ABSTRACT

Extracting isosurfaces represented as high quality meshes from three-dimensional scalar fields is needed for many important applications, particularly visualization and numerical simulations. One recent advance for extracting high quality meshes for isosurface computation is based on a dynamic particle system. Unfortunately, this state-of-the-art particle placement technique requires a significant amount of time to produce a satisfactory mesh. To address this issue, we study the parallelism property of the particle placement and make use of CUDA, a parallel programming technique on the GPU, to significantly improve the performance of particle placement. This paper describes the curvature dependent sampling method used to extract high quality meshes and describes its implementation using CUDA on the GPU.

Categories and Subject Descriptors

Keywords
CUDA, GPGPU, volumetric data mesh extraction, particle systems

1. INTRODUCTION

Extracting isosurfaces is a popular technique to visualize three-dimensional scalar fields. Given a scalar value for the desired isosurface, marching cubes [13], a fast and robust technique, can be used to extract a mesh for the isosurface from a scalar field defined on a 3D regular grid. However marching cubes has a few shortcomings which may result in a mesh that does not satisfy the needs of the specific applications. First, vertices of the extracted mesh are only linearly interpolated along the edges. No vertices can be placed in the interior of the cells. Therefore, detailed information, showing high curvature features smaller than the size of a cell, can be lost. Second, marching cubes does not provide any guarantees on the quality of the triangles that are extracted. Ill-conditioned triangles can be produced and the mesh can be problematic for numerical simulations.

To overcome these shortcomings, Meyer et al. introduced a solution based on particle placement to extract meshes [15]. The idea is to place dense particles on the isosurface that is represented as an implicit surface. A potential energy is computed based on the density of the particles. The particles are moved along the negative gradient of the potential energy to minimize the energy. With this approach, particles can be placed anywhere on the implicit surface and are no longer constrained to the grid edges. Further, a mesh with well-shaped triangles (i.e. equilateral triangles) is generated because minimizing the potential energy of the particle system induces a hexagonal configuration. This well-shaped mesh is suitable for numerical simulations where the volumetric tetrahedral meshes are typically generated from the boundary triangular meshes. In addition, the energy can be adjusted by the curvature of the surface to place more particles in areas of high curvature and fewer particles in flat regions.

While a surface extracted using the curvature dependent particle system has well distributed particles in areas of interest and well-shaped triangles, it comes with significantly increased computational cost over other methods. To speed-up the system, binning was introduced. By partitioning the space, the neighboring search in the energy and velocity computation can be carried out in a smaller region, thus improving performance [8]. However, even with binning, the computational costs are still too high. The excessive computational cost to generate a well-shaped mesh has hindered the use of the particle system by the bioengineering community for numerical simulations [18]. Therefore, improving the performance would increase the use of the particle system for various numerical simulation tasks.

In this paper we devise an efficient implementation of a particle system on the graphics processing unit (GPU) to reduce the runtime of the particle system. Our contributions are as follows:

- We study the potential parallelization of the particle placement and propose a simple strategy to segment the particles into groups that can be processed concurrently.
- We explore the parallel feature provided by the recent advance of CUDA programming on the Graphics Processing Unit (GPU), which allows us to parallelize the computations when processing each particle in a group.
- We have applied our GPU-based particle system to a number of medical data. The obtained meshes have comparable quality to those generated using a CPU-based particle placement, while the computation of our implementation is at least one order of magnitude faster than the CPU version for most cases.
The rest of the paper is organized as follows. Section 2 reviews the most related work. Section 3 reviews the particle system. Section 4 discusses the parallelization of the particle system and 5 provides the details of implementing the system using CUDA. Finally, Section 6 provides the experiment results which show that the GPU implementation is 6 to 44 times faster than a single threaded CPU implementation.

2. RELATED WORKS

Particle systems on the GPU were first introduced by Kolb et al. [11] and Kipfer et al. [9] for real-time animation and rendering of particles in OpenGL. For real-time 3D fluid visualization, Kruger et al. used a particle system on the GPU because the CPU was too slow [12]. Extending the particle system beyond computer graphics, the GPU was subsequently used for simulating fluid motion with smooth particle hydrodynamics (SPH) [10]. A good overview of state-of-the-art in SPH on the GPU can be found in Goswami et al. [5].

Although there are shared characteristics between these particle systems and the system presented in our paper, such as how particles are stored and accessed on the GPU, due to different application purposes each has a different parallelization strategy. The particle system by Kolb et al. [11] and Kruger et al. [12] do not require neighborhood information and are easily parallelizable (i.e., assigning a thread for each particle). On the other hand, Kipfer et al. [9] and the SPH implementations [10, 5] require local neighbors for collision detection and advection of the particles. However, both of these are Forward-Euler solutions which could use a small uniform time step to adjust the particle velocity to allow the systems to converge. In our implementation each particle determines its step size based on its energy and the local curvature and does not have a uniform time step. This allows faster convergence for the purpose of mesh extraction. In what follows, we focus on the most relevant work of particle placement for surface extraction.

Witkin and Heckbert were one of the first to use particles for visualization [20]. They used an energy based particle system to visualize implicit functions. They chose to use a Gaussian energy function based upon the distance from a particle to its neighbors to evenly place particles on the surface. The energy of a particle repelled its neighbors which, after a number of iterations, placed particles evenly on the surface. Following the lead of Witkin and Heckbert in the use of particles for visualization, Crossno et al. used a particle system to extract isosurfaces from scalar fields [3].

More recently Meyer et al. employed an energy based particle system for visualizing implicit surfaces [14] and extracting high quality meshes from scalar fields [15]. Instead of the Gaussian energy function used by Witkin and Heckbert [20], Meyer et al. applied a compact cotangent energy function because it is approximately scale invariant. Additionally Witkin and Heckbert used a gradient descent to minimize the energy which requires a tuning parameter. Meyer et al. replaced it with a Gauss-Seidel update and used an inverse Hessian scheme to automatically tune the energy minimization, removing this tuning parameter. Finally, this method allowed for the placement of more particles near areas of high curvature, while leaving regions of low curvature with fewer particles and fewer tuning parameters. Bronson et al. introduced a particle-based system for generating adaptive triangular surfaces and tetrahedral meshes for CAD models [2]. Instead of pre-computing feature size, their system adapts to curvature and moves the particles in the parameter space.

Our work is based on the particle system by Meyer et al. [15]. While this particle system generates a well-formed triangle mesh, it comes with a significantly increased computational cost. We introduce the parallelization of the particle system to process particles and reduce the run-time. Furthermore, we utilize the parallel computation of the GPU hardware to achieve substantial speed-up, increasing the likelihood of adoption for numerical simulation.

![Figure 1: Overview of the particle system.](image)

3. PARTICLE SYSTEM

The particle system used in this work is based on the dynamic particle system described by Meyer et al [14, 15]. A brief overview of the system is in Figure 1. Initially, a distance field and a sizing field are precomputed to represent the isosurface as an implicit function, \( F \), and to encode the distance between points on \( F \), respectively. Next, particles are seeded on the isosurface based on the results of marching cubes. Then, the particles are processed sequentially: determine neighbors, compute energy and velocity, and update position. A particle only moves if the new position has lower energy than its original position. Once every particle has been processed, the density of the particles are checked to delete or add particles. The above particle process is repeated until the system energy has converged.

### 3.1 Initialization

Before placing the particles, a distance field and a sizing field are precomputed respectively. A distance field of the implicit surface is computed from the scalar data and is used with reconstruction filters to generate the implicit function, \( F \) [15, 16, 19]. The sizing field, \( h \), is based on the local feature size and \textit{curvature} of the implicit surface, and is used by the particle system to meet \( \epsilon \)-sampling distribution requirements [15]. The distance between particles is scaled based on the sizing field in order to control the sampling density which also reflects the local curvature of the implicit surface (Eq. 2). For more information on the construction of the sizing field, see Meyer et al [15]. Once the distance and sizing fields are computed, the system is initialized with a set of particles. The positions of the particles are determined from a marching cubes triangulation. This is done to ensure that the entire isosurface is seeded, even the disconnected regions. The initial seeds are then projected onto \( F \) (Eq. 5).

### 3.2 Per Particle Processing

Processing a particle is a four step process (Figure 2). First, the neighbors of \( p_i \) are determined. Consider all other particles, \( p_j \), in the system where \( i \neq j \), \( p_j \) is a neighbor of \( p_i \) if \( d_{ij} \leq 1.0 \), where \( d_{ij} \) is the scaled distance from \( p_i \) to \( p_j \). Second, the energy, \( E_i \), of \( p_i \), is computed based on its neighbors. Third, the velocity, \( v_i \), at the position of \( p_i \) is computed to give a magnitude and direction for \( p_i \) to move in. Finally, an iterative process (the red blocks in Figure 2) is conducted to update the position of the particle, depending on whether the energy, \( E_{new} \), at the updated particle position \( p'_i = p_i + v_i \) is less than the current energy, \( E_i \). If \( E_{new} \) is less than \( E_i \) then the particle position is updated to \( p'_i \), otherwise we iterate, with a smaller step size, until the new particle position has a lower energy than the previous position.

#### 3.2.1 Energy and Velocity Computation
To compute the energy and the velocity, Meyer et al. proposed the cotangent energy function because of its scale invariance and compactness [14]. The energy, $E_i$, of $p_i$ is the sum of the energies $E_{ij}$ between $p_i$ and $p_j$ such that

$$ E_{ij} = \begin{cases} \cot(|d_{ij}| \frac{\pi}{2}) + |d_{ij}| \frac{\pi}{2} & |d_{ij}| \leq 1.0 \\ 0 & |d_{ij}| > 1.0 \end{cases} $$

(1)

and

$$ d_{ij} = \frac{|p_i - p_j|}{2 \times \cos(\frac{\pi}{6}) \times \min(h_i, h_j)} $$

(2)

where $d_{ij}$ is the scaled distance between $p_i$ and $p_j$ ($i \neq j$) and $|p_i - p_j|$ is the Euclidean distance between particles $p_i$ and $p_j$. Without confusion, we will refer to $d_{ij}$ as the distance between $p_i$ and $p_j$, in the rest of the paper.

To compute distance, $d_{ij}$, between $p_i$ and $p_j$, the sizing values, $h_i$ and $h_j$, at $p_i$ and $p_j$ are used (Eq. 2). The distance between $p_i$ and $p_j$ is scaled by the $\min(h_i, h_j)$. Because the distance and energy are scaled by the surface curvature as in Eq. 2, when the distance is less than 1.0 (i.e. within the neighborhood of the desired radius), the energy $E_{ij}$ is computed between the two particles using Eq. 1. Otherwise, there is no energy between them and $E_{ij} = 0$.

The energy of a particle is used to determine whether a new position, $p_i'$, is at a lower energy state than the original position. However, to move $p_i$, the velocity of $p_i$ is computed. The velocity, $v_i$, is the derivative of the energy function. The velocity for $p_i$ is computed as the sum of all the velocities, $v_{ij}$, between $p_i$ and $p_j$ and ($i \neq j$) where

$$ v_{ij} = -(\tilde{H}_i)^{-1} \left( \frac{\partial E_{ij}}{\partial [d_{ij}]} d_{ij} \right) $$

(3)

and

$$ \frac{\partial E_{ij}}{\partial [d_{ij}]} = \begin{cases} \frac{\pi}{2} & |d_{ij}| \leq 1.0 \\ \sin^{-2}(\frac{\pi}{2}) |d_{ij}| & |d_{ij}| > 1.0 \end{cases} $$

(4)

where $\tilde{H}_i$ is the Hessian of $p_i$’s potential with the diagonal of $\tilde{H}_i$ adjusted by $\lambda$ according to the Levenberg-Marquardt algorithm. The L-M algorithm is discussed further in Section 3.2.2. The velocity is used to move $p_i$ in the tangent plane of the $F$ at $p_i$. Once $p_i$ is moved in the tangent plane, it is projected back onto the surface,

$$ p_i \leftarrow p_i + v_i $$

(5)

where $F_i$ is the implicit function and $\nabla F_i$ is the gradient of the implicit function at $p_i$.

3.2.2 Update Position

Updating the position of the particle is an iterative process to find the appropriate step size for $v_i$. The Levenberg-Marquardt algorithm (L-M) is used because with the current step size of $v_i$, the particle may not be moved to a place with lower energy. Each particle has a $\lambda$ value which it maintains throughout the entire run of the particle system. Increasing $\lambda$ decreases the step size of $v_i$. As $\lambda$ is increased (or decreased), the step size of $v_i$ is converging to a good step size, i.e. the step will produce a proper velocity that leads to a lower energy state. In practice, $\lambda$ is incremented by 10. For more details on the Levenberg-Marquardt algorithm, see Meyer et al [14].

Algorithm 1 is used to update the position of $p_i$. A possible new position, $p_i' = p_i + v_i$, is computed. The energy of $p_i'$, $E_{new}$, is computed using Eq. 1. If $E_{new} < E_i$, then $p_i$ is updated to its new position $p_i'$. Otherwise, the particle system iteratively increases $\lambda$ and computes a new particle position $p_i' = p_i + v_i$ and energy, $E_{new}$, until $E_{new} < E_i$ or $\lambda \geq \lambda_{max}$. If $\lambda \geq \lambda_{max}$, then the particle’s position is not updated and $\lambda$ is reset to its value at the beginning of the iteration process. Otherwise, the position of $p_i$ is updated to $p_i'$.

3.3 Density Control

Controlling the density of the particles is an important aspect in the placement of the particles. Recall that the particle system is initially seeded with particles on the surface from marching cubes. However, the number of particles needed to create the proper density is not known a priori. Therefore, we may seed too many or too few particles. If that is the case, no matter how the particles are moved an optimal configuration may not be achieved.

Therefore, at the end of every iteration, the energy, $E_i$, of every particle $p_i$ is checked against an ideal energy, $E_{ideal}$. Recall that $E_i$ is calculated from the distance, $d_{ij}$, of $p_i$ to its neighbors $p_j$ and $d_{ij}$ is adjusted by the sizing field, $h_i$ (Eq. 2). If the energy is too high, then there are too many particles close to $p_i$. If the energy is too low then there are not enough particles close to $p_i$. The ideal energy of a particle, $E_{ideal} = 3.462$, is based on the energy computed from a natural hexagonal configuration [14]. In other words, the desired configuration is to have six neighboring particles. Achieving $E_{ideal}$ is controlled through the addition and deletion of particles. The addition or deletion of particles is biased with a random value from $[0, 1]$ to prevent mass addition or deletion [20].

3.4 Binning and Neighborhoods

The complexity of the aforementioned particle system as explained is $O(N^2)$. A particle’s energy and force is determined by the distance to every other particle in the system. Heckbert introduced binning as an acceleration structure [8]. Instead of computing energy between a particle and every other particle in the system, he subdivided the space according to a parameter, $\sigma$. Thus, it was only necessary to compare a particle with its immediate neighbors. By setting the bin length to at least $\sigma$ it is guaranteed that all possible neighbors are located within the current bin.
plus all the surrounding bins, i.e. the neighborhood. The neighboring bins must be included since a particle may lie near the edge of the bin and therefore its neighbors would be in the surrounding bins. Because the sizing field contains the distance between particles needed for a quality reconstruction, it is used to determine the bin size as $\sigma = \max(h)$, the global maximum of the sizing field. This acceleration structure is used to speed-up the particle system described by Meyer et al. and is implemented in BioMesh3D.

4. PARALLELIZATION

Although binning reduces the complexity of the particle system, the computation is still slow. Therefore, we need to explore other options to further improve performance. Parallelization is one possible solution to improve performance. The naive approach to parallelize the particle system is to map every particle to a thread and have the thread gather from the neighbors their locations and then calculate energy and force. Unfortunately, this method may fail to converge. Assume a particle, $p_i$, calculates its energy and force while neighboring particles, $p_j$, move. The energy and force calculations of $p_i$ will be incorrect if any $p_j$ move. If all the particles move concurrently, then all the movements could be incorrect and the system may never converge. Although the preceding problem could be mitigated by directly manipulating the time step, it is still problematic. Because the velocity step size is adjusted automatically through the L-M algorithm, any direct manipulation of the step size with a small time step could be compromised by the L-M. Further, because the velocity step size is dependent on the local curvature, manipulating the time step to prevent particles from occupying the same space in areas of high curvature would heavily penalize particles in areas of low curvature. Finally, this requires another tuning parameter, something we wish to avoid.

4.1 Bin Processing

Instead of trying to process all of the particles concurrently, groups of particles can be processed simultaneously if their neighborhoods do not overlap. The binning structure provides the necessary knowledge for such a grouping since every particle contained in a bin is a potential neighbor to every other particle within the same bin. To guarantee a correct energy and velocity computation, the particles in the neighboring bins of the current bin are also considered as neighbors of every particle in the current bin. That said, the particles in the neighboring bins cannot be processed simultaneously while the particles in the central bin are being processed. Therefore, no overlapping neighborhoods are allowed for any groups of particles that are being processed concurrently. Before attempting to run groups of particles concurrently though, how the particles are processed needs to be changed. Previously, all particles in the system are processed serially as described in Figure 1. Instead, since the particles are binned, the particle system can process the groups of particles. Thus, for each bin, $B$, and its neighborhood, $NH$ in the particle system, all the particles $p_i \in B$ are processed serially as shown in Figure 3. Although this change does not effect serial processing of the particles within a bin, it allows particles to be processed concurrently by executing bins with non-overlapping neighborhoods.

If the particles are grouped (and processed) by their bins, then the bins can be processed in parallel but only if the neighborhoods do not overlap. Recall that the bin size is $\max(h)$. The step size is limited to a maximum of the sizing field, $h$, which means the particle can travel into an adjacent bin. Therefore, given a bin $B(a,b)$ and its neighborhood, $NH = \bigcup_{i=a+1,i=b+1} B(i,j)$, if $B(a,b)$ is currently processed, then the other bins that can run concurrently are $B(a+3k,b+3m)$. An example of processing multiple bins concurrently is given in 2D in Figure 4. The bins in Figure 4a that are about to be processed are labeled $W$ through $Z$. Bin $W$ is at position $(0,0)$ therefore the next bins that are processed concurrently are at positions $(3,0)$, $(0,3)$ and $(3,3)$ for $X$, $Y$ and $Z$ respectively. Once all the particles in bins $W$ through $Z$ have been processed, the next bins are processed as in 4c and 4d. This procedure is repeated until all the bins in the 3x3 space, i.e. the compute block, are processed.

5. CUDA IMPLEMENTATION

In the previous sections we described how particles are moved and how bins can be run concurrently. Now, we explain how the particle system is run on the GPU. The motivation for using the GPU is simple. For the past several years, processing power on the GPU has outstripped the CPU [17]. Further, parallel computing architectures like CUDA have made that processing power more accessible than what was previously available with GPU shaders alone. Although the GPU has more processing power than the CPU, it also has limitations. In particular, the GPU is a massively parallel system with many hardware threads. Unfortunately these hardware threads do not handle divergence well, where control statements may cause threads to follow different execution paths.

![Figure 3: Processing particles by their bins.](image)

![Figure 4: Running multiple neighborhoods concurrently in 2D.](image)
which serializes the computations [17]. With the use of the Levenberg-Marquardt algorithm, it is not possible to run a particle per thread because there is no way to know a priori how many iterations the L-M algorithm will take to find an appropriate velocity step size. If every particle requires a different number of iterations to determine the step size, some of the threads would have to be run serially, which hinders performance. Beyond the thread divergence limitation, memory management is important as well. In particular, coalescing memory fetches is very important. This requires memory to be aligned when fetched from global memory.

With divergence and memory management in mind, running the particle system on the GPU is as follows. First, bins are run concurrently (Section 4.1) by processing a bin in a CUDA thread block because processing a bin per thread is not possible due to thread divergence. Second, note that a thread block is composed of tens to hundreds of CUDA threads, so for every particle run in a thread block, multiple threads are available for processing. Thus, the pairwise energy and velocity computations can be processed in parallel. Finally, memory management is discussed. To coalesce memory access, neighborhoods are copied into contiguous memory. Further, preprocessed data, i.e. the sizing and distance fields, use texture memory for automated memory management.

5.1 Bin Processing
Bins are processed concurrently by executing a CUDA block per bin. Assign each bin $B_i$ and its neighborhood (see Fig. 4), to a CUDA block $CB_i$. Processing all the bins in the particle system means iteratively processing bins in a compute block. Thus, once a group of bins is processed, the adjacent bins are processed next. We continue until all the bins have been processed, as illustrated in Figure 4. This is the block level parallelization.

5.2 Energy and Velocity Computation
Since a thread block is run per bin and particles are run serially within a bin, when $p_i \in B$ is processed, multiple CUDA threads are used to calculate the energy and velocity. A CUDA thread, $t_j$, is assigned to do the pair-wise energy computation from $p_i$ to one other $p_j \in NH$. Once the pair-wise energy calculations are finished, a parallel sum reduction is conducted to compute $E_i$ from the array of energy values, $E_{ij}$ [7]. The velocity is computed in a similar manner to the energy computation. By running a CUDA block per bin, the computation is parallelized at both block (bins) and thread (energy and velocity computation) levels.

5.3 Memory Management
The method to build the bins efficiently in CUDA is similar to the one used to build spatial subdivision for uniform grids in Green [6]. To coalesce memory access, at the beginning of every iteration the indexes of the particles are binned in global memory. Additionally, a particle count is generated for every bin, $B_CNT$. Before each neighborhood is processed, the particles are copied into a contiguous span of global memory. As $p_i$ is processed serially in bin $B$, and the energy (or velocity) is computed according to Eq. 1 (or Eq. 3) a thread, $t_j$, is assigned for the pair-wise computation. Copying the particles to coalesce memory access constitutes less than 4% of the total run-time required.

To create multiple neighborhoods, $NH_k$, in global memory, $NH$, compactly and concurrently, a three step approach is used as outlined in Algorithm 2. First, the number of particles in each $NH_k$ are counted (Figure 5a). For each $NH_k$, and for each bin $B_i \in NH_k$, $NH_CNT_k += B_CNT_i$. Second, the particle system computes the memory location, $NH_IDX_k$ of $NH_k$ (Figure 5b). Recursively, it is defined as

$$NH_IDX_k = NH_IDX_{k-1} + NH_CNT_k$$  \hspace{1cm} (6a)

with

$$NH_IDX_0 = 0$$  \hspace{1cm} (6b)

To determine the neighborhoods concurrently in CUDA, Eq. 6 the CUDA atomicInc() function and a global integer, ptr, are used to create the array of indexes. The atomicInc() function takes two values, a memory reference ptr and an integer val, and returns the previous value, prev, at $P$ atomically. Thus, although every neighborhood in the particle system is calling atomicInc(), it is serialized because the ptr can only be incremented by $NH_CNT_k$ atomically. Therefore, $NH_IDX_k = ptr + NH_CNT_k$ where $ptr = NH_IDX_{k-1}$. Third, with an index, $NH_IDX_k$ into the span of global memory reserved for $NH$, it is easy to copy particles into their respective neighborhoods (Figure 5b). This procedure produces a per neighborhood count of particles for each neighborhood, a per neighborhood index into the list of particles and a copy of all the particles binned into their neighborhoods. As mentioned before, this is done to copy a neighborhood into contiguous memory to coalesce memory access.

The sizing field is precomputed in a separate process and therefore the data is read into a 3D texture to take advantage of texture caching. However, the built in interpolation function was not accurate enough. The hardware trilinear interpolation is only a “9-bit fixed point format with 8 bits of fractional value” [17]. Instead a full float type trilinear interpolation function was used. Every thread block has a shared memory variable for the sizing field value at its location for better localized access. Likewise, the distance field is precomputed and read into a texture for the same reasons the sizing field was put into a texture. However instead of linear interpolation, B-Spline kernels were used to reconstruct the surface, its gradient and Hessian.

Finally, because of the addition and deletion of particles, the particles are double buffered between iterations. The addition or deletion of a particle is carried out after all the particles have been processed. If the energy of the particle is not within a certain threshold of $E_{ideal}$, then its either added or deleted. In practice, if $E_i < .75 \times E_{ideal}$ then a particle is added and if $E_i > 1.35 \times E_{ideal}$ then the particle is deleted. The energy calculation for adding or deleting particles is done in the same manner as moving the particles, with the block level and thread level parallelization. Although adding or deleting can be performed without the double buffer, this helps cluster the particles by region and allows for faster binning in the next iteration.

6. RESULTS
A CPU version of the particle system, BioMesh3D [1], is used to generate the CPU mesh. A level set method [19] is used to generate
the distance field and the sizing field $h$ in the pre-computation steps. A B-spline reconstruction kernel is used to interpolate values and compute the gradient and the hessian of $F$. For the sizing field, $h$, linear interpolation is used to look up the values at $p_i$.

Once the particles have been saved from BioMesh3D or the CUDA implementation, TIGHT COCONE [4] is used to create a watertight mesh. The three-dimensional scalar fields are $268x129x177$ volume data of a human ribcage, human heart, human heart and human lungs. The results of the ribcage, heart and lungs (CPU and GPU) are in Figures 9a through 9d. Marching cubes is used to seed the particles and is generated on the CPU. Once the initial particles are seeded and projected onto the surface, they are copied to the GPU and the system processes the particles as described in the previous sections. Once 50 iterations are completed or the energy has stagnated where $E_{\text{prev}} - E < E_{\text{min}}$ the process is terminated. We have found in practice that $E_{\text{min}} = 0.0015$ produces good meshes. All tests were run on an nVidia Tesla c2070 with 6GB of RAM and an Intel Xeon X5650 2.67Ghz with 196GB of RAM.

### 6.1 Quality

To evaluate the quality of the obtained mesh, the ratio of the inscribed and circumscribed radii is computed for every triangle on the mesh and the mean radius ratio of the mesh is calculated. The higher the ratio between inscribed and circumscribed radii, the closer a triangle is to being equilateral. The radius ratio is a common quality metric which allows a direct comparison between two meshes.

<p>| Table 1: Multiple data sets including heart, lungs and ribcage on the CPU and GPU, are compared for quality. Qualitative comparison is done by calculating the mean radius ratio of the resulting meshes. |</p>
<table>
<thead>
<tr>
<th>CPU</th>
<th>GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Heart</td>
<td>0.92114</td>
</tr>
<tr>
<td>Lungs</td>
<td>0.912578</td>
</tr>
<tr>
<td>Ribcage</td>
<td>0.914975</td>
</tr>
</tbody>
</table>

Table 1 has the qualitative results. The mean ratio of a mesh generated through the GPU system is within 1% of the mean radius ratio of the CPU implementation. Thus, the GPU meshes have a very similar quality to the CPU meshes. The histograms in Figures 9d through 9e generated for the heart, lungs and ribcage respectively, shows that the distributions of the ratios are dominated by good triangles and that both the CPU and GPU meshes have similar profiles. The close-up images in Figures 9a through 9f show that the quality of the mesh using our GPU particle system is similar to or comparable to the one using the CPU version.

### 6.2 Speed-up

While the quality of the meshes are nearly the same there is a substantial performance gain with the GPU version (Table 2). The GPU version is 7.8x to 35.2x faster than the single threaded CPU implementation. The reductions in the run-time are from 835.26 to 107.64 seconds for the lungs, 3150.38 to 245.77 seconds for the heart, and 9460.29 to 269.1 seconds for the ribcage (Table 2). Those are 7.8, 12.8, and 35.2 times speed-up of the GPU over the CPU respectively.

### 6.3 Scaling

In the previous section, there is a correlation between the number of particles and the speed-up. As the number of particles increases so does the speed up, but this is across different implicit functions.

**Table 2: The amount of time to place particles on the surface is compared in this table. Multiple data sets including heart, lungs and ribcage on the CPU and GPU, are listed along with the time, in seconds, to place the particles and the final number of particles for the CPU and the GPU respectively.**

<table>
<thead>
<tr>
<th>data set</th>
<th>Time # Particles</th>
<th>Time # Particles</th>
<th>Speed-up</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lungs</td>
<td>835.26 74153</td>
<td>107.64 74129</td>
<td>7.8x</td>
</tr>
<tr>
<td>Heart</td>
<td>3050.38 200125</td>
<td>245.77 180594</td>
<td>12.8x</td>
</tr>
<tr>
<td>Ribcage</td>
<td>9460.29 468877</td>
<td>269.12 166823</td>
<td>35.2x</td>
</tr>
</tbody>
</table>

To measure the speed-up, we conducted a real world test and a synthetic test using the ribcage dataset. The real world test controls the number of particles by varying $\epsilon$ and $\delta$ parameters when generating the sizing field around the isosurface. The $\epsilon$ and $\delta$ parameters control the density of the particles, where the smaller the values of $\epsilon$ and $\delta$, the denser the particles [15].

However, for the ribcage data set, the fewest number of particles generated by manipulating the $\epsilon$ and $\delta$ values in the precomputed phase was 320,000. Generating a sizing field using $\epsilon > 8.0$ and $\delta > 0.2$ resulted in an incomplete mesh. For instance, with $\epsilon = 10.0$ and $\delta = 5.0$, the ribs of the ribcage were removed. Therefore, a synthetic test was created. The synthetic test removes the *add new particles* stage and seeds a user defined number of particles. This creates an upper bound on the number of particles in the mesh. This seeding is done through marching cubes and generates an initial seeding that is closer to the original implicit function than using large $\epsilon$ and $\delta$ values.

**Table 3: Synthetic test data for scaling the ribcage data set without adding any particles to give an upper bound on the number of particles. The details are the initial number of particles (60,000 to 300,000), the time and final number of particles for the CPU system, the time and final number of particles for the GPU system and the speed-up.**

<table>
<thead>
<tr>
<th>Initial Parts</th>
<th>Time # Particles</th>
<th>Time # Particles</th>
<th>Speed-up</th>
</tr>
</thead>
<tbody>
<tr>
<td>60000</td>
<td>213.22 75475</td>
<td>34.75 56844</td>
<td>6.14x</td>
</tr>
<tr>
<td>90000</td>
<td>444.62 81432</td>
<td>48.82 80208</td>
<td>9.1x</td>
</tr>
<tr>
<td>120000</td>
<td>756.8 103913</td>
<td>66.35 103716</td>
<td>11.4x</td>
</tr>
<tr>
<td>150000</td>
<td>1360.87 131145</td>
<td>98.26 133792</td>
<td>13.8x</td>
</tr>
<tr>
<td>180000</td>
<td>1571.96 145958</td>
<td>100.76 146754</td>
<td>15.6x</td>
</tr>
<tr>
<td>210000</td>
<td>2354.04 170805</td>
<td>141.4 175775</td>
<td>16.6x</td>
</tr>
<tr>
<td>240000</td>
<td>2860.53 185035</td>
<td>160.28 194354</td>
<td>17.8x</td>
</tr>
<tr>
<td>270000</td>
<td>3545.14 200925</td>
<td>172.31 208866</td>
<td>20.1x</td>
</tr>
<tr>
<td>300000</td>
<td>4042.60 225054</td>
<td>183.98 237921</td>
<td>22.0x</td>
</tr>
</tbody>
</table>

For the synthetic test, the seed numbers were 60,000 to 300,000 increasing by 30,000. Note in Table 3 that although adding particles is disallowed, removing particles is still active. Therefore, the final particle count is less than the initial number seeded. Figure 7 shows a plot of the amount of time to generate a mesh vs the number of particles. As the number of particles increase, the speed-up increases as well, from 6.14 times speed-up of the GPU over the CPU with 57,000 particles to 22.0 times speed-up with 230,000 particles. Therefore, for the synthetic test, as the number of particles increase, the speed-up increases in a linear manner.

While the synthetic test is useful to verify linear speed-up when the number of desired particles is not achievable by changing the sizing field, the real world test is a more accurate reflection of attainable speed-ups. Table 3 contains the data from generating different sizing fields dependent on the $\epsilon$ and $\delta$ values. Further, the
We have presented a particle system for surface extraction on the GPU. The method is parallelized by processing bins concurrently. Further, on the GPU, by mapping bins to thread blocks, the energy and velocity computations are parallelized as well. We have presented a variety of data sets that show a reliable speed-up can be achieved regardless of the number of particles. We compared the accuracy of the GPU particle system against a CPU particle system and demonstrated that the resulting meshes are similar as measured by the mean ratio of the triangles. Finally, we have shown that as the number of particles increase so does the speed-up of the GPU system over the CPU system.

A current constraint of the system is the use of a global parameter, the maximum of the sizing field, to bin the space. Instead, an adaptive binning strategy could be used to decrease the size of the bin in areas of high curvature. This could lead to further decrease computational time because the number of neighbors are restricted.

Looking forward, the techniques presented could be applied to different applications as well. For instance, the binning technique could be applied to PIC (particle in cell) or MPM (material point method) for the GPU. Further, the successful speed-up of the particle system could change how the system is used. Currently for BioMesh3D, a command line interface with Python scripts is used with no user interaction or feedback because of the required amount of time to generate the mesh. Instead, it would be interesting to allow user interactions such as adding more particles to areas or features that are of particular interest to the user. Further, extending the current particle systems for extracting the conformal meshes is a useful addition to the present work.

8. ACKNOWLEDGMENTS
The authors would like to thank Zhisong Fu for the ribcage, heart and lung data sets.

9. REFERENCES
Table 4: Real world test data for scaling the ribcage data set by varying the $\epsilon$ and $\delta$ when generating the distance field. The fields are the $\epsilon$, $\delta$ and iteration count used to generate the sizing field, the time and final number of particles for the CPU implementation, the time and number of particles for the GPU implementation and the speed-up of the GPU system over the CPU system.

<table>
<thead>
<tr>
<th>$\epsilon$</th>
<th>$\delta$</th>
<th>Iterations</th>
<th>Time (s)</th>
<th># Particles</th>
<th>Speed-up</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.0</td>
<td>2.0</td>
<td>4</td>
<td>3798.74</td>
<td>317809</td>
<td>1.01</td>
</tr>
<tr>
<td>2.0</td>
<td>1.0</td>
<td>4</td>
<td>5526.21</td>
<td>377681</td>
<td>200.6</td>
</tr>
<tr>
<td>0.5</td>
<td>0.5</td>
<td>4</td>
<td>5952.6</td>
<td>398838</td>
<td>212.19</td>
</tr>
<tr>
<td>0.25</td>
<td>0.25</td>
<td>4</td>
<td>8805.9</td>
<td>428885</td>
<td>265.8</td>
</tr>
<tr>
<td>0.125</td>
<td>0.125</td>
<td>4</td>
<td>9460.29</td>
<td>464265</td>
<td>269.12</td>
</tr>
<tr>
<td>0.01</td>
<td>0.01</td>
<td>4</td>
<td>11356.3</td>
<td>493697</td>
<td>285.51</td>
</tr>
<tr>
<td>0.01</td>
<td>0.01</td>
<td>7</td>
<td>19750.2</td>
<td>530565</td>
<td>444.59</td>
</tr>
</tbody>
</table>

10. APPENDIX

Algorithm 1 Update Particle Position

iterate ← true
while iterate do
  increase $\lambda$ by 10
  $p_i' \leftarrow p_i + v_i$
  Project $p_i'$ onto surface as in Eq. 5.
  for all particles $p_j$ in neighborhood NH do
    if $p_i' \neq p_j$ AND $distance(p_i, p_j) \leq 1.0$ then
      $E_{ij} \leftarrow \text{calcEnergy()}$ as in Eq. 1
    end if
  end for
  $E_{\text{new}} = \text{sum } E_{ij} \text{ over } NH$
  if $E_{\text{new}} < E_i$ then
    Save $\lambda$
    $p_i \leftarrow p_i'$
    iterate ← false
  else if $\lambda \geq \lambda_{\text{max}}$ then
    iterate ← false
    reset $\lambda$ to its original value.
  end if
end while

Algorithm 2 buildNeighborhoods()

for all neighborhoods NH do
  for all bins $B \in NH$ do
    $NH_{CNT} \leftarrow \text{sum of particles } p \in B$
  end for
  $NH_{IDX} = \text{atomicInc}(ptr, NH_{CNT})$
  Copy particles in NH to NH IDX
end for
Figure 9: Images of the heart, lungs and ribcage data sets, CPU and GPU, respectively. Further, embedded is a zoomed in area for each image and the histogram for the data sets. The visual quality of the CPU implementation compared to the GPU implementation is very similar across the data sets. The histograms show that both the CPU and GPU systems are dominated by well-shaped triangles.