Finely Threaded Topological Connection Reconstruction

Category: Research

Abstract— Many algorithms applied to mesh-based data structures are easily parallelizable because operations can often be divided by the constituent elements of the mesh and parallelized with a large number of threads. The problem with this technique is that when these threads generate new topological features, they tend to generate redundant information such as coincident vertices. This "soup" of elements requires significant additional memory to store and fails to capture the adjacency of neighboring elements. We present a novel approach for the efficient construction of connected topologies on multi- and many-core systems, which we demonstrate with GPU and OpenMP implementations. In this approach, we use input topological features as a basis for efficient location of coincident features in the output. We provide examples and performance numbers of the technique for isosurface generation and multiple forms of tetrahedralization.

Index Terms—Parallel Systems, GPUs and Multi-core Architectures, CPU and GPU clusters, Petascale Techniques, Scalability Issues

1 Introduction

Many existing parallel graphics techniques are designed to generate geometry from some form of input mesh. Such algorithms tend to have the attribute that computation of any given output feature depends solely on a small set of input features. This provides a natural partitioning of the input which can then be used to develop parallel algorithms. As an example, the Marching Cubes [18] algorithm can be parallelized per voxel, so that only the data values at the voxel corners are necessary to generate the geometry within the voxel (if any).

The problem with this naive parallelization is that it generates a 'soup' of output primitives, with no information about connectivity. The aforementioned Marching Cubes algorithm simply produces a set of disconnected triangles. This is not a concern so long as all further computation on the output depends exclusively on the local attributes of individual primitives, such as is the case with rendering. Rendering has often been the ultimate goal for the generated output of parallel graphics algorithms and the size of the output is not a concern, so these techniques have been sufficient. When additional processing of the output is desired that requires information about the connectivity between multiple primitives, the situation becomes more difficult. Consider the calculation of the curvature of the isosurface generated by Marching Cubes. Given only a triangle soup, there is no simple method to determine the neighbors of any given triangle, which is a required step to compute the curvature. A more complex example would be the case of a visualization pipeline, where different types of topological connectivity information may be necessary depending on the structure of the pipeline.

This is not a new problem, and techniques exist to determine topological information about a primitive soup [19]. Generally this approach starts by finding and coalescing duplicate vertices, which may require a bounded radius nearest neighbor search to resolve vertices that are identical save for floating point error. Next, the algorithm finds all primitives that share two or more vertices, and links them together as neighbors. Finally, some 'duplicate' vertices may not actually represent desirable connections, such as is the case where two cones meet at their apices, so these vertices may need to be split in a final pass. This process is computationally complex, and is specific to the topological connections between triangles.

We present a technique which applies equally well to other types of topological connections, such as determining the neighboring facets of tetrahedra. To this end, we have explored the advantages of making a small modification to the algorithms that generate the geometry, where information is stored about what feature of the input topology was used to generate each output feature. For instance, Marching Cubes generates all output vertices on edges that exist between the voxel coordinates of the input mesh.

To summarize, this paper makes the following contributions:

- Provide a generalized technique for generation of topological connectivity information on highly parallel systems.
- Require no modifications to the algorithms generating geometry save for the storage of information about the input topological features used to generate each output feature.
- Make use of features of input topology to enhance the performance of the topology resolution technique.
- Demonstrate the effectiveness of our technique on the output of the Marching Cubes algorithm and a tetrahedralization algorithm.
- Show how knowledge of the output topology of an algorithm can be used to perform simple mesh coarsening in the same pass as the topology resolution.

We developed our technique for topology resolution so that it could be implemented into the upcoming DAX framework, which will provide data analysis capabilities similar to those in VTK, but targeted toward massively parallel architectures, beginning with the GPU.

2 Previous Work

We use Marching Cubes as an example of parallelized geometry generation [18]. There are myriad existing isosurfacing implementations on the GPU, dating from Rottger's implementation [21]. Dyken provides a fairly detailed overview of the advancements in GPU implementations of Marching Cubes and Marching Tetrahedra [11] .

The basic topology resolution technique described in the introduction section is introduced by Park et al. [19]. Additionally, Kipfer mentions implementing a topology construction technique along with their GPU Marching Cubes algorithm [16]. Kipfer's technique involves the use of the edges that vertices will be placed on as a key for determining which vertices would be duplicated. Kipfer thus avoids redundant computation of such vertices and avoids the need for coincident point resolution. To have accurate normals at the generated vertices, Kipfer's technique requires precomputation of the gradient of the volume at the voxel vertices. Any other interpolable attributes to be derived from the volume must also be precomputed at the voxel vertices. Our method for determining duplicate vertices extends Kipfer's method in that we may key off of any input feature.

Stream compaction is an important element for efficient geometry generation on the GPU. Horn provides an early method for stream compaction on the GPU [14], which was improved upon by Sengupta [22], who also provides a CUDA implementation. These approaches rely on the data parallelism technique and the application of prefix sums as described by Blelloch [4]. Dyken further optimizes GPU

compaction techniques by making use of a data structure called a Histogram Pyramid [11]. For our examples, we use geometry generation methods that make use of the prefix-sum method of compaction, but other compaction techniques such as histogram pyramids could be substituted for the prefix sum technique for increased performance in any case where our method is applied.

We use coincident point resolution (vertex welding) in Marching Cubes with the Thrust C++ template library [3]. Bell, a primary developer of the Thrust library, also presents a vertex welding technique as an example application of Thrust [2]. Bell also notes that GPU sorting techniques can be instrumental in the development of high-performance GPU algorithms. Hoberock uses a lexicographic sort directly on the vertices generated by Marching Cubes, then collapses duplicates to get the final welded surface. Our approach has many similarities, but we differ in the basis for the sort and in the method of collapsing duplicates. We can apply the technique to arbitrary input topologies and also avoid invalid results when floating point errors exist in the vertex coordinates.

Carr et al. [7] describe several ways to subdivide voxel grids into tetrahedral grids. To fairly test performance of our topology generation on unstructured grids, we generate each of these subdivisions and demonstrate that the performance of our approach is similar in all cases.

Observant readers may notice a similarity between our proposed algorithms and those supported by the MapReduce framework [9]. Like MapReduce, our algorithms first map keys to values then collect keys and reduce the values. Although we believe our algorithms could be implemented in a MapReduce framework (an exercise we leave to the reader), it would require additional collection operations to resolve topological connections. Other researchers [24][25] have implemented visualization algorithms directly in MapReduce, but they serve purposes other than those we address.

3 Merging Algorithm

Without significant modification of the algorithms generating topological features, we present an approach that reconstructs the topological connections between these features. By topological connections, we are referring to the information necessary to determine all topological features that are adjacent to any individual feature of interest. We assume and provide a representation based on point arrays and either implicit or explicit cell connection lists. Generating incidence lists, such as the list of all cells incident to a point, is a trivial extension to our algorithm. Given a point P generated by the Marching Cubes algorithm, the topological connections could be a list of all faces that contain point P as a vertex. Depending on the intended use of the topology, it may be preferable to store the list of points Q where there exists an edge (P,Q). Our approach can be applied to generate any of these forms of connectivity information.

3.1 Goals

In developing our technique for reconstruction of topological connections, we have the following requirements.

- Support at least the following feature types: vertices, edges, faces and facets.
- Require minimal modification to the algorithms used to generate topological features.
- Work as an efficient data-parallel method

3.2 Algorithm

Our technique was developed by generalizing and extending the vertex welding example from the Thrust library [3]. Since many vertices are redundant when all triangles in a mesh are stored discretely, it is often desirable to 'weld' the duplicate vertices to single instances. We base our technique on the Thrust method because it has a similar structure to the approach we take for topology generation, and because it is designed to work with nVidia CUDA platforms as well as platforms supported by OpenMP. The algorithm, hereafter referred to as VERTEX-WELD, works as follows:

VERTEX-WELD(*vertices*)
vertices: Array of vertex data (i.e. coordinates).

▷ Sort *vertices* to make identical vertices adjacent.
1 *sorted-vertices* ← LEXICOGRAPHIC-SORT(*vertices*)
▷ Copy the first element from each group of duplicates.
2 *welded-vertices* ← UNIQUE(*sorted-vertices*)
▷ For each item in *vertices* find the corresponding index
▷ in *welded-vertices*.
3 *cell-connections* ← VECTORIZED-FIND(*welded-vertices*, *vertices*)
4 **return** (*welded-vertices*, *cell-connections*)

Alg. 1: VERTEX-WELD, as demonstrated in the Thrust library examples, takes geometric soup as input and produces a welded topology.

VERTEX-WELD can be implemented using well-studied parallel algorithms. For example, the functions LEXICOGRAPHIC-SORT, UNIQUE, and VECTORIZED-FIND are performed using the parallel Thrust functions `sort`, `unique`, and `lower_bound`, respectively.

VERTEX-WELD is a good basis for an algorithm to generate topological connectivity because it allows for grouping and processing of duplicate vertices. Since we can store which triangle generated each copy of a duplicate vertex, the above algorithm can be adapted to calculate a neighboring face list for each point. Therefore, using VERTEX-WELD as a basis, we set out to create a technique that could be used for generating topological connectivity information from multiple kinds of disconnected input geometric elements. There are several requirements for this which the VERTEX-WELD algorithm does not immediately meet.

Unlike the creators of the VERTEX-WELD algorithm, we are not interested specifically in the elimination of duplicate elements. Instead, we want to form groups of geometric elements and then merge these into an output. These elements may not be simple vertices as in the VERTEX-WELD example. To use this for the generation of topological connections, we will need to associate attributes with each vertex. To this end, rather than taking as input an array of vertices, we take in an array of geometric elements of some unspecified type. Often, the input will be an array of 'tuples' which may consist of vertices and their associated attributes such as normals or polygon IDs. Hereafter, we will refer to this input array as the 'mergeable' array.

The VERTEX-WELD algorithm groups bitwise-equal elements together. This would be inappropriate if additional attributes were associated with each sample of a given vertex. For instance, three vertices at the same location that possess different normal vectors would not compare bitwise equal, but we may wish to process them as a group. To resolve this, we take as input an array of *keys* where each key corresponds to an element of the mergeable array. The key for each mergeable element will be compared against the keys for other mergeable elements using a user defined comparison operator *keycomp*. *keycomp* is required to be analagous to the less-than operator, so it takes two keys as input and must return *true* if the first element should be placed earlier in the array than the second element, and $false$ otherwise. If $keycomp(k_1,k_2) = false$ and $keycomp(k_2,k_1) = false$, then the elements are assumed to be part of the same group and will be merged.

The keys can be based off of point coordinates as is done in the VERTEX-WELD example. However, keys based on floating point numbers such as coordinates are inferior because of the large number of bits required to represent them and the high probability of small numerical errors leading to lexicographic differences. Later in this paper we present algorithms with more efficient keys.

Since we no longer require elements of each group to be bitwise equal, we cannot simply use the first element of the group for our output vector. Instead, we will rely on two user-defined operators which together will merge the grouped elements to a single output. The reason for having two operators is that some operations, such as averaging, require two steps; in this case, the first operator would be a sum,

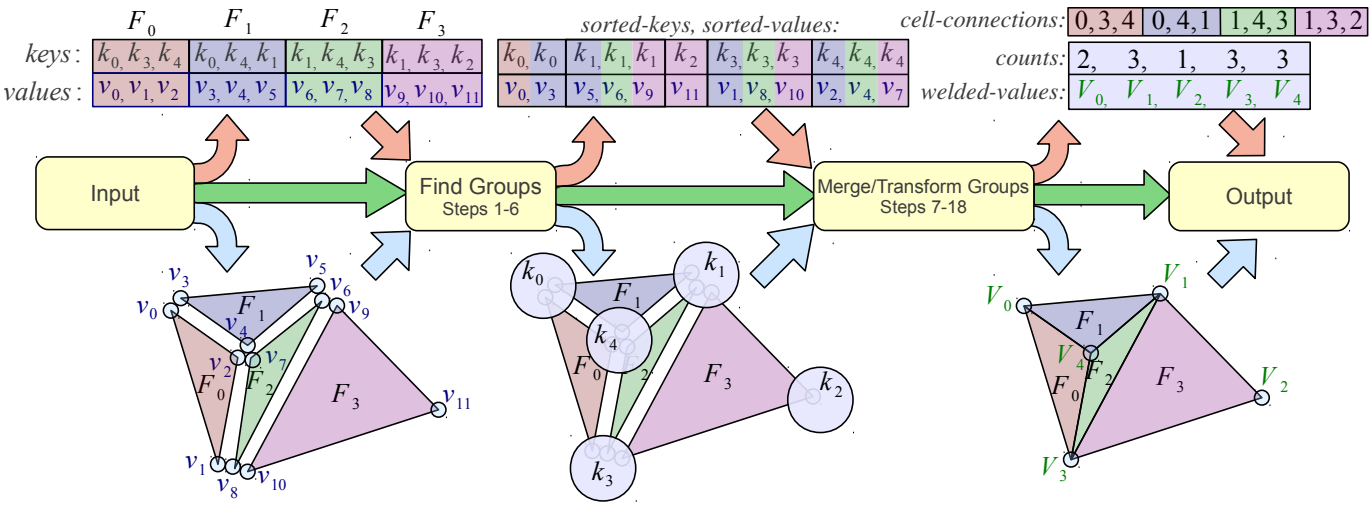

Fig. 1: An example of the KEY-WELD procedure on a very small set of triangles. Data values within the algorithm at several stages are shown above the flow diagram. The represented geometry is shown below the flow diagram. Background color in the upper tables reflect the color of the associated triangle in the lower diagrams. After processing, we have a map *cell-connections* that lists the new vertices for each triangle. We also have the number of faces attached to each vertex in *counts*, and the welded vertices and their associated attributes in *welded-values*.

and the second operator would be a divide. In our implementation, we restrict the first operator, hereafter referred to as a merge, to be analagous to a sum: a binary operator that is commutative and associative. We restrict the input and output types of this operator so that it must take in two geometric elements and output a version that merges the elements' properties in some way. Likewise, we restrict the second operator, hereafter referred to as a transform, to be analagous to scalar division: a binary operator taking one geometric element and the number of elements in the group as input. The return type of the transformation operator is unrestricted, and the result of these transformations will be passed as the output of our algorithm.

Finally, for this technique to be useful it must be possible to determine where each element of the mergeable array is mapped to in the output array. Our algorithm will therefore return an array of the new indices for each of the input elements.

With these modifications, we provide a generalized method named KEY-WELD that groups various types of geometric elements and then merge each group into single outputs.

Steps 7 through 18 in KEY-WELD are described at a high level for clarity. For increased data parallelism, we actually implement this portion of the algorithm using the Thrust library's `inclusive_scan_by_key` and `zip_iterator`. An example of how this algorithm works is provided in Figure 1.

4 MESH GENERATION

In order to test our method for generation of topological connectivities on a range of data types, we made use of several techniques to generate geometry. To test with data calculated along a structured grid, we used the Marching Cubes algorithm to generate contour surfaces from input volume data. Next, to test with data from unstructured grids, we tetrahedralized the volume in each of the manners as presented by Carr, et al. [7]. We generated contour surfaces from these tetrahedralizations to get similar data as from the Marching Cubes results but from an unstructured grid. Additionally, we use the tetrahedralizations directly as the input to test our technique to find the topological connections of a 3D structure.

4.1 Testing Environment and Datasets

We measure performance on a test system containing an Intel Core i7 975 Processor, which has a clock speed of 3.33GHz. Our test system has 12GB of RAM, and contains 2 nVidia Tesla C2070 cards. The

KEY-WELD(*keys*, *values*, *compare*, *merge*, *transform*)
keys: An array of keys uniquely identifying elements.
values: An array of mergeable elements the same size as *keys*.
compare: A key comparator $compare(k_1, k_2) \rightarrow$ BOOL.
merge: A merging operator $merge(m_1, m_2) \rightarrow m_3$.
transform: A transformation $transform(m_1, size) \rightarrow out$.

▷ Sort *values* using *keys* to make groups.
1 (*sorted-keys*, *sorted-values*)
  ← KEY-SORT(*keys*, *values*, *compare*)
▷ Determine the new indices for each element.
2 *unique-keys* ← UNIQUE(*sorted-keys*)
3 *cell-connections* ← VECTORIZED-FIND(*unique-keys*, *keys*)
▷ Get a reverse map from output array to sorted arrays.
4 *reverse-map*
  ← VECTORIZED-FIND(*sorted-keys*, *unique-keys*)
▷ Get the size of each group (one per unique key).
5 *weld-array-size* ← LENGTH(*reverse-map*)
6 *counts* ← {}
7 **for** *i* ← 0 **to** *weld-array-size* − 2
  do in parallel
8 *counts*[*i*] ← *reverse-map*[*i* + 1] − *reverse-map*[*i*]
9 *counts*[*weld-array-size* − 1]
  ← LENGTH(*sorted-keys*)
  − *reverse-map*[*weld-array-size* − 1]
▷ Merge elements of each group into a single element.
10 *welded-values* ← {}
11 **for** *weld-index* ← 0 **to** *weld-array-size* − 1
  do in parallel
12 *sort-index-start* ← *reverse-map*[*weld-index*]
13 *count* ← *counts*[*weld-index*]
14 *v* ← *sorted-values*[*sort-index-start*]
15 **for** *group-index* ← 1 **to** *count* − 1
    do
16 *sort-index* ← *sort-index-start*
      + *group-index*
17 *v* ← *merge*(*sorted-values*[*sort-index*]
18 *welded-values*[*weld-index*]
    ← *transform*(*v*, *count*)
19 **return** (*welded-values*, *cell-connections*, *counts*)

Alg. 2: The KEY-WELD procedure welds vertices and allows multiple local samples of vertex attributes to be merged to form a better sampling without nonlocal operation during geometry generation.

Table 1: Test Datasets

Name	Dims	Triangles	Tetrahedrons
Spherical Distance A	64^3	35,996	1,250,235
Spherical Distance B	128^3	149,420	10,241,915
Spherical Distance C	256^3	607,388	82,906,875
Spherical Distance D	512^3	2,451,212	N/A
MRI Head A	64^3	138,510	1,250,235
MRI Head B	128^3	789,440	10,241,915
MRI Head C	256^3	4,947,294	82,906,875

datasets we use for timing are shown in Table 1. For all OpenMP tests, we set the number of threads to 8.

The isovalues used for contour generation are consistent for all tests within each dataset, and are chosen to show relevant features. We attach vertex normals and a scalar color value as associated attributes to each input vertex for surface contour data, but only include the color value for the tetrahedral tests due to memory constraints. In addition to the datasets listed for timings, we also use a concentric ripple dataset for some of our images for demonstration. Timings for this dataset were similar to those for the spherical dataset in all cases.

We have performed multiple tetrahedralizations as described in Carr [7] to generate tetrahedral grids for input to our method, but found that our method's performance is similar in all cases. We therefore report only the timings for the minimal tetrahedralization method described in the aforementioned paper due to its smaller memory requirements. We perform Marching Cubes on each dataset to produce contour geometry. We have found that our algorithm performs similarly with output from Marching Tetrahedrons, and therefore only report the timings for Marching Cubes. All reported times in all tables are in milliseconds unless otherwise noted. The Thrust library does not currently support spanning operations over multiple GPUs. We make use of two GPUs for our measurements by partitioning the input and performing the algorithm separately on each, then merging the results.

4.2 Method of Topological Connectivity Construction

Our technique is an extension of existing vertex welding methods. We will first analyze the original vertex welding approach and the manner in which we extended it for topological connectivity construction. We will then discuss methods of performance improvement of the original vertex welding method that we can achieve due to our extensions. Finally, we will discuss the usage of our extensions for the generation of topological connectivity information for various types of geometric inputs.

4.2.1 Vertex Welding

Because we made several modifications and extensions to the original vertex welding technique, we measure the additional performance cost of our implementation of the technique as compared to the original. To perform a fair comparison, it is necessary to apply our technique to the same task. Vertex welding with our KEY-WELD algorithm can be accomplished by specifying the following inputs:

- The array of mergeables *values* should be an input array of vertex coordinates.
- The *keys* array should be the same array as *values*.
- The key comparator *compare* should compare vertex coordinates lexicographically.
- The *merge* operator should simply return the first element.
- The *transform* operator should simply return the given element.

For the purposes of vertex welding, we ignore the *counts* output of the KEY-WELD algorithm, and treat the outputs *welded-values* and *cell-connections* as the original VERTEX-WELD algorithm's corresponding outputs with the same names. In this case, steps 7 through 18 can be optimized to a single call to UNIQUE, as in the original VERTEX-WELD algorithm. We therefore take no performance hit by generalizing VERTEX-WELD for the no-attribute case, as demonstrated in Table 2.

Although we were able to measure the performance differences using the KEY-WELD technique, it provides no benefit over the original VERTEX-WELD in this case. Also, our generated contour surfaces contain per-vertex normals and several scalar attributes, and this approach loses all information for these associated attributes except for those related to the first encountered instance of each vertex. To resolve this, we substitute a *merge* operator that sums these attributes, and a *transform* operator that divides the given element by *size* (and renormalizes in the case of the normal vector) so that our output vertices contain an averaged sample of the associated attributes from each instance of each vertex. We then measure the performance impact from this change.

To improve performance over VERTEX-WELD, we make use of our ability to use input keys other than the original vertices. Initially we considered the case of the output of our structured grid approach, Marching Cubes. In this case, we know a good deal about the input topology on which the output is generated that was not previously being used to help generate the topological connections. Specifically, the output vertices from Marching Cubes are generated only on the edges of input voxels. Each of these edges is shared by at most four voxels (ignoring degenerate vertex intersections), so there can be at most four instances of any one vertex. We assign each unique edge in the voxel grid an implicit integer ID, then store the corresponding edge ID for each vertex generated by Marching Cubes. There is no additional storage requirement because the ID can be implicitly determined from vertex position, but even when doing so this modification causes no measurable change in the performance of geometry generation. It does, however, significantly improve performance during vertex welding. Because it is known that these IDs are identical if and only if the vertices are identical, we hypothesize that by using these integer IDs as the *keys* in the KEY-WELD algorithm we noticeably improve the performance of the technique due to the simpler *compare* operation used during KEY-SORT, UNIQUE, and VECTORIZED-FIND.

We next consider the case of unstructured grid inputs. In this case edges cannot in general be implicitly assigned IDs, so they must be built from vertex IDs, which are implicit. We pair vertex IDs (least to greatest) to form a unique integral ID for each edge of the unstructured grid and store this as an attribute of each output vertex.

Using the integer ID approach described above carries an additional advantage. From our Marching Cubes and Marching Tetrahedra implementations, we find that the VERTEX-WELD approach often fails to weld meshes as expected. We trace this failure to the fact that some vertices that should be identical actually differ slightly due to floating point error, and thus are not bitwise equal as required by the VERTEX-WELD approach. Such errors are exceptionally easy to encounter: since floating-point operations are not associative, these errors can be generated simply by failing to ensure that we always use the same vertex order when generating interpolants. By assigning integral IDs to the output vertices, KEY-WELD becomes agnostic to such floating-point error in the input vertices, and thus does not require careful prevention of these errors in the geometry-generating implementation.

4.2.2 Vertex Welding Results

We measure performance of the VERTEX-WELD example on each set of generated geometry. Additionally, we measure KEY-WELD with and without merging of vertex properties, and with and without the integral IDs as *keys* to each vertex. Even when explicitly storing IDs in memory, the difference in recorded times for geometry generation are negligible for both structured and unstructured grids as opposed to the timings when these IDs are not generated. Because of this, and because geometry generation is independent of our technique, we

Table 2: Performance comparison for different approaches to Vertex Welding

Test	CUDA	OpenMP
VERTEX-WELD (no attribute merging)	281	1557
KEY-WELD (no attribute merging)	281	1557
KEY-WELD (with attribute merging)	527	2862
KEY-WELD (integral IDs, no merging)	204	1620
KEY-WELD (integral IDs, merging)	350	2516

Table 3: KEY-WELD timings for contours on structured grids.

	Spherical Distance				MRI Head		
Arch.	64^3	128^3	256^3	512^3	64^3	128^3	256^3
CUDA	7	23	254	349	22	116	755
OpenMP	30	125	550	2516	115	695	3377

Table 4: KEY-WELD timings for tetrahedra.

	Spherical Distance			MRI Head		
Arch.	64^3	128^3	256^3	64^3	128^3	256^3
CUDA	127	1137	9301	128	1136	9322
OpenMP	1228	11677	$>$ 5min	1231	11655	$>$ 5min

do not report timings for geometry generation here. We report the CUDA and OpenMP timings for each approach for vertex welding on the 512x512x512 spherical dataset in Table 2.

Because the timings are best for merging when using integral edge IDs as keys for vertex welding when merging is necessary, all further timings in this paper use that approach for the vertex welding component. KEY-WELD reduces to VERTEX-WELD when no attribute merging is performed, so the timings are identical. Also note that using integral IDs for the non-merging case with KEY-WELD yields better performance in CUDA than VERTEX-WELD, due to the less complex comparison operator within the sort.

The OpenMP non-merging case does not appear to benefit from the use of integral IDs. Timings for the KEY-WELD technique for Marching Cubes contours of all input datasets are shown in Table 3, and timings when using tetrahedralizations of the input dataset are shown in Table 4

The performance of the technique appears to be bound by the number of vertices passed as input. In all cases, performance scales approximately with the number of input vertices.

The tetrahedral case achieves approximately twice the performance per vertex of the contour case because of the absence of the normal vector as an attribute. The OpenMP version scales similarly to the CUDA version until system memory is exceeded in the largest tetrahedral dataset, at which point disk thrashing occurs during the sorts.

4.2.3 Incidence and adjacency list construction

When only considering vertex welding, we make no use of the output *counts* of KEY-WELD, which contains the sizes of the merged groups. This output is more useful when we wish to determine the topological connections between the merged elements in *welded-values*.

We demonstrate the use of our KEY-WELD technique as a basis for determination of the following kinds of topological connectivity information:

- Lists of adjacent vertices and incident edges, faces and tetrahedra for each input vertex
- Lists of faces incident to each edge or adjacent to each face, even those connected only by a single vertex
- Lists of faces adjacent to each face, sharing exactly one edge
- Lists of tetrahedra adjacent to each tetrahedron, including those connected only by an edge or point
- Lists of tetrahedra adjacent to each tetrahedron, sharing exactly one face

We begin with a procedure VERTEX-INCIDENCE-LISTS for the generation of incident faces for each input vertex on a generated surface. Example input is provided in Figure 2.

Though our example input *cell-ids* is for triangular faces, this process can be trivially extended by modifying the array *cell-ids* to represent other polygonal faces, tetrahedra or any other cell type. To determine edges surrounding a point, the process could be similarly modified to have each item in *cell-ids* label the represented edge. Note that when surrounding tetrahedra are generated for each vertex, surrounding faces are also generated as a subset. Similarly, when surrounding faces are generated surrounding edges are generated as a subset, which can be used to quickly determine the nearest neighboring vertices for each input vertices where an edge exists between the two.

Marching Cubes and Marching Tetrahedra both have the property that all generated surface geometry forms a manifold, with the exception of degenerate and boundary cases. Because of this, we can determine more information about the neighboring geometric features. To determine all faces touched by an edge, simply combine the incident face lists of the two endpoints. If only the two faces separated by the edge are desired, these can be detected because they will occur twice in the list. All other faces will occur only once.

Determination of adjacent face lists for each triangular face works similarly to edges. Combine the incident face lists of all three endpoints. The face that appears three times in the combined list is the current face. Faces that occur twice are separated from this face by an edge, and those that appear only once are separated by a point. Likewise for tetrahedra: The tetrahedron that appears four times in the resultant list is the current tetrahedron, those that appear three times are adjacent to it through a face, those that appear twice are adjacent

VERTEX-INCIDENCE-LISTS(*cell-ids*, *cell-connections*, *counts*)
cell-ids: An array of cell identifiers for each cell vertex.
Usually sequential repeated indices (e.g. $\{0,0,0,1,1,1,...\}$)
cell-connections: Result from KEY-WELD representing cell connectivity.
counts: Result from KEY-WELD, number of cells per vertex.

▷ Use a key-sort with cell connections as the keys to
▷ group cell identifiers by vertices.
1 $(-, links) \leftarrow$ KEY-SORT(*cell-connections*, *cell-ids*)
▷ The size of each incidence list is the same as *count*.
2 *links-counts* $\leftarrow$ *counts*
▷ Use an exclusive scan to find the offset to the incidence
▷ list for each vertex.
3 *links-offsets* $\leftarrow$ EXCLUSIVE-SCAN(*links-counts*)
4 **return** (*links*, *links-counts*, *links-offsets*)

Alg. 3: VERTEX-INCIDENCE-LISTS takes the output of the KEY-WELD algorithm and quickly generates an incidence list.

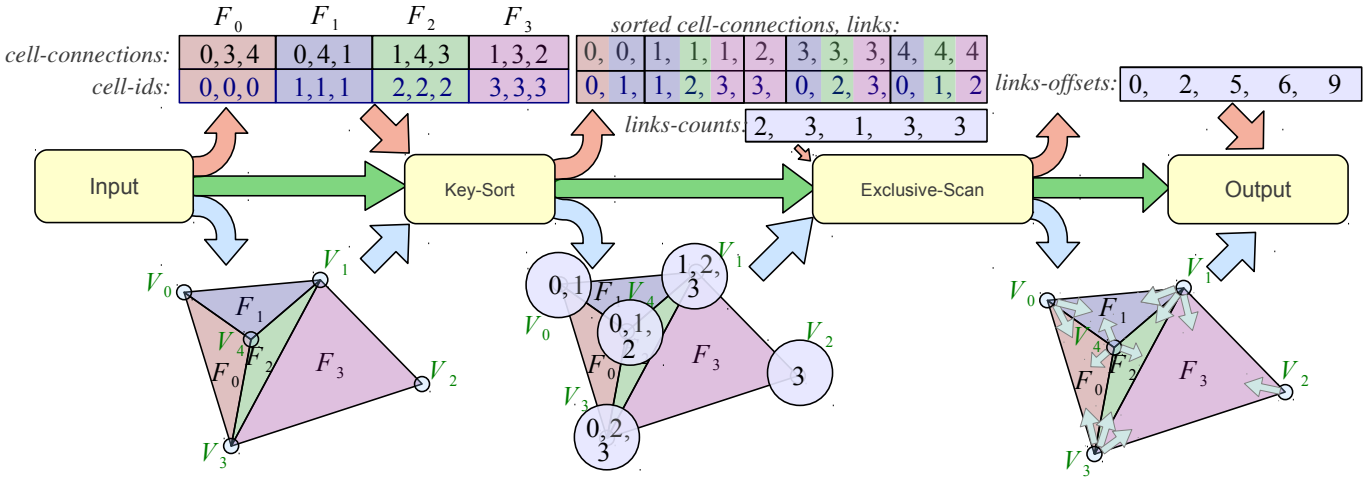

Fig. 2: An example of the VERTEX-INCIDENCE-LISTS procedure on a very small set of triangles. Data values within the algorithm at several stages are shown above the flow diagram. Upon output, *links-offsets* contains offsets to the beginning of each vertex's face list in *links*.

Table 5: Timings for VERTEX-INCIDENCE-LIST for contours on structured (SG) and unstructured (UG) grids.

	Spherical Distance				MRI Head		
Arch.	64^3	128^3	256^3	512^3	64^3	128^3	256^3
CUDA	1	2	6	25	2	7	47
OpenMP	6	32	125	563	33	191	1213

Table 6: Timings for VERTEX-INCIDENCE-LIST on tetrahedra.

	Spherical Distance			MRI Head		
Arch.	64^3	128^3	256^3	64^3	128^3	256^3
CUDA	15	139	1141	16	142	1128
OpenMP	378	3610	$>$ 5min	371	3621	$>$ 5min

to it through an edge, and those that appear once are adjacent to it through a point. Similar behavior occurs for other types of polyhedra and polygons.

Although the optimizations used to perform KEY-WELD change depending on whether the input is a structured or unstructured grid, the output is processed into topological connectivity information in the same manner for either case.

4.2.4 Incidence and adjacency list construction results

The time necessary to generate incidence lists for contours of structured and unstructured grids is shown in Table 5. The time necessary to generate incidence lists for tetrahedral volumes is shown in Table 6.

Performance for generation of incidence lists scales approximately linearly with the number of input vertices for CUDA. OpenMP scales similarly to CUDA until system memory is exceeded in the largest tetrahedral dataset.

5 APPLICATIONS

To test that the generated incidence and adjacency lists are useful on parallel systems once generated, we implement several calculations that make use this topological connectivity information. Specifically, we have implemented the following:

- Local averaging of vertex attributes
- A form of mesh coarsening, implemented as an extension to KEY-WELD without incurring additional performance cost
- Generation of a dual mesh

Table 7: Timings for local averaging of scalar (S) and vector (V) properties on contour surfaces

	Spherical Distance				MRI Head		
Type	64^3	128^3	256^3	512^3	64^3	128^3	256^3
(S) CUDA	18	47	158	650	44	51	203
(V) CUDA	18	48	158	649	45	51	203
(S) OMP	39	180	905	4035	184	1391	8611
(V) OMP	39	181	914	4039	191	1390	8625

The local averaging technique is an example that does not alter the topological connectivity information, and thus does not require an additional application of our technique after its completion. Mesh coarsening is an example that alters the input topology, and thus requires a regeneration of the topological connectivity information after its completion. Also, we use mesh coarsening to demonstrate that certain types of mesh processing can occur during the vertex welding phase of topological connectivity construction. Generation of a dual mesh is an example that generates a new topology.

5.1 Local Averaging

We perform a simple averaging of properties of the nearest neighboring vertices surrounding each processed vertex. This is accomplished by first obtaining the list of indices to neighboring points as described in 4.2.3. Duplicates in this list are removed, then the attributes of each are stored. We perform a segmented scan [5] to sum the properties of the neighboring vertices using a merge operator that is appropriate for the property type, then divide the summed properties by the number of neighbors for the group. Some vertex properties, such as normals, need an alternative operator to replace the division operation. Since we implemented our approach as a C++ template, this operator can be specified as a template parameter.

5.2 Local Averaging Results

Table 7 shows the timings for calculating the local averages of vector properties on the generated surfaces. Figure 3 shows a before and after comparison of averaging used for vector averaging.

Fig. 3: Surface normal artifacts are visible in the image of the MRI head dataset at left due to sampling error caused by the local operation of our Marching Cubes algorithm. The most common solution is to visit neighboring voxels while producing geometry to obtain a better sampling, but this can impede performance on the GPU. Without modifying Marching Cubes to do so, we can achieve better results by averaging local normals as shown in the central image, but this requires an additional computation pass. In section 5.3, we demonstrate a way to remove these artifacts without incurring the additional performance cost required by local averaging, as shown in right image.

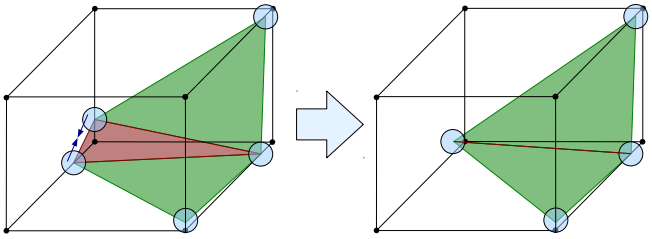

Fig. 4: When Marching Cubes and Marching Tetrahedra generate contour surfaces, the produced vertices can only occur on voxel edges (black). When several output vertices (blue) fall close to the same voxel vertex (black), small or skinny triangles may be produced (red). We eliminate these by merging all vertices that fall close to a voxel vertex into a single output. This collapses the skinny triangles into degenerate triangles, which are then removed.

Performance for averaging scales approximately linearly with the number of input vertices in all cases. We have profiled the performance of the algorithm and found that approximately 96% of the time taken is spent removing duplicate vertices from the incidence lists. Generation of the incidence list is the next largest fraction of the processing time, at 3%. The time taken to perform the sum and divide operations is negligible.

5.3 Mesh coarsening

Many algorithms that make use of an input topology must modify that topology to produce their output. To demonstrate how such an algorithm can be constructed with our technique, we implement a form of simple mesh coarsening. Traditional implementations of mesh coarsening operate without regard to the mesh's provenance by either iteratively collapsing features [20] or collectively clustering vertices [10]. However, by combining edge collapsing with vertex welding we can provide a first-level coarsening that is fast, has low error, provides well-formed cells, and adapts to the structure of the original mesh.

Rather than computing the topological connectivity of the original generated data and then coarsening it thereafter, we instead alter the input key array to our Vertex Welding technique to perform a type of coarsening in the process of topological connectivity generation. Furthermore, this technique is intended to be more practically useful than the first implementation in that it would improve mesh quality. Specifically, it is our goal to eliminate poor quality triangles in the mesh while performing minimal modifications to the others. We define triangle quality Q as in Equation 1.

$$Q = \frac{4a\sqrt{3}}{h_1^2 + h_2^2 + h_3^2} \tag{1}$$

This measure of triangle quality considers only aspect ratio, and is such that triangles where $Q > 0.6$ are considered to be of acceptable quality, and $Q = 1$ when the triangle is equilateral [1]. This is equivalent to the pdetriq function in MATLAB.

The basic idea for our coarsening is described in Figure 4. Marching Cubes generates small or skinny triangles when two adjacent output vertices are generated near the same input mesh vertex. To eliminate this possibility, we consider all vertices close to any particular endpoint as identical so that during Vertex Welding, these skinny triangles become degenerate and can be removed. Essentially we divide the edges on which geometry is generated into three regions. We reserve a parameter k to determine the size of these regions as a fraction of the edge length ℓ. For unstructured grids, we modify the ID assignment we used previously in vertex welding. As before, we initially assign each vertex in the input topology an ID. During geometry generation, we pair these vertex IDs to form edge IDs, which are attached as attributes to the output vertex. However, when a vertex falls within $\frac{k\ell}{2}$ of an endpoint of this edge, the ID of that endpoint is repeated as shown in Figure 5.

As with the original ID assignment for structured grids, we can use our knowledge of the input topology to assign the IDs implicitly so that there is no performance or storage cost. This is accomplished by subdividing space as shown in Figure 6 and assigning an integral ID to each cell. All vertices within these cells will be considered identical and merged during Vertex Welding. The value of k is restricted such that $0 \leq k \leq 1$, and k specifies the width of the cells that will group vertices. By placing the center of these cells over the intersections of the voxel grid, the 'skinny' triangles generated by Marching Cubes can be reduced or eliminated. In the structured grid case, when $k = 1$ we can guarantee a minimum possible triangle quality. For cubic voxel grids, this minimum is $Q = 0.275$, and the expected triangle quality is much higher, as shown in Figure 8. We cannot make such a guarantee about the output of the coarsening in the unstructured grid case because the triangle quality depends on the aspect ratios of the input cells. When $k = 0$ this technique is equivalent to the previous form of vertex welding.

After the coarsening has been applied, we expect many degenerate triangles in the output. These triangles may be quickly removed

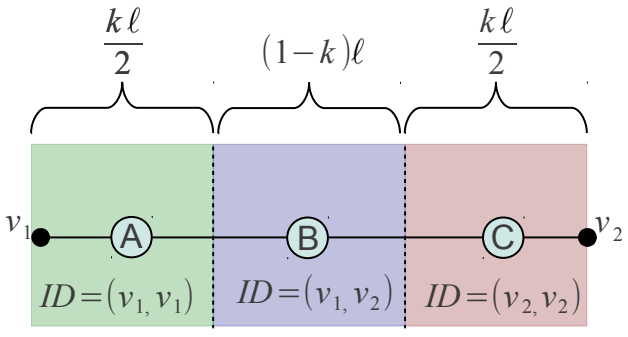

Fig. 5: Our goal is to merge output vertices (light blue) that are generated too close to the vertices of the input voxel grid (black). To do so, we split each input voxel edge into three regions along edges of length ℓ, using a control parameter k to control the size. If a vertex is generated in the central region (B), we assign an output ID as normal. However, if a vertex is generated in one of the edge regions (A or C), we repeat that region's vertex ID. All output vertices generated near a particular input grid vertex are thus assigned the same ID and merged, which collapses skinny triangles to degenerate cases.

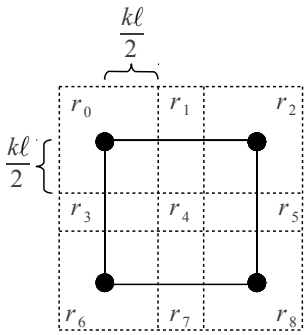

Fig. 6: 2D representation of implicit spatial subdivision into regions for the assignment of IDs to vertices generated on structured grids. Each region surrounding a voxel edge is shared by between 4 and 8 voxels. Because these regions can be determined directly from the output vertex coordinates, they need not be stored in memory. As such, no modification to the geometry generating function is required if the input voxel grid is known. Furthermore, determination of the integral region ID allows for faster sorting, as no lexicographic comparison of the vertices is required.

in parallel by removing all instances of faces containing two or more identical indices. This invalidates the output array of group sizes C from the Vertex Welding array, but a corrected group sizes C can be generated by sorting the output array $cell - connections$ and performing a segmented scan to determine the number of instances of each index.

A larger problem with the coarsening technique is that in some cases, such as the case shown in Figure 7, non-manifold surface elements may be generated. If such surface elements are undesirable, they may be eliminated by calculating the maximum edge curvature of each triangle, and removing all triangles containing an edge curvature of 180°.

5.4 Mesh Coarsening Results

Figure 9 shows a before and after comparison of our coarsening technique with $k = 1$. Figure 8 shows the change in triangle quality before and after coarsening. Table 8 shows the timings for coarsening the mesh as part of the vertex welding algorithm. Compare with Table 3

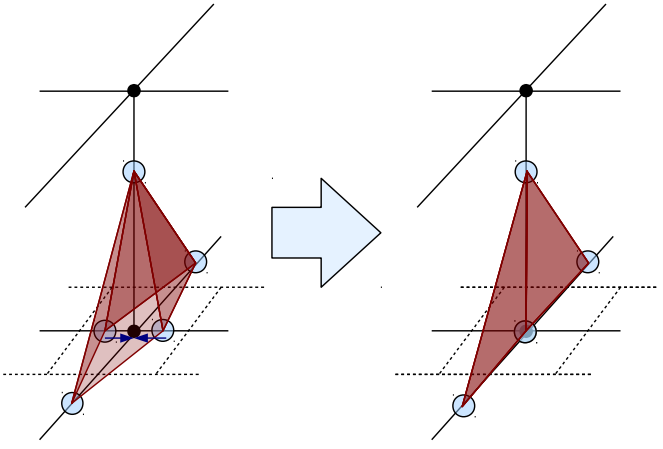

Fig. 7: A problem with our coarsening method is that in some cases nonmanifold faces may be produced. In the example above, two vertices from different faces are merged and a 3D volume is collapsed into a single plane. To recreate a manifold surface, we detect and remove these cases in a separate step.

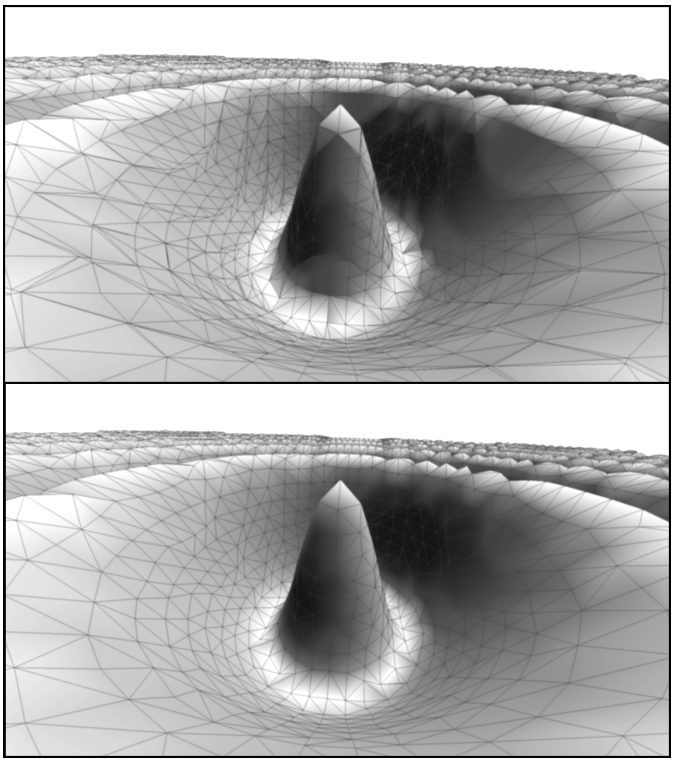

Fig. 9: Top: Before coarsening. Bottom: After coarsening. Note that skinny triangles caused by the underlying voxel grid are removed in the bottom image. Removal of these triangles eliminates the artifacts visible along the crests and valleys of the top image, and provides a better sampling of vertex properties across the contour surface. The coarsened mesh in the bottom image contains approximately half the triangle count of the mesh in the top image.

to observe that simultaneously performing the coarsening operation does not significantly affect the timing of the topology generation. In addition, sampling of vertex attributes is markedly improved in the coarsened graph. Although our Marching Cubes algorithm strictly determines surface normals by the information given in a single voxel, we find that our approach yields similar quality surface normals to those generated by the VTK Marching Cubes algorithm, which visits neighboring voxels to determine accurate normals. This is advantageous because it allows for completely local operation during geometry generation without loss of computed attribute quality, which provides better performance on the GPU.

Table 8: Timings for mesh coarsening of surface contours during vertex welding

	Spherical Distance				MRI Head		
Arch.	64^3	128^3	256^3	512^3	64^3	128^3	256^3
CUDA	7	24	86	351	22	114	716
OpenMP	28	122	561	2567	122	807	6208

5.5 Dual Mesh Generation

To generate the dual of our triangular input meshes, we perform the following procedure: For each face, weaverage the vertices to get the centroid, and store it in a new array at the same ID as the original face. Now the surrounding face lists from KEY-WELD for each of the vertices in the original mesh form a representation of each face of the dual mesh, with the vertices not necessarily in order. To order the vertices, we create an adjacency list for all faces in parallel as

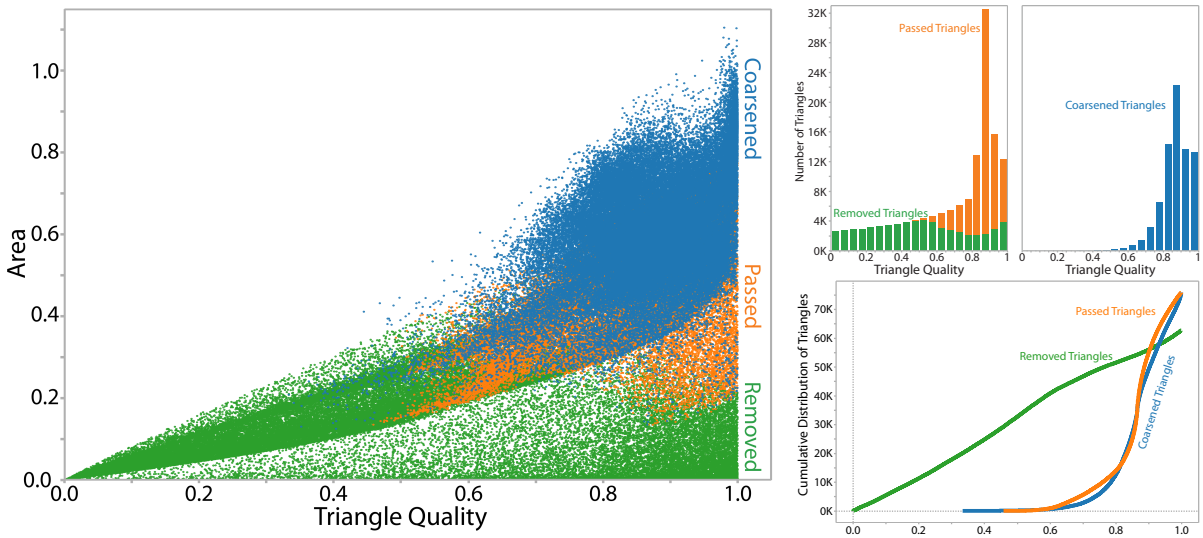

Fig. 8: Left: Scatterplot of triangle area and triangle quality for the MRI head dataset as calculated in Eq. 1 in Section 5.3. Note that in general the triangles that are removed by our coarsening (green) are either small or poor quality as compared with other triangles in the distribution. Triangles that are not removed by the coarsening (orange) are transformed by the coarsening into the output triangles (blue). While not all passed triangles are improved by this coarsening, there is a lower bound on produced triangle quality. After coarsening, most of the triangles are relatively large and high quality. Top Right: Histograms of the quality of passed, removed, and coarsened triangles. Bottom Right: Cumulative distributions of triangles before and after coarsening. Our coarsening generally reduces the size of the output mesh to approximately half of its original size. Median quality of the removed triangles is approximately $Q = 0.5$. Median quality of passed triangles is approximately $Q = 0.85$, and after coarsening the median quality of the output triangles remains at approximately $Q = 0.85$.

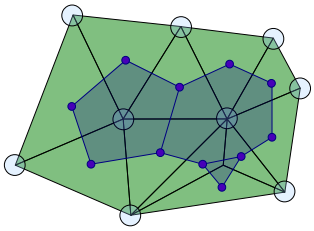

Fig. 10: 2D example of a dual mesh (blue) overlaid against a triangular mesh (green).

Table 9: Timings for dual mesh generation

	Spherical Distance				MRI Head		
Arch.	64^3	128^3	256^3	512^3	64^3	128^3	256^3
CUDA	20	64	255	1111	62	337	1480
OpenMP	24	113	522	2263	115	531	2302

described in 4.2.3, then use the adacency lists for each face to 'walk' around each dual mesh cell, storing the order as we go. We perform all of these operations in a data parallel manner using existing functions of the Thrust library. Figure 10 shows an example of this kind of dual mesh for a 2D case.

5.6 Dual Mesh Generation Results

Figure 11 shows a mesh and its dual overlaid as a skeleton. Table 9 shows the timings for generating dual meshes. Similarly to the case for local averaging, the time taken to generate a dual mesh is dominated by the time taken to remove unique elements from the incidence list, thereby generating an edge adjacency list. The time taken for generating a dual mesh exceeds that for the local averaging case because the face adjacency lists are larger than vertex incidence lists by a factor of 3.

6 Conclusions and Future Work

We provide an efficient, generalized method for parallel generation of topological connectivity information. We require little to no alteration to the algorithms generating geometry, although we leverage small modifications to allow for knowledge of input topological features for better performance. We demonstrate how to use such modifications to gain performance on structured grids, and to perform a simple mesh coarsening on either structured or unstructured grids. We demonstrate that our algorithm is effective on structured grids and on several different tetrahedral grids.

Future work includes measurement of how well this technique can scale on architectures approaching the exascale and determining what bottlenecks, if any, need to be addressed for this transition.

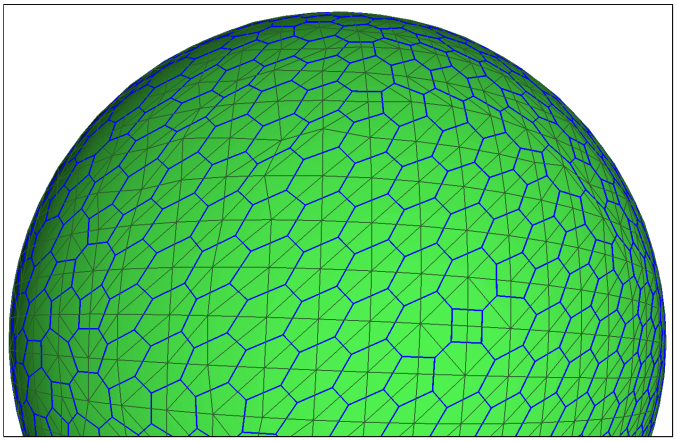

Fig. 11: Dual mesh (blue) overlaid as as a skeleton upon a triangular contour mesh (green) that was generated by Marching Cubes with the spherical distance dataset.

REFERENCES

[1] R. E. Bank and M. Holst. A New Paradigm for Parallel Adaptive Meshing Algorithms. *SIAM Review*, 45(2):291, 2003.

[2] N. Bell. High-Productivity CUDA Development with the Thrust Template Library, 2010.

[3] N. Bell and J. Hoberock. Thrust: A Productivity-Oriented Library for CUDA. *thrust.googlecode.com*, pages 359–373, 2012.

[4] G. Blelloch. Prefix sums and their applications. *Synthesis of Parallel Algorithms*, pages 35–60, 1990.

[5] G. E. Blelloch. Vector Models for Data-Parallel Computing. *Computing Control Engineering Journal*, 2(5):238, 1991.

[6] T. Boubekeur and C. Schlick. Generic mesh refinement on GPU. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on graphics hardware*, number July, pages 99–104. ACM, 2005.

[7] H. Carr, T. Möller, and J. Snoeyink. Artifacts caused by simplicial subdivision. *IEEE transactions on visualization and computer graphics*, 12(2):231–42, 2006.

[8] H. P. Computing. Scientific Visualization with VTK, The Visualization Toolkit. *Fluid Dynamics*, 12(1):1–9, 1992.

[9] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, January 2008.

[10] C. DeCoro and N. Tatarchuk. Real-time mesh simplification using the GPU. In *Proceedings of the 2007 Symposium on Interactive 3D Graphics and Games (I3D '07)*, pages 161–166, 2007. DOI 10.1145/1230100.1230128.

[11] C. Dyken, G. Ziegler, C. Theobalt, and H.-P. Seidel. High-speed Marching Cubes using HistoPyramids. *Computer Graphics Forum*, 27(8):2028–2039, 2008.

[12] M. Garland, A. Willmott, and P. S. Heckbert. Hierarchical face clustering on polygonal surfaces. *Proceedings of the 2001 symposium on Interactive 3D graphics SI3D 01*, pages 49–58, 2001.

[13] H. Hoppe. Progressive meshes. *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques SIGGRAPH 96*, 30(3):99–108, 1996.

[14] D. Horn. Stream Reduction Operations for GPGPU Applications. In M. Pharr, editor, *GPU Gems 2*, volume 2, chapter 36, pages 573–589. Addison Wesley, 2005.

[15] H. Hsieh and W. Tai. A simple GPU-based approach for 3D Voronoi diagram construction and visualization. *Simulation Modelling Practice and Theory*, 13(8):681–692, Nov. 2005.

[16] P. Kipfer and R. Westermann. GPU construction and transparent rendering of iso-surfaces. In *Proceedings Vision, Modeling and Visualization*, volume 5, 2005.

[17] P. Lindstrom. Out-of-core simplification of large polygonal models. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 259–262. ACM Press/Addison-Wesley Publishing Co., 2000.

[18] W. E. Lorensen and H. E. Cline. Marching cubes: A high resolution 3D surface construction algorithm. *Computer*, 21(4):163–169, 1987.

[19] J. Park, B. Choi, and Y. Chung. Efficient topology construction from triangle soup. In *Geometric Modeling and Processing, Proceedings*, volume 1, pages 359–364. IEEE, 2004.

[20] M. Potter. *Anisotropic mesh coarsening and refinement on GPU architecture*. PhD thesis, Imperial College London, Department of Computing, June 2011.

[21] S. Röttger, M. Kraus, and T. Ertl. Hardware-accelerated volume and isosurface rendering based on cell-projection. In *Proceedings of the conference on Visualization'00*, pages 109–116. IEEE Computer Society Press, 2000.

[22] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens. Scan Primitives for GPU Computing. *Computing*, 21(3):106, 2007.

[23] E. Sifakis. Arbitrary Cutting of Deformable Tetrahedralized Objects. *Computing*, 1, 2007.

[24] J. A. Stuart, C.-K. Chen, K.-L. Ma, and J. D. Owens. Multi-GPU volume rendering using MapReduce. In *1st International Workshop on MapReduce and its Applications*, June 2010.

[25] H. T. Vo, J. Bronson, B. Summa, J. L. Comba, J. Freire, B. Howe, V. Pascucci, and C. T. Silva. Parallel visualization on large clusters using MapReduce. In *Proceedings of the IEEE Symposium on Large-Scale Data Analysis and Visualization*, pages 81–88, October 2011. DOI 10.1109/LDAV.2011.6092321.