Fast Sparse Level Sets on Graphics Hardware

Andrei C. Jalba, Wladimir J. van der Laan, and Jos B.T.M. Roerdink, *Senior Member*, *IEEE*

Abstract—The level-set method is one of the most popular techniques for capturing and tracking deformable interfaces. Although level sets have demonstrated great potential in visualization and computer graphics applications, such as surface editing and physically based modeling, their use for interactive simulations has been limited due to the high computational demands involved. In this paper, we address this computational challenge by leveraging the increased computing power of graphics processors, to achieve fast simulations based on level sets. Our efficient, sparse GPU level-set method is substantially faster than other state-of-the-art, parallel approaches on both CPU and GPU hardware. We further investigate its performance through a method for surface reconstruction, based on GPU level sets. Our novel multiresolution method for surface reconstruction from unorganized point clouds compares favorably with recent, existing techniques and other parallel implementations. Finally, we point out that both level-set computations and rendering of level-set surfaces can be performed at interactive rates, even on large volumetric grids. Therefore, many applications based on level sets can benefit from our sparse level-set method.

Index Terms—Level-set method, sparse representation, sorted tile list, surface reconstruction, octree

1 Introduction

Since its inception by Osher and Sethian [28], the level-set method has become the favorite technique for capturing and tracking moving interfaces, and found a host of applications in a wide variety of research fields ranging from chemistry and physics to computer vision and graphics. The basic idea is to represent the dynamic interface (e.g., contour or surface) implicitly and embed it as the zero level set of a time-dependent, higher dimensional function. Evolving the interface with a given velocity in the normal direction then becomes equivalent to solving a time-dependent PDE for the embedding level-set function. The main advantages of the level-set method are that it allows the interface to undergo arbitrary topological changes and conveniently provides intrinsic geometric properties such as normal and curvature information.

Unfortunately, although the level-set method is well suited for tracking highly deformable models such as mud and water in accurate, physically based simulations [5], [9], [21], its use for interactive systems has been hampered due to the high computational demands. The cost which has to be paid for the flexibility offered by the level-set method is of twofold nature: first, *computationally*, one has to solve the level-set PDE in a higher dimensional space than that of the embedded interface, and second, the *memory requirements* are higher than the size of the interface, as one needs to explicitly store a uniform Cartesian grid for solving the level-set PDE. To address these issues, a number of techniques have been proposed, see Section 2. These methods rely on the fact that it suffices to solve the PDE only in the vicinity of the interface in order to preserve the embedding. Thus, the computational requirements scale with the size of the interface.

In this paper, we leverage the increased computing power of the GPU to achieve interactive simulations based on level sets. This requires specially designed data structures, as most CPU methods rely on complex data structures that do not fit well in the streaming model of GPU computing. Although interactive level-set methods on the GPU do exist, they are constrained to small grid resolutions, see Section 2. The Sorted Tile List (STL), which we introduced recently [37], constitutes an efficient data structure for tracking dynamic interfaces through the level-set method. Inspired by the increased potential for parallelism of the STL method, we present efficient and scalable parallel algorithms for the level-set method on graphics hardware, see Section 3. Although we focus on Nvidia's Compute Unified Device Architecture (CUDA) parallel programming model in our GPU implementation, our algorithms can certainly be adapted to OpenCL [14], as both CUDA and OpenCL share the same SIMD computation model.

Our fast GPU method further improves upon the work by Lefohn et al. [18] to bring the use of level-set methods into the realm of interactive simulations. Specifically, the main contributions of this paper are:

- A scalable parallel mapping of the STL method, running entirely on graphics hardware.
- A multiresolution and a tile-caching scheme, which together with a simple tile-based convergence criterion, improves the overall efficiency of our GPU-STL method.
- A method for surface reconstruction based on GPU level sets, which compares favorably with existing state-of-the-art methods.

- *A.C. Jalba is with the Department of Mathematics and Computing Science, Institute for Mathematics and Computing Science, Eindhoven University of Technology, PO Box 513, 5600 MB Eindhoven, The Netherlands. E-mail: A.C.Jalba@tue.nl.*
- *W.J. van der Laan and J.B.T.M. Roerdink are with the Johann Bernoulli Institute for Mathematics and Computer Science, University of Groningen, Nijenborgh 9, 9747 AG GRONINGEN, The Netherlands. E-mail: laanwj@gmail.com, j.b.t.m.roerdink@rug.nl.*

Manuscript received 14 Dec. 2010; revised 16 June 2011; accepted 1 Feb. 2012; published online 29 Feb. 2012.
Recommended for acceptance by L. Kobbelt.
For information on obtaining reprints of this article, please send e-mail to: tvcg@computer.org, and reference IEEECS Log Number TVCG-2010-12-0293.
Digital Object Identifier no. 10.1109/TVCG.2012.69.

The method can also be made to run on similar highly parallel computing platforms. The simple schemes mentioned above are rather generic and additionally allow a decoupling of the level-set computation engine from the specifics of the problem at hand, thus increasing the usability of our GPU method. Therefore, through our GPU-STL method, many classical applications of level sets can now be substantially accelerated, at high grid resolutions of up to $4{,}096^3$ voxels, which in terms of GPU level sets have not previously been achieved. Finally, for the sake of reproducibility, we provide detailed descriptions and pseudocode of our methods and algorithms.

2 Previous and Related Work

As the level-set method is a tremendously popular approach for tracking moving interfaces, there has been continuous interest in developing improved algorithms to address the computational issues. Here, we focus on closely related methods and state-of-the-art results. For general issues in designing GPU-based algorithms, see [17], [41].

2.1 Efficient Level-Set Methods on the CPU

The computational issue was first addressed with the introduction of the narrow-band schemes [1], [29], [38], which restrict the computations to a small vicinity around the zero level set representing the deforming interface. However, narrow-band methods still need to explicitly store a full grid and additional data structures. Thus, such methods have storage complexities scaling with the size of the grid. Quadtree and octree-based methods [21], [23], [36] do achieve smaller memory footprints, although they usually require uniform refinement along the interface. An alternative approach for reducing memory requirements, called the "Sparse Block Grid" method, was presented by Bridson [4]. In 3D, this method divides the volume of size n^3 into small blocks of size m^3 voxels each. A coarse grid of size $(n/m)^3$ stores at each location a single pointer to a full-resolution block that intersects the interface. Although this method has nonoptimal storage complexity, it maintains constant access time similar to the full-grid method.

2.2 Level-Set GPU Methods

The first GPU implementation of level sets is due to Rumpf and Strzodka [32]. More recently, parallel implementations of particle [7] and marker [22] level sets were also proposed. These methods achieve 15 and 24 fps, respectively, on full grids of size 128^3 voxels. Both methods are more computationally involved than the pure level-set method, and thus a direct comparison with regard to efficiency is unfair. While not sparse, the recent methods of Roberts et al. [31] and Klar [15] represent approaches to implement narrow-band approaches on the GPU. Unfortunately, both methods have rather high memory requirements. However, Roberts et al. [31] show that their method is both work and step efficient.

To the best of our knowledge, the only memory-adaptive model for the level-set representation on the GPU is due to Lefohn et al. [18]. In this method, the domain is decomposed into small 2D tiles, of which only the tiles with nonzero derivatives are stored on the GPU. A lookup table spanning the entire domain stores a pointer to the data for every tile. Memory management is performed by transferring a bit-vector image of about 64 kB from the GPU in every iteration, after which the CPU loads and unloads tiles based on their necessity for the computation during the next iteration.

2.3 Sparse CPU Methods

Recently, Nielsen and Museth introduced the "Dynamic Tubular Grid" (DT-Grid) [25], a recursive, compressed level-set representation inspired by the compressed-row-storage technique used to represent sparse matrices. They showed that the memory requirement of DT-Grid is optimal, i.e., it is proportional to the size of the interface. Moreover, they found the 3D DT-Grid to be faster and more memory efficient than state-of-the-art octree-based approaches.

Huston et al. [11] use hierarchical run-length encoding (RLE) in a dimensional-recursive fashion to encode the domain in a series of runs, each associated with a specific run code. Regions away from the narrow band are encoded to just their sign representation, while the narrow band is stored in full precision. Although this method is more flexible than DT-Grid, the price paid is a slight increase in computation time and memory usage.

Similar to the methods by Lefohn et al. [18] and Bridson [4], the Sorted Tile List method [37] divides the domain into fixed-size *tiles*, such that each tile represents a part of the domain of the level-set function ϕ. Tiles outside the narrow band are discarded. The remaining narrow-band tiles form the "active list." The active neighboring tiles of a given tile can be found in constant time, so that updating the level-set values of all tiles is linear in the number of active tiles. The STL method was found to be faster than the recent approaches mentioned above [4], [11], [25] and more importantly, the algorithm can greatly benefit from both fine- and coarse-grain parallelization by leveraging SIMD and/or multicore configurations.

2.4 Surface Reconstruction

Surface reconstruction from unorganized point clouds has been intensively studied; see [3], [12], [13], [16], [27], [40] and references therein. Despite the increased availability of commodity parallel platforms, there has been very little work on parallel algorithms for surface reconstruction. Only Zhou et al. [40] and Bolitho et al. [3] implemented the Poisson method [13] on the GPU, and on shared and distributed-memory parallel platforms, respectively. In [40], significant speedups were obtained on the GPU at small grid resolutions. As pointed out by Bolitho et al. [3], a limitation is the need to maintain the entire octree and additional data structures in GPU memory, thus drastically limiting the maximum resolution. Moreover, the method is more susceptible to noise than the original one, due to some computational simplifications that were introduced. Finally, the results of Bolitho et al. [3] indicate that the Poisson method scales well on distributed-memory parallel computers and badly on shared-memory architectures.

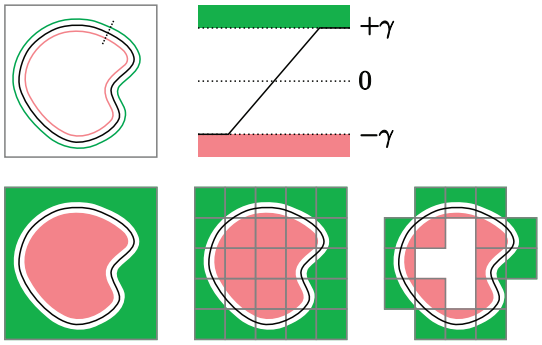

Fig. 1. Sparse tile-based representation in 2D. *Top row, left-to-right* : interface (black), and the $-\gamma$ and γ iso-contours inside/outside the black curve; profile of level-set function ϕ. *Bottom row, left-to-right* : domain with $\phi \geq \gamma$ outside the interface, and $\phi \leq -\gamma$ inside the interface; domain divided into tiles; sparse domain with inactive tiles (completely inside or outside) removed.

3 Proposed GPU Level-Set Method

After a brief overview on level sets, we provide a summary of the CPU-STL method [37]. Then, we introduce our fast GPU method, based on the CUDA paradigm [19].

3.1 Generic Level-Set Equation

In the level-set method, a closed $(d-1)$-dimensional hypersurface $\mathbf{\Gamma}(t=0)$ is implicitly defined as the zero set of a d-dimensional Lipschitz continuous function $\phi(\mathbf{x}, t=0) : \mathbb{R}^d \rightarrow \mathbb{R}$, e.g., the signed distance to $\mathbf{\Gamma}(t=0)$, with $\mathbf{x} \in \mathbb{R}^d$. A generic equation for $\phi(\mathbf{x}, t)$, representing the evolution of $\mathbf{\Gamma}(t)$, is [28]

$$\frac{\partial \phi}{\partial t} = -F(\mathbf{x})|\nabla \phi| - \mathbf{U}(\mathbf{x}, t) \cdot \nabla \phi + \alpha \kappa |\nabla \phi|, \qquad (1)$$

where α is a constant and κ is the (mean) curvature of the hypersurface. Accordingly, the interface moves under three simultaneous influences. The first right-hand side term, involving the position-dependent signed scalar function $F(\mathbf{x})$, defines its motion in the normal direction. Second, it is being passively convected by an external velocity field $\mathbf{U}(\mathbf{x}, t)$, whose direction and strength depend on position and (possibly) time, but not on the front itself. Third, the interface collapses with a speed proportional to its curvature.

The curvature term is discretized using central differences, whereas all the other terms are discretized using upwind differences in the appropriate direction. Here, we use either first-order upwind differencing, or the fifth-order accurate HJ-WENO scheme [20] ensuring less numerical dissipation. For the time derivative, we either use forward differences, or the third-order accurate TVD Runge-Kutta scheme (TVD-RK3). In what follows, whenever the HJ-WENO scheme is employed for spatial derivatives, the TVD-RK3 scheme is used for time integration.

3.2 CPU-STL Method

The Sorted Tile List method [37] divides the domain into fixed-size *tiles* (blocks), such that each tile represents a part of the domain of the level-set function ϕ. Tiles outside the narrow band, with values outside the range $(-\gamma, \gamma)$, are discarded, see Fig. 1. The remaining narrow-band tiles form the active set. A key aspect of the STL method is that the active set is just a list of (active) tiles, *lexicographically ordered* by coordinates. To update the level-set values of a tile, values from the direct neighboring tiles are needed. To do this, the sequential STL method in 3D keeps track of 3^3 pointers to the current and neighboring (active) tiles. In a sequential traversal of the active list, each of these pointers is increased until either the corresponding neighboring tile is found (if it exists in the active list), or the next larger tile (in lexicographical order) is located, if the neighboring tile is not in the active list. Therefore, ordering active tiles by coordinates ensures that the active list is traversed at most 3^3 times, which means that updating the level-set values of all tiles is linear in the number of active tiles.

As the interface evolves, tiles that are no longer close to the zero level set have to be removed, and new tiles needed during the next time step must be added to the active list. This is accomplished in the "tile-management" step, whose basic idea is as follows: For each currently-active tile, it is first determined which of the neighboring tiles are needed in the next time step. The borders of a tile that are being approached by the evolving interface are signalled through a set of activity flags. If the activity flag for a certain border is set, a tile has to be created if it is not yet present in the direction of the border. If the interface has just left a certain tile, all activity flags for that tile are set to zero. If none of the neighboring tiles request for the tile to be retained, it is safely removed from memory. During tile management an expensive resorting step is avoided by carefully tracking newly added and removed tiles. For full details on the STL method, see [37].

3.3 GPU-STL Method

Similar to the STL method, our GPU method, which we call the *GPU-STL method*, relies on lexicographically ordered tiles to represent the narrow band. In our method, each level-set simulation step consists of one *computation* (Section 3.3.4) and one *tile management* (Section 3.3.5) substep. During the computation substep, the active tiles are visited and the level-set PDE is used to update function ϕ. In the tile management substep, new tiles are created and existing tiles are either removed or kept. Although each substep requires a different traversal of the active list (see Algorithms 1 and 2), the general idea is the same, as explained in Section 3.3.2.

Algorithm 1. Iterating over the sorted tile list with T parallel threads and access to the $N = 26$ potential neighboring tiles. Function *iter* is called for each tile by each of the T threads.

Input: *unit* {unit number}, *sub* {thread in unit}, *size* {active list size}, *coord[size]* {tile coords}, *neighborhood[N]* {neighbor coords}

1: *offset* $\leftarrow$ $unit * size/num\_units$ {begin of work for this unit}
2: *end* $\leftarrow$ $(unit+1) * size/num\_units$ {end of work}
3: *ptr* $\leftarrow$ BSEARCH(*size*, *offset*, *sub*) {locate neighbor using binary search}
4: **while** $offset < end$ **do**
5: $cur \leftarrow coord[offset]$ {take coord of current tile}
6: $c \leftarrow cur + neighborhood[sub]$ {neighbor coord}

7: **while** $coord[ptr] < c$ **do**
8: $ptr \leftarrow ptr + 1$ {track neighbor}
9: $match \leftarrow coord[ptr] = c$ {if match, neighbor exists}
10: iter (*match*, *ptr*, *cur*, *unit*, *sub*) {call functor}
11: $offset \leftarrow offset + 1$ {advance to next tile}

Algorithm 2. Iterating over a dilated version of a lexicographically-ordered list of tile coordinates, maintaining the order. Function *iter* is called for each tile by each thread.

Input: *unit* {unit number}, *sub* {thread in unit}, *size* {active list size}, *coord[size]* {tile coords}, *neighborhood[N]* {neighbor coords}

1: *offset* $\leftarrow unit * size/num\_units$ {begin of work for this unit}
2: *end* $\leftarrow (unit + 1) * size/num\_units$ {end of work}
3: *ptr* $\leftarrow$ bsearch(*size*, *offset*, *sub*) {locate neighbor}
4: **loop**
5: $cur \leftarrow coord[ptr] - neighborhood[sub]$
6: $cur \leftarrow$ REDUCE32(min, *cur*) {reduction, minimum value}
7: **if** $cur >= coord[end]$ **then**
8: **break** {end of tile-set reached}
9: **if** $coord[ptr] = (cur + neighborhood[sub])$ **then**
10: $my\_match \leftarrow 2^{sub}$ {if match, this neighbor exists}
11: **else**
12: $my\_match \leftarrow 0$ {otherwise, it does not exist}
13: $match \leftarrow$ REDUCE32(or, *my_match*) {reduce, bitwise-or}
14: iter (cur, match, ptr, unit, sub) {call the functor}
15: **if** my_match **then**
16: $ptr \leftarrow ptr + 1$ {advance structuring element position}

Although the STL method maps well to the CPU computational model, since it relies on tiles to represent the narrow band, an analogous, sparse, tile-based GPU method is more complex. For example, in the STL method, tile management is performed in one traversal of the active list. In contrast, to achieve good memory throughput and thus efficiency, our GPU method performs five passes, using more conceptually involved algorithms for iterating over the active list.

First, we introduce the tile-based data structure that is central to our approach and then we present the details of both computation and tile-management steps.

3.3.1 Data Structure

(see Fig. 2)

- The *active list* is an array of 8-byte structures serving as storage for two integers: *position* (concatenated tile coordinates) and *tile id* (an index into other arrays holding, e.g., the data or border flags of this tile).
- The *data array*, stores values of function ϕ for each tile. We use tiles of size 4^3 voxels, stored in single precision, as we found this to represent the best tradeoff between efficiency and memory footprint. Depending on the time-integration scheme, two or three data arrays have to be maintained on the GPU, see Section 3.1.

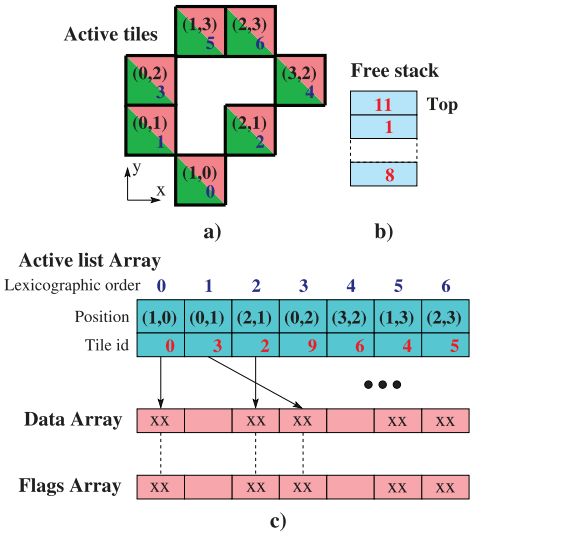

Fig. 2. GPU data structures (arrays). a): active tiles with numbers representing tile positions and indices in the active list, according to the lexicographic order by tile position; b): the stack of free tile identifiers; when a tile becomes inactive, its *tile id* is pushed on the stack; conversely, if a new tile has to be created, its *tile id* is popped from the stack; c): the active list contains tile positions and identifiers; The *tile id* is used as index in the data and (border) flags arrays.

- The array of *border* and *activity flags* of each tile. This array stores the sign of the data outside a tile in each of the 26 directions, in case the tile has no direct neighbor there. The overall activity bit signifies whether a tile is active or not during the next iteration.
- An array of unused indices, the *free stack*, is also maintained, in the form of a stack of *tile id* values.

Tile coordinates are stored as 32-bit integers, as this makes lexicographic comparison very efficient and one can apply neighbor offsets by simple addition and subtraction operations. In 3D, we have at our disposal only 10 bits on average for each dimension, which limits the maximum size of the volume to $(2^{10} \times 4)^3$, using 4^3 tiles. If larger volumes are needed, 64-bit position identifiers could be used at the expense of some speed and storage space.

When a tile becomes active/inactive, its identifier is popped from/pushed on the free stack, see Fig. 2. Only if the stack is empty, new memory must be allocated by the CPU. Maintaining a stack in CUDA is nontrivial, as it must be accessible by all threads in parallel, and there is no way to synchronize thread accesses across blocks. Although it is possible to use atomic integer operations to maintain a stack pointer, this results in a lot of added communication with the global memory. A better option is to count the number of push and pop operations that each thread needs to perform, and then use a parallel prefix-sum algorithm [10] to make sure each thread only accesses its own part of the stack.

3.3.2 Traversal of the Active List

An efficient CUDA algorithm divides the work between CUDA *blocks* and *threads* in a way that yields minimal overhead. For this, we introduce the concept of *units*, i.e., groups of T threads working together on one tile at a time. Each thread among the first $N = 26$ threads of a unit, with $T = 64$ threads, first tracks a pointer to a neighboring

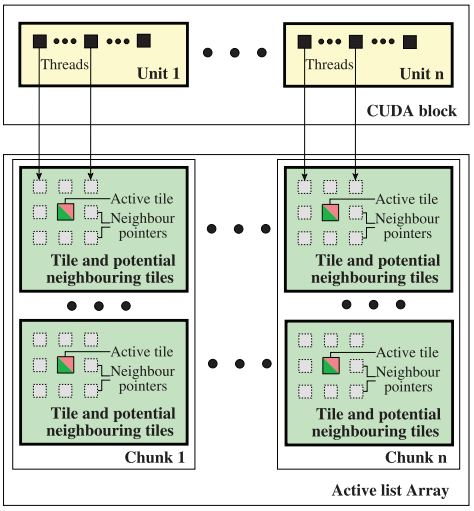

Fig. 3. Tile iteration with access to neighboring tiles. Each of the first N threads of a unit tracks the pointer of a neighboring tile. A CUDA block consists of multiple units, and each unit processes a chunk of the active list.

tile, see Fig. 3. The remaining $T - N = 38$ threads wait until the other (active) threads track the pointers to the neighbors of the currently processed tile. After tracking, each thread of the unit computes one level-set value within the current tile. Finally, the unit advances to the next tile, so that in total, each unit processes a number of consecutive tiles (a chunk) of the active list. A CUDA block can consist of a multiple of these units, which act independently of each other, see Fig. 3. The total number of units is chosen so as to saturate all GPU multiprocessors.

3.3.3 Parallel Tile Iteration with Compute Stencil

Tile iteration with compute stencil, i.e., with access to neighboring tiles, is an essential building block of the GPU-STL method, as it is used for updating the level-set function (Section 3.3.4) and for rendering the deformable surface (Section 3.4). Its pseudocode is given in Algorithm 1. First, on a coarse-grained level, the active list is divided evenly into chunks and each chunk is assigned to a unit (lines 1 and 2). Second, each of the first N threads of a unit assumes responsibility for one neighbor pointer, see also Fig. 3.

At the beginning of the kernel (line 3), each of the first N threads of a unit performs a *binary search* for one of the neighbors of the first tile in the chunk. This is the only binary search that is required, as advancing to the next tile can be done linearly, lines 4-11. The algorithm advances until it reaches the beginning of the next chunk (variable *end*), which is handled by the next unit. For each tile, in each thread, function (object) *iter* is called. The unit (*unit*) and thread (*sub*) identifiers are also passed to the function for convenience, as the values of *match* and *ptr* are different for each thread of a unit. Note that the code in lines 3 and 7-9 (binary search and neighbor tracking) should be protected by a conditional statement (omitted in the pseudocode of Algorithm 1), so that the code is executed only by the first N threads of a unit.

3.3.4 Computation

The data flow in this step is illustrated in Fig. 4. This step is implemented by one CUDA kernel (**calc_kernel**), which

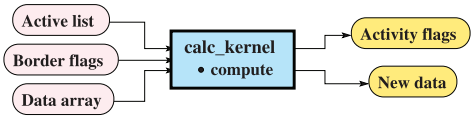

Fig. 4. Data flow in the computation step. CUDA kernel **calc_kernel** receives as input the list of active tiles and their attributes, and updates the tile data according to the level-set PDE. It also updates the *activity flags* used in the tile management step for deciding which tiles are needed at the next iteration.

requires one iteration through the active tile list, using Algorithm 1 (functor **iter** is set to function **compute**). For this kernel, each unit consists of $T = 64$ threads, such that in the **compute** function, all threads of a unit collectively update the 4^3 data elements of a tile. Again, note that only the first $N = 26$ threads of a unit track the pointers of the neighboring tiles, in Algorithm 1. However, since clearly the most expensive part in this step is the actual computation of ϕ values, the resulting CUDA kernel is highly efficient. Also, note that the parameter γ that defines the narrow band is set to $\gamma = 1.5$, as in [37]. Through the **compute** function, thread units perform the following:

1. Read an entire tile into shared memory, having each thread of a unit read a floating-point value. Given that tiles are stored in a consecutive and aligned fashion, this read is coalesced. Tile data are stored in the center of a $6 \times 6 \times 6$ cubic array in shared memory.
2. Read the $6^3 - 4^3 = 152$ border elements. If a neighbor exists in the direction of the border, read the value from device memory, otherwise substitute $-\gamma$ or γ depending on the border flag.
3. Each thread updates ϕ at its location, using (1).
4. If the resulting value is outside the range $(-\gamma, \gamma)$, it is set to the nearest value within the range. Otherwise, the activity bit for this thread is set.
5. Write the entire tile back to device memory, having each thread writing a floating-point value. Similar to the reading step, this write operation is coalesced.
6. Determine the activity flags for this tile by doing a reduction (bitwise-OR the activity flags of the threads), and write the result to device memory. This write is not coalesced, as only the first thread of the unit performs the write operation. However, due to the small amount of data written, this write operation is also efficient.

Since tiles contain 4^3 voxels, this implies that the larger finite-difference discretization stencil required by the HJ-WENO scheme (a 3D cross centered in a 5^3 axis-aligned cube), can also be implemented using only direct neighboring tiles.

3.3.5 Tile Management

For each currently active tile, it is first determined which of the neighboring tiles are needed in the next time step. If the interface approaches a tile border, the tile at the other side of that border has to be present in the next time step to continue the computation. If the activity flag for a certain border is set, a tile has to be created if it is not yet present in the direction of that flag. The basic idea then is to iterate over the list of tiles, and for each tile to expand the tile set by

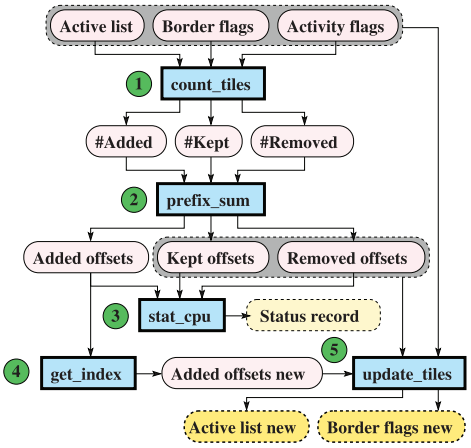

Fig. 5. The overall data flow in the tile management step. Rounded boxes represent data streams, rectangular boxes denote CUDA kernels, and arrows depict directions of data flows; the order in which the kernels are invoked is shown with numbers inside circular boxes.

creating tiles in the directions whose activity flags are set. A straightforward implementation of this idea is to perform a morphological "dilation" of the set of active tiles by a 3^3 structuring element [34]. That is, for each active tile, all its $3^3 - 1$ potential neighboring tiles (dilated set) are considered when expanding the active tiles set. We call the union of all dilated sets (due to each active tile)—the dilated active list (set). Then, for each element in the dilated version of the active list, it should then be determined whether to create, remove, or keep the tile at that position. This assures that new tiles will only be created at most one time.

During the tile management step, Algorithm 2 is used twice to iterate over the dilated version of the active list. This is parallelized similarly to the tile iteration step from Section 3.3.3.

The minimal unit size of $T = 32$ threads is chosen, which equals the warp size of the underlying hardware. Since threads within a warp are automatically synchronized, it is convenient to use this approach if values need to be combined from the variables of individual threads, such as in lines 6 and 13 of Algorithm 2.

In the GPU implementation, tile management cannot be done in just one pass over the active list as in [37], because it is not known in advance how many tiles will be created, removed, or kept. For this reason, we split the tile-management step in multiple passes, each implemented by a separate CUDA kernel, see also Fig. 5:

- **count_tiles**. In the first pass, Algorithm 2 is used to iterate over the data structure. This CUDA kernel simply counts for each unit how many tiles are created, removed, and kept, using the activity and border flags. For each unit, it outputs the number of tiles in each of these categories.
- **prefix_sum**. The second pass performs a parallel prefix-sum [10] on the previously computed counts, to calculate offsets into the old and new active list, for each unit. As the number of thread blocks and units is limited and fixed, this scan can be done quickly in shared memory and then the result written back to device memory. This step converts the number of tiles which are created into an offset into the stack, the number of tiles which are removed into an offset in a list of tiles added to the stack, and finally, the number of tiles which are either kept or created into an offset into the new active list. After this pass, it is possible to check whether there is enough space left to accommodate the new tile set, or if new memory needs to be allocated. To implement this, the total counts are passed back to the CPU.
- **stat_cpu**. The third pass is a very small, one-thread kernel that computes and outputs a status record for the CPU, to determine how much memory is used and whether the structure should be resized. This record contains the following fields: the total number of tiles added, removed, and kept, and the starting offset into the list of free tiles, so that the new tiles are allocated from the end, thus maintaining the LIFO ordering.
- **get_index**. In the following step, a full pass over the offsets of newly added tiles is performed, and a constant value is added so that they can be used as index into the list of free tiles.
- **update_tiles**. In the final pass, Algorithm 2 is reused; however, this time the new active list is created, new tiles are initialized, and the border flags are updated. Note that the resulting active list is already lexicographically ordered by coordinates, without any sorting step being required to assure the preservation of the initial ordering.

When a tile is removed, one of the border bits of each of its neighbors must be updated to reflect whether the tile was inside or outside the interface. This results in a race condition, since more tiles could be changing the same neighbor at the same time. To avoid this, one could use atomic bitwise operations, but these are not supported on all hardware. Therefore, we use another, albeit slower, possibility, by storing a "will be deleted" bit during the first pass in the activity flags for each deleted tile. This is then taken into account in the last pass, when collecting the border-flag mutations of all neighbors and integrating them into the new value for itself.

3.4 Rendering Interface Using CUDA and OpenGL

Our rendering method uses implicit surface polygonization, employing marching cubes. With modern programmable GPUs that support *geometry shaders*, it is possible to generate geometry on the fly. This means that we can use a compact storage format, where rendering is performed directly from our data structure and runs entirely on the GPU. The idea of using a geometry shader to accelerate the marching cubes algorithm was first proposed by Crassin [6]. Our contribution here is an efficient intermediate-storage structure for the cube attributes, so that the output of a CUDA or CPU algorithm that computes a sparse volume can be directly visualized, without the need to access the entire volume in the geometry shader (in the form of a 3D texture, or otherwise).

Our polygonization algorithm is split into two parts. The first part, based on CUDA, iterates through the level-set data structure and generates a compact record for each cube

(voxel) that is intersected by the surface. The following attributes are written for each cube: 1) intersection case; 2) position of the base; 3) values at the corners, relative to the iso-value, quantized using 8 bits. The resulting structure is only 15 bytes: 3×2 for the coordinates, 1 for the intersection case, and 8 for the quantized corner values. For comparison, the marching cubes algorithm can generate up to five triangles (15 vertices) per cube, resulting in a worst case of $15 \times 3 \times 4 = 180$ bytes per cube if one were to store the triangles. Therefore, our representation can potentially improve memory efficiency by a factor of 12, see Section 6.3 for actual results.

The second part, using a geometry shader, processes the list of cube attributes, generates triangle positions and normals, and sends these through the rendering pipeline. The geometry shader takes points as input and generates triangles, thus implementing the second half of the marching cubes algorithm.

A simple extension, consisting in storing additional values required to estimate the surface normal through the gradient of the level-set function, allows the GPU-STL method to render the surface with per-vertex normals for higher quality, at the cost of nearly doubling the memory usage.

4 Proposed Surface Reconstruction Method

Our multiresolution method for surface reconstruction employs convection of the evolving level-set surface (current approximation to the final reconstructed surface) toward the input sample points. Using velocity fields based on the distance transform as in [39] is not an option, since at very large grid resolutions, e.g., $2{,}048^3$ voxels, the storage requirements would be more than 30 GB of RAM. Instead, we use inverse-distance velocity fields similar to [12], which can be evaluated on-the-fly using memory-efficient octree grids. Specifically, the velocity field is based on Shepard interpolation [35] of normalized direction vectors between locations in the narrow band and input point samples.

Formally, let S denote the input set of point samples lying on or near the surface $\partial\mathbf{M}$ of an unknown object M. The problem is to accurately reconstruct the indicator function of M, and then to approximate its surface $\partial\mathbf{M}$ by a smooth triangulated iso-surface. Given a flexible, enclosing level-set surface $\Phi = \{\mathbf{x} \mid \phi(\mathbf{x}, t) = 0\}$, we formulate the reconstruction problem as the convection of Φ in the velocity field $\mathbf{V}$ due to the input samples $\mathbf{x}_i$ given by

$$\mathbf{V}(\mathbf{x}) = \sum_{i=1}^{N} w_i(\mathbf{x})\hat{\mathbf{f}}_i(\mathbf{x}), \qquad \text{where} \tag{2}$$

$$w_i(\mathbf{x}) = \frac{d_i(\mathbf{x})^{-p}}{\sum_{j=1}^{N} d_j(\mathbf{x})^{-p}}, \qquad \hat{\mathbf{f}}_i(\mathbf{x}) = \frac{\mathbf{x} - \mathbf{x}_i}{d_i(\mathbf{x})}, \tag{3}$$

with $d_i(\mathbf{x}) = \sqrt{D_i^2(\mathbf{x}) + \varepsilon^2}$, $D_i(\mathbf{x}) = ||\mathbf{x} - \mathbf{x}_i||$ is the euclidean distance between $\mathbf{x}$ and $\mathbf{x}_i$, $\varepsilon > 0$ is a small softening constant, $N = |S|$ denotes the number of input samples, and p is a constant parameter. Thus, $\mathbf{V}$ is the minimizer of a functional $E(\mathbf{x}, \mathbf{V})$ measuring deviations between tuples of interpolating points $\{\mathbf{x}, \mathbf{V}\}$ and N tuples of interpolated points $\{\mathbf{x}_i, \hat{\mathbf{f}}_i\}$, defined as

$$E(\mathbf{x}, \mathbf{V}) = \frac{1}{2}\sum_{i=1}^{N} d_i(\mathbf{x})^{-p}\|\mathbf{V} - \hat{\mathbf{f}}_i\|^2. \tag{4}$$

Passive convection in the velocity field $\mathbf{V}$ and flexibility of the level-set surface Φ can be obtained using (1), and setting $F(\mathbf{x}) \equiv 0$, $\mathbf{U}(\mathbf{x}, t) \equiv \mathbf{V}(\mathbf{x})$, and $\alpha > 0$. Note that the velocity field $\mathbf{V}$ is evaluated at all locations in the narrow band, not only at Φ, so that *extending the velocity* to all level sets (within the narrow band) is avoided.

4.1 Efficiency and Multiresolution

Letting $P(\mathbf{x}) = \sum_{i=1}^{N} d_i(\mathbf{x})^{-p+1}$ and assuming $p > 1$, $\varepsilon > 0$, it can be shown by the triangle inequality that $\mathbf{V}(\mathbf{x}) = c \cdot (-\nabla P(\mathbf{x})/||\nabla P(\mathbf{x})||)$, where $0 < c < 1$. Thus, since $-\nabla P(\mathbf{x})$ and $\mathbf{V}(\mathbf{x})$ have the same directions and $-\nabla P(\mathbf{x})/||\nabla P(\mathbf{x})||$ is a unit vector, convecting the level set function toward the sample points by taking unit-size steps is optimal. Therefore, evaluating $\mathbf{V}$ for an active tile can be done as follows: First, the potential P is efficiently approximated using the Barnes-Hut algorithm [2] similar to [12]. Then, the resulting *scalar* values are used to compute the normalized gradient of P (using central differences), to finally yield $\mathbf{V}$.

The following simple *caching scheme* was devised to improve the overall efficiency of the method. An additional array is used on the GPU, indexed by *tile id* (see Section 3.3.1), that stores the P values of the currently active tiles. Whenever new tiles are created during the tile-management step, their corresponding pages (containing tile coordinates and identifiers) are stored in the array, so that later P can be evaluated at all locations within these tiles, as required during the next time step to update the level-set function. When inactive tiles are removed, their storage is easily reclaimed and reused, as the array is indexed by tile identifier.

A simple *multiresolution* scheme was also deployed, so as to further improve the efficiency of the method. That is, instead of convecting the level-set surface at the highest resolution, we successively evolve it at gradually increasing resolutions, see Algorithm 3. After initializing the level-set surface to a box at a small starting resolution d (line 1), the octree $\mathcal{O}$ required by the Barnes-Hut algorithm [2] is built (line 2). Given a maximum depth D (corresponding to a grid of size 2^{3D} voxels), the octree $\mathcal{O}$ is built in an attempt to allocate one sample point per octree leaf. If upon insertion of a new sample point, a leaf at depth D is reached that already contains a sample, both samples are discarded and replaced by their centroid. In the process, the number of samples contained by each leaf is also stored. After the tree is built, the centroid and number of samples is computed for each octree node (line 3). Next, the level-set surface Φ is convected at the starting resolution d. Following the Barnes-Hut algorithm, to evaluate P at a location $\mathbf{x}$ in the narrow band, octree nodes are traversed in *depth-first order*. If $\mathbf{x}$ is far from the centroid of a given node n, P is computed using the total number and centroid of the samples in n. Thus, instead of computing *all* contributions of the samples in n, only one contribution due to all samples within the node is considered. Otherwise, if $\mathbf{x}$ is close to centroid of node n, the traversal continues with the child nodes of n.

Algorithm 3. Multi-resolution surface reconstruction.
1: Initialize level-set surface Φ to a box at resolution d
2: Construct octree $\mathcal{O}$ with maximum resolution D
3: Compute centroid and number of samples of each node in $\mathcal{O}$
4: Evolve level-set surface at resolution d
5: **for** $r = d + 1$ to D **do**
6: Upsample level-set surface to resolution r
7: Evolve level-set surface at resolution r
8: Output is the final reconstructed surface Φ at resolution D.

Convergence of the level-set surface at any resolution $r = d, \ldots, D$ is automatically detected, as follows: For each active tile, an 8-bit variable is stored, signifying the "age" of the tile. During the tile-management step, this variable is incremented for each tile that remains active during the next time step, whereas for newly added tiles it is set to zero. When the age of all active tiles is larger than a given value (we use 5), the level-set surface is assumed to have converged. This criterion is efficiently implemented in CUDA using a reduction primitive.

Before advancing to the next resolution, the list of active tiles has to be *upsampled* (line 6). Upsampling the narrow band is accomplished in two steps:

1. Create a new active list using Algorithm 1, by replacing each active tile by eight new tiles. The coordinates of the new tiles are set to $\mathbf{x}' \equiv 2\mathbf{x} + [b_0, b_1, b_2]$, where $b_0 b_1 b_2$ is the binary representation of $c = 0, \ldots, 7$, the index of the newly created tile. The data for the new tiles are computed by trilinear interpolation of the initial tile data.
2. Sort the resulting active list in lexicographic order, using the radix-sort algorithm [33].

The first step is implemented in CUDA in only one compute pass, since each thread working on one initial tile generates a constant number of tiles in the new active list. Note that, although the new active list is eight times larger than the initial one, after the first tile-management step, usually the number of active tiles is halved.

To evolve the level-set surface Φ at resolution $r = d, \ldots, D$, given the octree $\mathcal{O}$ built at resolution D, we proceed as follows: When evaluating P at location $\mathbf{x}$, at resolution r, we in fact evaluate it at location $\mathbf{x}'' \equiv 2^{D-r}\mathbf{x}$. Moreover, we limit the maximum depth during the octree traversal to r, i.e., instead of visiting children of nodes at level r, their centroid and number of samples are used. At the end of the algorithm, the final reconstructed surface is given by the level-set surface Φ at resolution D (line 8).

Note that our multiresolution approach can be applied more broadly to other, related level-set problems, e.g., image segmentation, object tracking, etc.

4.2 All-GPU Method

In an "all-GPU" method, both the convection of the level-set surface and the on-the-fly evaluation of the potential P is performed on the GPU. Furthermore, the octree $\mathcal{O}$ with maximum depth D is also constructed on the GPU, see [40].

As mentioned above, the core of the Barnes-Hut algorithm consists of a depth-first order traversal of the octree. The standard approach to implement such traversals on the GPU uses a stack data structure [40]. However, Popov et al. [30] showed that eliminating the need for maintaining a stack on the GPU, for kd-tree traversals, significantly improved the performance of ray-tracing applications. Therefore, to achieve good efficiency of the Barnes-Hut algorithm in CUDA, we propose the following stackless traversal, see Algorithm 4.

Algorithm 4. Stackless Barnes-Hut algorithm in CUDA. Function *add_pot* accumulates the contribution of the samples within an octree node to the potential P at $\mathbf{x}$.
Input: *tid* {thread id within block}, *root* {root node}, $\mathbf{x}$ {position corresponding to *tid* within current tile}, *node.children_addr* {starting address of *node*'s children in the linear octree represented by *nodes*}, *node.parent* {parent of *node*}, *node.idx* {index of *node* among the children of its parent}.
Output: $P(\mathbf{x})$.
1: $idx \leftarrow 0$ {child index}
2: $P(\mathbf{x}) \leftarrow 0$ {initialize P}
3: **repeat**
4: {visit left-most (first) children}
5: **while** $idx < 8$ **do**
6: **while** *has_children*($root$) **do**
7: $root \leftarrow nodes[root.children\_addr + 0]$ {first child}
8: **if** not $add\_pot(root, \mathbf{x}, P)$ **then**
9: **break**
10: {visit remaining children}
11: **if** *has_parent*($root$) **then**
12: $parent \leftarrow root.parent$
13: **for** $idx \leftarrow root.idx + 1$ to 8 **do**
14: $child \leftarrow nodes[parent.children\_addr + idx]$
15: **if** $add\_pot(child, \mathbf{x}, P)$ **then**
16: **break**
17: {up propagation}
18: **while** *has_parent*($root$) **and** $root.idx = 7$ **do**
19: $root \leftarrow root.parent$
20: {visit next children}
21: **if** *has_parent*($root$) **then**
22: $parent \leftarrow root.parent$
23: **for** $idx \leftarrow root.idx + 1$ to 8 **do**
24: $child \leftarrow nodes[parent.children\_addr + idx]$
25: **if** $add\_pot(child, \mathbf{x}, P)$ **then**
26: **break**
27: **until** not *has_parent*($root$)

First, each CUDA thread tid of the $B = 64$ threads of a CUDA block computes the position $\mathbf{x}$ where P has to be evaluated (based on tile coordinates). Then, each thread executes Algorithm 4. The main algorithm consists of three iterative phases: 1) visiting the left-most subtree, lines 4 to 16, 2) up-propagation to find the next subtree, lines 17 to 19, and 3) visiting of the remaining subtrees, lines 20 to 26. The algorithm terminates when the root node is reached, line 27. Function *add_pot* accumulates the contribution to the potential $P(\mathbf{x})$ of the samples within its $node$ argument and evaluates to *false* if $\mathbf{x}$ is far from the centroid of the samples, and simply returns *true*, otherwise.

4.3 Octree on the CPU

An alternative approach, allowing us to achieve even higher resolutions of the reconstructed surfaces, consists in moving the evaluation of potentials P through the Barnes-Hut algorithm to the CPU. That is, octree $\mathcal{O}$ with maximum depth D is constructed and maintained on the CPU, to overcome the memory-shortage limitations of mainstream GPUs. Thus, in this case, we leverage the computational power of both GPU and CPU, and accordingly, the convection of the level-set surface runs entirely on the GPU, whereas the computation of the velocity field in which the surface is convected runs in parallel on the multiple cores of the host CPU.

The simple caching scheme proposed above, also demonstrates how the communication between the host CPU and the GPU, potentially required by other level-set applications, can be *effectively* accommodated by our level-set method. Clearly, this scheme minimizes GPU-CPU memory transfers, which can only improve the performance of the application at hand. Similar to the GPU approach, when new tiles are created on the GPU, their corresponding pages are sent to the CPU, so that the CPU threads can start immediately evaluating P at the required locations. After the CPU computations terminate, the P values of the new tiles are transferred back to the GPU, so that it can continue with updating the level-set function.

5 Comparison to Previous Approaches

In this section, we make a methodological comparison with previous approaches which were reviewed in Section 2.

Our GPU sparse level-set method is similar to that of Lefohn et al. [18], in that it uses small, fixed-size blocks, i.e., tiles, to represent the narrow band around the interface. However, in contrast to Lefohn's method, we do not need to store a map of the complete domain nor have we to maintain a list of neighbors for each tile. Furthermore, the complex paging mechanism involved by Lefohn's method is avoided altogether, and updating the active tiles is a simple list traversal. Moreover, the GPU-STL method is not bound to a fixed domain. Instead, it allocates and deallocates new tiles as the interface propagates to accommodate the deformations. Another difference is that in our method the GPU handles the tile management step; only a very small data structure (16 bytes) has to be transferred to the CPU in every iteration so that the CPU can check whether the tile list is large enough, or has to be resized.

Unlike our method, the GPU method of Roberts et al. [31] is not sparse. Actually, the latter method has rather high memory requirements, which drastically limits the maximum resolution that can be achieved. The method requires three 3D buffers of size equal to that of the 3D computational domain, and eight additional 1D buffers. It is, however, a narrow-band method, since it only performs computations in a (very) small area around the interface. Further, both level-set computations and management of the active domain (i.e., locations within the narrow band) are performed on the GPU using CUDA. The narrow band contains only positions where the value of the level-set function changed in the previous step (both in time and space) and a subset of neighbors of these locations, i.e., the parameter γ defining the size of the narrow band in our method would have the value $\gamma = 1$. Unfortunately, such a small narrow band limits the accuracy of the finite-difference computations, which might be problematic for certain applications. Note that the method is not based on tiles, but instead it works directly with sets of 3D locations. In [31], it is shown that the method is both work-efficient and step-efficient, and it is faster than Lefohn's method.

Our approach using fixed-size tiles fits very well the computational model of CUDA. By contrast, the DT-Grid [25] requires potentially different handling of every voxel, and furthermore it relies on more complex iterator structures specific for every neighboring voxel, which would result in more registers being used in a CUDA implementation. Additionally, to implement in CUDA the "push" operation used to insert grid points to the DT-Grid data structure, one has to compare the last inserted coordinate to the current one and execute potentially different code consisting of write operations in a random-access fashion. This also means that in a parallel CUDA implementation, the merging step would be more complex. By contrast, in our approach similar to the STL method, implementing the same operation is trivial, as all one has to do is simply append the new tile to the active list. The DT-Grid needs more complex steps to maintain the narrow band, whereas the STL method requires one tile-management step that updates the active list in linear time [37]. Finally, when rebuilding the tubular grid, the entire current domain is dilated, while in the STL method only the tiles which are active at the next time step are added to the active list. Since the hierarchical RLE level-set method of Huston et al. [11] is based on the DT-Grid enhanced with RLE compression, the resulting data structure is even more complex and thus even more difficult to parallelize efficiently. In our tile-based method, access to neighboring tiles has a similar pattern (i.e., all GPU threads execute similar operations), which results in very fast parallel execution. Moreover, tiles can be read or written using coalesced accesses, which is desirable in CUDA to achieve maximum performance [26]. Thus, the proposed data structure maximizes the potential for parallelism.

Although the STL requires about twice as much memory as the DT-Grid, it was shown that the CPU-STL method is about nine times faster than the latter [37]. The increased memory usage of the STL method stems from the fact that the narrow band is tile based, see Fig. 1. Note that, as the DT-Grid was inspired by the compressed-row layout for representing sparse matrices, the STL and our GPU methods are similar to the compressed-block layout. However, since values in the narrow band are in a small range, i.e., $(-1.5, 1.5)$ when $\gamma = 1.5$, fixed-point representations on 16-bits can be used to store single-precision ϕ values, thus improving the memory usage by almost a factor of two, see Section 6.4.

The proposed multiresolution method for surface reconstruction is related to the work by Zhao et al. [39] in that we rely on the level-set method to represent the approximating surface, which unlike [39] is convected in an inverse-distance velocity field based on Shepard interpolation [35].

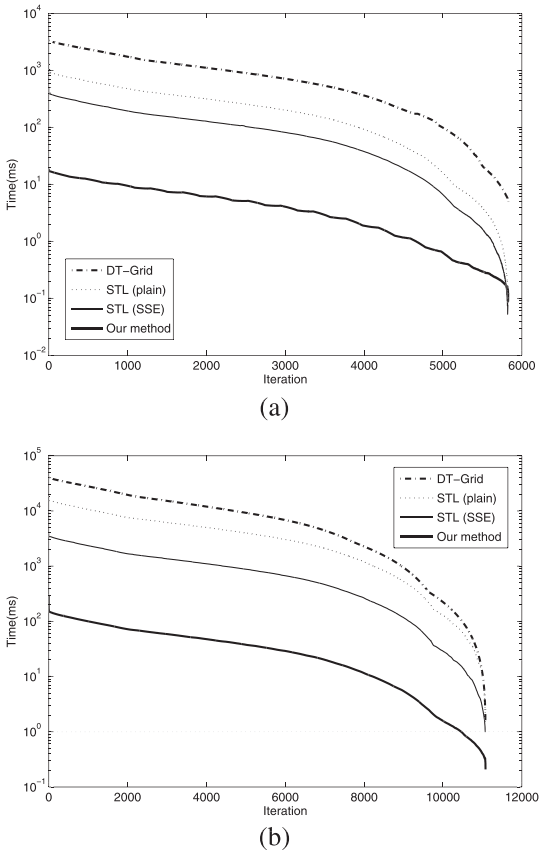

Fig. 6. Efficiency (timing/iteration). (a): first-order numerical scheme; (b): HJ-WENO scheme. Logarithmic plot: time (ms) per iteration for the STL method (both SSE-hand optimized and plain), DT-Grid (open-source implementation [24]) and our GPU-STL method. Initial surface was the Lucy model reconstructed by our method on a $1{,}024^3$ grid.

The field is evaluated on the fly using the "tree algorithm" of Barnes and Hut [2], see Section 4 and [12].

6 Results

In this section, we present experimental results obtained by the proposed methods. All experiments were performed on a machine equipped with an Intel Core 2 Quad CPU at 2.4 GHz, 6 GB RAM and a GeForce GTX 280 GPU (1 GB).

6.1 Efficiency: Comparison to Other Sparse CPU Methods

We performed a direct comparison of our GPU level-set method with state-of-the-art, sparse counterparts running on the CPU. The parameters of all level-set methods were set to $F(\mathbf{x}) = 0.1$, $\alpha = 1$, $\mathbf{U}(\mathbf{x}, t) = 0$, so that the interface collapses to a point, mostly due to motion with speed proportional to its curvature, see (1). In all cases, the initialization was the surface of the Lucy model (see Section 6.4) reconstructed by the GPU-STL method (Section 4) on a $1{,}024^3$ grid; the initial number of active tiles was 127,535. The average timings per iteration in five runs of the STL method (SSE-optimized and plain), DT-Grid (original open-source [24]), and our GPU-STL method are shown in Fig. 6. Since during the simulation, ϕ should be maintained close to a distance transform, all STL methods use for this benchmark a simplified PDE (as frequently used in image segmentation) with an additional rescaling-speed term $\mathrm{sgn}(\phi)(1 - |\nabla\phi|)$ [18], which enforces $|\nabla\phi| = 1$. The open-source DT-Grid performs every time step one iteration of the so-called reinitialization PDE, $\phi_t = \mathrm{sgn}(\phi)(1 - |\nabla\phi|)$. All compared methods use constant time stepping, the same "safe" Courant-Friedrichs-Lewy (CFL) number, $\nu = 0.3$, and the same size of the narrow band, i.e., $\gamma = 1.5$.

TABLE 1
Total Timings and Overall Performance for Both STL Methods (SSE and Plain), DT-Grid [24], and Our GPU-STL Method

Method	Timing (s)	Speedup
DT-Grid	7,051	1.0
STL (plain)	1,506	4.7
STL (SSE)	616.0	11.4
GPU-STL method	29.5	239.0

As can be seen, all methods show similar performance patterns. Only after 4,000 iterations, when the number of active tiles becomes smaller than 10,000, our GPU method becomes slightly less efficient, which is to be expected as the GPU compute resources are not fully used. Note that all methods converge in the same number of iterations (5,830), and also visually the evolutions of the surfaces were the same.

Total timings of the simulation and speedups with respect to the slowest method (the open-source DT-Grid [24]) are given in Table 1. Accordingly, our GPU-STL method is about 20 times faster than the sequential, SSE-optimized STL method and two orders of magnitude faster than the DT-Grid method. Part of this speed difference may be attributed to the use of the simplified PDE. However, in Section 6.3, we use two further benchmarks to show that the order of magnitude difference is maintained when, instead of using an additional rescaling-speed term, reinitialization is separately performed, as in the DT-Grid implementation. Fig. 6b and Table 2 illustrate the performance of the GPU-STL method when discretizing the level-set PDE using the more-computationally involved HJ-WENO scheme. Note that the performance pattern remains approximately the same as with the simpler numerical scheme.

We have also compared our GPU-STL method to a multithreaded, SSE-optimized implementation of the STL method, see Fig. 7. As can be seen, the multithreaded STL implementation is about 2.5 times faster than its sequential version. Note that in this implementation, only the level-set computations have been parallelized. Further, our GPU-STL method is about 6.8 times faster than the multithreaded, SSE-optimized STL.

The memory requirements of the GPU-STL method are similar to those of the STL method, which in turn are about 2.5 times larger than those of the storage-optimal DT-Grid.

TABLE 2
Total Timings and Performance: SSE and Plain STL Methods, DT-Grid [24] and the GPU-STL Method—HJ-WENO Scheme

Method	Timing (s)	Speedup
DT-Grid	136,493	1.0
STL (plain)	47,794	2.9
STL (SSE)	10,541	12.9
GPU-STL method	725	188.1

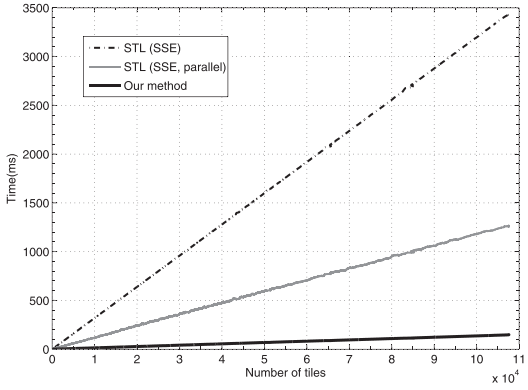

Fig. 7. Performance (timing/no. active tiles) comparison between our GPU-STL method, STL (using SSE) and multithreaded STL SSE, using the HJ-WENO numerical scheme.

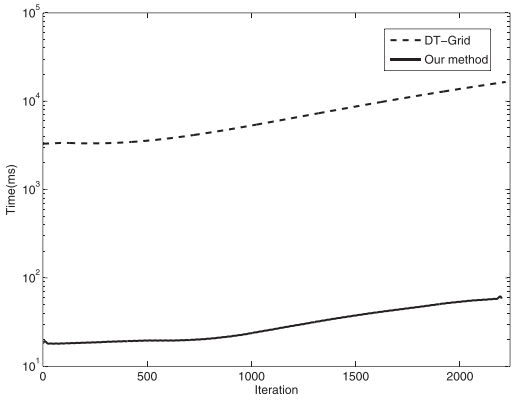

Fig. 9. Performance comparison between our GPU-STL method and the DT-Grid [24] for the "Enright test" [8], using the reinitialization PDE with HJ-WENO numerical scheme in both cases.

6.2 Comparison to Other GPU Methods

We have also performed a direct comparison between the GPU method of Roberts et al. [31] and ours. For both methods, we used the first-order numerical scheme, the parameter settings indicated in Section 6.1 and the same test machine. Further, the size of the narrow band of our method was set using $\gamma = 1.5$, whereas the method of Roberts et al. effectively uses $\gamma = 1.0$, see Section 5. The benchmark consisted in collapsing the surface of the Stanford Dragon model reconstructed on a smaller grid (512^3 voxels), so that the nonsparse method of Roberts et al. could be run. The results of this benchmark agree with those reported in [31] and are shown in Fig. 8. As can be seen, even though our method uses a slightly larger narrow band, it is about five times faster than the method of Roberts et al., which in turn is faster than Lefohn's method. Thus, even though our method performs a few binary searches, which increase its theoretical complexity, in practice the method is both work- and step-efficient, as shown in Figs. 7 and 8.

6.3 Additional Performance Considerations

In this section, we briefly discuss additional performance considerations for the GPU-STL method.

To perform a one-to-one comparison between the STL and DT-Grid methods, we repeated the curvature-collapse benchmark from Section 6.1, but we removed the extra speed term and modified all STL methods to perform one iteration of the reinitialization PDE, per simulation step, similar to the DT-Grid. We found that all STL methods become about 17 percent slower than when using the extra speed term. It is, however, worth mentioning that for this benchmark no noticeable differences were observed when comparing the results of the STL variants, with extra-speed term versus no speed term but separate reinitialization, with respect to 1) number of iterations, 2) size of the active list, and 3) and convergence speed.

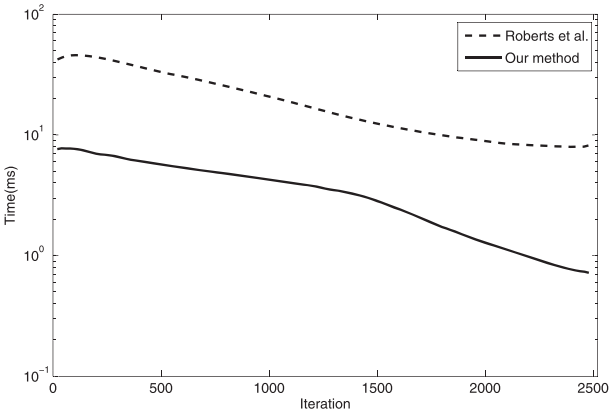

Fig. 8. Performance comparison between our GPU-STL method and the GPU method of Roberts et al. [31].

Although STL methods use fixed-size tiles (with 4^3 voxels) for efficiency reasons, the size of the narrow band can be increased, so as to increase the accuracy of the computations for certain applications. When increasing the size of the narrow band by setting $\gamma = 3.0$, we noticed an overhead of about 15 percent for all STL methods.

The so-called "Enright test" [8] (pure advection) was also performed, to further investigate the performance of the GPU-STL method. The test was performed by both the DT-Grid method [24] and ours, on a grid of size 512^3 voxels, using the HJ-WENO scheme with identical time steps and an even larger narrow band, obtained by setting $\gamma = 5$. Both methods performed one reinitialization step per iteration, using the reinitialization PDE. The test was stopped at time $t = 1.1$, i.e., before the interface starts to break due to extreme stretching and limited grid resolution. The observed behavior was indentical for both methods. Performance figures are shown in Fig. 9, showing that our method is still about two orders of magnitude faster than the DT-Grid implementation, although reinitialization was separately performed, and an even larger narrow band was used. Also, the more reinitialization iterations are performed, the larger the performance gap between the methods will get, in favor of the GPU-STL method. Please note that for this benchmark using the reinitialization PDE instead of the extra-speed term does improve the stability of the simulation.

We also compared the performance of our rendering method (see Section 3.4), which uses a geometry shader to generate geometry (triangle strips), to that of a standard, all-CUDA implementation of the Marching Cubes algorithm. In the latter method, a CUDA kernel is responsible for generating triangle vertices. Geometry (triangles) is then

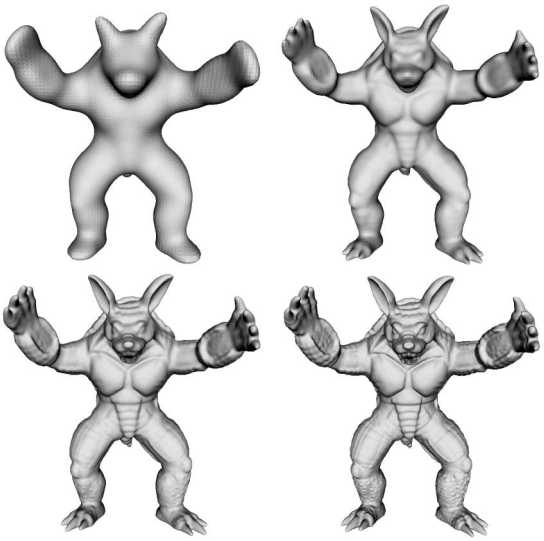

Fig. 10. Multiresolution surface reconstruction of the Armadillo data set (source: Stanford 3D Scanning Repository). *Left-to-right, top-to-bottom* : octree depths $d = 7, 8, 9, 10$.

rendered using standard OpenGL vertex arrays. For any of the models from Figs. 10, 11, and 12, we observed an eightfold improvement in GPU memory usage, by our method. This happens because the all-CUDA method needs to store all triangle vertices on the GPU, whereas ours only stores one small record per intersecting cube. However, our method is about two times slower than the all-CUDA one, due to the relatively low performance of geometry shaders on commodity GPU hardware. Back-face culling, implemented in the geometry shader, can be used to improve the performance by a factor of about two. Thus, any of the models above, reconstructed on an equivalent $1{,}024^3$ grid, can be rendered by our method at least at 10 fps, with substantial savings in GPU memory usage.

6.4 Surface Reconstruction

Throughout all our surface reconstruction experiments, we set $p = 3$ in (3), and use *flat shading* when rendering the reconstructed models, so as to emphasize the smoothness of the surfaces delivered by our method.

According to our discussion from Section 4, the proposed method for surface reconstruction delivers multiresolution representations at increasing grid resolutions. In the first experiment, the octree depth was set to $D = 10$, and we started the reconstruction process at level $d = 7$ from a cube surrounding the Armadillo model, see Fig. 10. The evolution of the level-set surface (1), is steered using $F(\mathbf{x}) = 0$ and $\alpha = 0.1$, where $\mathbf{U}$ is evaluated on the GPU from inverse-distance potentials, see Section 4.1. It took 700 iterations for the level-set surface to converge and reconstruct the Armadillo model at level $d = 7$. Before advancing to the next resolutions, 10 curvature flow iterations were used to produce the models in Fig. 10. Further, at each resolution $r = 8, 9, 10$, less than 100 full iterations were necessary for the level-set surface to converge. Since directly evolving the level-set surface at the maximum resolution $D = 10$ would require well in excess of 1,000 iterations $(700 + 3 \times 100)$, which are also more computationally expensive, our choice of using a multiresolution scheme is, in our opinion, justified.

Fig. 11. Noise behavior. *Left* : noisy range data with $2 \cdot 10^6$ samples and 4,000 outliers. *Right* : reconstructed surface, $D = 10$. *Angel data set courtesy of the U.C. Berkeley Computer Animation and Modeling Group.*

Fig. 12. Reconstruction of two models: Thai statue and Lucy (source: Stanford 3D Scanning Repository) at very large resolutions (octree level $D = 12$).

The proposed method for surface reconstruction withstands large numbers of outliers, due to its reliance on inverse-distance potentials [12]. Moreover, increasing the stiffness of the interface, by adjusting the curvature term in (1), allows it to bypass outlier locations during its evolution, see Fig. 11.

Our *all-GPU method* allows reconstruction of large models on octree grids of up to $2^{11\times3}$ voxels, i.e., the maximum octree depth is set to $D = 11$. Statistics of our all-GPU reconstruction method for the Thai-statue and Lucy models (see Fig. 12) are given in Tables 5 and 6. The time required for updating the level-set function (second column) is comparable to that of the tile management step (fourth column). The evaluation of the potential P, implemented on the GPU using our stackless Barnes-Hut algorithm (third column), accounts for about half of the total time of tile management step. In fact, evaluating P at the required tile locations is about two orders of magnitude faster than its CPU counterpart (results not shown). Finally, the octree construction time at level $D = 11$ on the GPU is less than 0.5 seconds.

The last columns in Tables 5 and 6 show the approximation error $\mathcal{E}$, computed as the average distance from the input samples to the reconstructed (implicit) surface,

TABLE 3
Reconstruction Statistics for the Thai-Statue Model (5 Million Samples)

Octree depth d	Number of iterations	Number of tiles	GPU time(s)	CPU time(s)	Total time(s)
7	704	1,724	1.3	0.9	2.2
8	297	7,127	0.4	1.8	4.4
9	128	38,425	1.3	4.4	10.1
10	107	138,812	2.2	18.2	30.5
11	101	557,030	11.6	80.8	122.9
12	100	2,246,782	60.2	323.2	**506.3**

Octree construction time was 13 seconds (s), the peak memory usage was 860 MB on the GPU and 2.5 GB on the CPU.

i.e., $\mathcal{E} = \sum_{i=1}^{N} |\phi(\mathbf{x}_i)|/N$, where $\phi(\mathbf{x}_i)$ is the level-set function evaluated at the position of an input sample $\mathbf{x}_i \in S$ using trilinear interpolation of the ϕ-values stored in tiles. $\mathcal{E}$ is given as a percentage of the length of the main diagonal of the bounding box (see, e.g., [12], [27]), and it represents an upper bound for the true average distance from the data points to the reconstructed surface. Accordingly, at octree depths $D = 10, 11$, our method attains reconstruction errors comparable to those in [12], [27].

At even larger grid resolutions, the storage requirements of the narrow band on the GPU become larger than the GPU memory. However, by trading accuracy for storage space, we can push the maximum resolution to one octree-level higher, without introducing any visible artifacts. Since within the narrow band, the level-set function satisfies $|\phi| < 1.5$ (assuming $\gamma = 1.5$), a *fixed-point* representation on 16-bits is used to store its 32-bit, single precision values. Similarly, after clamping P values (see Section 3.1) in the range $(0, 10^3)$ and adaptively-compressing them using a logarithmic function, so that more precision is used toward the maximum end of the range, the resulting values are stored again using a fixed-point representation. This simple storage scheme reduces the overall GPU memory footprint by almost a factor of two, allowing the reconstruction of large models at even higher resolution grids. Results and statistics of two such experiments are shown in Fig. 12 and Tables 3 and 4. With increasing resolution, the size of the narrow band becomes about four times larger than that at the previous resolution, resulting in about the same penalty factor at which the speed of both CPU and GPU computations decreases. Thus, at any resolution, both CPU and GPU computational requirements scale with the size of the interface, making our method very efficient. For comparison, the method in [12], which was considerably faster than other approaches at the time, reconstructs the Thai and Lucy models at octree depth $D = 11$ in 28 and 21 minutes, while our all-GPU method needs only 25.7 and 21.7 seconds, respectively, making the GPU-STL method at least one order of magnitude faster. Storing the octree on the CPU and performing reconstruction at level $D = 12$ takes in our method 9 and 8 minutes, respectively. As another comparison, the parallel Poisson surface reconstruction yields the Lucy model at depth $D = 12$ on a distributed-memory cluster with 12 processors in 17 minutes [3], at the expense of data replication across the three workstations constituting the cluster.

TABLE 4
Reconstruction Statistics for the Lucy Model (14 Million Samples)

Octree depth d	Number of iterations	Number of tiles	GPU time(s)	CPU time(s)	Total time(s)
7	602	1,320	1.1	0.8	1.9
8	210	6,691	0.2	0.4	2.5
9	180	35,956	1.1	2.8	6.4
10	120	127,537	2.7	13.7	22.8
11	112	408,362	10.1	62.1	95.0
12	114	2,047,288	58.2	279.2	**432.4**

Octree construction time was 17 seconds (s), the peak memory usage was 800 MB on the GPU and 1.8 GB on the CPU.

TABLE 5
Reconstruction Statistics for the Thai-Statue Model, All-GPU Method

Octree depth d	Compute time(s)	Evaluate time(s)	Update time(s)	Total time(s)	Error (%)
7	1.2	0.7	1.4	2.6	0.1
8	0.3	0.2	0.4	3.3	0.05
9	1.2	0.6	1.0	5.5	0.03
10	2.2	0.9	1.8	9.5	0.02
11	9.9	3.7	6.3	**25.7**	0.009

Octree construction time was 0.3 seconds (s), the peak memory usage was 960 MB on the GPU and 550 MB on the CPU.

6.5 Discussion and Limitations

Our current method has a number of limitations. First, the entire data set has to be kept in GPU memory, i.e., there is no out-of-core support. Therefore, unlike other state-of-the-art CPU methods (e.g., [11], [25]), our method is limited in the maximum resolution that can be achieved, by the (relatively small) amount of memory available on mainstream GPUs. However, in terms of GPU level sets, resolutions of $4{,}096^3$ voxels have not previously been achieved. To overcome this limitation, we could use our convergence criterion to remove from GPU memory those tiles in which the level-set computation already converged, to make space for new tiles. Second, merging two (unsorted) tile lists requires a resorting of the resulting list. On the other hand, if the two lists are already sorted, this can be done in a less expensive merge pass. However, this would only work for steady-state problems.

Tile deletion requires a pass over the entire active list. Moreover, since the active tiles are ordered lexicographically by coordinate, random accesses are logarithmic. For these purposes, using a hierarchical structure or a hash table would be more efficient.

Currently, our method allows accessing only the direct neighbors of each active tile. Thus, if larger discretization

TABLE 6
Reconstruction Statistics for the Lucy Model, All-GPU Method

Octree depth d	Compute time(s)	Evaluate time(s)	Update time(s)	Total time(s)	Error (%)
7	1.0	0.6	1.3	2.3	0.08
8	0.2	0.1	0.2	2.7	0.04
9	0.9	0.5	0.9	4.5	0.02
10	2.4	0.8	1.4	8.3	0.01
11	8.2	2.6	5.2	**21.7**	0.006

Octree construction time was 0.4 seconds (s), the peak memory usage was 940 MB on the GPU and 480 MB on the CPU.

stencils are needed for improved accuracy, the tile size would have to be increased, at the expense of a larger computational burden.

7 Conclusions and Future Work

We have presented an efficient, sparse, tile-based level-set method, called the GPU-STL method, which runs entirely on commodity graphics hardware. We compared our method to other state-of-the-art CPU and GPU approaches, and have shown that ours is substantially faster. Many level-set applications can benefit from our level-set GPU infrastructure. To demonstrate its efficiency, we presented a method for surface reconstruction from point clouds. Our novel multiresolution method for surface reconstruction compared favorably with recent, existing techniques and parallel implementations.

In future work, we shall investigate the possibility of extending our GPU level-set framework to accommodate the particles of the Particle/Marker level-set method. Moreover, work is in progress to adapt our data structure to implement simulations based on the Smoothed Particle Hydrodynamics (SPH) method. Finally, we plan to combine the GPU octree implementation of [40] with our GPU level sets, across a number of GPUs, to achieve very efficient, scalable parallel surface reconstruction, with out-of-core extensions.

Acknowledgments

The authors would like to thank Mike Roberts for making the source code of his GPU level-set method available.

References

[1] D. Adalsteinsson and J.A. Sethian, "A Fast Level Set Method for Propagating Interfaces," *J. Computational Physics,* vol. 118, no. 2, pp. 269-277, 1995.

[2] J.E. Barnes and P. Hut, "A Hierarchical O(N Log N) Force Calculation Algorithm," *Nature,* vol. 324, no. 4, pp. 446-449, 1986.

[3] M. Bolitho, M. Kazhdan, R. Burns, and H. Hoppe, "Parallel Poisson Surface Reconstruction," *ISVC '09: Proc. Fifth Int'l Symp. Advances in Visual Computing : Part I,* pp. 678-689, 2009.

[4] R.E. Bridson, "Computational Aspects of Dynamic Surfaces," PhD thesis, Stanford Univ., Stanford, CA, USA, 2003.

[5] K. Crane, I. Llamas, and S. Tariq, "Real-Time Simulation and Rendering of 3D Fluid," *GPU Gems 3,* H. Nguyen, ed., ch. 30, pp. 633-675, Addison Wesley, Aug. 2007.

[6] C. Crassin, "OpenGL Geometry Shader Marching Cubes," http://www.icare3d.org/blog_techno/gpu/opengl_geometry_shader_marching_cubes.html, 2007.

[7] N. Cuntz, A. Kolb, R. Strzodka, and D. Weiskopf, "Particle Level Set Advection for the Interactive Visualization of Unsteady 3D Flow," *Computer Graphics Forum,* vol. 27, no. 3, pp. 719-726, May 2008.

[8] D. Enright, R. Fedkiw, J. Ferziger, and I. Mitchell, "A Hybrid Particle Level Set Method for Improved Interface Capturing," *J. Computational Physics,* vol. 183, pp. 83-116, 2002.

[9] R. Fedkiw, "Simulating Natural Phenomena for Computer Graphics," *Geometric Level Sets in Imaging, Vision and Graphics,* pp. 461-479, Springer, 2002.

[10] M. Harris, S. Sengupta, and J.D. Owens, "Parallel Prefix Sum (Scan) with CUDA," *GPU Gems 3,* H. Nguyen, ed., Addison Wesley, Aug. 2007.

[11] B. Houston, M.B. Nielsen, C. Batty, O. Nilsson, and K. Museth, "Hierarchical RLE Level Set: A Compact and Versatile Deformable Surface Representation," *ACM Trans. Graphics,* vol. 25, no. 1, pp. 151-175, 2006.

[12] A.C. Jalba and J.B.T.M. Roerdink, "Efficient Surface Reconstruction Using Generalized Coulomb Potentials," *IEEE Trans. Visualization and Computer Graphics,* vol. 13, no. 6, pp. 1512-1519, Nov./Dec. 2007.

[13] M. Kazhdan, M. Bolitho, and H. Hoppe, "Poisson Surface Reconstruction," *Proc. Eurographics Symp. Geometry Processing,* pp. 61-70, 2006.

[14] Khronos OpenCL Working Group, *The OpenCL Specification, version 1.0.29,* Dec. 2008.

[15] O. Klar, "Interactive GPU-Based Segmentation of Large Medical Volume Data with Level-Sets," master's thesis, VRVis and Univ. Koblenz-Landau, 2006.

[16] R. Kolluri, J.R. Shewchuk, and J.F. O'Brien, "Spectral Surface Reconstruction from Noisy Point Clouds," *Proc. Symp. Geometry Processing,* pp. 11-21, July 2004.

[17] C. Lauterbach, M. Garland, S. Sengupta, D. Luebke, and D. Manocha, "Fast BVH Construction on GPUs," *Computer Graphics Forum,* vol. 28, no. 2, pp. 375-384, 2009.

[18] A.E. Lefohn, J.M. Kniss, C.D. Hansen, and R.T. Whitaker, "A Streaming Narrow-Band Algorithm: Interactive Computation and Visualization of Level Sets," *IEEE Trans. Visualization Computer Graphics,* vol. 10, no. 4, pp. 422-433, July/Aug. 2004.

[19] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "NVIDIA Tesla: A Unified Graphics and Computing Architecture," *IEEE Micro,* vol. 28, no. 2, pp. 39-55, Mar./Apr. 2008.

[20] X.-D. Liu, S. Osher, and T. Chan, "Weighted Essentially Non-Oscillatory Schemes," *J. Computational Physics,* vol. 115, no. 1, pp. 200-212, 1994.

[21] F. Losasso, F. Gibou, and R. Fedkiw, "Simulating Water and Smoke with an Octree Data Structure," *Proc. SIGGRAPH '04,* pp. 457-462, 2004.

[22] X. Mei, P. Decaudin, B.-G. Hu, and X. Zhang, "Real-Time Marker Level Set on GPU," *Proc. Int'l Conf. Cyberworlds (CW '08),* Sept. 2008.

[23] C. Min, "Local Level Set Method in High Dimension and Codimension," *J. Computational Physics,* vol. 200, no. 1, pp. 368-382, 2004.

[24] M.B. Nielsen, "DT-Grid Open Source Implementation," http://code.google.com/p/dt-grid, 2012.

[25] M.B. Nielsen and K. Museth, "Dynamic Tubular Grid: An Efficient Data Structure and Algorithms for High Resolution Level Sets," *J. Scientific Computing,* vol. 26, no. 3, pp. 261-299, 2006.

[26] NVIDIA Corporation "NVIDIA CUDA C Programming Best Practices Guide," *CUDA Toolkit 2.3.,* 2009.

[27] Y. Ohtake, A. Belyaev, M. Alexa, G. Turk, and H. Seidel, "Multi-Level Partition of Unity Implicits," *Proc. SIGGRAPH '03,* pp. 463-470, 2003.

[28] S. Osher and J.A. Sethian, "Fronts Propagating with Curvature-Dependent Speed: Algorithms Based on Hamilton-Jacobi Formulations," *J. Computational Physics,* vol. 79, no. 1, pp. 12-49, 1988.

[29] D. Peng, B. Merriman, S. Osher, H. Zhao, and M. Kang, "A PDE-Based Fast Local Level Set Method," *J. Computational Physics,* vol. 155, no. 2, pp. 410-438, 1999.

[30] S. Popov, J. Günther, H.-P. Seidel, and P. Slusallek, "Stackless KD-Tree Traversal for High Performance GPU Ray Tracing," *Computer Graphics Forum,* vol. 26, no. 3, pp. 415-424, Sept. 2007.

[31] M. Roberts, J. Packer, M.C. Sousa, and J.R. Mitchell, "A Work-Efficient GPU Algorithm for Level Set Segmentation," *Proc. Conf. High Performance Graphics (HPG '10),* pp. 123-132, 2010.

[32] M. Rumpf and R. Strzodka, "Level Set Segmentation in Graphics Hardware," *Proc. IEEE Int'l Conf. Image Processing (ICIP '01),* vol. 3, pp. 1103-1106, 2001.

[33] N. Satish, M. Harris, and M. Garland, "Designing Efficient Sorting Algorithms for Manycore GPUs," *Proc. IEEE Int'l Symp. Parallel & Distributed Processing,* pp. 1-10, 2009.

[34] J. Serra, *Image Analysis and Mathematical Morphology.* Academic Press, 1982.

[35] D. Shepard, "A Two-Dimensional Interpolation Function for Irregularly-Spaced Data," *Proc. 23rd ACM Nat'l Conf.,* pp. 517-524, 1968.

[36] J. Strain, "Tree Methods for Moving Interfaces," *J. Computational Physics,* vol. 151, no. 2, pp. 616-648, 1999.

[37] W.J. van der Laan, A.C. Jalba, and J.B.T.M. Roerdink, "A Memory and Computation Efficient Sparse Level-Set Method," *J. Scientific Computing,* vol. 46, no. 2, pp. 243-264, http://dx.doi.org/10.1007/s10915-010-9399-5, 2011.

[38] R.T. Whitaker, "A Level-Set Approach to 3D Reconstruction from Range Data," *Int'l J. Computer Vision*, vol. 29, no. 3, pp. 203-231, 1998.
[39] H. Zhao, S. Osher, and R. Fedkiw, "Fast Surface Reconstruction Using the Level Set Method," *Proc. IEEE Workshop Variational and Level Set Methods in Computer Vision*, pp. 194-202, 2001.
[40] K. Zhou, M. Gong, X. Huang, and B. Guo, "Highly Parallel Surface Reconstruction," technical report, Microsoft Research, 2008.
[41] K. Zhou, Q. Hou, R. Wang, and B. Guo, "Real-Time kd-Tree Construction on Graphics Hardware," *ACM Trans. Graphics*, vol. 27, no. 5, pp. 1-11, 2008.

Andrei C. Jalba received the BSc and MSc degrees in applied electronics and information engineering from "Politehnica" University of Bucharest, Romania, in 1998 and 1999, respectively. He received the PhD degree from the Institute for Mathematics and Computing Science, University of Groningen in 2004. Currently, he is an assistant professor at the Eindhoven University of Technology. His research interests include computer vision, pattern recognition, image processing, and parallel computing.

Wladimir J. van der Laan received the MSc degree in computing science from the University of Groningen in 2005. He recently received the PhD degree from the Johann Bernoulli Institute for Mathematics and Computer Science, University of Groningen. Currently, he is employed by ASML company in Veldhoven, The Netherlands. His research interests include computer graphics, visualization and GPU computing.

Jos B.T.M. Roerdink received the PhD degree in theoretical physics from the University of Utrecht, The Netherlands, in 1983. After a postdoctoral position (1983-1985) at the University of California, San Diego, he joined the Centre for Mathematics and Computer Science in Amsterdam, working on image processing and tomographic reconstruction. He was appointed associate professor in 1992 and full professor in 2003, respectively, at the Johann Bernoulli Institute for Mathematics and Computer Science of the University of Groningen, where he currently holds a chair in Scientific Visualization and Computer Graphics. He is a senior member of the IEEE.