A Fast Iterative Method for a Class of Hamilton-Jacobi Equations on Parallel Systems

Won-Ki Jeong and Ross T. Whitaker

School of Computing, University of Utah

April 18, 2007

Abstract

In this paper we propose a novel computational technique, which we call the Fast Iterative Method (FIM), to solve a class of Hamilton-Jacobi (H-J) equations on massively parallel systems. The proposed method manages the list of active nodes and iteratively updates the solutions on those nodes until they converge. Nodes are added to or removed from the list based on a convergence measure, but the management of this list does not entail the the extra burden of expensive ordered data structures or special updating sequences. The proposed method has suboptimal worst-case performance, but in practice, on real and synthetic datasets, performs fewer computations per node than guaranteed-optimal alternatives. Furthermore, the proposed method uses only local, synchronous updates and therefore has better cache coherency, is simple to implement, and scales efficiently on parallel architectures, such as cluster systems or graphics processing units (GPUs). This paper describes the method, the implementation on the GPU, and a performance analysis that compares the proposed method against the state-of-the-art H-J solvers.

1 Introduction

The applications of solutions to the H-J equation are numerous. The equation arises in the fields of computer vision, image processing, geoscience, and medical imaging and analysis. For example in computer vision, the shape-from-shading problem, which infers 3D surface shape from the intensity values in 2D image, can be modeled and solved with the Eikonal equation [3, 14], which is a special form of the H-J equation. Extracting the medial axis or skeleton of the shape can be done by analyzing solutions of the H-J equation with the boundaries specified at the shape contour [20]. Solutions to the H-J equation have been proposed for noise removal, feature detection and segmentation [8, 17]. In physics, the H-J equation arises in models of wavefront propagation. For instance, the calculation of the travel times of the optimal trajectories of seismic waves is a critical process for seismic tomography [13, 19]. Several methods based on the H-J equation have recently been introduced as a means for describing connectivity in white matter in medical image analysis [10, 9, 4, 11].

The Hamilton-Jacobi partial differential equations (PDEs), is given by

$$H(\nabla u, \mathbf{x}) = \sqrt{(\nabla u) M (\nabla u)^T} = 1, \ \ \forall \mathbf{x} \in \Omega, \tag{1}$$

where Ω is a domain in R^n, $u(\mathbf{x})$ is the travel time or distance from the source, and M is the speed tensor matrix defined on Ω. We use the Hamiltonian defined below for our model equation:

$$H(p, q, r) = \sqrt{ap^2 + dq^2 + fr^2 + 2(bpq + cpr + eqr)} \tag{2}$$

$$M = \begin{bmatrix} a & b & c \\ b & d & e \\ c & e & f \end{bmatrix}, \ p = \frac{\partial H}{\partial x}, q = \frac{\partial H}{\partial y}, r = \frac{\partial H}{\partial z}$$

where p, q, and r are partial derivatives of u_i at $\mathbf{x}$ along x, y, and z axis, and a, b, c, d, e, and f are upper triangular elements of the matrix M. Equation 1 becomes the Eikonal equation when M is an identity matrix.

A number of different numerical strategies have been proposed to efficiently solve the H-J equation. These methods can be classified into two groups. One is a class of iterative methods based on a fixed-point update using Jacobi or Gauss-Seidel schemes. An early work by Rouy et al. [14] solves the Eikonal equation, a special case of H-J equation, by updating the solutions of the grid using a pre-defined updating order and Godunov upwind Hamiltonian until they converge. The method is simple to implement and produces viscosity solutions, but takes many iterations to converge and worst case complexity can reach up to $O(N^2)$. Zhao [22] proposed the Fast Sweeping method, which uses a Gauss-Seidel updating order for fast convergence. The Fast Sweeping method has a computational complexity of $O(kN)$ where k depends on the complexity of the speed function. Tsai et al. [21] employed the Fast Sweeping method and a Godunov upwind discretization of the class of convex Hamiltonians to solve anisotropic H-J equations. The proposed Godunov Hamiltonian uses only 1-neighborhood pixels, so it maps well on iterative schemes. However, there are many cases to check for the correct solution of the Hamiltonian, e.g., eight cases for 2D and 26 cases for 3D. solution, in up to 26 times for 3D. Kao et al. [6] introduced a new interpretation of Hamiltonians based on the Legendre transformation and showed that it is in fact a Godunov Hamiltonian. In the following paper [5] Kao et al. employed the Lax-Friedrichs Hamiltonian for arbitrary static H-J equations. The proposed method is simple to implement and can be used widely on both convex and non-convex H-J equations, but it requires many more iterations than the Godunov Hamiltonian and the solution shows excessive diffusion due to the nature of the scheme. In general, the iterative methods are slow to converge and are not suitable for interactive applications.

Another class of H-J solvers is based on adaptive updating schemes and sorting data structures. An earlier work by Qin et al. [12] and later Sethian et al. [15, 16, 18] used a Dijkstra-type shortest path algorithm to solve convex H-J equations, which is generally referred to as the Fast Marching method. The main idea behind this method is that solutions for a convex Hamiltonian depend only on the upwind neighbors along the characteristics, so the causality relationship can be determined uniquely and the correct solutions can be computed by only a single pass update. The complexity of the Fast Marching method is $O(N \log N)$, which is worst-case optimal, and the running time is not much affected by the complexity of the speed. However, for a class of general H-J equations [18], tracing the characteristics can cause expensive searching among a wider range of neighborhoods than solving equations using an iterative numerical method. In addition, the method uses a global sorting data structure, e.g., a heap, and therefore the parallelization is not straightforward.

In this paper we focus on the development of a parallel algorithm for the H-J equation and the implementation on the GPU in order to make comparisons against other state-of-the-art methods. While the worse-case performance of the proposed algorithm is not optimal, it performs much better than worst case on a variety of complex data sets even on a single processor, and scales well on many parallel architectures for a further performance benefit. The main contribution of this paper is introducing a novel numerical algorithm to solve the H-J equation that can be well-adapted to various parallel architectures, an improved Godunov Hamiltonian computation, and a GPU implementation of the proposed H-J solver.

The remainder of this paper proceeds as follows. In the next section we introduce the proposed *fast iterative method* (FIM) algorithm for parallel systems. In Section 3 we introduce the 3D Godunov Hamiltonian for the H-J equation and its implementation in detail. In Section 4, we introduce GPU implementation of the proposed method. In Section 5 we show numerical results on several synthetic and real tensor volumes and compare with the existing state-of-the-art CPU methods. In section 6 we summarize the paper and discuss the future research directions related to this work.

2 Fast Iterative Method (FIM)

To solve Equation 1 efficiently, we introduce a novel numerical algorithm that scales well on parallel architectures. As discussed in Section 1, existing H-J solvers do not scale well on parallel architectures due to the use of global data structures and fixed updating orders. Therefore, the main design goals in order to produce good overall performance, cache coherence, and scalability across multiple processors are:

- the algorithm should not impose a particular update order
- the algorithm should not use a separate, heterogeneous data structure for sorting, and
- the algorithm should be able to simultaneously update multiple points

2.1 Algorithm description

FIM is a numerical algorithm to solve PDEs, such as Equation 1, on parallel architectures. The main idea of FIM is to solve the H-J equation selectively on the grid nodes without maintaining expensive data structures. FIM maintains a narrow band, called the *active list*, for storing the index of grid nodes to be updated. Instead of using a special data structure to keep track of exact causal relationships, we maintain a looser relationship and update all nodes in the active list simultaneously (i.e., Jacobi update). During each iteration, we expand the list of active nodes, and the band thickens or expands to include all nodes that could be influenced by the current updates. A node can be removed from the active list when the solution is converged, and re-inserted when any changes of its adjacent neighbors affect the solution of the current node. Note that newly inserted nodes must be updated in the following update iteration to ensure a correct Jacobi update. To compute the solutions of the nodes in the active list, we use the Godunov upwind discretization of the Hamiltonian (section 3). The key ideas of the proposed algorithm are two fold: allowing multiple updates per node by reinserting nodes to the active list, and using a Jacobi update for parallel computation. It turns out that the proposed algorithm is classified as a class of *label-correcting* algorithms. The pseudo code of the FIM is as follows ($U_{\mathbf{x}}$ is a discrete approximation of $u(\mathbf{x})$, and $g(U_{\mathbf{x}})$ is a new solution at $\mathbf{x}$ that satisfies Equation 1 computed using a Godunov Hamiltonian H_G in Equation 3).

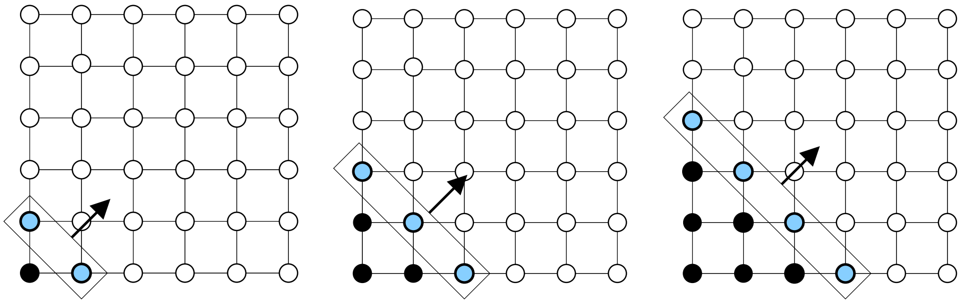
Algorithm 2.1: FIM($\mathbf{X}$)

```
comment: 1. Initialization (X : set of all grid nodes, L : active list)
for each x ∈ X
  do { if x is source
         then U_x ← 0
         else U_x ← ∞
for each x ∈ X
  do { if any neighbor of x is source
         then add x to L

comment: 2. Update nodes in L
while L is not empty
  do { for each x ∈ L
         do { p ← U_x
              q ← g(U_x)
              if p > q
                then { U_x ← q
              if |p − q| < ε
                then { for each 1-neighbor x_nb of x
                         do { if x_nb is not in L
                                then { p ← U_{x_nb}
                                       q ← g(U_{x_nb})
                                       if p > q
                                         then { U_{x_nb} ← q
                                                add x_nb to L
                       remove x from L
```

2.2 Properties of the algorithm

In this section we describe how the algorithm works in detail. Figure 1 shows the schematic 2D example of FIM frontwave expanding in the first quadrant. The lower-left corner point is the source point, the black points are fixed points, the diagonal rectangle containing blue points is the active list, and the black arrow represents the narrow band's advancing direction. Figure 1 (a) is the initial stage, (b) is after the first update step, and (c) is after the second update step. Because blue points depend only on the neighboring black points, all of the blue points in the active list can be updated at the same time. If the characteristic path does not change its direction to the other quadrant, then all the updated blue points will be fixed (become black points) and their 1-neighbor white points will form a new narrow band.



(a) Initial stage (b) After first update (c) After second update

Figure 1: Schematic 2D example of FIM frontwave propagation.

FIM is an iterative method, meaning that a point is updated until its solution converges. However, for many data sets most points require only a single update to converge. This can be interpreted as follows. If the angle between the direction of the characteristic path and the the narrow band's advancing direction is smaller than 45 degree, then the exact solution at the point can be found only in a single update, as in the fast sweeping method. If the angle is larger than 45 degrees, the point at the location where the characteristic path changes the direction will have an initial value that is computed using the wrong up-wind neighborhood, and it will be revised in successive iterations as neighbors refine their values. Thus, that point will not be removed from the active list and will be updated until the correct value is computed. Figure 2 shows this situation. Unlike FMM, where the wavefront propagates with closed, 1-point-thick curves, the FIM can result in thicker bands that split in places where the characteristic path changes the direction (Fig 2 (a) red point). Also, the wavefront can move over solutions that have already converged, and reactivate them to correct values as new information is propagated across the image. Thus, the worst-case performance of FIM is suboptimal. The following section gives the results of empirical studies, including situations where this worst-case behavior undermines computational efficiency of FIM and compares the results with those of the other state-of-the-art solvers.

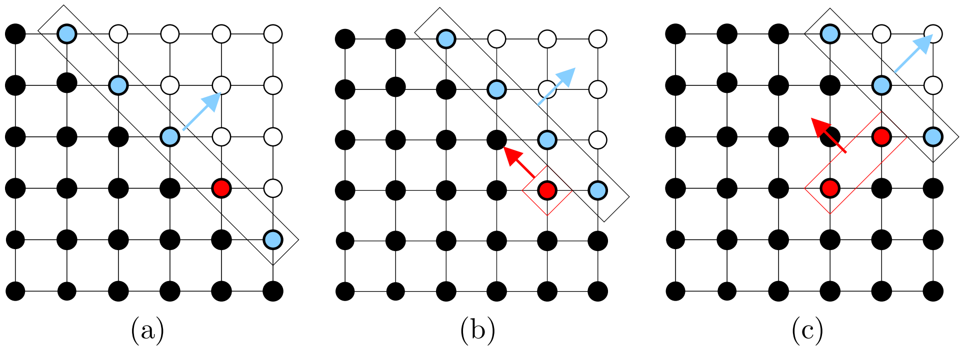


Figure 2: Schematic 2D example of the change of the characteristic direction.

To prove correctness of the algorithm, we follow reasoning similar to that described in [14].

Lemma 2.1. *FIM algorithm converges.*

Proof. For this we rely on monotonicity (decreasing) of the solution and boundedness (positive). From the pseudo code 2.1 we see that a point is added to the active list and its tentative solution is updated only when the new solution is smaller than the previous one. All updates are positive by construction. □

Lemma 2.2. *The solution U at the completion of FIM algorithm with $\epsilon = 0$ (error threshold) is consistent with the corresponding Hamiltonian given in Equation 1.*

Proof. Each point in the domain is appended to the active list at least once. Each point $\mathbf{x}$ is finally removed from $\mathcal{L}$ only when $g(U, \mathbf{x}) = 0$ and the upwind neighbors (which impact this calculation) are also inactive. Any change in those neighbors causes $\mathbf{x}$ to be re-appended to the active list. Thus, when the active list is empty (the condition for completion), $g(U, \mathbf{x}) = 0$ for the entire domain. □

Theorem 2.3. *FIM algorithm, for $\epsilon = 0$ gives an approximate solution to Equation 1 on the discrete grid.*

Proof. The proof of the theorem is given by the convergence and consistency of the solution, as given lemmas above. □

3 Godunov Hamiltonian for the Hamilton-Jacobi equation

In this section we introduce the details of Godunov discretization of H-J Hamiltonian on a 3D grid, which is an extension of the 2D case introduced by Tsai et al. [21]. The simplest way to solve Equation 1 is computing p, q, and r using a central difference method and solve a quadratic equation as in [5], but this approach requires global updates to converge. However, since convex Hamiltonians have strict causality relations with adjacent neighbors, there is a more efficient way to solve it. One approach is using only one-sided derivatives to compute Hamiltonians, e.g., Godunov upwind scheme. We employ a similar Godunov upwind Hamiltonian as in [21], but we have derived an efficient method to evaluate the Hamiltonian.

3.1 Definition

The Godunov Hamiltonian H_G for the H-J equation can be defined as follows [21]:

$$H_G(p,q,r) = \mathrm{ext}_{p \in I[p_-,p_+]} \mathrm{ext}_{q \in I[q_-,q_+]} \mathrm{ext}_{r \in I[r_-,r_+]} H(p,q,r) \tag{3}$$

where

$$\begin{aligned} \mathrm{ext}_{x \in I[a,b]} &= \min_{x \in [a,b]} \ \text{ if } a \le b \\ \mathrm{ext}_{x \in I[a,b]} &= \max_{x \in [b,a]} \ \text{ if } a > b \end{aligned}$$

$p_\pm = D^x_\pm u$, $q_\pm = D^y_\pm u$, $r_\pm = D^z_\pm u$, and $I[a,b]$ is the closed interval bounded by a and b. This definition of the Godunov Hamiltonian looks complicated, but the main idea is evaluating the Hamiltonian $H(p,q,r)$ with all possible combination of $p = \{p_-, p_+, p_\sigma\}, q = \{q_-, q_+, q_\sigma\}$, and $r = \{r_-, r_+, r_\sigma\}$ where p_σ, q_σ, and r_σ are critical points (because the extremum of a convex Hamiltonian occurs only on either the end of the interval or the critical point), and taking the valid minimum solution that satisfies Equation 1. We have eight cases for 2D and 26 cases for 3D to evaluate the Hamiltonian (we do not evaluate for $H(p_\sigma, q_\sigma, r_\sigma)$). To check the validity of the solution for $H(p,q,r)$, Tsai et al. proposed the following conditions [21].

$$\begin{aligned} H(\mathrm{sgn}\max\{(p_- - p_\sigma)^+, (p_+ - p_\sigma)^-\} + p_\sigma, q, r) &= 1 \\ H(p, \mathrm{sgn}\max\{(q_- - q_\sigma)^+, (q_+ - q_\sigma)^-\} + q_\sigma, r) &= 1 \\ H(p, q, \mathrm{sgn}\max\{(r_- - r_\sigma)^+, (r_+ - r_\sigma)^-\} + r_\sigma) &= 1 \end{aligned}$$

Even though the above test to check the validity of the solution looks mathematically clean and works well, practically it is not efficient due to two reasons. First, this test requires three evaluations of the Hamiltonian, which is an expensive operation. Second, we need to use a threshold to numerically check the float equality ($|H-1|<\epsilon$), which may induce numerical errors. The new validity test we propose is based on the observation that if the solution is valid then p, q, and r used to compute the solution must be correct values. For example, if we use $p = p_-$, then $\operatorname{sgn}\max\{(p_- - p_\sigma)^+, (p_+ - p_\sigma)^-\} + p_\sigma = p_-$ must hold. Checking equality for this equation can be done efficiently because we can encode the left and the right side of the equation using integers, +1, 0, and -1, and compare equality of the integers. The right side index is determined by p, and the left side index is determined by p_-, p_+, and p_σ based on the new solution.

$$\text{Right side index} = \begin{cases} 0 & \text{if } p = p_\sigma \\ +1 & \text{if } p = p_+ \\ -1 & \text{if } p = p_- \end{cases}$$

$$\text{Left side index} = \begin{cases} 0 & \text{if } p_- < p_\sigma < p_+ \\ +1 & \text{else if } (p_- + p_+)/2 < p_\sigma \\ -1 & \text{else} \end{cases}$$

The proposed test does not entail an extra burden of Hamiltonian computations, and can be done using only simple integer equality and float inequality comparisons. Our experiments show that using the new validity test can increase the performance about 50% compared to the original method [21].

3.2 Implementation Detail

2D implementation of Godunov Hamiltonian was introduced in [21], but it is not straightforward to extend the method to 3D cases. Therefore, in this section we introduce the implementation of the 3D Godunov Hamiltonian defined in Section 3.1 in detail. Note that we assume the grid sizes along x, y, and z are all 1 for simplicity. First, solving a H-J equation with given neighborhood values can be implemented as follows.

```
==============================================
Function : solve_HJ(p, q, r, u, v, w)
==============================================
// Solve Hamilton-Jacobi equation H(u(x-p),v(x-q),w(x-r)=1
// a,b,c,d,e,f : Upper triangular elements of tensor matrix
```

```
float A = a*u*u + d*v*v + f*w*w + 2.0*(b*u*v + c*u*w + e*v*w);
float B = -2.0*(a*p*u*u + d*q*v*v + f*r*w*w +
                b*u*v*(p+q) + c*u*w*(p+r) + e*v*w*(q+r));
float C = a*u*u*p*p + d*v*v*q*q + f*w*w*r*r +
          2.0*(b*u*v*p*q + c*u*w*p*r + e*v*w*q*r) - _f*_f;
float D = B*B-4.0*A*C;

if(D < 0) return INF;
else return= (-B+sqrt(D))/(2.0*A);
```

Once we solve the H-J equation using the function solve_HJ(), then we need to perform the solution validity check using the method introduced in Section 3.1, which can be implemented as follows.

```
=========================================================
Function : check_valid(mode, i, j, k, newT, ext, rt_idx)
=========================================================
float Df, Db; // forward/backward difference
if(mode == 0)
{
  Df = U(i+1,j,k) - newT;
  Db = newT - U(i-1,j,k);
}
else if(mode == 1)
{
  Df = U(i,j+1,k) - newT;
  Db = newT - U(i,j-1,k);
}
else
{
  Df = U(i,j,k+1) - newT;
  Db = newT - U(i,j,k-1);
}

int lf_idx;
if(Db < ext && ext < Df) lf_idx = 0;
else if((Db+Df)/2 < ext) lf_idx = 1;
else lf_idx = -1;
```

```
return (rt_idx == lf_idx);
```

Godunov Hamiltonian is defined as the extremum of the Hamiltonian in the domain $[p_-, p_+] \times [q_-, q_+] \times [r_-, r_+]$ (Equation 3). Because the extremum for a convex function only occurs either on the boundary of the domain (e.g., p_- or p_+) or the critical point (e.g., p_σ), we can classify the evaluation of the Hamiltonian into three cases, which are corners, edges, and faces. Corner case is when the extremum occurs only on the boundary of the domain, so there are eight cases to evaluate the Hamiltonian, where $p = p_\pm, q = q_\pm$, and $r = r_\pm$ for $H(p, q, r)$. Edge case is when the extremum occurs on the boundary of two axis and critical point on the other axis. For example, if we fix $p = p_\sigma$, then there are four cases, $q = q_\pm$ and $r = r_\pm$ to evaluate the Hamiltonian. We can fix either p, q, or r, so there are 12 different edge cases. Face case is when the extremum occurs on the boundary of one axis and critical points on the other two axis, for example $p = p_\pm$, $q = q_\sigma$ and $r = r_\sigma$. Therefore, for a given 3D node $\mathbf{x} = (i, j, k)$, new solution $g(U_\mathbf{x})$ (Algorithm 2.1) can be computed using the function $g(i, j, k)$ defined as follows.

```
============================================
Function : g(i,j,k)
============================================
// U(i,j,k) : value at grid node (i,j,k)
// a,b,c,d,e,f : Upper triangular elements of tensor matrix

float U_new = INF;

--------------------------------------------
1. Corners
--------------------------------------------
int nu[] = {+1,+1,+1,+1,-1,-1,-1,-1};
int nv[] = {+1,+1,-1,-1,+1,+1,-1,-1};
int nw[] = {+1,-1,+1,-1,+1,-1,+1,-1};

for(int n=0; n<8; n++)
{
  float p = U(i+nu[n],j,k);
  float q = U(i,j+nv[n],k);
  float r = U(i,j,k+nw[n]);
  if(p < INF && q < INF && r < INF)
```

```
  {
    float u = nu[n];
    float v = nv[n];
    float w = nw[n];
    float U_tmp = solve_HJ(p,q,r,u,v,w);
    if(check_valid(0,i,j,k,U_tmp,-(b*v*(q-U_tmp)+c*w*(r-U_tmp))/a,nu[i]) &&
       check_valid(1,i,j,k,U_tmp,-(b*u*(p-U_tmp)+e*w*(r-U_tmp))/d,nv[i]) &&
       check_valid(2,i,j,k,U_tmp,-(c*u*(p-U_tmp)+e*v*(q-U_tmp))/f,nw[i]) &&
       U_tmp >= min(p,min(q,r)) &&
       U_tmp < U_new)
    {
      U_new = U_tmp;
    }
  }
}


--------------------------------------------
2. Edges
--------------------------------------------
int nu[] = {-1,-1,+1,+1,-1,-1,+1,+1, 0, 0, 0, 0};
int nv[] = {-1,+1,-1,+1, 0, 0, 0, 0,-1,-1,+1,+1};
int nw[] = { 0, 0, 0, 0,-1,+1,-1,+1,-1,+1,-1,+1};

for(int n=0; n<4; n++)
{
  float p = U(i+u[n],j,k);
  float q = U(i,j+v[n],k);
  if(p < INF && q < INF)
  {
    float u = nu[n];
    float v = nv[n];
    float w = -(c*u+e*v)/f;
    float r = (c*u*p+e*v*q)/(c*u+e*v); // check divide by zero in actual code
    float U_tmp = solve_HJ(p,q,r,u,v,w);
    if(check_valid(0,i,j,k,U_tmp,-(b*v*(q-U_tmp)+c*w*(r-U_tmp))/a,nu[i]) &&
       check_valid(1,i,j,k,U_tmp,-(b*u*(p-U_tmp)+e*w*(r-U_tmp))/d,nv[i]) &&
       U_tmp >= min(p,q) &&
       U_tmp < U_new)
    {
```

```
      U_new = U_tmp;
    }
  }
}

for(int n=4; n<8; n++)
{
  float p = U(i+u[n],j,k);
  float r = U(i,j,k+w[n]);
  if(p < INF && r < INF)
  {
    float u = nu[n];
    float w = nw[n];
    float v =  -(b*u+e*w)/d;
    float q = (b*u*p+e*w*r)/(b*u+e*w);
    float U_tmp = solve_HJ(p,q,r,u,v,w);
    if(check_valid(0,i,j,k,U_tmp,-(b*v*(q-U_tmp)+c*w*(r-U_tmp))/a,nu[i]) &&
       check_valid(2,i,j,k,U_tmp,-(c*u*(p-U_tmp)+e*v*(q-U_tmp))/f,nw[i]) &&
       U_tmp >= min(p,r) &&
       U_tmp < U_new)
    {
      U_new = U_tmp;
    }
  }
}

for(int n=8; n<12; n++)
{
  float q = U(i,j+nv[n],k);
  float r = U(i,j,k+nw[n]);
  if(q < INF && r < INF)
  {
    float v = nv[n];
    float w = nw[n];
    float u = -(b*v+c*w)/a;
    float p = (b*v*q+c*w*r)/(b*v+c*w);
    float U_tmp = solve_HJ(p,q,r,u,v,w);
    if(check_valid(1,i,j,k,U_tmp,-(b*u*(p-U_tmp)+e*w*(r-U_tmp))/d,nv[i]) &&
       check_valid(2,i,j,k,U_tmp,-(c*u*(p-U_tmp)+e*v*(q-U_tmp))/f,nw[i]) &&
       U_tmp >= min(q,r) &&
```

```
       U_tmp < U_new)
    {
      U_new = U_tmp;
    }
  }
}
```

```
--------------------------------------------
2. Faces
--------------------------------------------
int nu[] = {-1,+1, 0, 0, 0, 0};
int nv[] = { 0, 0,-1,+1, 0, 0};
int nw[] = { 0, 0, 0, 0,-1,+1};

for(int n=0; n<2; n++)
{
  float p = U(i+u[n],j,k);
  if(p < INF)
  {
    float q = p;
    float r = p;
    float u = nu[n];
    float v = u*(c*e-b*f)/(f*d-e*e);
    float w = u*(b*e-c*d)/(f*d-e*e);
    float U_tmp = solve_HJ(p,q,r,u,v,w);
    if(check_valid(0,i,j,k,U_tmp,-(b*v*(q-U_tmp)+c*w*(r-U_tmp))/a,nu[i]) &&
       U_tmp >= p && U_tmp < U_new)
    {
      U_new = U_tmp;
    }
  }
}

for(int n=2; n<4; n++)
{
  float q = U(i,j+nv[n],k);
  if(q < INF)
  {
    float p = q;
```

```
    float r = q;
    float v = nv[n];
    float u = v*(c*e-f*b)/(f*a-c*c);
    float w = v*(c*b-e*a)/(f*a-c*c);
    float U_tmp = solve_HJ(p,q,r,u,v,w);
    if(check_valid(1,i,j,k,U_tmp,-(b*u*(p-U_tmp)+e*w*(r-U_tmp))/d,nv[i]) &&
       U_tmp >= q && U_tmp < U_new)
    {
      U_new = U_tmp;
    }
  }
}

for(int n=4; n<6; n++)
{
  float r = U(i,j,k+nw[n]);
  if(r < INF)
  {
    float p = r;
    float q = r;
    float w = nw[n];
    float u = w*(e*b-c*d)/(a*d-b*b);
    float v = w*(b*c-a*e)/(a*d-b*b);
    float U_tmp = solve_HJ(p,q,r,u,v,w);
    if(check_valid(2,i,j,k,U_tmp,-(c*u*(p-U_tmp)+e*v*(q-U_tmp))/f,nw[i]) &&
       U_tmp >= r && U_tmp < U_new)
    {
      U_new = U_tmp;
    }
  }
}

return U_new;
```

4 GPU Implementation

4.1 GPU FIM for H-J Solver

The FIM algorithm should scale well on various parallel architectures, e.g., multi-core processors, shared memory multiprocessor machines, or cluster systems. We chose the GPU to implement FIM to solve the H-J equation because the current GPUs are massively parallel SIMD processors, providing a very powerful general-purpose computational platform.

The major difference between the CPU and the GPU implementation of FIM is that the GPU employs a block-based updating scheme, as proposed in [7], because the GPU architecture favors coherent memory access and control flows. The original node-based FIM (Algorithm 2.1) can be easily extended to a block-based FIM as shown in Algorithm 4.1. For a block-based update, the domain is decomposed into pre-defined size blocks (we use a 4^3 cube for 3D in the GPU implementation), and solutions of the pixels in the same block are updated simultaneously with a Jacobi update scheme. Therefore, the active list of the GPU maintains the list of active *blocks* instead of nodes.

The GPU FIM algorithm consists of three steps. First, each active block is updated with a pre-defined number of iterations. During each iteration, a new solution of Equation 1 is computed, replace the old solution if the new solution is smaller, and its convergence is encoded as a boolean value. After the update step, we perform a reduction (Section 4.2.3) on each active block to check whether it is converged or not. If a block is converged, we mark it as *to-be-removed*. The second step is checking which neighbor blocks of to-be-removed blocks need to be re-activated. To do this, all the adjacent neighbor blocks of to-be-removed blocks are updated once, and another reduction operation is applied on each of the neighbor blocks. The final step is updating the active list by checking the convergence of each block and remove or insert only active blocks to the list. The following is a GPU FIM pseudo code for updating active blocks (C_p and C_b are introduced in Section 4.2).

Algorithm 4.1: GPU FIM(L, V)

comment: Update blocks b in active list L, V:list of all blocks

while L is not empty

do
- **comment:** Step 1 - Update Active Blocks
- **for each** $b \in L$
  - **do**
    - **for** $i = 0$ **to** n
      - **do** $(b, C_p(b)) \leftarrow g(b)$
    - $C_b(b) \leftarrow \text{reduction}(C_p(b))$
- **comment:** Step 2 - Check Neighbors
- **for each** $b \in L$ **and**
  - **do**
    - **if** $C_b(b) = \text{true}$
    - **then**
      - **for each** 1-neighbor b_{nb} of b
        - **do**
          - $(b_{nb}, C_p(b_{nb})) \leftarrow g(b_{nb})$
          - $C_b(b_{nb}) \leftarrow \text{reduction}(C_p(b_{nb}))$
- **comment:** Step 3 - Update Active List
- clear(L)
- **for each** $b \in V$
  - **do**
    - **if** $C_b(b) = \text{false}$
    - **then** Insert b to L

4.2 GPU Implementation Detail

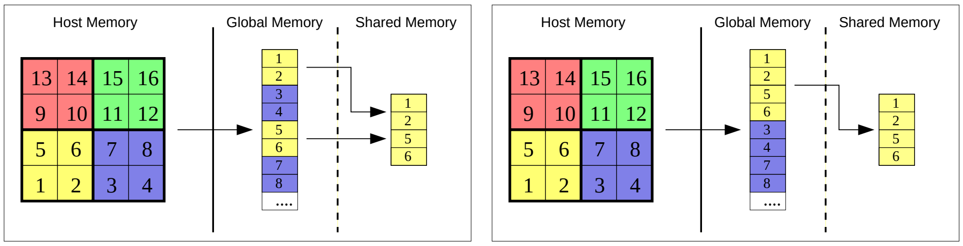
Our GPU H-J solver is implemented on an NVIDIA GeForce 8800 GTX graphics card. NVIDIA CUDA [1] is used for GPU programming, and we will explain the GPU implementation details based on the CUDA programming model, so please refer the CUDA programming guide [1] for more details about the GPGPU programming using CUDA. Computing on the GPU is running a kernel with a batch process of a large group of fixed size thread blocks, which matches well the block-based update method used in the FIM algorithm. We fix the block size to 4^3, so 64 threads share the same shared memory and are executed in parallel on the same processor unit.

Since it is not necessary to use special data structures, e.g., list or vector, to implement the active list on the GPU, we use a simple 1D integer array whose size is the total number of blocks to store active blocks. Only the array elements of index ranging between 0 to (number of total active blocks-1) are valid at any given time. For each CUDA kernel call, the grid size is adjusted to the current number of active blocks, and when a block is being processed, its block index is retrieved from the active list on the GPU. Updating solutions and reductions, which are computationally dominant in the overall process, are done entirely on the GPU.

On the GPU memory, we create two sets of boolean arrays, one C_p with a size of # of pixels (i.e., nodes), and the other C_b with a size of # of blocks, to store convergence of pixels and blocks, in addition to a float array with a size of # of pixels to store solutions. To check the convergence of blocks, we run a reduction on C_p to get C_b. Managing the active list, e.g., inserting or deleting blocks from the list, is efficiently done on the CPU by reading back C_b to the CPU and looping over it to insert only non-converged blocks to the active list. When the list is completely updated on the CPU, it is copied to the GPU, but only a small part of the active list is actually used at any given time (index 0 to (# of active blocks-1)), so only a small fraction of contiguous memory needs to be copied to the GPU.

4.2.1 Data Packing for Coalesced Global Memory Access

To efficiently move data from global to shared memory on the GPU, we need to pack the data on the GPU memory space in a certain way to access global memory as coalesced as possible. A volume is stored in memory as an 1D array with a certain traversing order. Figure 3 shows an example of two different cases of storing a 4x4 image in the GPU global memory space as 1D array when a block is copied to shared memory. Host memory is the CPU side memory, and global / shared memory is the GPU side memory. Color represents the pixels in the same block. Usually pixels are stored from the fastest to the slowest axis order, as shown in Figure 3 (a). In this case, a block is split into two regions in global memory space, which leads to split block accesses. However, if we re-order global memory as shown in Figure 3 (b), accessing a block can be a single coalesced memory access, which is the most efficient way to access global memory on the GPU. Hence, whenever input volumes are copied from the CPU to the GPU memory, a proper re-ordering should be applied so that the block access can be done through a coalesced memory access.



(a) Non-coalesced (b) Coalesced

Figure 3: Example of coalesced/non-coalesced global memory access

4.2.2 Efficient Neighbor Access using Shared Memory

Another factor that affects the GPU performance is accessing shared memory. The shared memory space in the NVIDIA G80 architecture is divided into 16 banks, and 16 shared memory accesses can be done simultaneously as long as all the memory requests refer to different memory banks or to the same memory bank. If any two memory requests, but not all, refer to the same memory bank, i.e., bank conflict, then this request must be serialized and impairs the performance. Because the block size is fixed as 4^3, there is no bank conflict to access pixels inside blocks (block size is a multiple of warp size [1]). However, since we need adjacent neighbor pixels to solve the H-J equation, we should set up an additional shared memory space for left/right/up/down/top/bottom neighbors of the boundary pixels of each block. To avoid bank conflicts, we assign the neighbor pixels to pre-defined banks, which requires a slightly larger extra shared memory space. Figure 4 shows a 2D example of the bank assignment that avoids bank conflicts for neighbor pixel access. The block size for this example is 16 (4x4), which is drawn as a yellow box on the leftmost image in Figure 4. The extra four pixels on each left/right/up/down side of the block are neighbor pixels. The number on each pixel represents the bank number to be assigned. By assigning pixels to shared memory in this pattern, memory requests for left/right/up/down neighbors can be done simultaneously without a bank conflict (Figure 4 red : left neighbors, cyan : right neighbors, green : up neighbors, blue : down neighbors). We need shared memory of size 3*blocksize to store a block and its neighbors because some bank numbers appear twice (1, 4, 13, and 16 in Figure 4). Figure 5 shows an example of actual pixel assignment in shared memory. Figure 5 (a) shows a 2D block diagram with pixel indices (not bank numbers). Figure 5 (b) shows which bank each pixel is actually assigned to. Figure 5 (c) shows a snapshot of a shared memory access pattern when left neighbors are accessed (same case as the second diagram from left in Figure 4). Pixels colored in red are accessed by 16 threads in parallel, and since there is no bank conflict, this memory request can be processed simultaneously. The bank assignment technique shown here can be easily extended to 3D cases.

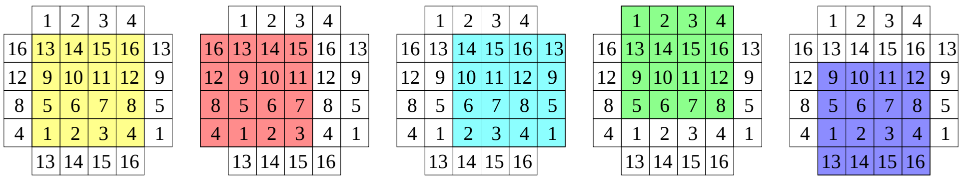
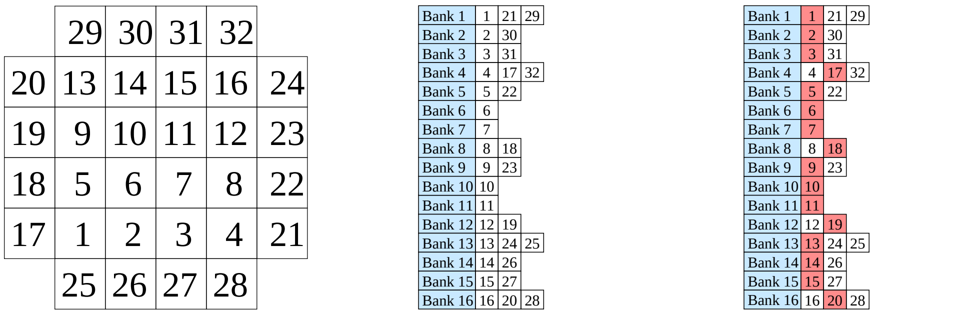


Figure 4: Neighbor pixel access without shared memory bank-conflict



(a) Pixel indices (b) Bank assignment (c) Left neighbors

Figure 5: Bank assignment example

4.2.3 Reduction

Reduction is one of the commonly used computational techniques in the streaming programming model to produce a smaller stream from a larger input stream. To check the convergence of a block, we need to check the convergence of every pixel in the block. Therefore, we need to reduce a block down to a single pixel that represents the convergence of the block. Since CUDA provides a block-wise thread synchronization mechanism, we can perform a parallel reduction [2] in a single kernel execution. To reduce a block of size n, start with $\frac{n}{2}$ threads. For each iteration, every thread participating in reduction reads two convergence values from the current block and write a true or false to one of the original locations (both converge : true, else false). In the next iteration, the number of participating threads is halved and the same reduction is performed. This process is repeated until a block is reduced to a single pixel.

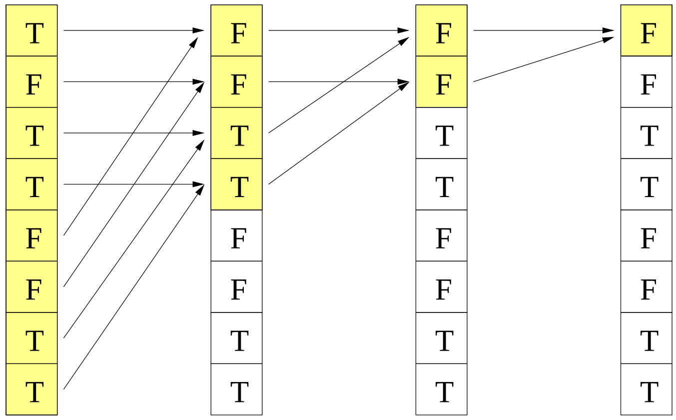


Figure 6: Reduction on a block of size 8

5 Results

Table 1 and Figure 7, 8 show the running time of three H-J equation solvers (GPU, CPU Fast Sweeping with Godunov Hamiltonian, and CPU Fast Sweeping with Lax-Friedrichs Hamiltonian) and their solutions on three synthetic and real tensor volumes. We have tested H-J solvers on a PC equipped with a Intel Core 2 Duo 2.4GHz processor and an NVIDIA GeForce 8800 GTX graphics card.

	Example 1	Example 2	Example 3
GPU FIM	1 sec	1.5 sec	2.8 sec
CPU FS Gdv	54 sec	76 sec	301 sec
CPU FS L-F	142 sec	220 sec	N/A

Table 1: Running time on 3D tensor volumes

Example 1 is a 64^3 volume with a constant tensor elongated along the diagonal direction ($a = d = f = 1.0$ and $b = c = e = 0.9$). The level sets of the solution on this volume is shown in Figure 7 left. The GPU solver took

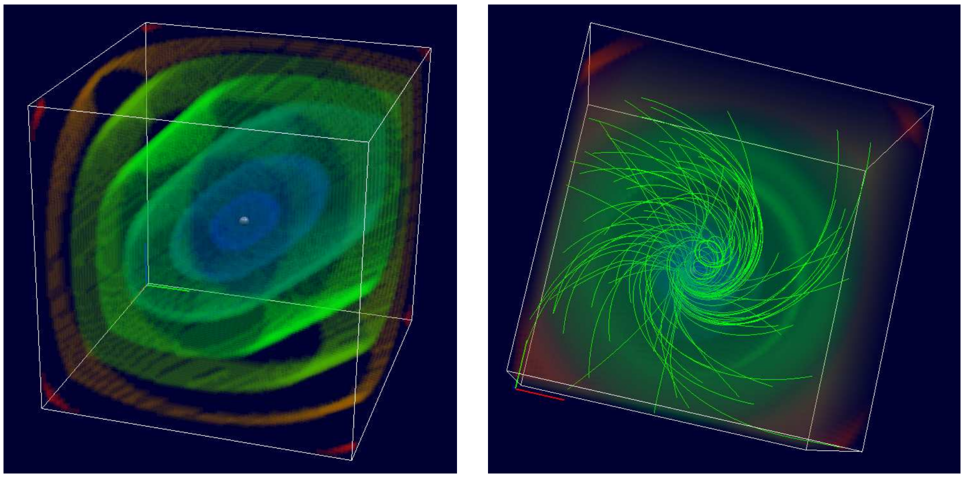


Figure 7: Visualization of distance from a seed point. Left (Example 1) : tensor elongated toward diagonal direction; Right(Example 2) : helix.

only 1 sec while the CPU solvers take about 1–2 minutes to compute the solution on this volume.

Example 2 is a 64^3 volume with tensors aligned to a helix. We built a tensor whose dominant eigenvector is parallel to the tangent vector of the helix curve, and set the dominant eigenvalue as 1 and the other two eigenvalues as 0.1. Figure 7 right is the solution and characteristic paths tracing from randomly distributed points to the seed point placed on the center of the bottom slice. The GPU solver took 1.5 second, while the CPU solvers took 1–3 minutes on this volume.

Example 3 is a DT-MRI brain volume of size 256x256x100, with the number of effective pixels is 196K (we only run the solver inside the white matter mask), and the solution is given in Figure 8. We put a seed at the center of the white matter region. The GPU solver runs less than 3 seconds while the CPU solver took 5 minutes on this volume. We have not tested the Lax-Friedrichs method on this volume because the L-F boundary conditions on the arbitrary boundary is not implemented yet. Overall, the proposed GPU H-J solver runs roughly 50–100 times faster than the commonly used CPU-based methods, allowing users for interactive volumetric paths extraction in DT-MRI volumes.

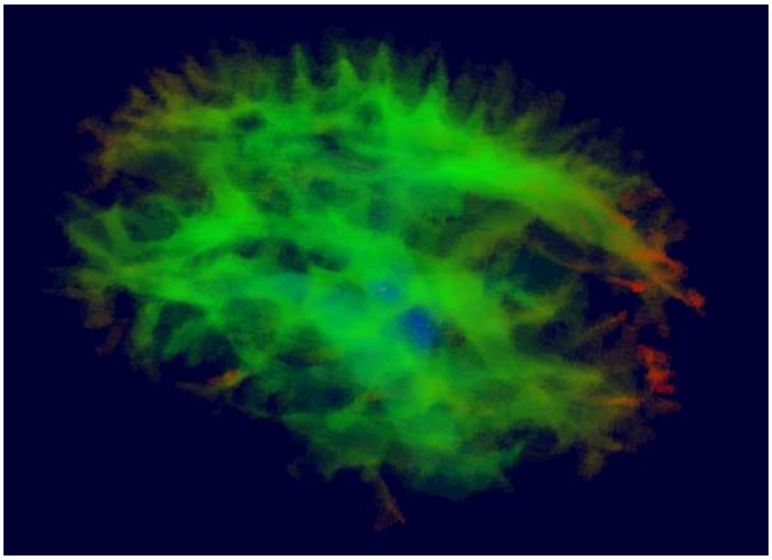


Figure 8: Visualization of distance from a seed point. Example 3: DT-MRI brain data.

6 Conclusion and Future Work

In this paper we propose a parallel H-J solver based on the selective iterative method. The proposed method employs the narrow band approach to keep track of the points to be updated, and iteratively updates the solutions until they converge. Instead of using an expensive sorting data structure to keep the causality, the proposed method uses a simple list to store active points and updates all of them in parallel until they converge. The points in the list can be removed from or added to the list based on the convergence measure. The proposed method is simple to implement and runs faster than the existing solvers on a class of convex Hamilton-Jacobi equations. Our prototype implementation on the GPU runs roughly 50–100 times faster than the state-of-the-art CPU H-J solvers.

Introducing a fast parallel H-J solver opens up a numerous interesting future research directions. Since the GPU implementation allows rapid computation of distance computation on DT-MRI volumes, this makes interactive white matter connectivity analysis feasible. Seismic wave propagation simulation in an anisotropic speed volume will be an interesting application of the proposed method. Fast geodesic computation on parameteric surfaces or volumes can be also interesting future work related to the proposed method.

References

[1] *NVIDIA CUDA Programming Guide, http://developer.nvidia.com/object/cuda.html*, 2007.

[2] NVIDIA CUDA SDK, http://developer.nvidia.com/object/cuda.html, 2007.

[3] A. Bruss. The eikonal equation: some results applicable to computer vision. *J. Math. Phy.*, 23(5):890–896, 1982.

[4] M. Jackowski, C. Y. Kao, M. Qiu, R. T. Constable, and L. H. Staib. Estimation of anatomical connectivity by anisotropic front propagation and diffusion tensor imaging. In *MICCAI*, pages 663–667, 2004.

[5] C. Kao, S. Osher, and J. Qian. Lax-friedrichs sweeping scheme for static Hamilton-Jacobi equations. *Journal of Computational Physics*, 196(1):367–391, 2004.

[6] C. Kao, S. Osher, and Y. Tsai. Fast sweeping methods for static Hamilton-Jacobi equations. Technical report, Department of Mathematics, University of California, Los Angeles, 2002.

[7] 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.

[8] 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.

[9] L. O'Donnell, S. Haker, and C.-F. Westin. New approaches to estimation of white matter connectivity in diffusion tensor MRI: elliptic PDEs and geodesics in a tensor-warped space. In *MICCAI*, pages 459–466, 2002.

[10] G. Parker, C. Wheeler-Kingshott, and G. Barker. Estimating distributed anatomical connectivity using fast marching methods and diffusion tensor imaging. *Transactions on Medical Imaging*, 21:505–512, 2002.

[11] E. Pichon, C.-F. Westin, and A. Tannenbaum. A Hamilton-Jacobi-Bellman approach to high angular resolution diffusion tractography. In *MICCAI*, pages 180–187, 2005.

[12] 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.

[13] 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.

[14] E. Rouy and A. Tourin. A viscosity solutions approach to shape-from-shading. *SIAM Journal of Numerical Analysis*, 29:867–884, 1992.

[15] J. Sethian. A fast marching level set method for monotonically advancing fronts. In *Proc. Natl. Acad. Sci.*, volume 93, pages 1591–1595, February 1996.

[16] J. Sethian. Fast marching methods. *SIAM Review*, 41(2):199–235, 1999.

[17] J. Sethian. *Level set methods and fast marching methods*. Cambridge University Press, 2002.

[18] James A. Sethian and Alexander Vladimirsky. Ordered upwind methods for static Hamilton-Jacobi equations: Theory and algorithms. *SIAM Journal of Numerical Analysis*, 41(1):325–363, 2003.

[19] R. Sheriff and L. Geldart. *Exploration Seismology*. Cambridge University Press, 1995.

[20] 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.

[21] Y.-H. R. Tsai, L.-T. Cheng, S. Osher, and H.-K. Zhao. Fast sweeping algorithms for a class of Hamilton-Jacobi equations. *SIAM Journal of Numerical Analysis*, 41(2):659–672, 2003.

[22] H. Zhao. A fast sweeping method for eikonal equations. *Mathematics of Computation*, 74:603–627, 2004.