Mesh Simplification via Embedded Tree Collapsing Implemented in CUDA

Nicolas Mignucci, George Ralph

URL:

https://github.com/gdr22/15-418-Final-Project

Summary:

We implemented and tested a parallel mesh simplification algorithm in CUDA on the GPUs in the GHC servers. Our implementation is based on a parallel algorithm proposed by Lee and Kyung. We compared the performance of our CUDA implementation with Open3D's sequential implementation, and determined that while the output meshes generated by our implementation have some less desirable characteristics, the CUDA implementation presents a 400 to 1200 factor speedup, depending on the complexity of mesh provided.

Background:

Quadric mesh simplification algorithms use a quadric error metric to determine which edges in a manifold mesh can be collapsed to reduce the mesh's vertex count while retaining its original shape. The original implementation, devised by Garland and Heckbert computes an error metric for each edge on the mesh and iteratively collapses edges with the lowest cost until the mesh is of desired size.

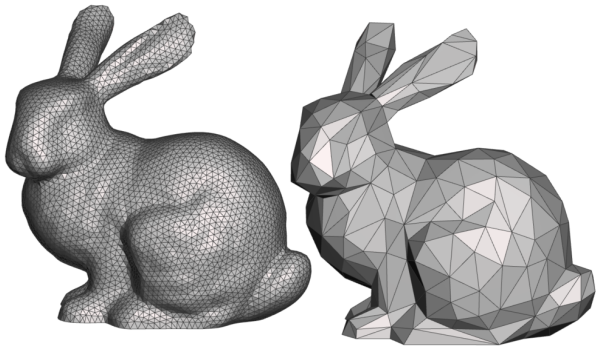

Figure 1: Example of mesh simplification

The quadric decimation algorithm relies on a mesh data structure known as a halfedge mesh. In this structure, which resembles a linked-list, triangle halfedges contain the indices of their neighboring edges, as well as the vertices and triangles they are a part of. While useful at performing local updates on a mesh, such as collapsing edges and faces to a single vertex (as used in the decimation algorithm) these operations involve changing values in up to six non-adjacent fields in the mesh. This presents a challenge for performing these collapsing operations in parallel. If these fields are not updated atomically, it is possible for the mesh to become non-manifold, gaining undesirable features like holes and infinitely thin volumes.

Also contained in the mesh is a list of vertex positions, stored as an array of three dimensional vectors. In order to compute the collapsing error of each edge, which determines what edges can be removed from the mesh while retaining as much detail as possible, we compute the quadric error matrix for each vertex. After computing the error matrices for each vertex, and in turn using these matrices to determine the error of each potential edge collapse in the mesh, the Garland and Heckbert algorithm places all edges into a priority queue, and collapses the edge with minimal error until the mesh has reached the desired complexity.

The algorithm then functions as follows:

1. Compute quadric error matrices for all vertices
2. Compute the optimal contraction target for each vertex.
3. Place all of the contracting pairs in a heap ordered by the quadric error invoked by the contraction..
4. Iteratively collapse the lowest-error pairs, updating the costs of other pairs, until the desired number of vertices is reached.

Here, we see many opportunities for parallelism. If a set of edges within the mesh is unrelated, i.e. no vertices would be collapsed into a vertex which is being collapsed by another edge, it would be simple to collapse all of the edges in the set in parallel. We also see opportunities for parallelism in the computation of quadric error matrices for each vertex, as this process is completely independent for each vertex. Both of these ways of parallelizing this algorithm involve performing the same operation many times simultaneously, meaning our parallel algorithm would be quite amenable to data parallelism.

While there are parts of the program with a high potential to be parallelized, there are significant barriers to parallelization. As edges are collapsed, the cost of other edges must be updated based on the new construction of the mesh. Additionally, as discussed above, it is not easy to atomically collapse halfedges due to the number of fields which must all be updated simultaneously. Efficiently selecting a set of edges to collapse such that there isn't overlap within the set proves to be a difficult task. This makes the selection of edges to be collapsed in an

iteration potentially the most computationally expensive part in a parallel setting, as this task is rife with sequential dependencies.

Approach:

In order to parallelize quadric mesh simplification, we implemented an algorithm proposed by Lee and Kyung in CUDA. Our algorithm performs the following pipeline for mesh simplification starting with a 3D mesh stored in a .ply file.

1. We use a python script to convert the .ply file to a text file containing the halfedge data structure associated with the mesh. (This step is done manually)
2. Using this file we set up our mesh in C++, allocating an array for vertices, an array mapping the index of each vertex to a halfedge, and an array mapping the index of each triangle to a halfedge. We then load these data structures onto the GPU, at which point we begin the actual algorithm.
3. We launch a kernel which computes a plane equation for each triangle, with each thread pertaining to a triangle. We then launch another kernel where each thread traverses the faces adjacent to a vertex and uses the plane equations to compute the quadric error matrix for the given vertex.
4. We launch a kernel which picks an edge to collapse for each vertex. By the criteria we use to select edges, we can ensure that all edges form a set of trees which can each be collapsed into their root vertices. We launch an additional kernel for each vertex that verifies that the tree is acyclic, removing the edge of its tree if it creates a cycle.
5. To perform edge collapses, we iterate over all vertices, and perform a local collapse operation. For each of the 3 to 6 edges e removed in this operation, we store an edge

which can replace it in an array. We construct a similar array to represent vertex collapses.

6. Once we have constructed arrays which acyclicly represent how to replace removed halfedges, we iterate over all existing halfedges to update the indices of removed halfedges and vertices with their replacements. Because each thread only writes to its associated halfedge, we never need to worry about data races while updating these fields.
7. To improve the quality of our output mesh, we compute new positions for the remaining vertices. We launch a thread for each vertex. If the vertex is a root of a tree, we minimize its accumulated quadric error by solving a linear system using its error matrix. We solve this linear system using the conjugate gradient method. If our solution converges within a margin of error of the original position, we update the vertex position.
8. To minimize the amount of data communicated back to the host, we use a parallel scan to count the number of remaining halfedges and vertices, and make the halfedge and vertex arrays contiguous again. Because this step involves changing indices, we must once again update the index fields in all halfedges.
9. At this point, the host can copy the results back from the GPU, and write this halfedge mesh back to our txt format.
10. We then use another Python script to convert the text file into a .ply file which can be visually inspected.

Results:

Both the sequential and CUDA implementations were run on four meshes selected from the Stanford 3D Scanning Repository. To ensure both algorithms performed a comparable number of

edge collapses, the sequential implementation was configured to target the number of triangles the mesh generated by the CUDA implementation contained. The Open3D implementation was tested on an 3.9GHz Intel i7-8750H CPU with 16 GB RAM, while the CUDA implementation was tested on an NVIDIA GeForce RTX 2080 with 8GB of global memory.

Compared to the sequential implementation, our CUDA implementation performed between 400 to 1200 times faster, well above our goal of a ten times speedup. Note that as mesh complexity increases, our speedup also improves. This is likely due to the fact that our implementation primarily uses global memory due to the size of the mesh, while the CPU implementation can exploit some memory locality through the hardware cache. As the mesh size increases though, our working set becomes too large for the cache to contain, thus adding a similar memory access latency to the sequential implementation.

CUDA vs Open3D simplify_quadric_decimation			
	CUDA (ms)	Sequential Time (ms)	Speedup:
Bunny	1.045	416.000	398
Dragon	9.889	6,900.712	698
Dragon2	81.355	67,140.217	825
Statue	106.595	125,624.24	1179

Table 1 : Execution times and speedup of our CUDA implementation compared to the sequential open 3D implementation

In terms of mesh complexity, our CUDA implementation was able to reduce triangle counts in one iteration of parallel edge collapsing to about 40% of what they were initially. This is more than a factor of 2 reduction in the complexity of the provided mesh.

	Initial		Reduced		
	Verts	Triangles	Verts	Triangles	Triangle Reduction
Bunny	34834	69451	10054	27505	40%
Dragon	435545	871306	109595	333592	38%
Dragon2	3609600	7219045	889843	2729957	38%
Statue	4999996	10000000	1202707	3792661	38%

Table 2 : Mesh complexities before and after performing our algorithm

When comparing the execution times of each individual kernel our implementation performed, we found that a majority of time was spent in the prefix-sum counting and relabeling of all halfedges and vertices. This is reasonable considering due to the implementation of parallel scan we used, we could only perform this operation in one thread block, while most other steps in our pipeline are able to be performed across all thread blocks in parallel. Following the relabeling step, our next most time consuming kernel is the construction of embedded trees for us to collapse. This is also reasonable given that this step performs many non-contiguous accesses to global memory while traversing all edges adjacent to each vertex in the mesh.

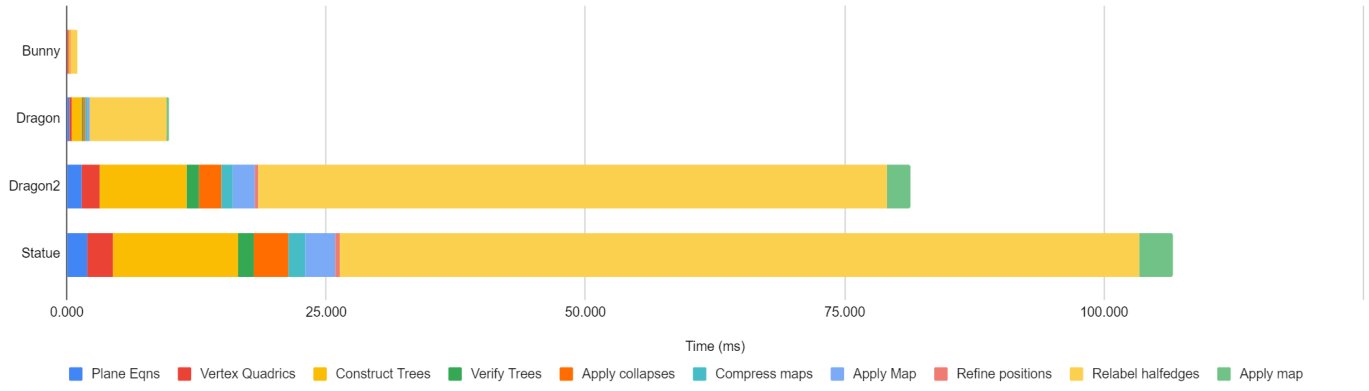

Figure 2: Time spent on each kernel for each mesh (measured in milliseconds)

In terms of quality, our implementation does produce reasonable output meshes, although there are some undesirable features in our output data compared to the sequential implementation. Specifically, the triangulations produced by our mesh are less even, and our generated meshes have some points of normal inversion, which appear as dark spots when rendered. These issues were expected however, as the Lee and Kyung paper acknowledges them, and proposes some additional filters to the collapsed edges to prevent illegal mesh operations such as triangle flipping from occurring.

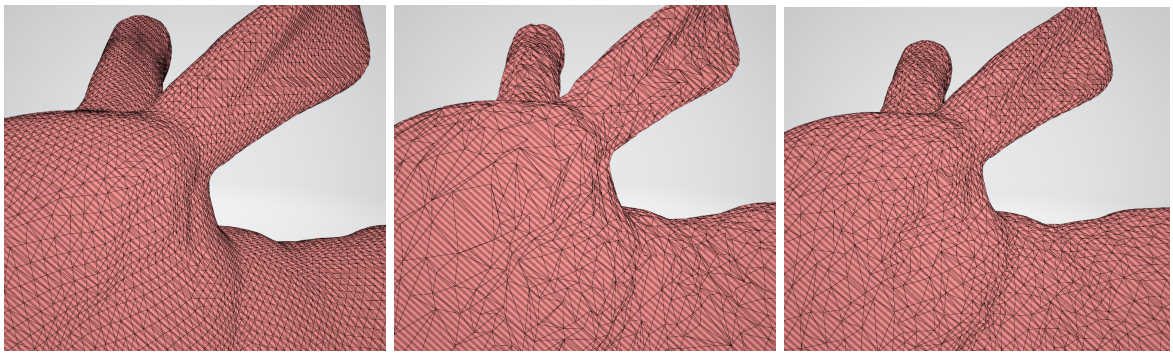

Figure 3: Original bunny mesh, our reduction, and the reference reduction (left to right)

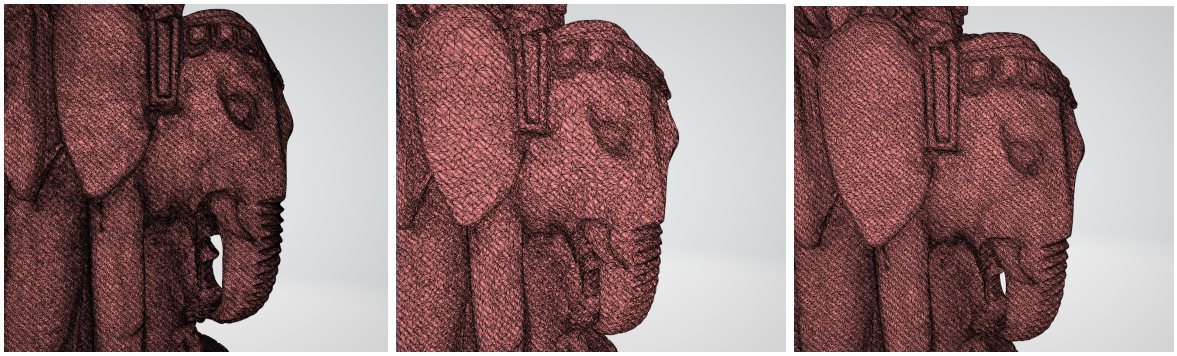

Figure 4: Original statue mesh, our reduction, and the reference reduction (left to right)

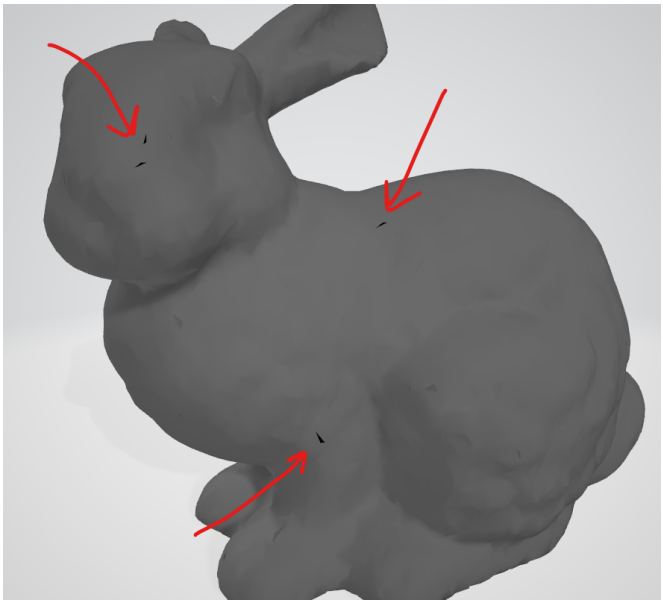

Figure 5: Points on the bunny mesh where triangle normals were inverted, causing incorrect-looking shading. Note that these occurrences are rare.

Credit Distribution:

Nico (50%)

- Implemented quadric error kernel.
- Implemented embedded tree construction and verification kernels along with helper function for calculating quadric error.
- Implemented vertex refinement.

George (50%)

- Implemented scripts for converting from a .ply to a text file with halfedges and vice versa.
- Implemented framework to load halfedge file and copy it onto the GPU.

- Implemented parallel tree collapsing and halfedge/vertex relabeling
- Implemented exclusive scan to remove deleted halfedges and vertices
- Wrote timing code and measured the time taken by the sequential and parallel implementations.

References:

Garland, Michael, and Paul S. Heckbert. “Surface Simplification Using Quadric Error Metrics.” *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques - SIGGRAPH '97*, 1997, https://doi.org/10.1145/258734.258849.

Lee, Hyunho, and Min-Ho Kyung. “Parallel Mesh Simplification Using Embedded Tree Collapsing.” *The Visual Computer*, vol. 32, no. 6-8, 2016, pp. 967–976., https://doi.org/10.1007/s00371-016-1242-z.