Moving Least-Squares Reconstruction of Large Models with GPUs

Bruce Merry, James Gain, *Member*, *IEEE*, and Patrick Marais, *Member*, *IEEE*

Abstract—Modern laser range scanning campaigns produce extremely large point clouds, and reconstructing a triangulated surface thus requires both out-of-core techniques and significant computational power. We present a GPU-accelerated implementation of the moving least-squares (MLS) surface reconstruction technique. We believe this to be the first GPU-accelerated, out-of-core implementation of surface reconstruction that is suitable for laser range-scanned data. While several previous out-of-core approaches use a sweep-plane approach, we subdivide the space into cubic regions that are processed independently. This independence allows the algorithm to be parallelized using multiple GPUs, either in a single machine or a cluster. It also allows data sets with billions of point samples to be processed on a standard desktop PC. We show that our implementation is an order of magnitude faster than a CPU-based implementation when using a single GPU, and scales well to 8 GPUs.

Index Terms—Moving least squares, surface reconstruction, GPU, out of core

1 Introduction

Laser range scanning is a useful tool in recording a digital model of a building or large site, but the huge amount of data produced by modern scanning campaigns means that efficient processing remains a challenge. With billions of point samples, practical software tools must use out-of-core techniques to deal with limited RAM, and must also be extremely efficient. We have implemented a GPU-accelerated reconstruction system that is able to process massive data sets (billions of points). Although our system extends standard algorithms, to our knowledge it is the first surface reconstruction implementation that is simultaneously GPU-accelerated, out-of-core, and suitable for use with range-scanned data. It also supports execution using multiple GPUs, either in a single machine or in a cluster.

Our data are often noisy and contain small alignment errors which must be smoothed, making methods that simply triangulate the given sample points unsuitable. To produce an accurate historical record, it is important that gaps in the scanner coverage are *not* filled in using interpolation, which rules out methods based on an indicator function [1], [2], [3]. We chose the moving least-squares (MLS) surface definition as it handles noise well without the oversmoothing associated with Poisson reconstruction, is able to deal with holes in the surface, and the quality is frequently better than other modern surface reconstruction techniques [4].

Our system assumes that scans have already been cleaned, registered, and transformed to a common coordinate system [5]. We also assume that each scan sample has an associated oriented normal and an estimate of the local sample spacing—if not already present, these are easily computed using information such as the position of the scanner and the layout of the scanning grid. We refer to such augmented samples as *splats*. Each splat has a *sphere of influence* which defines a local neighborhood: The sphere is centered on the sample position and the radius is the sample spacing estimate scaled by a global user-provided smoothing factor.

To handle the huge data sizes, we spatially partition the data into bins which are small enough to be processed on a GPU. Plane-sweep approaches can require up to 10 percent of the data set to be resident in memory at once, which is infeasible for our largest data sets; hence, we use variably-sized cubic bins [6]. The MLS surface is an implicit surface, so we apply an isosurface extraction algorithm to each bin to produce a mesh. The per-bin operations all run on a GPU, and are discussed in more detail in Section 3. Once all the bins have been processed, these per-bin meshes are stitched together and further processed to produce the output file. This is covered in Section 4.

Even using a GPU for acceleration, a large data set can take hours to process. To further accelerate processing, we can distribute different bins to different GPUs. In Section 5, we describe our implementation for multiple GPUs in a single machine or in a cluster.

Section 6 shows that our single-GPU implementation provides an order-of-magnitude speedup over previous CPU-based approaches, with billions of samples processed in hours rather than days; furthermore, our multi-GPU implementation achieves almost linear speedup with up to 8 GPUs on large data sets. This is achieved while providing a hard bound on GPU memory usage.

2 Background

Surface reconstruction from point clouds is a well-studied field, and it is not possible to provide a complete survey

• *The authors are with the Department of Computer Science, University of Cape Town, Private Bag X3, Rondebosch, 7701, South Africa, and with the South African Centre for High Performance Computing, Cape Town 7701, South Africa. E-mail: {bmerry, jgain, patrick}@cs.uct.ac.za.*

Manuscript received 15 Apr. 2013; revised 24 June 2013; accepted 22 Aug. 2013; published online 5 Sept. 2013.
Recommended for acceptance by A. Sheffer.
For information on obtaining reprints of this article, please send e-mail to: tvcg@computer.org, and reference IEEECS Log Number TVCG-2013-04-0105.
Digital Object Identifier no. 10.1109/TVCG.2013.118.

here: Interested readers are referred to previous surveys [4], [7]. We will list only a few key contributions, focusing on the methods we have implemented.

Surface reconstruction methods generally fall into one of two categories. Interpolation methods use the point samples as vertices in the reconstruction, and compute a triangulation of the point samples. Because scans are not always perfectly aligned, interpolation methods may also create new vertices in overlap regions, but the majority of vertices are still placed at the point samples. Approximation methods define a smooth surface that does not pass exactly through the samples. The smooth surface is often defined implicitly and so standard isosurface extraction methods are used to produce an explicit representation. Approximation methods (including the MLS method we used) are better able to handle noise in the data.

Kil and Amenta [8] use constrained Delaunay triangulations to compute edges, from which they extract and triangulate polygons. This interpolation-based scheme is noteworthy as it is the only scheme we are aware of that is both GPU-accelerated and supports out-of-core operation. However, it requires the points to be locally uniformly distributed, which makes it inapplicable to range-scanned data.

The Poisson approach takes points with oriented normals, interpolates a normal field, and uses a discretized Poisson equation to solve for the indicator function. This algorithm has been extended to support parallel and out-of-core execution [2], [9] and GPU-based acceleration [3]. Because it computes the indicator function, it is guaranteed to produce a water-tight surface. While this is useful for filling small holes, it is unable to reconstruct open surfaces such as those shown in Fig. 11.

Another approximation method that has proven popular is MLS. As this is the method implemented in our work, it is covered in more detail in the next section.

2.1 Moving Least-Squares Surfaces

Alexa et al. [10] define an implicit surface using MLS data approximation. The MLS approach is based on local approximations to the surface. From an initial point $\mathbf{x}$, a surface patch is defined that approximates the surface in the neighborhood of $\mathbf{x}$. A projection operator P maps $\mathbf{x}$ onto this local approximation. The MLS surface is defined as the set of points $\mathbf{x}$ such that $P(\mathbf{x}) = \mathbf{x}$. There are many variations on this basic formulation, which mostly differ in how the local patches are defined.

MLS techniques use a per-sample Gaussian-like weight function to determine the importance of each sample at a point in space, based on the distance to that sample. Some authors use a Gaussian [10], [11], [12], while others use functions with a similar shape but finite support [13], [14]. The function parameters may also vary per sample. We use w_i to denote the function applicable to sample i; in some cases, we also use w_i to mean the value of this function.

The standard deviation of the weight function is either global, or set per-sample based to the local sampling density. Cuccuru et al. [15] and Fiorin et al. [12] also scale the weight function based on an estimate of the sample quality, to prevent low-quality, potentially inaccurate samples from adversely affecting the reconstruction.

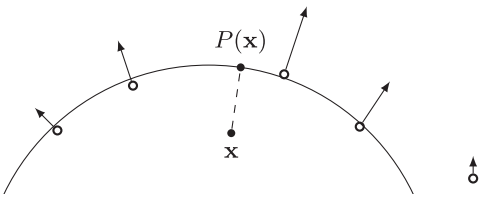


Fig. 1. Projection operation for APSS. The hollow circles represent splats, with arrows representing normal direction and arrow length representing the weight $w_i(\mathbf{x})$. The initial point $\mathbf{x}$ is projected onto the fitted sphere to give $P(\mathbf{x})$.

2.1.1 Algebraic Point Set Surfaces

Guennebaud and Gross [13] introduce algebraic point set surfaces (APSS), which we use in our implementation. As with other MLS methods, the surface is defined in terms of a projection procedure. The difference is that the surface is locally approximated by a sphere rather than a plane or polynomial patch. Compared to a plane, a sphere is better able to capture curvature in the original surface. Given a point $\mathbf{x}$ in space near the surface, the projection $P(\mathbf{x})$ is computed as follows (see Fig. 1):

1. The splats in the local neighborhood of $\mathbf{x}$ are identified, i.e., those for which $\mathbf{x}$ falls within the sphere of influence of the splat.
2. The splats are weighted based on their distance from $\mathbf{x}$, using the weight functions w_i.
3. A sphere is fitted to the splats, using weighted least squares to match the positions and normals. The sphere is represented in an algebraic (implicit) form, which simplifies the fitting procedure and robustly decays to a plane.
4. $\mathbf{x}$ is projected onto the sphere.

The MLS surface is defined as the set of points for which $P(\mathbf{x}) = \mathbf{x}$. P is not a true projection: In general, $P(\mathbf{x}) \neq P(P(\mathbf{x}))$ because the weights at $P(\mathbf{x})$ are not the same as at $\mathbf{x}$. Alexa et al. [16] describe several iterative procedures for projecting points onto the MLS surface.

The sphere is described by a vector $\mathbf{u}$ of five parameters, defining the implicit surface

$$0 = S_{\mathbf{u}}(\mathbf{x}) = (\mathbf{x}^T \ \mathbf{x}^T\mathbf{x} \ 1)\mathbf{u}. \tag{1}$$

Since a sphere can be described by four parameters, there is an extra degree of freedom: Specifically, $\lambda\mathbf{u}$ describes the same surface for any $\lambda > 0$. The gradient of the implicit function is

$$\nabla S_{\mathbf{u}}(\mathbf{x}) = (I_3 \ 2\mathbf{x} \ \mathbf{0})\mathbf{u}. \tag{2}$$

For each splat with position $\mathbf{p}_i$ and normal $\mathbf{n}_i$, we have two linear constraints:

$$S_{\mathbf{u}}(\mathbf{p}_i) = 0, \tag{3}$$

$$\nabla S_{\mathbf{u}}(\mathbf{p}_i) = \mathbf{n}_i. \tag{4}$$

The normal constraints are necessary to compensate for the extra degree of freedom, because the position constraints are trivially satisfied by $\mathbf{u} = \mathbf{0}$. Guennebaud et al. [17] solve for the normal constraints first to compute four of the coefficients, and then use the position constraints only to

compute u_4 (which determines the radius of the sphere). They give the following explicit formulae:

$$u_3 = \frac{(\sum w_i)(\sum w_i \mathbf{p}_i^T \mathbf{n}_i) - (\sum w_i \mathbf{p}_i)^T (\sum w_i \mathbf{n}_i)}{(\sum w_i)(\sum w_i \mathbf{p}_i^T \mathbf{p}_i) - (\sum w_i \mathbf{p}_i)^T (\sum w_i \mathbf{p}_i)}, \quad (5)$$

$$\begin{pmatrix} u_0 \\ u_1 \\ u_2 \end{pmatrix} = \frac{\sum w_i \mathbf{n}_i - 2u_3 \sum w_i \mathbf{p}_i}{\sum w_i}, \quad (6)$$

$$u_4 = \frac{-(u_1 u_2 u_3) \sum w_i \mathbf{p}_i - u_3 \sum w_i \mathbf{p}_i^T \mathbf{p}_i}{\sum w_i}. \quad (7)$$

APSS is attractive for data-parallel implementation because each projection step is given by closed formulae, and hence the number of operations does not depend on the data.

2.2 Out-of-Core Reconstruction

Fiorin et al. [18] sort the samples in each scan in Morton order [19] and construct an in-memory octree over them. The sort ensures that each octree node corresponds to a contiguous part of the file. Octree leaves are then loaded on-demand for MLS-based reconstruction. Because each sample is stored in only one octree node, evaluating the MLS projection may require many nodes to be in memory at once. There is no analysis of the cache size necessary to prevent excessive loading and unloading of octree nodes.

Cuccuru et al. [15] also perform out-of-core MLS reconstruction, but use a streaming approach. The points are sorted along an axis, then processed in a single pass, keeping an active set in memory. Bolitho et al. [2] also use a streaming approach but use a Poisson method for reconstruction. A major limitation of streaming approaches is that the in-core memory requirements are heavily data-dependent, and can be as much as 10 percent of the entire data set [20].

Bernardini et al. [21] extend their ball-pivoting algorithm to run out-of-core by processing the data in slices. Since the algorithm assumes a global bound ρ on the radius of a neighborhood, it is a simple matter to determine which points must be memory resident. Rather than sorting the points along an axis, they load a complete scan when it intersects the active slice, potentially requiring even more memory than streaming approaches.

2.3 Acceleration Using GPUs

Graphics processing units (GPUs) were originally developed for rendering 3D graphics, but have since been used for more general-purpose processing due to their high performance on data-parallel tasks. Our implementation uses OpenCL [22] to target GPUs for the performance-critical steps in surface reconstruction. In OpenCL terminology, computation is done at three scales: kernel, work-group, and work-item. Each time the API is invoked to execute code on the GPU, it executes a single kernel (piece of code) on multiple work-groups, each of which contains multiple work-items. Work-groups are significant because the work-items in a work-group can share data and synchronize with each other. Readers more familiar with CUDA [23] may substitute thread-block and thread for work-group and work-item, respectively.

While OpenCL is portable across a range of devices (including CPUs), maximizing performance still requires tuning for a particular architecture. We have targeted the NVIDIA Fermi architecture [23]. Fermi devices contain up to 16 compute units, each of which contains an L1 cache and a fast local memory space that is shared by a work-group. Work-items are scheduled in groups of 32, called *warps*, which execute in lockstep. Divergent flow control within a warp is inefficient as both sides of the branch are executed. Tens of thousands of threads are required to fully saturate a device. The design, thus, encourages data parallelism, where large numbers of work-items run the identical sequence of operations. Memory transactions are also performed at the warp level, with fewer transactions (and hence greater throughput) if the work-items in a warp all access the same cache line rather than performing scattered accesses.

3 In-Core Processing

We start by describing our algorithm for a subset of data that can be held and processed entirely within GPU memory. The input is a cuboid volume to process along with all samples whose spheres of influence intersect this volume, and the output is a triangle mesh for the portion of the MLS surface that falls inside the volume. This forms the basis for our out-of-core algorithm, which divides the full data set into bins that can be processed by this in-core portion. A bin is further subdivided into cubic *cells*, which form the grid used for isosurface extraction.

A key goal in our design is to bound GPU memory usage. Thus, some design choices are made for their worst-case rather than average-case memory usage. In particular, we limit not only the number of splats, but also the spatial dimensions of bins. By default our system imposes a maximum bin size of $256 \times 256 \times 256$ cells.

We use APSS (see Section 2.1.1) to define the implicit surface. Although APSS has been implemented on a GPU before [17], tuning for GPU architectures has not previously been discussed in detail.

We initially used the following weight function:

$$\phi(d) = \begin{cases} (1-d^2)^4, & \text{if } d < 1, \\ 0, & \text{otherwise}, \end{cases} \quad (8)$$

$$w_i(\mathbf{x}) = \frac{1}{r_i^2} \phi\left(\frac{\|\mathbf{p}_i - \mathbf{x}\|}{r_i \cdot h}\right), \quad (9)$$

where ϕ is an approximation to a Gaussian (but with finite support), r_i is the sample spacing estimate around point i, and h is a global smoothing factor. We later modified this function slightly to avoid numerical instabilities, as described in Section 3.2.

The algorithm is driven by a fully refined octree built over the point samples (see Section 3.1). Section 3.2 describes how we measure the implicit function that defines the MLS surface. We combine this with Marching Tetrahedra to produce a triangle soup (see Section 3.3), which is then converted to a triangle mesh by welding shared vertices (see Section 3.4). The box marked "GPU" in Fig. 2 shows the data flow between these stages.

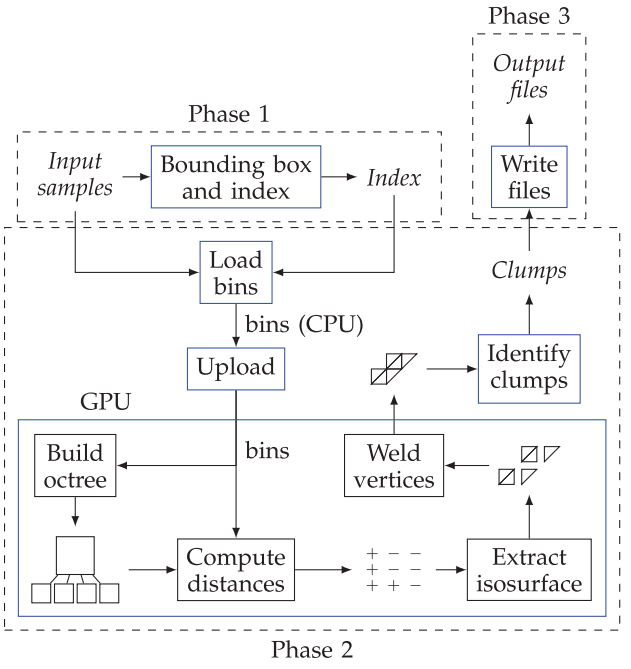


Fig. 2. Data flow in the system. The blue boxes represent independent threads connected by queues, while italics indicate data on disk.

3.1 Octree Construction

When sampling the implicit function giving the signed distance to the MLS surface, we have a point $\mathbf{x}$ in space and need to iterate over all samples whose spheres of influence intersect $\mathbf{x}$. This is a standard spatial indexing problem, which can be accelerated by a variety of data structures. We use an octree as it is simple both to construct and to query, and has successfully been used in previous work on surface reconstruction [2], [15].

Guennebaud et al. [17] investigated several octree implementations for MLS projections on a GPU, and found that a "redundant pyramid" gives high performance and low construction times. The term "pyramid" indicates that the octree is fully refined, with each level stored as a dense array in memory. It is "redundant" in that each splat may be listed in multiple nodes, so that all the splats intersecting a leaf are listed in that leaf or an ancestor. This makes the data structure larger, but eliminates the need to consult neighbors during a walk of the tree. Although Zhou et al. [3] have since devised a practical nonredundant octree representation for GPUs, we use a redundant pyramid as the octree query time is critical to performance, and a pyramid has good worst-case memory usage.

We depart from previous work in performing queries bottom-up rather than top-down, which avoids the need to determine which child to visit as there is only a single parent. When assigning each node an index, we use Morton order [19], which has better spatial coherence than a scan-line ordering, and allows the index of the parent to be found using simple arithmetic on the index. Because the memory usage increases exponentially with the depth of the octree, it must be bounded: this is why we impose a maximum spatial size on bins.

We must still decide at which nodes of the octree to place each splat. Similarly to Guennebaud et al., we list each splat in the nodes it intersects on a single level of the tree, chosen as the finest level whose nodes' side length is greater than the diameter of the sphere of influence. In effect, the sphere is rasterized into a grid at the chosen resolution. Fig. 3 shows a 2D example. This choice guarantees that the sphere will intersect at most eight nodes, and hence the required memory per splat is bounded. This choice of level is also invariant to the alignment of the octree, which is important as it ensures that queries at the boundary between two bins will return samples in a fixed order, independent of which octree is consulted.

We designed the octree encoding to allow the relevant sample IDs to be examined in parallel. A single array of integers, the *sample ID array*, contains both the sample IDs for each node and metadata indicating the start and end of each node. Specifically, a node with M samples is described by a contiguous block of $M+2$ integers, as shown in Fig. 4:

1. the index of the last integer in the block (*end index*);
2. the sample IDs of the samples contained in node; and
3. the index of the start of the nearest nonempty ancestor node, or -1 if there is none (*parent index*).

This metadata allows walking from a node upward through the tree, but additional information is needed to start a walk. A *start array* is indexed by the Morton code of a leaf, and indicates where to begin a walk in the sample ID array.

This octree structure is produced in parallel on the GPU. First, each sample is examined to determine the

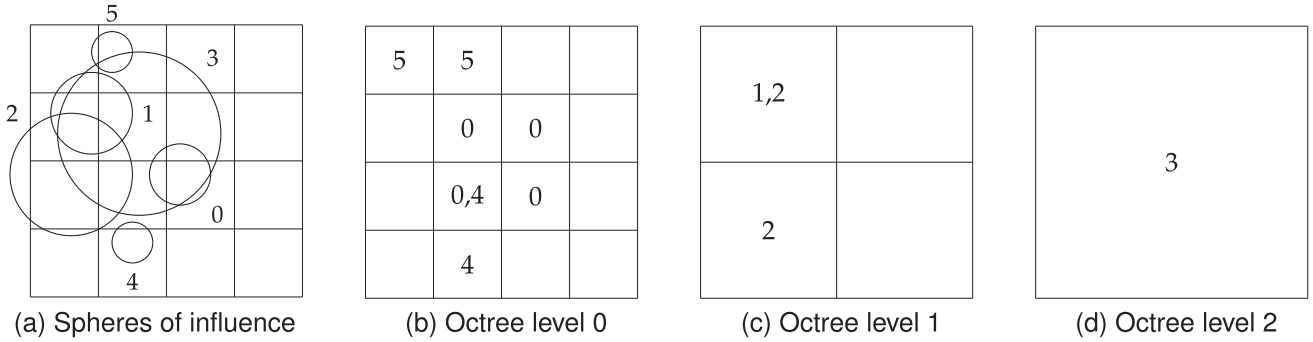


Fig. 3. Example 2D analogue of an octree. Levels are shown fine-to-coarse from left to right. Numbers in the nodes are the IDs of samples that should be considered when visiting the node.

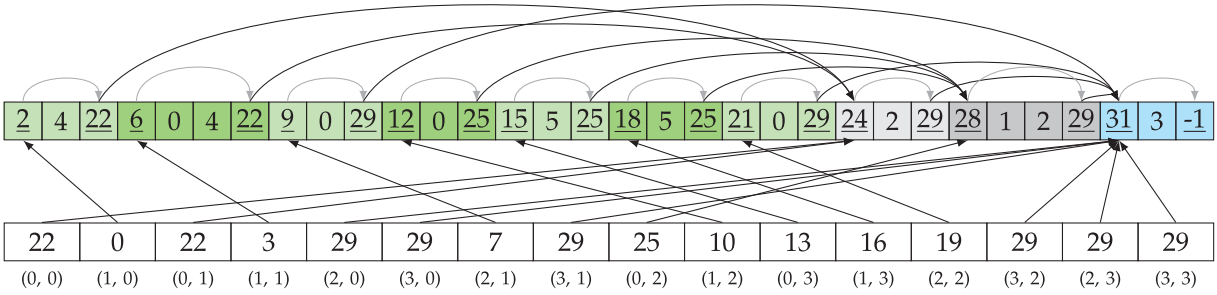


Fig. 4. Sample ID array (top) and start array (below) for the example above. In the sample ID array, underlined numbers encode indices within the array (indicated with arrows) while the other numbers index the actual samples. Above the command array, gray arrows point from the start to the end of the array region corresponding to one octree node, while black arrows point from the end of one node to the start of the nearest nonempty ancestor. Colors delineate portions of the array specific to each level. The start array contains indices into the command array (shown by arrows), and the indices (Morton order) are shown below.

octree nodes in which it will be entered; these are written as eight entries in an array of node indices, with a sentinel value used to fill unused slots. This array is then used as sort keys to radix-sort a corresponding array of sample IDs, such that all sample IDs for a node are sorted together.

At this point the sample IDs are in the correct order, but the end and parent indices are missing. To make space for them, we make use of a parallel prefix sum [24]. We first determine the appropriate gap in the final array between each sample ID and the next, namely 3 for the final sample ID in a node and 1 elsewhere. We then prefix-sum these gaps to obtain the correct location for each sample ID. Subsequent passes copy the sample IDs to these locations and then compute the end and parent indices.

This construction algorithm has similarities with that of Zhou et al. [3], but it is simpler as the nodes do not store information about children or neighbors, and our encoding is different.

3.2 Computing Signed Distances

We use Marching Tetrahedra [25] to reconstruct the surface, which means that we must sample a signed distance field across a regular grid. Rather than computing an exact projection onto the MLS surface, we use a single iteration and take the distance from $\mathbf{x}$ to $P(\mathbf{x})$. We found that this has little impact on quality, confirming previous work [15].

To perform the calculations on the GPU, we use one work-item per grid point. Achieving high performance is challenging, as each grid point has a different neighborhood, possibly with a different size, and so computations are not purely data-parallel. We reduce these effects and also save memory by using a lower resolution octree. Each octree leaf corresponds to an $8 \times 8 \times 8$ set of cells, and we also use this as the work-group size. This loss of resolution leads to more point-inside-sphere tests, but since the octree walk is now common to all work-items in a work-group it can be amortized across them. Within each octree cell, we alternate between loading up to 256 sample IDs in parallel to local memory, and processing those sample IDs—see Algorithm 1. The value 256 was found experimentally to give a good balance between providing enough parallel work for latency-hiding and not using too much local memory; this is likely to be quite hardware-specific.

```
Compute Morton code c for work-group;
p ← start[c];
if p ≥ 0 then
    e ← IDs[p];                    // end position
    p ← p + 1;
    while p < e do
        n ← min(256, e − p);
        foreach i ∈ [p, p + n) do in parallel
            Read IDs[i] to local mem;
            Read position and radius to local mem;
        p ← p + n;
        if p ≥ e then
            p ← IDs[e];            // parent index
            if p ≥ 0 then
                e ← IDs[p];        // end position
                p ← p + 1;
            else
                e ← −∞
        foreach cell corner in work-group do in parallel
            foreach splat in local mem do
                if cell corner inside splat then
                    Process splat;
```

Algorithm 1: Walking the octree to find neighbors. This code describes the actions for a work-group, with the work-items used to parallelize the loops marked as parallel.

When using a 3D work-group, the Fermi architecture arranges the work-items using a scan-line order, causing each warp to cover an $8 \times 4 \times 1$ region. We instead use Morton ordering to manually map a work-item ID to 3D coordinates, giving a $4 \times 4 \times 2$ region per warp. This compact shape is more likely to branch coherently in point-inside-sphere tests and so performance is improved. Note that while there is some cost to decoding the Morton code, this is done only once per work-item and is insignificant compared to the cost of iterating over the neighborhood.

Standard isosurface extraction will create a watertight surface that separates the inside from the outside. However, it is common for there to be gaps in scanner coverage, making it necessary to have holes in the output. We achieve this by using a sentinel value, namely, floating-point not-a-number (NaN), to indicate that a cell corner has neither positive nor negative distance and that no geometry should be generated for the incident cells. We use this in a number

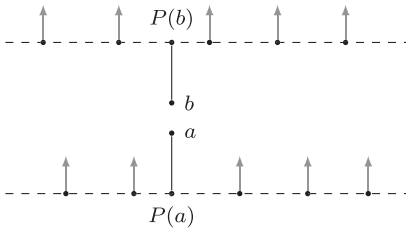


Fig. 5. Misalignment of scans causes a discontinuity in the signed distance. The dashed lines represent the two different scans of the same surface, which have been misaligned. Query points a and b are close, but a projects onto the lower sheet and is considered outside, while b projects onto the upper sheet and is considered inside. Isosurface extraction infers a spurious isovertex between a and b.

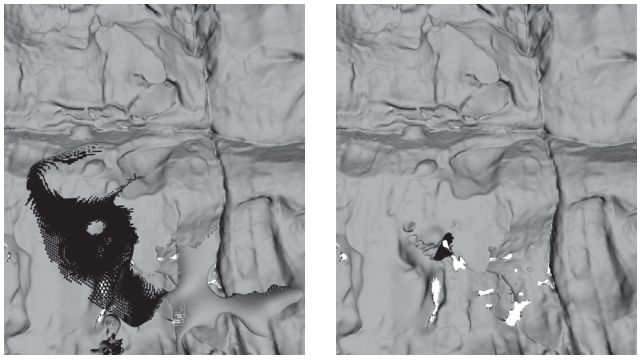
of cases. First, if fewer than four splats are found in the neighborhood (which can only happen far from the surface), the fitting problem can become ill-conditioned. Ill-conditioning is also a problem if all but a few weights are very close to zero: To prevent this, we truncate the weight function to

$$\phi(d) = \begin{cases} (1-d^2)^4, & \text{if } d^2 < 0.99, \\ 0, & \text{otherwise.} \end{cases} \quad (10)$$

While this introduces a discontinuity in ϕ, it is very small and we observed no ill-effects.

We found that in some cases the signed distance function becomes discontinuous far from the surface. Where the discontinuity causes a sign change, the isosurface extraction incorrectly interpolates a vertex. Fig. 5 shows how poorly registered scans can cause this, although we found it to be a problem in other cases as well. The issue usually occurs at a large distance from the surface, so we solve it by replacing any distances greater than the cell diameter with NaN. The isosurface extraction only requires distances at the corners of cells that intersect the isosurface, so larger distances can safely be discarded without introducing holes.

The final case in which we use NaN is for explicit detection of boundaries, to prevent extrapolation. While the finite support of ϕ limits extrapolation, Fig. 6 shows that it remains a problem. We use the same approach as Adamson and Alexa [14]: a point $\mathbf{v}$ on the isosurface is classified based on its distance to the weighted mean $\overline{\mathbf{p}}$ of its



(a) No pruning (b) With pruning

Fig. 6. Without boundary pruning there are significant artifacts. Boundary pruning eliminates the artifacts, although it also opens up some holes that were otherwise filled.

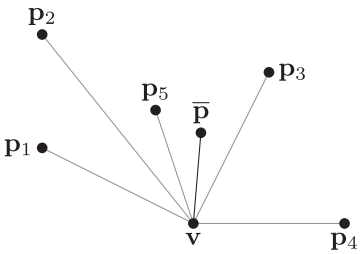


Fig. 7. Boundary detection. Given a point $\mathbf{v}$ on the MLS surface, the neighborhood $\mathbf{p}_i$ is used to compute a weighted mean $\overline{\mathbf{p}}$. If $\mathbf{v}$ lies outside the boundary, the neighborhood will be skewed to one side of $\mathbf{v}$ and so $\overline{\mathbf{p}}$ will be far from $\mathbf{v}$.

neighboring samples. If $\mathbf{v}$ is in the interior of the surface, then $\overline{\mathbf{p}}$ will be close to $\mathbf{v}$, but if $\mathbf{v}$ lies outside the boundary (as in Fig. 7) then they will be far apart.

To make the test scale-invariant, we classify $P(\mathbf{x})$ as extrapolated if

$$\left\| \frac{\sum w_i(\mathbf{x})\mathbf{p}_i}{\sum w_i(\mathbf{x})} - P(\mathbf{x}) \right\| > \gamma \sqrt{\frac{\sum w_i(\mathbf{x}) \|\mathbf{p}_i - P(\mathbf{x})\|^2}{\sum w_i(\mathbf{x})}}. \quad (11)$$

Here, γ is a tuning factor which allows the user to either increase boundary removal to obtain more certain results, or reduce it to cause more small holes to be filled. The default value is $\gamma = \frac{512\sqrt{6}}{693\pi}$; the derivation of this value can be found in the appendix, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TVCG.2013.118. Note that we have used $w_i(\mathbf{x})$ rather than $w_i(P(\mathbf{x}))$: while marginally less accurate, it is significantly faster as it allows all the sums computed for (5)-(7) to be reused without a second pass over the neighborhood to compute new weights.

3.3 Isosurface Extraction

To keep the implementation simple, we avoid using an isosurface extraction technique that adapts its resolution to the sample density. In many cases, the variations in sampling density are unwanted, such as oversampling where scans overlap and undersampling in hard-to-reach areas, and adaptive reconstruction will lead to unnecessary detail in the former and visible edges in the latter. This assumption can still be a limitation, such as when a building is to be reconstructed at high resolution while a large area of surrounding landscape needs only low resolution.

We chose Marching Tetrahedra for isosurface extraction as it is simple to implement while avoiding the ambiguities in Marching Cubes [26]. It also interpolates more vertices and hence potentially gives a smoother surface for the same number of distance field evaluations. As in standard Marching Cubes implementations, the sign of the distance field at each corner of a cell is used to build an 8-bit cell code, which is used to index lookup tables defining the geometry.

Isosurface extraction is performed on the GPU, in a manner similar to previous work [3], [27]. Since the isosurface extraction accounts for less than 20 percent of runtime, we have not done any extensive optimization or comparison of alternative techniques.

Most cells in the grid produce no geometry, and using a work-item for every cell would waste a large number of

work-items. Instead, we do an initial pass to build a list of all cells that will produce geometry (see Algorithm 2), and following passes use a work-item per element in this list. As described in the previous section, cell corners may have the distance set to NaN to signal a hole: This is handled in this pass by skipping cells where at least one corner is a NaN.

Data: N^v: table of number of vertices per cell code
Data: N^t: table of number of triangles per cell code
Input: D: distance field
Input: $[z_0, z_1)$: range of slices to process
Output: L: list of non-empty cell coordinates
Output: S^v: number of vertices per slice
Output: S^t: number of triangles per slice
foreach cell *in slices* $[z_0, z_1)$ **do** in parallel
  Generate cell code c from D;
  if $c \neq 0$ **and** $c \neq 255$ **and** *no corner is a NaN* **then**
    Append cell to L;
    Increment $S^v_{\text{cell}.z}$ by N^v_c;
    Increment $S^t_{\text{cell}.z}$ by N^t_c;

Algorithm 2: GPU isosurface preprocessing (`GenerateCells`). This runs on the GPU, with a separate work-item per cell.

Subsequent passes determine the number of vertices and triangles per cell, compute prefix sums of those counts, and generate the geometry to output buffers. The triangles generated within a single cell form an indexed mesh, but vertices on cell boundaries are duplicated for each available cell. While this increases the number of interpolations, it ensures that cells can be processed in parallel without dependencies. Another pass, described in the next section, is used to merge the shared vertices and produce a connected mesh.

There is not enough GPU memory to guarantee that an entire 256^3-cell bin can be processed in a single pass: In the theoretical worst case, each cell will generate 13 vertices and 36 indices, or 300 bytes per cell. Instead, we process the bin in *swaths* of $N = 24$ slices at a time. Since unextended OpenCL does not allow writes to 3D images, we store the distance field in a 2D image with slices packed along the Y-axis; using $N = 24$ guarantees that this image will not exceed 8,192 pixels in the Y direction (the minimum required support for OpenCL). We alternate between computing the distance field for N slices and generating geometry for those slices. Note that since each slice has corners both above and below, we actually need $N + 1$ slices of the distance field at a time. The extra slice is copied from the previous swath.

Even with this reduction in memory requirements, we still do not have enough memory for the worst case. We allocate enough memory to hold the worst case for just two slices, which in actual use is usually sufficient to hold the output for the entire bin. As we prepare a swath, we count the number of vertices and triangles that will be emitted for each slice, and if necessary we subdivide the swath into manageable pieces. This is shown in Algorithm 3. If the buffers are not large enough to hold the output for the entire bin, the bin is split. Each sub-bin produces a separate mesh, with boundary vertices duplicated. We deal with stitching the meshes back together as part of our out-of-core algorithm in Section 4.2. In our test cases, fewer than 10 percent of bins are split, and fewer than 0.01 percent of swaths are subdivided.

Input: $[z_0, z_1)$: a range of slices to process
Input: D: distance field
In/Out: B^v, B^t: buffers for vertices/triangles
$L, S^v, S^t \leftarrow$ `GenerateCells`(z_0, z_1);
$C^t \leftarrow B^t$.capacity;
$C^v \leftarrow B^v$.capacity;
if $\sum S^v_i > C^v$ **or** $\sum S^t_i > C^t$ **then**
  $a \leftarrow z_0$;
  while $a < z_1$ **do**
    `// Find range` $[a, b)$ `that fits`
    $b \leftarrow a + 1$;
    while $\sum_a^b S^v_i \leq C^v$ **and** $\sum_a^b S^t_i \leq C^t$ **do**
      $b \leftarrow b + 1$;
      if $b = z_1$ **then break**;
    `DoSlices(`D, a, b, B^v, B^t`)`;
    $a \leftarrow b$;
else
  $R^t \leftarrow B^t$.remaining;
  $R^v \leftarrow B^v$.remaining;
  if $\sum S^v_i > R^v$ **or** $\sum S^t_i > R^t$ **then**
    Split bin at z_0;
    `// Also clears the buffers`
  `MakeGeometry(`D, L, B^v, B^t`)`

Algorithm 3: CPU code for processing a range of slices (`DoSlices`). Splitting the bin means completing the rest of the GPU pipeline (particularly welding, Section 3.4) on the currently buffered geometry and emitting a completed mesh, before clearing the buffers and starting a new mesh.

3.4 Welding

Our isosurface extraction algorithm emits only a partially indexed mesh for each bin: Vertices shared between cells are duplicated, and must be unified. When a bin is complete, either by splitting or at the normal completion of a bin, we postprocess the buffers on the GPU to achieve this.

During the isosurface extraction, we emit a 64-bit *vertex key* alongside each vertex. Each vertex is generated by interpolation along an edge, so we use the coordinates of the midpoint of the edge to form the vertex key, encoded in fixed-point and packed into the 64-bit integer. Since the vertex key depends only on the edge and not on the cell that generated the vertex, duplicates can easily be identified as they have the same key.

The welding process is performed on the GPU using several passes. The vertices are first sorted by key, then compacted to preserve only one copy of each vertex. Vertices are stored with their original index in the otherwise unused w component, which we use during compaction to build a table mapping the original to the compacted index. This table is then used to rewrite the triangles in-place to use the compacted IDs.

4 Out-of-Core Processing

To extend the in-core algorithm from the previous section to an out-of-core algorithm, we partition the space into cubic

bins and then process each bin in-core. To process a bin correctly, we must load all samples whose spheres of influence intersect the bin, even if the sample point lies outside the bin. This causes boundary samples to be loaded multiple times. We, thus, wish to minimize the size of the boundaries, which we do by choosing the bin sizes adaptively to maximize use of the available memory.

The in-core algorithm is run independently on each bin to produce a mesh. The per-bin meshes are then stitched together to produce a single output mesh. The in-core algorithm needs to be carefully implemented to avoid introducing cracks at the boundaries between bins, which can easily happen if floating-point computations are not performed identically.

The process runs in three phases, shown in Fig. 2:

- *Phase 1.* The splats are read to compute a bounding box, and to identify runs of splats that lie close to each other. This phase is normally I/O-bound, although we have observed it to be CPU-bound when using a high-performance distributed filesystem.
- *Phase 2.* Most of the work occurs in this phase. The bins are computed and their splats loaded, the in-core algorithm is run and the resulting per-bin meshes are postprocessed and written to disk. This phase is usually GPU-bound.
- *Phase 3.* The per-bin meshes are loaded from disk. Spurious components are removed, and the remaining geometry is written to the output file. This phase is normally I/O-bound.

4.1 Binning

The technique used for binning is described in our previous work [6]; we provide only a brief overview here. The bounding box is divided into a grid, with a sufficiently high density that the majority of grid cells will be small enough, both spatially and in number of splats, for the in-core algorithm. We have found heuristically that making this grid 63 times coarser than the isosurface extraction grid works well. During Phase 1, we determine which grid cells contain each splat, and use run-length encoding to compress runs of splats that intersect the same grid cells. At the start of Phase 2, we build a hierarchical histogram over this grid, and use it to select the bins. Each bin is a cube of grid cells. In some cases, a single grid cell will intersect too many splats, in which case it is recursively reprocessed using a higher resolution grid covering only the original cell.

After bins are chosen, the runs are reprocessed to build a list of splat IDs per bin. To process a bin, we use this list to load the splats from disk and then apply the in-core pipeline described in Section 3.

This approach minimizes I/O bandwidth by avoiding an external sort, but at the cost of more random accesses to the original data. The efficiency of the I/O operations improves with larger bins: in the limit, there would be only one bin, and all the points would be loaded in a single pass. However, the number of splats per bin is constrained by the GPU memory requirements of the in-core algorithm, which needs space for not just the splat data but also the octree. We found that I/O performance was improved by grouping bins together for loading: As bins are generated, they are held in a buffer until a threshold on their combined splat count is reached. The splat ID lists for these bins are then merged to generate a list of ranges to load from disk to CPU memory, after which the splats are distributed to their respective bins.

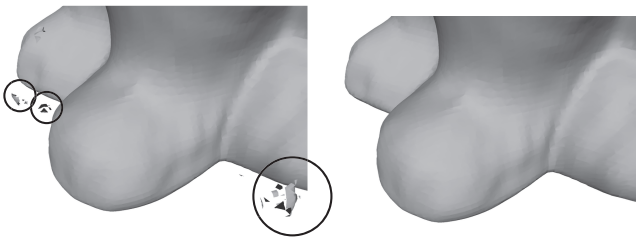


Fig. 8. Removal of isolated components.

This batching process causes incoming splat data to become available in bursts with long gaps in between, which could easily stall the GPU processing. To smooth out the delivery of data to the GPU, we use a large buffer on the CPU, and a separate upload thread which transfers samples from this buffer to GPU memory a bin at a time, as GPU memory becomes available.

4.2 Stitching

After the bins are processed, we have a stream of mesh fragments that need to be stitched back together into a whole mesh. Note that the boundaries are already consistent, as the fragments are pieces of the same global surface[1]; the stitching is needed only to unify the boundary vertices that are duplicated in the respective fragments. At this stage, we also apply another cleaning step: there are often spurious isolated pieces of surface disconnected from the main body, as shown in Fig. 8. We follow Meshlab [28] in removing components whose size (in vertices) falls below a threshold percentage of the total. Since the total is not known until the end of Phase 2, it is not possible to do this pruning until after Phase 2 is complete.

The stitching is distributed across two phases. Most of the work is done in Phase 2, in parallel with GPU processing. Phase 2 is dominated by the GPU work and so this part of the stitching has minimal impact on overall runtime. In Phase 3, we are able to quickly determine which components are to be retained, and the bulk of the work consists of reading this geometry from temporary storage and writing it to PLY files.

The stitching process shares some similarities with the intra-bin welding described in Section 3.4, particularly the idea of a vertex key. We modify the definition of vertex keys used in that step so that the top-most bit indicates whether a vertex is internal or external. External vertices are those on the boundary of the bin, and hence may be duplicated in a neighboring bin. Setting the top bit in the vertex key causes the external vertices to be sorted together, making it easier to extract them. The stitching works on the premise that the bins are large enough that there are relatively few external vertices, and hence information about them can be kept in-core while internal vertices are kept out-of-core.

1. This does require careful implementation to ensure that floating-point computations are performed identically on each side of the boundary.

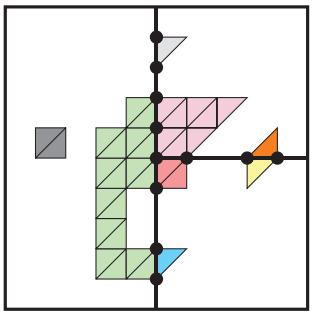


Fig. 9. Identifying components. The thick lines delineate bins, and each clump is given a different color. The external vertices are marked with dots. Note that global information is required to correctly identify small components for deletion: The blue clump (bottom) contains only one triangle, but is a part of the largest global component.

As each per-bin mesh fragment is received during Phase 2, we apply several processing steps. First, we identify components within the fragment; we call these *clumps* (see Fig. 9). Each component of the entire model consists of one or more clumps, joined together by their duplicated external vertices, and we arrange them in a disjoint set data structure [29]. To update this data structure, we use a hash table that maps each external vertex key to a clump that contains it. We update the data structures by iterating over the external vertices: If a vertex key already appears in the table then we can merge components; otherwise, we update the table.

The final step in Phase 2 is to move each clump out-of-core by writing it to temporary files. In this step, we also eliminate external vertices that have already been written.

In Phase 3, we use the disjoint set data structure to identify the components. The data structure is augmented to track the number of vertices in each component, so we can quickly determine which components should be eliminated. Then, we write the clumps that are in retained components to the output file. This involves yet another reindexing operation, as the final position of vertices in the output is not known until this point.

5 Scalability

We can further improve performance by utilizing multiple GPUs. Note that this will only accelerate Phase 2, as Phases 1 and 3 do not use GPUs. The degree of acceleration will depend on the extent to which Phase 2 is GPU-bound (rather than I/O-bound), but we have obtained good results with up to eight GPUs.

5.1 Single Machine, Multiple GPUs

Our binning scheme makes multi-GPU support almost trivial. In Fig. 2, we replace the single thread for GPU processing with a separate thread per GPU. No inter-GPU communication is required, and in fact we use a separate OpenCL context for each GPU. It is even possible to use a heterogeneous configuration combining GPUs and CPUs, although we found that, in practice, this starves the other CPU threads and leads to load balancing issues due to the large mismatch in performance. Heterogeneous environments also risk introducing cracks at the boundaries between bins, due to variations in floating-point computation.

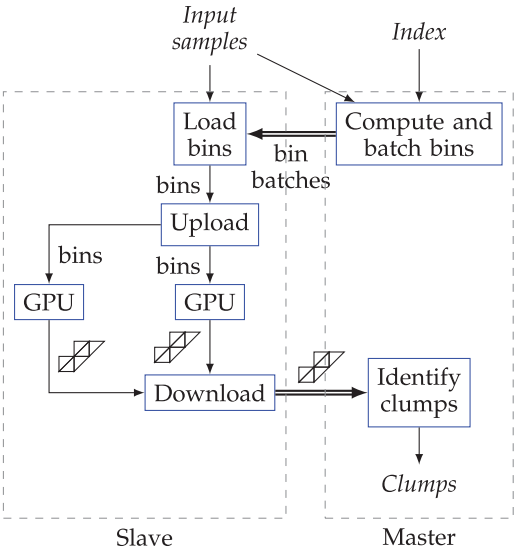


Fig. 10. Phase 2 data flow in a cluster. Each dotted box represents one node in the cluster and the double arrows show data passed by MPI messages. Only one of the slaves is shown. The master may also act as a slave if it has GPUs. Color-coding is as for Fig. 2.

Because memory is not shared, each GPU has a separate queue for incoming work. A dedicated upload thread blocks until any GPU queue has sufficient space for the next bin, and then transfers it. If there are multiple options, the emptiest queue is selected. This tie-breaker is mainly important when starting up, to ensure that the GPUs are all given some work as soon as possible.

5.2 Distributed Memory Clusters

We further extended our implementation to run on a cluster using MPI [30]. All three phases are distributed across nodes. Because the system is designed for out-of-core operation, most large-scale data movement occurs implicitly through the shared filesystem, with MPI messages used for passing control messages.

Phase 1 is almost trivially parallelizable: The splats are partitioned, and each node loads and processes its assigned portion of the splats. This produces a separate index file per node. Rather than stitching these files back into a single index, we keep them separate and iterate over them during Phase 2.

In Phase 2, we compute the bins on only the master node, as this is reasonably fast and would be complex to distribute. As before, the master node batches bins into groups that are more efficient to load, but it does not actually perform the loading. Instead, it sends the metadata for a batch (bounding boxes and splat ID ranges) to the next available slave node. The slave loads the splats and processes them on its GPU(s). At the end of the pipeline, the mesh is sent back to the master for component detection and to be written to file. This is shown in Fig. 10.

In Phase 3, we use one of two parallelization strategies, depending on whether the results are being split into multiple output files (see Section 5.3). If they are, then each node takes responsibility for a subset of the output files. If not, then each node takes responsibility for a subset of the clumps, and the MPI-IO routines are used to ensure safe parallel access to the single output file.

TABLE 1
Data Sets Used, Tuning Parameters, and Statistics for a Single GPU

Name	Samples ($\times 10^6$)	Size (GiB)	Bounding box (m)	Grid (mm)	h	Output Verts ($\times 10^6$)	Output (GiB)	Time (s)	Peak mem (GiB)
Amman Tower	13	0.4	15 × 15 × 7	10	4	11	0.4	19	4.40
Pisa Cathedral	157	4.1	119 × 89 × 55	20	5	192	6.7	251	7.42
Siq	5,654	163.2	580 × 1,195 × 165	20	4	3,243	114.0	8,628	8.61
Songo Mnara	6,248	180.4	332 × 291 × 24	10	4	3,107	108.3	20,791	10.09

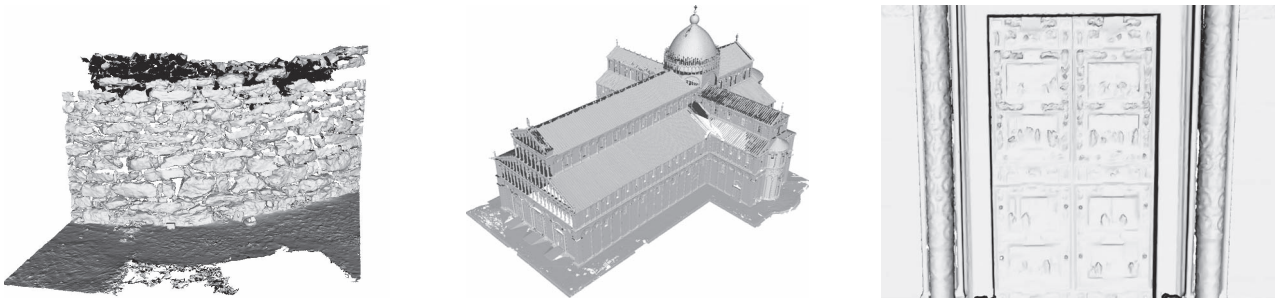
The peak memory measures only host memory allocated by our code: Additional memory is used by the OpenCL driver, libraries, OS and so on.

These design choices are sufficient to achieve good scaling on the small number of GPU-equipped nodes available in our cluster, but we expect that for larger scale use (dozens of GPUs) it may be necessary to move more of the mesh processing onto the slave nodes.

5.3 Chunked Output

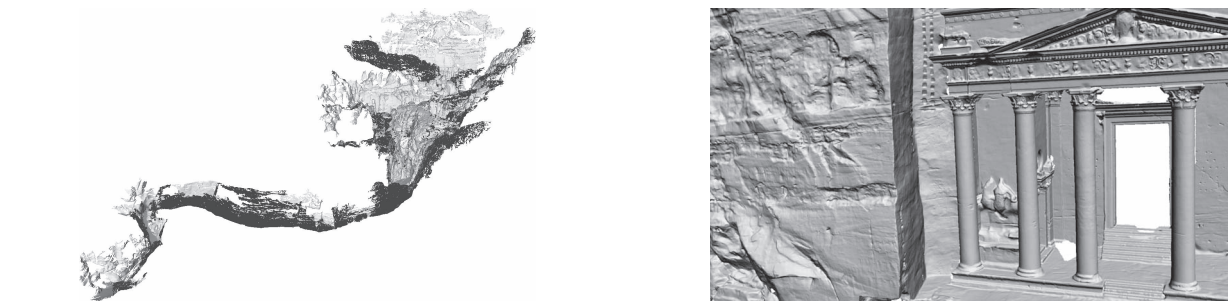
Many software tools do not operate out-of-core and so are unable to handle very large output files. To allow the output of our system to be used with such tools, we have implemented the ability to split the output into chunks using a regular grid, with each chunk written to a separate file. Boundary vertices are duplicated so that every triangle appears in exactly one output file.

When one GPU is used, this requires only a few minor changes to the pipeline. We choose bins independently for each chunk in the grid, and the chunk ID is passed down the pipeline. We write each output file in turn before starting on the next one (although there is some overlap because we use asynchronous I/O).

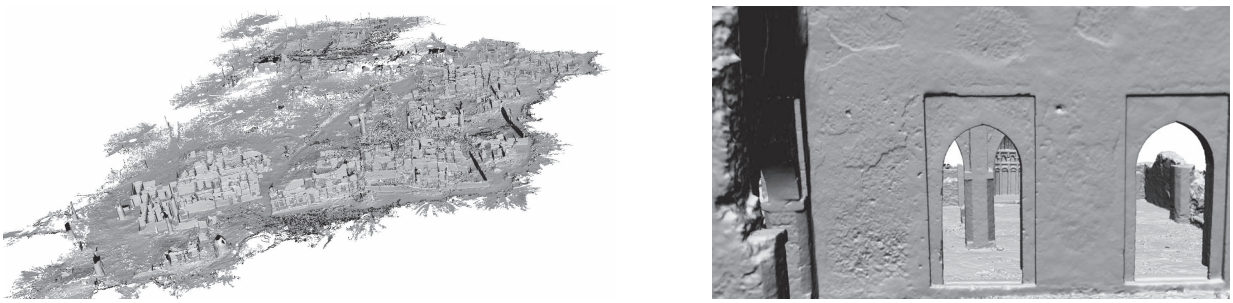


(a) Part of Rujm Al-Malfoouf watch-tower in Amman, Jordan

(b) Pisa Cathedral



(c) The Siq, entrance canyon to the Nabataen city of Petra in Jordan



(d) Ruins of Songo Mnara in Tanzania

Fig. 11. Reconstructions produced by our system. Left column: overview shots of the whole model. Right: fine detail is captured.

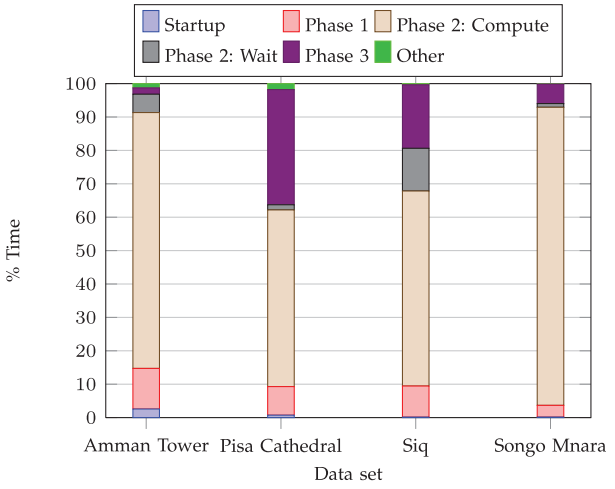


Fig. 12. Time breakdown. Phase 2 is subdivided into time spent in the GPU thread ("Compute") and time spent waiting either for more input to arrive or for more space in the output buffer ("Wait"). "Other" is time not accounted for in any of the phases, particularly freeing of resources.

When using multiple GPUs, bins do not necessarily complete in the same order as they are dispatched, which causes the intermediate storage of the clumps for a single output file to be noncontiguous. We found that, particularly on a cluster, this severely affects I/O performance. We experimented with placing barriers in the queues to constrain ordering, but these barriers became a bottleneck that prevented full utilization of the GPUs. Instead, we use a *reordering buffer*: before the intermediate results are written to disk, they are stored in an in-memory buffer. Once the buffer size exceeds a threshold it is flushed to disk, with the clumps written in order of increasing output chunk ID. This does not completely order the intermediate data, but it improves it to the point where OS-level caching is effective.

6 Results

For testing we used two systems. The first is a desktop machine with a Core i7-2600 (four cores, eight threads, 3.4 GHz), 16 GiB RAM, two 3 TB hard drives with software RAID-0 giving read speeds of 250 MiB/s, and a single NVIDIA GeForce GTX 480. The second system is a GPU cluster, where each node has two Xeon E5530 CPUs (four cores, eight threads, 2.4 GHz), 12 GiB of usable RAM, and up to three Tesla C2070 GPUs; the nodes are connected to storage using Infiniband and the filesystem can deliver up to 3 GiB/s of read bandwidth.

Table 1 lists the data sets we used in our experiments. The Pisa Cathedral data set came without density estimates, so we used Meshlab [28] to estimate them on a per-scan basis and then clamped them to 100 mm to prevent outliers from causing artifacts. We also used a larger smoothing factor for Pisa to handle registration errors. Fig. 11 shows our reconstructions.

The times in Table 1 are total running time for the desktop system. Fig. 12 shows how this time is split across the phases. It is clear that Phase 2 dominates the running time. In most cases, Phase 2 is GPU-bound, but for the Siq, Phase 2 is at times I/O-bound on splat loading. This suggests that the input files have less spatial coherence than in the other data sets.

TABLE 2
Memory Usage for Songo Mnara on the Desktop System

Usage	CPU RAM (MiB)	GPU RAM (MiB)
Input file buffer	**32.0**	
Splat loading buffer	**1,024.0**	
Binning	*808.4*	
Binned splats	**2,048.0**	**256.0**
Pinned memory	128.0	
Octree construction		384.1
Octree		128.4
Distance field		6.2
Isosurface extraction		87.4
Mesh data	**512.0**	56.6
External vertices	*2,847.1*	
Reorder buffer	**4,032.0**	
Output file buffer	34.6	
Other	*449.5*	
Peak total usage	10,332.0	918.7

Bold indicates a tuning parameter while italics indicate an allocation whose size is unbounded. The CPU memory is not all allocated simultaneously and so the peak usage is less than the sum of the individual allocations.

Unfortunately, we found the driver support for OpenCL event profiling to be unreliable, so we cannot provide an accurate profile of GPU activity. Discarding obviously invalid profiling results suggests that distance field computations dominate GPU execution time (75-95 percent), with the remaining time more-or-less evenly split between octree construction and isosurface extraction.

Table 2 shows how memory is allocated for the largest data set. We have allocated a large buffer to hold splats, but this could be reduced if necessary at the expense of more scattered I/O. The unbounded allocations are dominated by the data held for each external vertex. Most of this is consumed by hash table overheads: Each external vertex requires 24 bytes of storage, but the hash tables consume 130 bytes per external vertex. It is, thus, possible to reduce memory requirements by using a slower but more memory-efficient data structure.

Table 3 compares performance against that of previous work. Since results are sensitive to hardware, data sets and tuning parameters, this is only useful for order-of-magnitude comparisons. It is clear that GPU acceleration gives an order-of-magnitude speedup. Our implementation is significantly faster than previous out-of-core systems, and has comparable performance with in-core-only GPU-accelerated Poisson reconstruction.

We can more directly measure the speedup due to the GPU by running our code using a CPU-based OpenCL implementation. This took 5,399 s, indicating a 21.5× speedup on the GPU. It should be noted that while the kernels execute across all CPU cores, they have been tuned for a Fermi GPU and are likely to be suboptimal for a CPU.

Fig. 13 shows the effect of using multiple GPUs and nodes, on Phase 2 and on the total runtime. It is clear that scalability is best for the larger and slower data sets—which are the ones where acceleration is the most valuable. We believe this variation is due to the coarse-grained distribution of work and the deep queues: Once there is no further

TABLE 3
Comparison of Output Rates against Previous Work

Method	Algorithm	Out-of-core	GPU accelerated	Hardware	Rate (KVert/s)
Bolitho et al. [2]	Poisson	Yes	No	Not specified	0.7 – 1.2
Kazhdan and Hoppe [31] ($\alpha = 4$)	Screened Poisson	No	No	Quad-core Core i7	21.8 – 25.4
Kazhdan and Hoppe [31] ($\alpha = 0$)	Poisson	No	No	Quad-core Core i7	24.5 – 29.5
Cuccuru et al. [15]	MLS	Yes	No	Core 2 Quad 2.4GHz	31.0 – 40.6
Zhou et al. [3]	Poisson	No	Yes	GeForce 8800 Ultra	240.2 – 601.7
Ours	**MLS**	**Yes**	**Yes**	**GeForce GTX 480**	**149.4 – 763.4**
Ours (Phase 2 only)	**MLS**	**No**	**Yes**	**GeForce GTX 480**	**165.4 – 1402.3**

The rates are based on reported results, so vary substantially in the hardware and data sets used. We also list the rate for Phase 2 of our method, which is useful for comparison to in-core methods that do not include the I/O time in their results.

work to distribute, some nodes will drain their queues earlier than others and will then sit idle. For larger data sets, this is a smaller proportion of the total time and so has less impact. The smaller data sets do particularly badly on the 7 and 8 GPU configurations because these are unbalanced: They each add a node with only one GPU, so it is much more likely that the two 3-GPU nodes will finish early and sit idle.

7 Conclusions and Future Work

We have presented a complete system for GPU-accelerated reconstruction of extremely large range-scanned surfaces—to our knowledge, the first such system. Although MLS surfaces are not an ideal fit for the data-parallel execution model of modern GPUs, we still achieve an order-of-magnitude speed-up over a multicore CPU implementation. Our approach is also easy to adapt to multiple GPUs, and it scales well, at least for the relatively small number of GPUs in our cluster. Spatial binning provides a hard bound on the GPU memory required, and thus GPU memory does not limit the sizes of models that can be processed.

The main barrier to scalability is currently the memory required for external vertices. With the right data structures, it is likely that external vertices could be kept on disk and loaded only as needed for stitching. When a bin is completed, only its neighbors' external vertices are relevant to stitching, and these could be loaded from disk on-demand. Thus, the relatively large amount of memory for external vertices seen in Table 2 should not be seen as a barrier to processing larger data sets.

The quality of our approach is limited by the single-resolution isosurface extraction. In particular, areas that are sampled at low density can actually slow down extraction, because the total volume of the spheres of influence scales inversely with density. It would, thus, be useful to use variable-resolution isosurface extraction. This will complicate the implementation, as cells on the face of each bin may be affected by the size of their neighbors in adjacent bins. A totally unconstrained octree [32] may also be difficult to handle in a data-parallel way: Performance may be improved by constraining each bin to a single resolution. Because sampling the distance function currently dominates the GPU time, we expect that using variable resolution will improve performance in spite of the overheads.

It would be interesting to apply our bin-based approach to other reconstruction algorithms, particularly Poisson reconstruction. The main challenge is that the Poisson equation is a global linear system, and current out-of-core solutions [2], [9] are tightly coupled with the plane sweep used to stream the data.

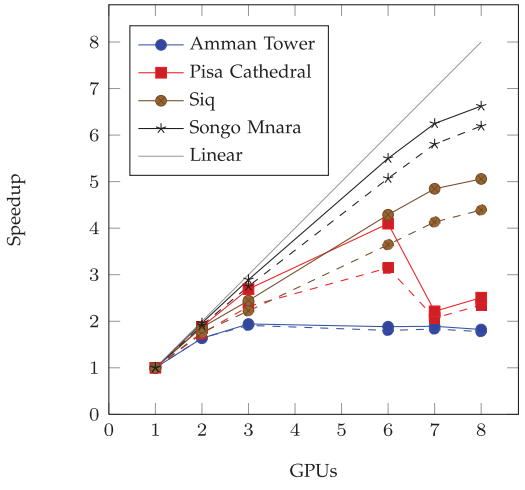


Fig. 13. Speedup of Phase 2 (solid) and overall (dashed) with multiple GPUs. For up to three GPUs, a single node is used. The remaining data points use two, three, and four nodes, respectively. There are two 3-GPU and two 1-GPU nodes.

Acknowledgments

The authors would like to thank the African Cultural Heritage and Landscape Database (particularly Heinz Rüther and Roshan Bhurtha) and the Visual Computing Lab ISTI-CNR Pisa for providing data. Federico Ponchio's Nexus tool was used to produce Fig. 11.

References

[1] M. Kazhdan, M. Bolitho, and H. Hoppe, "Poisson Surface Reconstruction," *Proc. Fourth Eurographics Symp. Geometry Processing (SGP '06)*, pp. 61-70, 2006.
[2] M. Bolitho, M. Kazhdan, R. Burns, and H. Hoppe, "Multilevel Streaming for Out-of-Core Surface Reconstruction," *Proc. Fifth Eurographics Symp. Geometry processing*, pp. 69-78, 2007.
[3] K. Zhou, M. Gong, X. Huang, and B. Guo, "Data-Parallel Octrees for Surface Reconstruction," *IEEE Trans. Visualization and Computer Graphics*, vol. 17, no. 5, pp. 669-681, May 2011.
[4] M. Berger, J.A. Levine, L.G. Nonato, G. Taubin, and C.T. Silva, "A Benchmark for Surface Reconstruction," *ACM Trans. Graphics*, vol. 32, no. 2, pp. 20:1-20:17, Apr. 2013.
[5] H. Rüther, C. Held, R. Bhurtha, R. Schröder, and S. Wessels, "Challenges in Heritage Documentation with Terrestrial Laser Scanning," *Proc. First AfricaGEO Conf.*, 2011.

[6] B. Merry, J. Gain, and P. Marais, "Fast in-Place Binning of Laser Range-Scanned Point Sets," *J. Computing and Cultural Heritage,* vol. 6, pp. 14:1-14:19, 2013.
[7] O. Schall and M. Samozino, "Surface from Scattered Points: A Brief Survey of Recent Developments," *Proc. First Int'l Workshop Semantic Virtual Environments,* pp. 138-147, 2005.
[8] Y.J. Kil and N. Amenta, "GPU-Assisted Surface Reconstruction on Locally-Uniform Samples," *Proc. 17th Int'l Meshing Roundtable,* pp. 369-385, 2008.
[9] M. Bolitho, M. Kazhdan, R. Burns, and H. Hoppe, "Parallel Poisson Surface Reconstruction," *Proc. Int'l Symp. Visual Computing,* 2009.
[10] M. Alexa, J. Behr, D. Cohen-Or, S. Fleishman, D. Levin, and C.T. Silva, "Point Set Surfaces," *Proc. Conf. Visualization,* pp. 21-28, 2001.
[11] A. Adamson and M. Alexa, "Ray Tracing Point Set Surfaces," *Proc. Shape Modeling Int'l,* pp. 272-279, 2003.
[12] V. Fiorin, P. Cignoni, and R. Scopigno, "Practical and Robust MLS-Based Integration of Scanned Data," *Proc. Sixth Eurographics Italian Chapter Conf.,* pp. 57-64, 2008.
[13] G. Guennebaud and M. Gross, "Algebraic Point Set Surfaces," *ACM Trans. Graphics,* vol. 26, no. 3, pp. 23-1-23-9, July 2007.
[14] A. Adamson and M. Alexa, "Approximating Bounded, Non-Orientable Surfaces from Points," *Proc. Shape Modeling Int'l,* pp. 243-252, 2004.
[15] G. Cuccuru, E. Gobbetti, F. Marton, R. Pajarola, and R. Pintus, "Fast Low-Memory Streaming MLS Reconstruction of Point-Sampled Surfaces," *Proc. Graphics Interface,* pp. 15-22, 2009.
[16] M. Alexa, S. Rusinkiewicz, M. Alexa, and A. Adamson, "On Normals and Projection Operators for Surfaces Defined by Point Sets," *Proc. Eurographics Symp. Point-Based Graphics,* pp. 149-155, 2004.
[17] G. Guennebaud, M. Germann, and M. Gross, "Dynamic Sampling and Rendering of Algebraic Point Set Surfaces," *Computer Graphics Forum,* vol. 27, no. 2, pp. 653-662, 2008.
[18] V. Fiorin, P. Cignoni, and R. Scopigno, "Out-of-Core MLS Reconstruction," *Proc. Ninth IASTED Int'l Conf. Computer Graphics and Imaging,* pp. 27-34, 2007.
[19] G.M. Morton, "A Computer Oriented Geodetic Data Base; and a New Technique in File Sequencing," technical report, IBM Ltd., 1966.
[20] J. Boesch and R. Pajarola, "Flexible Configurable Stream Processing of Point Data," *Proc. 17th Int'l Conf. in Central Europe on Computer Graphics, Visualization and Computer Vision (WSCG '09),* pp. 49-56, Feb. 2009.
[21] F. Bernardini, J. Mittleman, H. Rushmeier, C. Silva, and G. Taubin, "The Ball-Pivoting Algorithm for Surface Reconstruction," *IEEE Trans. Visualization and Computer Graphics,* vol. 5, no. 4, pp. 349-359, Oct. 1999.
[22] Khronos OpenCL Working Group, "The OpenCL Specification," http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf, June 2011.
[23] NVIDIA, *NVIDIA CUDA C Programming Guide (version 4.1),* Nov. 2011.
[24] D. Merrill and A. Grimshaw, "Parallel Scan for Stream Architectures," Technical Report CS2009-14, Dept. of Computer Science, Univ. of Virgina, Dec. 2009.
[25] B. Payne and A. Toga, "Surface Mapping Brain Function on 3D Models," *IEEE Computer Graphics and Applications,* vol. 10, no. 5, pp. 33-41, Sept. 1990.
[26] G.M. Nielson and B. Hamann, "The Asymptotic Decider: Resolving the Ambiguity in Marching Cubes," *Proc. Second Conf. Visualization '91,* pp. 83-91, 1991.
[27] S. Dias, K. Bora, and A. Gomes, "CUDA-Based Triangulations of Convolution Molecular Surfaces," *Proc. 19th ACM Int'l Symp. High Performance Distributed Computing,* pp. 531-540, 2010.
[28] P. Cignoni, M. Callieri, M. Corsini, M. Dellepiane, F. Ganovelli, and G. Ranzuglia, "MeshLab: An Open-Source Mesh Processing Tool," *Proc. Sixth Eurographics Italian Chapter Conf.,* pp. 129-136, 2008.
[29] R. Sedgewick, *Algorithms in C.* Addison-Wesley, 1990.
[30] Message Passing Interface Forum, "MPI: A Message-Passing Interface Standard," http://www.mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf, Sept. 2009.
[31] M. Kazhdan and H. Hoppe, "Screened Poisson Surface Reconstruction," *ACM Trans. Graphics,* vol. 32, no. 3, pp. 29:1-29:13, June 2013.
[32] M. Kazhdan, A. Klein, K. Dalal, and H. Hoppe, "Unconstrained Isosurface Extraction on Arbitrary Octrees," *Proc. Fifth Eurographics Symp. Geometry Processing,* pp. 125-133, 2007.

Bruce Merry received the BSc (Hons) and PhD degrees from the University of Cape Town (UCT), in 2003 and 2007, respectively. He is currently a postdoctoral research fellow at the UCT and the Centre for High Performance Computing, investigating the use of GPUs as high-performance processors for heritage and visualization problems.

James Gain received the BSc (Hons) and MSc degrees in computer science from Rhodes University, South Africa, in 1994 and 1996, respectively, and the PhD degree from the University of Cambridge in 2000. He is currently an associate professor in the Computer Science Department at the University of Cape Town and an affiliate of the Centre for High Performance Computing, and his research is focused on computer graphics, visualization and high-performance computing. He is a member of the IEEE.

Patrick Marais received the BSc (Hons) degree in applied mathematics and the MSc degree in computer science both from the University of Cape Town (UCT), in 1991 and 1994, respectively, and the PhD degree in medical imaging at the University of Oxford in 1998. He is currently a senior lecturer in the Department of Computer Science at the UCT and an affiliate of the Centre for High Performance Computing. His research is centered on aspects of computer graphics (specifically procedural content generation), data compression and GPU-based methods for simulation and visualization. He is a member of the IEEE.