid has converged. The other related approach is FMM [10], which uses the idea of a narrow band of points on the wavefront, and thereby updates points selectively (one at a time) by managing a heap.

The main idea of FIM is to solve the eikonal equation selectively on the grid points in parallel without maintaining expensive data structures. FIM maintains a narrow band, called *active list*, for storing the points that are being updated. Instead of using a special data structure to keep track of exact causal relationships, we maintain a looser relationship and update all such points simultaneously. At each iteration, we expand the list of active points, and the band thickens or expands to include all points that could be influenced by the current updates. A point can be removed from active list only when its solution has converged—that is, it solves the local H-J approximation relative to its upwind neighbors. To update the solutions of the points in the active list, we use the Godunov upwind discretization of the eikonal equation (Eq 2.1) and solve the equation for $U_{i,j,k}$ using a quadratic solver.

$$\left[\frac{(U_{i,j,k}-U^{x\min}_{i,j,k})^+}{h_x}\right]^2+\left[\frac{(U_{i,j,k}-U^{y\min}_{i,j,k})^+}{h_y}\right]^2+\left[\frac{(U_{i,j,k}-U^{z\min}_{i,j,k})^+}{h_z}\right]^2=\frac{1}{f^2_{i,j,k}} \tag{2.1}$$

where $U_{i,j,k}$ is the discrete approximation to ϕ at $\mathbf{x}=(i,j,k)$, h_p is a grid spacing along the axis p, $U^{p\min}_{i,j,k}$ is the smallest neighbor value along the axis p, and $(x)^+=\max(x,0)$.

The main algorithm consists of two parts, the initialization and the updating. In the initialization step, we set the boundary conditions on the grid, and set the values of the rest of the grid points to infinity (or a numerical placeholder for that value). The 1-neighbors of the source points are then added to the *active list* L. In the updating step, for every point $\mathbf{x}=(i,j,k)$ we compute the new $U_{i,j,k}$ by solving Eq 2.1 and check if the point is converged by comparing the previous and the new $U_{i,j,k}$ values. If so, we remove the point from L and any 1-neighbor points that are not in L and have values that are down wind (greater than $U_{i,j,k}$) are added to L. Note that newly added points must be updated in the next updating iteration. The updating step is repeated until L is empty.

3. GPU Implementation. The GPU has evolved into a highly parallelized SIMD multiprocessor machine, meaning that the same instructions can be applied to multiple streams of fragments in parallel. In addition, the GPU supports the high precision computation (up to 32bit float), and is fully programmable. Due to their increasing computational power and flexibility, current GPUs are becoming a very powerful computational platform for general-purpose computation problems as well as traditional computer graphics tasks. Therefore, we have chosen the GPU to implement our parallel eikonal equation solver and conducted the performance analysis. Note that the proposed method can be also easily ported to the other parallel architectures, e.g., multicore processors, shared memory multiprocessor machines, or cluster systems.

The major difference between the CPU and the GPU implementation is that the GPU employs a tile-based updating scheme, as proposed in [4]. The idea is that the domain is decomposed into the pre-defined size tiles (in our implementation we use a 8^3 cube for 3D), and solutions of all pixels in the tiles are updated in parallel. Therefore, the active list of the GPU maintains the list of active *tiles* instead of points. Since the computation on the GPU is done by drawing quads using the graphics API, a tile-based update scales much better than a pixel-based update on the GPU.

The GPU algorithm consists of four steps. First, each active tile is updated with a given number of iterations. During each iteration, a new solution of Eq 2.1 is computed, and its convergence is encoded using negative/positive signs, e.g., converged values are stored in positive but not-yet-converged values are stored in negative values, to minimize GPU memory use. Note that only the new solution that is smaller than the previous solution will be updated, and the convergence is determined if the difference of previous and new solution is within the threshold. After the update step, we perform a reduction on each tile, i.e., 8-to-1 mapping on 3D, to check whether the tile is converged or not. For a n^3 sized tile, we need $\log_2 n$ reduction iterations to shrink a tile into a pixel (e.g., 8^3 tile requires $\log_2 8 = 3$ reduction iterations). The first reduction iteration converts the positive/negative signs into 0 or 1, and the following reduction iterations check if any of eight pixel values is 1. If so, set the pixel value after reduction as 1, or set it as 0 otherwise. The third step is checking the neighborhood tiles if any pixels in the current tile is upwind of its neighbors to update the active list. In this step, all the neighborhood tiles of the to-be-removed tiles are updated once and check if any pixel value is changed. The final step is updating the active list. To do this, we perform another reduction on the neighborhood tiles. Each pixel of resulting volume of the reduction operation represents whether the corresponding tile should be added to or removed from the active list.

We use OpenGL API and Nvidia Cg shader language to implement the proposed algorithm on the GPU. A 3D volume on the GPU is implemented as a set of 2D float textures, and each texture packs four z-slice to utilize vector ALUs on the GPU. Instead of reading back the active list volume to the CPU memory and selectively drawing quads, we store all geometry on the GPU side using the vertex buffer object and selectively draw quads by checking the corresponding value in the active list volume in the vertex shader stage using the vertex texture fetch function. For each update step, a target texture is attached to the framebuffer object and off-screen rendering is performed. To quickly determine the convergence of the solver, we use the occlusion query test to see if the active list is empty or not.

4. Results. We have tested our GPU eikonal solver on several synthetic datasets and compared the running time with a Fast Marching solver on the CPU. The test PC is equipped with an Intel Core 2 Duo 2.4Ghz CPU and an Nvidia 7900 GTX GPU. We use an optimized *ITK* [1] implementation of the Fast Marching solver for a fair comparison.

Table 4.1 lists the running time on the various input volumes of size 256^3, and figure 4.1 shows the level sets of the solution on each input volume. As shown in table 4.1, the GPU eikonal solver runs about seven times faster than the CPU solver on all input speed volumes.

Table 4.1
Running time (in second) on various input speed volumes

	Single	Three	Random	Coherent
FMM (CPU)	69.75	69.563	81.875	80.625
FIM (GPU)	11.328	14.937	14.25	13.687

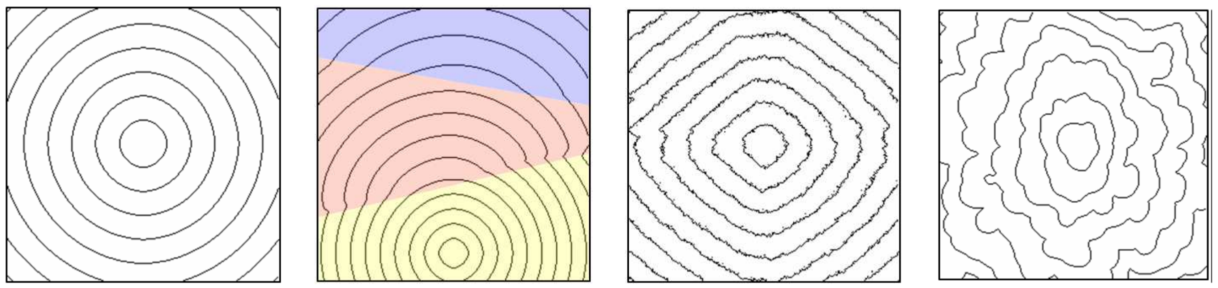
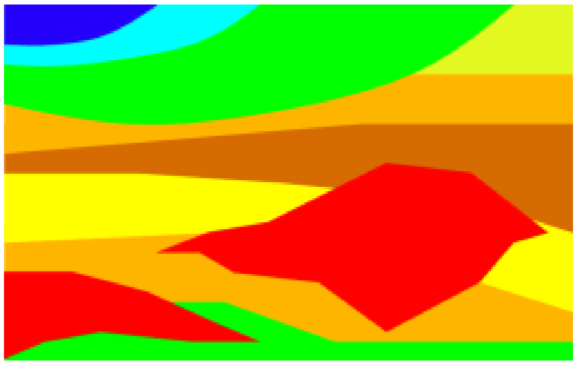

(a) Single speed (b) Three speed (c) Random speed (d) Coherent speed

Fig. 4.1. *Level set of the solutions from the synthetic speed volumes*

Figure 4.2 shows a geoscience application of the eikonal solver. Due to the heterogeneous rock materials, the seismic wave propagation speed varies depending on the rock properties. Figure 4.2 (a) shows the input speed volume (blue is the slowest while red is the fastest speed). Figure 4.2 (b) shows the level set of the seismic wave propagation from the seed point. We use a 256x160x256 size volume, and a single propagation takes about eight seconds on the GPU solver and a minute on the CPU solver. Because the GPU solver takes only a few seconds to run, users can change the seed position and simulate seismic propagation interactively on reasonably sized datasets. Moreover, the GPU solvers can greatly reduce the computation time for the applications of using multiple solutions of eikonal equations, e.g., tracking the multiple phases of seismic wave reflections and seismic tomography [8], and seismic traveltimes computation [6].


(a) Synthetic speed volume

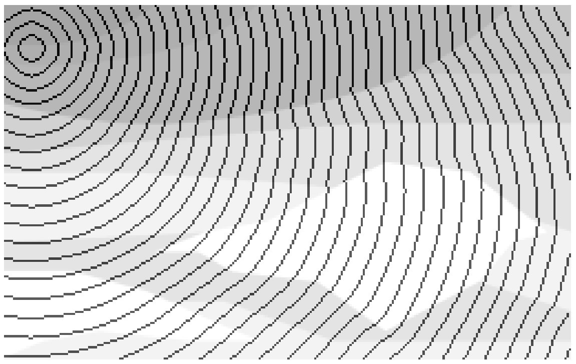

(b) Level set of solution

FIG. 4.2. *Wave propagation in the synthetic speed volume*

5. Conclusion. In this paper we describe a novel parallel algorithm for an eikonal equation solver and its implementation on the GPU. The proposed method scales well on the massively parallel architectures, and our GPU implementation shows about seven times performance gain over the traditional CPU-based eikonal solver. For future work, developing a parallel algorithm for general Hamilton-Jacobi equations will be an extension of this method. Porting the proposed algorithm to the newer GPU architecture that supports GPGPU (General-Purpose computation on the GPU) API will be another interesting area of future work and should result in further improvements in run-time performance.

REFERENCES

[1] The insight segmentation and registration toolkit. http://www.itk.org.
[2] A. Bruss. The eikonal equation: some results applicable to computer vision. *J. Math. Phy.*, 23(5):890–896, 1982.
[3] W. Schneider Jr., K. Ranzinger, A. Balch, and C. Kruse. A dynamic programming approach to first arrival traveltime computation in media with arbitrarily distributed velocities. *Geophysics*, 57(1):39–50, 1992.
[4] A. Lefohn, J. Kniss, C. Hansen, and R. Whitaker. Interactive deformation and visualization of level set surfaces using graphics hardware. In *IEEE Visualization 2003 Conference Proceedings*, pages 75–82, 2003.
[5] R. Malladi and J. Sethian. A unified approach to noise removal, image enhancement, and shape recovery. *IEEE Trans. on Image Processing*, 5(11):1554–1568, 1996.
[6] A. Popovici and J. Sethian. 3-d imaging using higher order fast marching traveltimes. *Geophysics*, 67(2):604–609, March-April 2002.
[7] F. Qin, Y. Luo, K. Olsen, W. Cai, and G. Schuster. Finite-difference solution of the eikonal equation along expanding wavefronts. *Geophysics*, 57(3):478–487, 1992.
[8] N. Rawlinson and M. Sambridge. The fast marching method: an effective tool for tomographics imaging and tracking multiple phases in complex layered media. *Exploration Geophysics*, 36:341–350, 2005.
[9] E. Rouy and A. Tourin. A viscosity solutions approach to shape-from-shading. *SIAM Journal of Numerical Analysis*, 29:867–884, 1992.
[10] J. Sethian. A fast marching level set method for monotonically advancing fronts. In *Proc. Natl. Acad. Sci.*, volume 93, pages 1591–1595, February 1996.
[11] J. Sethian. *Level set methods and fast marching methods.* Cambridge University Press, 2002.
[12] R. Sheriff and L. Geldart. *Exploration Seismology.* Cambridge University Press, 1995.
[13] K. Siddiqi, S. Bouix, A. Tannenbaum, and S. Zucker. The hamilton-jacobi skeleton. In *Proc. International Conference on Computer Vision*, pages 828–834, September 1999.
[14] J. Trier and W. Symes. Upwind finite-difference calculation of traveltimes. *Geophysics*, 56(6):812–821, June 1991.
[15] Y. Tsai. Rapid and accurate computation of the distance function using grids. *J. Comp. Phys.*, 178(1):175–195, 2002.
[16] J. Tsitsiklis. Efficient algorithms for globally optimal trajectories. *IEEE Trans. on Automatic Control*, 40(9):1528–1538, September 1995.
[17] J. Vidale. Finite-difference calculation of traveltimes in three dimensions. *Geophysics*, 55(5):521–526, 1990.
[18] H. Zhao. A fast sweeping method for eikonal equations. *Mathematics of Computation*, 74:603–627, 2004.