
CPU-GPU Algorithms for Triangular Surface Mesh Simplification

Suzanne M. Shontz¹ and Dragos M. Nistor²

¹ Department of Mathematics and Statistics, Mississippi State University,
Mississippi State, MS, U.S.A., sshontz@math.msstate.edu

² Department of Computer Science and Engineering, The Pennsylvania State
University, University Park, PA, U.S.A., nistor.mn@gmail.com

Summary. Mesh simplification and mesh compression are important processes in computer graphics and scientific computing, as such contexts allow for a mesh which takes up less memory than the original mesh. Current simplification and compression algorithms do not take advantage of both the central processing unit (CPU) and the graphics processing unit (GPU). We propose three simplification algorithms, one of which runs on the CPU and two of which run on the GPU. We combine these algorithms into two CPU-GPU algorithms for mesh simplification. Our CPU-GPU algorithms are the naïve marking algorithm and the inverse reduction algorithm. Experimental results show that when the algorithms take advantage of both the CPU and the GPU, there is a decrease in running time for simplification compared to performing all of the computation on the CPU. The marking algorithm provides higher simplification rates than the inverse reduction algorithm, whereas the inverse reduction algorithm has a lower running time than the marking algorithm.

Key words: mesh simplification, mesh compression, graphics processing unit (GPU), visualization

1 Introduction

Three-dimensional geometric models of varying detail are useful for solving problems in such areas as computer graphics [20], surface reconstruction [15], computer vision [18], and communication [4]. Such models give rise to the need for mesh simplification, mesh compression [14], and mesh optimization [16]. This paper proposes new algorithms for mesh simplification utilizing both the central processing unit (CPU) and the graphics processing unit (GPU) found on most modern computers.

Mesh simplification is the process of removing elements and vertices from a mesh to create a simpler model. Mesh simplification can be applied in surface reconstruction [3], three-dimensional scanning [17], computer animation [12],

and terrain rendering [10]. For example, when rendering a movie scene, a model with extremely high detail is not required for a far away object.

Several serial CPU-based mesh simplification algorithms have been proposed. Some algorithms use a simple edge-collapse operation [14], which involves repeatedly collapsing edges into vertices to obtain a simplified mesh. Others use a triangle-collapse operation [23]. While the triangle-collapse operation yields more simplified meshes per operation [6], there are more cases to handle when performing this operation. The algorithms we propose are based on the edge-collapse operation for its simplicity.

Other algorithms focus on controlled vertex, edge, or element decimation [24], where a vertex, edge, or element is removed from the mesh if it meets the decimation criteria. Any resulting holes in the mesh are patched through available methods such as triangulation. One other method for mesh simplification is vertex clustering [19]. When performing mesh simplification using vertex clustering, vertices are clustered by topological location, and a new vertex is created to represent each cluster. Elements can then be created through surface reconstruction [15]. Our algorithms focus on neither decimation nor clustering, as neither of these processes are designed to take advantage of the available GPU concurrency.

A few parallel CPU-based algorithms based on the corresponding serial algorithms have been proposed. One such parallel algorithm is based on vertex decimation [11], where the importance of each vertex is evaluated, and vertices with low importance are removed. A GPU-based implementation of this algorithm has also been proposed [13] and is discussed later in this section. However, we are interested in designing an algorithm that takes advantage of both the CPU and GPU by splitting the simplification workload between them. Thus, we avoid operations which involve a significant amount of communication between the CPU and GPU, as the transfer rate between the CPU and GPU is a significant bottleneck [25].

Other simplification algorithms focus on distributed systems [5] and efficient communication between nodes. Such algorithms would suffer from the same communication latency between the CPU and the GPU if implemented to take advantage of the GPU. Another algorithm [9] focuses on greedily splitting a mesh into equal submeshes and assigning each part to a CPU core to be simplified by applying the edge-collapse operation. These techniques could be used to implement a GPU-based algorithm. However, there is no need to create submeshes for such an algorithm, as the GPU can simultaneously support many more threads than can a multi-core CPU. Instead, each GPU thread can focus on one element.

Some GPU-based simplification algorithms have also been proposed. One such algorithm offloads the computationally-intensive parts of vertex decimation to the GPU [13], while leaving the data structure representing the mesh in main memory. While this approach is valid, it assumes that the CPU will be available during the whole process. Another popular method [8] is based on vertex clustering. A downside of this algorithm is that it assumes that the

surface mesh is closed and that there is access to the entire mesh during the simplification process. This excludes streaming input models.

We propose three simplification algorithms, one of which runs on the CPU and two of which run on the GPU. We combine these algorithms into two CPU-GPU algorithms for mesh simplification. The algorithms are based on the edge-collapse operation, as it is an extremely simple and small-scale operation, and it works even if there is no access to the entire mesh. The CPU algorithm visits every available element and performs the edge-collapse operation on the element if it is not yet marked as affected (defined in Section 2), as does one of the GPU algorithms. The other GPU algorithm takes full advantage of the concurrency of the GPU and attempts to collapse a greater number of edges each iteration.

In Section 2, we present our three mesh simplification algorithms: the CPU simplification algorithm, a naïve GPU simplification algorithm, and a GPU simplification algorithm based on reductions, which we combined into two CPU-GPU algorithms. We discuss the correctness of the algorithms, as well. In Section 3, we describe our experiments to test our CPU-GPU algorithms for various CPU-GPU workload splits and then discuss our results. In Section 4, we conclude our work and discuss several possibilities for future work.

2 Simplification Algorithms

We propose three CPU- and GPU-based algorithms which work in tandem to simplify a mesh. We combine our algorithms to produce two CPU-GPU mesh simplification algorithms. The algorithms simplify meshes uniformly and exhaustively, ensuring maximal simplification occurs. All proposed algorithms rely on the edge-collapse operation, which is defined below. Additionally, the algorithms are lossless, which means that the operations can be applied in reverse, and the original mesh can be recovered.

Our simplification algorithms rely on the edge-collapse operation [14], which is defined as follows for an input mesh containing a set of vertices V and a set of elements T .

For an edge $e = (v_1, v_2)$ shared by elements $T_1 = (v_1, v_2, v_3)$ and $T_2 = (v_4, v_2, v_1)$, define v_m to be the midpoint of e . To collapse edge e , T_1 and T_2 are removed from the mesh, and any references to v_1 or v_2 are updated to refer to v_m . Figure 1 shows an edge-collapse operation on edge (v_1, v_2) .

For the edge-collapse operation, the order of the vertices that make up an element does not matter. If additional information, such as the original positions of v_1 and v_2 , is stored, the edge-collapse operation is reversible. Therefore, compression or simplification algorithms based on this operation are lossless, and the original mesh can be recovered by reversing these steps.

Our simplification algorithms allocate a portion of the mesh to the CPU and the rest of it to the GPU to be simplified. We propose an extremely simple method for allocation. For a CPU-GPU split $k\%$ where $0 \leq k \leq 100$

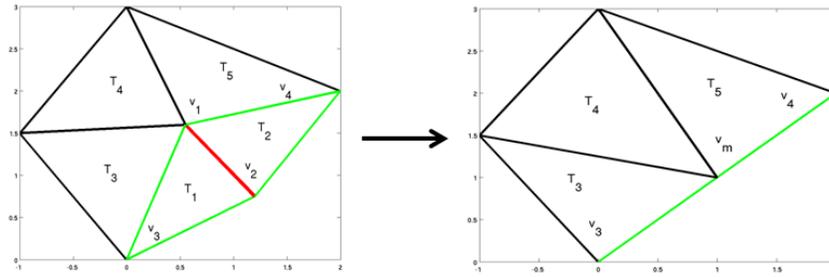


Fig. 1. An edge-collapse on $e = (v_1, v_2)$.

of a mesh $M = (\text{vertices}, \text{elements})$, the CPU simplifies the first $k\%$ of the elements, and the GPU simplifies the remaining elements. The workload splitting is performed once as a preprocessing step. Locks are used to ensure that conflicting updates are not made in the simplification process.

Since edge-collapse operations should be performed uniformly on the mesh, we mark the elements which have been affected by previous edge collapses so that they do not take part in subsequent edge-collapse operations.

Define $N(T)$ for $T = (v_a, v_b, v_c)$ to be elements which contain any of the vertices v_a, v_b , or v_c . If edge $e = (v_1, v_2)$ between elements $T_1 = (v_1, v_2, v_3)$ and $T_2 = (v_4, v_2, v_1)$, for example, has been collapsed, then the elements in $N(T_1) \cup N(T_2)$ are considered affected. The elements shaded in gray in Figure 2 would be considered affected if edge (v_1, v_2) were collapsed.

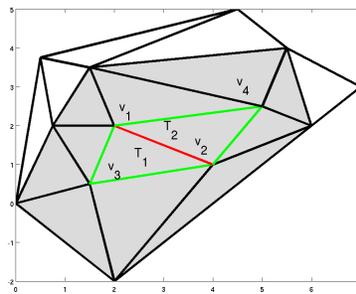


Fig. 2. Elements considered affected by edge-collapse of (v_1, v_2) .

Since each edge-collapse operation causes neighboring elements to become affected, there is a hard limit on the number of edge-collapse operations, and on the amount of simplification performed per iteration of the algorithm. Additional simplification may be obtained by performing additional iterations of the algorithms.

2.1 CPU Edge-Collapse Algorithm

To simplify portions of the mesh using the CPU, we propose a simple edge-collapse algorithm. The CPU edge-collapse algorithm iterates over all CPU-assigned elements. Each element is examined to see if it is affected, and if an element $T = (v_1, v_2, v_3)$ that is not affected is found, an edge-collapse is performed on (v_1, v_2) . When all elements are affected or have taken part in an edge-collapse operation, the algorithm terminates. This ensures that the edge-collapse operation is performed uniformly and exhaustively across the elements assigned to the CPU, and that no one area is more or less simplified or deformed. Pseudocode for the CPU edge-collapse simplification algorithm is given in Algorithm 1.

Algorithm 1 The CPU Edge-Collapse Simplification Algorithm

```

function MARK-AS-AFFECTED(element)
  for all  $v \in$  element do
    affected[ $v$ ]  $\leftarrow$  true
  end for
end function

function MARK-AS-COLLAPSED( $(v_1, v_2)$ )
  for all  $T \in$  elements  $\supset \{v_1, v_2\}$  do
    collapsed[ $T$ ]  $\leftarrow$  true
  end for
end function

function CPU-SIMPLIFY(elements, vertices)
  for all  $T = (v_1, v_2, v_3) \in$  elements do
    if  $T \in N(\text{affected elements})$  then
      mark-as-affected( $T$ )
    else
      collapse( $(v_1, v_2)$ )
      mark-as-collapsed( $(v_1, v_2)$ )
    end if
  end for
end function

```

2.2 GPU Marking Algorithm

We propose a naïve GPU algorithm to simplify portions of the mesh based on the edge-collapse operation. The naïve GPU marking simplification algorithm works by searching through all elements assigned to the GPU one at a time. If an element $T = (v_1, v_2, v_3)$ that is not affected is found, an edge-collapse is performed on (v_1, v_2) , and all $T_n \in N(T)$ are concurrently marked as affected. When all elements are affected or have taken part in an edge-collapse

operation, the algorithm terminates. This also ensures that the edge-collapse operation is performed uniformly and exhaustively across all elements assigned to the GPU, and that no one area is more or less simplified or deformed. The pseudocode for the naïve GPU marking simplification algorithm is given in Algorithm 2.

Algorithm 2 The GPU Marking Simplification Algorithm

```

function GPU-MARK(mark, elements, vertices)
  for all  $T \in N(\text{mark})$  do GPU thread  $T$  : mark-as-affected( $T$ )
  end for
end function

function GPU-MARK-SIMPLIFY(elements, vertices)
  for all  $T = (v_1, v_2, v_3) \in \text{elements}$  do
    if not marked( $T$ ) then
      collapse( $(v_1, v_2)$ )
      mark-as-collapsed( $(v_1, v_2)$ )
      GPU-Mark( $T$ )
    end if
  end for
end function

```

2.3 GPU Inverse Reduction Algorithm

We propose a GPU mesh simplification algorithm that leverages the full strength of the GPU. Unlike our previous algorithms in which one element was examined at each iteration of the main loop to determine which elements were not affected, we now examine twice as many elements per iteration, with each element examined by a different GPU thread. To take full advantage of the architecture of the GPU, a soft-grained blocking [21] method based on test-and-set [1] is used to decide if any edge of an element should be collapsed. We attempt to lock each vertex in an element by calling test-and-set on the affected bit of each vertex in the element. Pseudocode for the GPU inverse reduction simplification algorithm is given in Algorithm 3.

Correctness. The GPU-Simp-Try method attempts to lock each vertex of an element by checking to make sure the vertex has not already been collapsed. If it finds that a vertex has already been locked, it releases all previously-locked vertices. Therefore, if a thread successfully locks v_1, v_2 , and v_3 for element t , this means that no other thread has locked any $t_n \in N(t)$, either currently or previously. Therefore, the algorithm simplifies the mesh both uniformly and exhaustively, ensuring that no one area is too simplified or deformed.

Algorithm 3 The GPU Inverse Reduction Simplification Algorithm

```

function GPU-SIMP-TRY(target =  $(v_0, v_1, v_2)$ , elements, vertices)
  if affected(target) then
    return
  end if
  if test-and-set(collapsed[ $v_0$ ]) = 0 then
    if test-and-set(collapsed[ $v_1$ ]) = 0 then
      if test-and-set(collapsed[ $v_2$ ]) = 0 then
        collapse( $(v_0, v_1)$ )
        GPU-Mark(target)
      end if
      collapsed[ $v_1$ ]  $\leftarrow$  0
      collapsed[ $v_0$ ]  $\leftarrow$  0
    end if
    collapsed[ $v_0$ ]  $\leftarrow$  0
  end if
end function

function GPU-IR-SIMPLIFY(elements, vertices)
   $i = |\text{elements}|$ 
  while  $i \geq 1$  do
    if threadid mod  $i = 0$  then
      GPU-Simp-Try(elements[threadid])
    end if
     $i = i \text{ div } 2$ 
  end while
end function

```

3 Experiments

In this section, we describe the experiments which we designed to test the performance of our mesh simplification algorithms. We implemented our mesh simplification algorithms in C++ and compiled them using the NVIDIA C++ compiler included with the CUDA Toolkit [7]. We tested our algorithms on the following six triangular surface meshes from computer graphics: armadillo [26], bunny [26], gargoyle [2], hand [22], horse [22], and kitten [2], which are shown in Figure 3. The computer used for our experiments was a Dell XPS 17 laptop running Windows 7 Professional equipped with an NVIDIA GeForce GT 550M GPU and an Intel Core i5-2430M CPU running at 2.4 GHz with 3.90 GB of usable main memory.

The goals of our experiments were to determine how much simplification the meshes could withstand, how much time it takes to simplify the meshes for various CPU-GPU splits, and how much memory is required for various CPU-GPU splits to simplify the meshes using each algorithm.

For each mesh produced by our mesh simplification algorithms, we determine the numbers of vertices and elements, the minimum and maximum

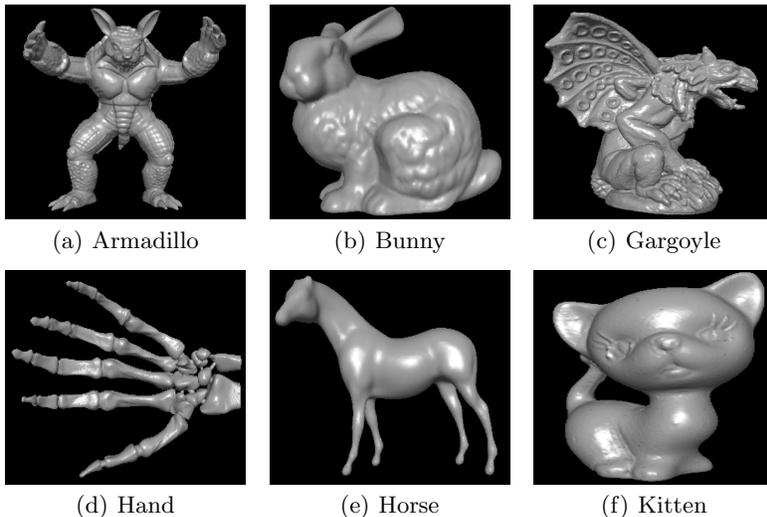


Fig. 3. Initial test meshes.

element angles (in degrees), the minimum and average element areas, the mesh volume, and the wall clock time (measured in seconds) for each algorithm. Using these metrics, we were able to assess the quality of the meshes and the algorithmic efficiency. For the wall clock time, we report averages over 100 runs of the algorithm. Table 1 contains the values of the metrics for the initial meshes.

mesh	# vertices	# faces	min \angle	max \angle	min area	avg area	volume
armadillo	172974	345944	0.034750	170.907	7.424e-08	1.840e-1	1.42690e+6
bunny	34834	69664	0.494800	177.515	7.925e-08	1.573e-2	7.54700e+3
gargoyle	863182	1726364	0.000215	179.820	3.638e-12	4.553e-2	1.63730e+6
hand	327323	654666	0.545000	177.995	5.161e-11	1.078e-4	1.64936e+1
horse	15366	30728	0.385500	177.119	1.118e-14	2.118e-6	1.58866e-3
kitten	137098	274196	0.004609	179.936	1.427e-08	8.846e-2	8.38625e+5

Table 1. Metric values for the initial meshes.

To examine the effects of splitting the mesh simplification workload between the CPU and the GPU using the naïve marking algorithm, we compared the time spent during the simplification process for the following CPU-GPU workload splits 100-0, 95-5, 90-10, \dots , and 0-100 on each test case. The time taken in seconds for the tested CPU-GPU splits for the bunny and gargoyle meshes can be seen in Figure 4. Such results are representative of those obtained on small and large meshes, respectively. The percentage of running time spent in the GPU is shown in Table 2; the 100-0 split is not shown since

all of the time is spent on the CPU. The GPU memory usage for the tested splits is shown in Table 3.

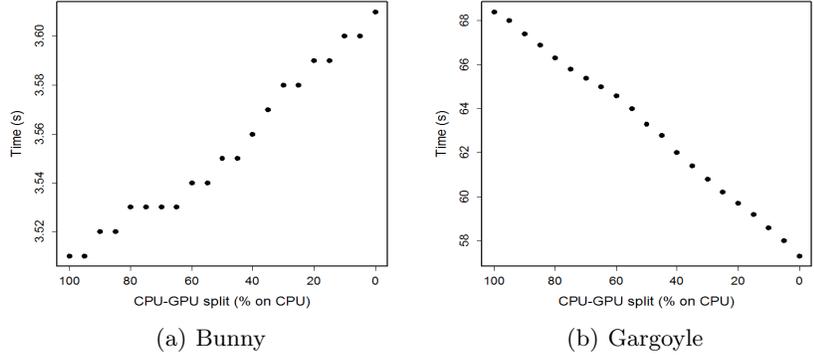


Fig. 4. The time taken to simplify the bunny and gargoyle meshes for each CPU-GPU split using the CPU-GPU naïve marking algorithm. As the CPU-GPU split increases, the CPU workload increases, and the GPU workload decreases.

% of GPU time										
mesh	95-5	90-10	85-15	80-20	75-25	70-30	65-35	60-40	55-45	50-50
armadillo	9.2	14.5	19.6	24.6	29.4	34.1	38.6	43.1	47.5	51.7
bunny	5.4	8.8	12.1	15.2	18.2	21.1	23.9	26.5	29.0	31.4
gargoyle	10.4	15.8	21.1	26.3	31.3	36.1	40.8	45.4	49.9	54.3
hand	9.4	14.8	20.0	25.1	29.9	34.7	39.3	43.9	48.4	52.7
horse	5.0	8.0	10.9	13.6	16.2	18.7	21.1	23.3	25.4	27.4
kitten	8.2	13.4	18.4	23.2	27.9	32.5	36.8	41.2	45.5	49.6
mesh	45-55	40-60	35-65	30-70	25-75	20-80	15-85	10-90	5-95	0-100
armadillo	55.8	59.8	63.6	67.4	70.9	74.3	77.5	80.6	83.4	86.0
bunny	33.6	35.7	37.7	39.5	41.3	42.9	44.4	45.8	47.2	48.5
gargoyle	58.6	62.7	66.7	70.5	74.2	77.7	81.0	84.2	87.2	89.1
hand	56.9	61.0	64.8	68.7	72.3	75.8	79.0	82.2	84.2	87.8
horse	29.2	30.9	32.5	33.9	35.3	36.5	37.6	38.6	39.4	40.1
kitten	53.6	57.4	61.1	64.8	68.2	71.4	74.5	77.4	80.1	82.5

Table 2. The percentage of time spent on the GPU for various CPU-GPU splits using the CPU-GPU naïve marking algorithm.

As seen in Figure 4, the gargoyle mesh shows an increase in running time, whereas the bunny mesh exhibits a decrease in running time, with respect to increasing the workload of the GPU. This indicates that if the mesh is

Amount of memory used on the GPU (KB)										
mesh	95-5	90-10	85-15	80-20	75-25	70-30	65-35	60-40	55-45	50-50
armadillo	2568	2770	2973	3176	3378	3581	3784	3986	4189	4392
bunny	517	558	599	640	680	721	762	803	844	884
gargoyle	12813	13824	14836	15847	16859	17871	18882	19894	20905	21917
hand	4859	5242	5626	6009	6393	6777	7160	7544	7927	8311
horse	228	246	264	282	300	318	336	354	372	390
kitten	2035	2196	2356	2517	2678	2838	2999	3160	3320	3481
mesh	45-55	40-60	35-65	30-70	25-75	20-80	15-85	10-90	5-95	0-100
armadillo	4595	4797	5000	5203	5405	5608	5811	6014	6216	6419
bunny	925	966	1007	1048	1089	1129	1170	1211	1252	1293
gargoyle	22928	23940	24951	25963	26974	27986	28998	30009	31021	32032
hand	8695	9078	9462	9845	10229	10613	10996	11380	11763	12147
horse	408	426	444	462	480	498	516	534	552	570
kitten	3642	3802	3963	4124	4284	4445	4606	4766	4927	5088

Table 3. The amount of memory used, in KB, by the GPU during simplification for each CPU-GPU split for both the CPU-GPU naïve algorithm and the CPU-GPU inverse reduction algorithm.

large, the running time decreases as the GPU workload increases, whereas if the mesh is small, the reverse is true. This can be attributed to the extra time taken to allocate memory in the GPU and to copy the data from main memory to the GPU cache in addition to the time required for simplification.

We obtain the following metrics after 10 iterations of the algorithm with a CPU-GPU split of 0-100: numbers of vertices and faces, minimum and maximum element ages, minimum and average element areas, mesh volume, and the percentages of vertex and face simplification. The values of the metrics can be seen in Table 4.

Simplification using the CPU-GPU naïve marking algorithm does not affect the mesh volume significantly. It does, however, increase the average area of each element, which is to be expected. Additionally, the simplification rate is approximately 14% to 15% for one iteration of the algorithm.

mesh	# vertices	# faces	min \angle	max \angle	min area	avg area	volume
armadillo	56469	111617	3.091e-2	179.86	3.901e-07	6.077e-1	1.42548e+6
bunny	9979	20129	1.477e-1	179.645	6.242e-06	5.759e-2	7.56014e+3
gargoyle	278305	629188	3.688e-4	179.981	8.967e-08	1.143e-1	1.64053e+6
hand	99635	213061	1.969e-2	179.912	2.674e-09	3.937e-4	1.65117e+1
horse	5428	10927	1.874e-2	179.339	1.032e-11	7.162e-6	1.58650e-3
kitten	39991	87997	1.530e-2	179.799	2.320e-06	5.259e-1	8.38547e+5

Table 4. Metric values for the meshes after 10 iterations of the naïve CPU-GPU marking algorithm.

We also consider the simplification rate after performing multiple iterations of the naïve CPU-GPU marking algorithm on the test meshes. The simplification rates for 10 iterations of the algorithm run on the bunny and gargoyle meshes can be see in Tables 5 and 6, respectively.

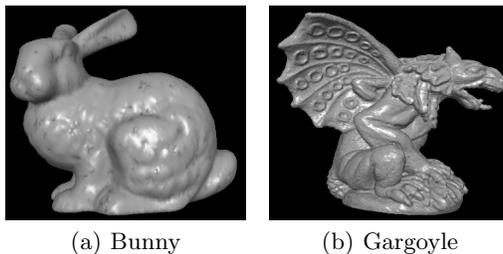


Fig. 5. The resulting meshes after three iterations of the CPU-GPU naïve marking algorithm.

iter	Simplification Percentage Naïve Marking Algorithm				Simplification Percentage Inverse Reduction Algorithm			
	# vertices	# faces	%v simp	%f simp	# vertices	# faces	%v simp	%f simp
0	34834	69664	—	—	34834	69664	—	—
1	30102	60194	13.6	13.6	31442	62278	10.6	10.6
2	26103	52183	25.1	25.1	27980	55954	19.7	19.7
3	22772	45487	34.6	34.7	25216	50383	27.6	27.7
4	19976	39846	42.7	42.8	22809	45567	34.5	34.6
5	17603	35026	49.5	49.7	20657	41261	40.7	40.8
6	15588	30942	55.3	55.6	18762	37463	46.1	46.2
7	13870	27413	60.2	60.6	17069	34064	51.0	51.1
8	12396	24391	64.4	65.0	15548	31013	55.4	55.5
9	11093	21703	68.2	68.8	14230	28349	59.1	59.3
10	9979	20129	71.4	71.1	13043	25936	62.6	62.8

Table 5. The simplification percentages of the bunny mesh over multiple iterations of the CPU-GPU naïve marking algorithm and the CPU-GPU inverse reduction algorithm.

The simplification rate hovers around 64% to 71% for both vertices and faces after 10 iterations. After three iterations, we can see that there are visible areas where simplification occurred on the bunny mesh as shown in Figure 5. This is not the case on the larger meshes. After 10 iterations, the bunny, horse, and kitten meshes as shown in Figure 6 exhibit an extreme loss of detail. The armadillo lost some detail in the head area, and the other meshes do not show

iter	Simplification Percentage Naïve Marking Algorithm				Simplification Percentage Inverse Reduction Algorithm			
	# vertices	# faces	%v simp	%f simp	# vertices	# faces	%v simp	%f simp
0	863182	1726364	—	—	863182	1726364	—	—
1	735345	1470688	14.8	14.8	779354	1558590	9.7	9.7
2	646764	1292036	25.1	25.2	704379	1408640	18.4	18.4
3	573431	1142601	33.6	33.8	639370	1276589	25.9	26.1
4	512166	1042650	40.7	39.6	581774	1161369	32.6	32.7
5	458718	952966	46.9	44.8	531048	1059586	38.5	38.6
6	412135	872794	52.3	49.4	486874	970716	43.6	43.8
7	371523	801356	57.0	53.6	446693	889413	48.3	48.5
8	336120	737405	61.1	57.3	410960	816781	52.4	52.7
9	305243	680268	64.6	60.6	379788	753007	56.0	56.4
10	278305	629188	67.8	63.6	351150	693992	59.3	59.8

Table 6. The simplification percentage of the gargoyle mesh over multiple iterations of the CPU-GPU naïve marking algorithm and the CPU-GPU inverse reduction algorithm.

too much loss of detail. This reinforces the notion that larger meshes can be simplified more without seeing any visible effects.

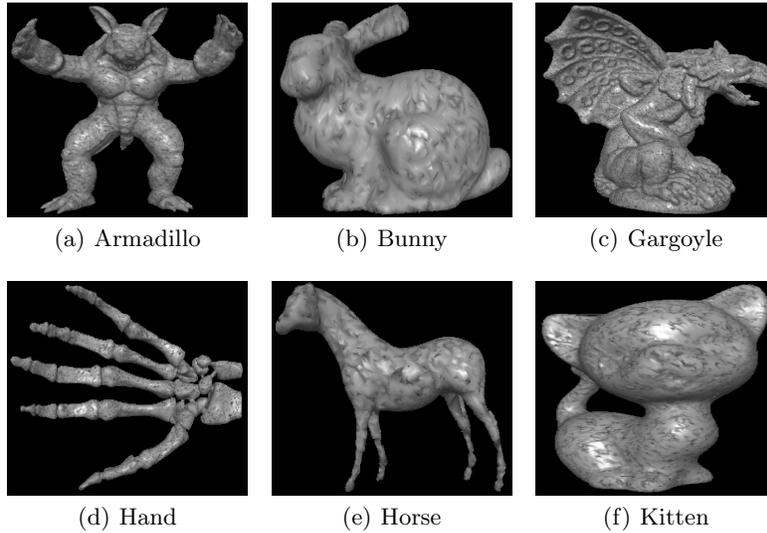


Fig. 6. The resulting meshes after 10 iterations of the CPU-GPU naïve marking algorithm.

Tables 5 and 6 show the vertex and face simplification percentages as a function of iteration in each mesh. Note that %v simp and %f simp represent

the percentages of vertex and face simplification, respectively, in these tables. The amount of simplification performed each iteration decreases. This suggests that as the vertex and element counts increase, the decrease in running time achieved by using the GPU simplification algorithm instead of the CPU simplification algorithm increases as well.

To examine the effects of splitting the mesh simplification workload between the CPU and the GPU using the inverse reduction algorithm, we recorded the time spent during the simplification process for various CPU-GPU splits on all test cases. We tested the following CPU-GPU workload splits: 100-0, 95-5, 90-10, \dots , and 0-100. The time taken in seconds for the CPU-GPU splits can be seen in Figure 7. The proportion of running time spent in the GPU for the tested is shown in Table 7. The GPU memory usage for the tested splits is shown in Table 3.

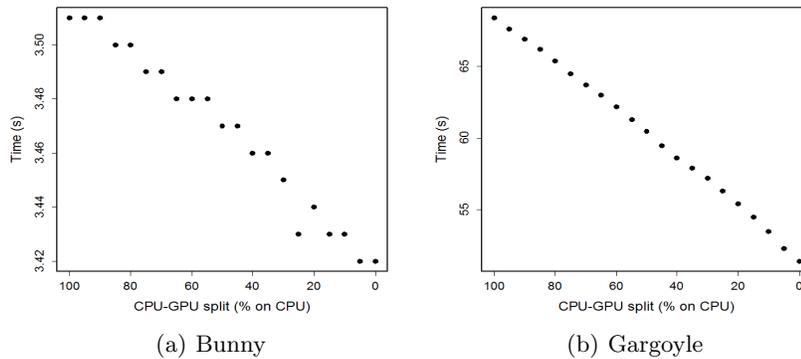


Fig. 7. The time taken to simplify each test case for every split using the CPU-GPU inverse reduction algorithm. As the CPU-GPU split increases, the CPU workload increases, and the GPU workload decreases.

The CPU-GPU inverse reduction algorithm shows an increase in running time on the horse mesh, whereas it shows a decrease in running time on the armadillo, bunny, gargoyle, hand, and kitten meshes with respect to an increasing workload on the GPU. If the mesh is large, the time taken decreases as the GPU workload increases. If the mesh is small, the reverse trend holds. This is due to the extra time taken for memory allocation in the GPU and for copying the data from main memory to the GPU cache, shown in Table 3, in addition to the time required for simplification. Also, because the inverse reduction algorithm is more efficient than the naïve marking algorithm, we see that the time taken to simplify the bunny mesh decreases as the GPU workload increases, now decreasing anywhere between 5.55% for the bunny mesh to 11.48% for the gargoyle mesh when the GPU is given the full workload when compared to the naïve algorithm.

% GPU time										
mesh	95-5	90-10	85-15	80-20	75-25	70-30	65-35	60-40	55-45	50-50
armadillo	8.2	13.3	18.2	22.8	27.4	31.9	36.2	40.5	44.7	48.7
bunny	5.0	8.3	11.5	14.4	17.3	20.1	22.7	25.5	27.6	29.8
gargoyle	9.2	14.4	19.5	24.4	29.2	33.8	38.3	42.6	46.8	51.0
hand	8.3	13.5	18.4	23.1	27.8	32.5	36.9	41.3	45.6	49.7
horse	4.7	7.7	10.5	13.1	15.7	18.1	20.4	22.5	24.6	26.5
kitten	7.4	12.4	17.2	21.8	26.3	30.7	34.8	38.9	43.0	46.8
mesh	45-55	40-60	35-65	30-70	25-75	20-80	15-85	10-90	5-95	0-100
armadillo	52.6	56.4	59.8	63.4	66.7	69.7	72.4	75.0	77.2	79.0
bunny	31.9	33.9	35.7	37.4	39.1	40.5	41.9	43.2	44.5	45.7
gargoyle	55.1	58.9	62.7	66.2	69.7	73.0	76.0	79.0	81.7	83.4
hand	53.7	57.6	61.1	64.9	68.3	71.4	74.2	76.9	79.2	81.1
horse	28.2	28.8	31.4	32.6	33.9	34.9	35.9	36.8	37.5	38.1
kitten	50.7	54.3	57.8	61.3	64.5	67.5	70.3	73.0	75.6	77.1

Table 7. The percentage of time spent using the GPU for various CPU-GPU splits using the inverse reduction algorithm.

The following metrics were collected after 10 iterations of the algorithm with a CPU-GPU split of 0-100: vertex and face counts, minimum and maximum element angles (in degrees), minimum and average element area, mesh volume, and vertex and face simplification percentages. The values for these metrics for the tenth iteration can be seen in Table 8.

mesh	# vertices	# faces	min \angle	max \angle	min area	avg area	volume
armadillo	65091	129131	2.271e-2	179.949	1.656e-07	5.190e-1	1.42659e+6
bunny	13043	25936	5.214e-2	179.805	1.000e-07	4.455e-2	7.55771e+3
gargoyle	351150	693992	3.044e-3	179.991	3.046e-08	1.211e-1	1.64145e+6
hand	132063	259374	1.449e-2	179.920	8.594e-10	2.940e-4	1.64892e+1
horse	6623	12719	1.251e-2	179.520	6.054e-11	5.698e-6	1.58656e-3
kitten	53725	106863	1.076e-2	179.912	1.476e-06	2.404e-1	8.38289e+5

Table 8. Values of the metrics for the meshes after 10 iterations of the CPU-GPU inverse reduction algorithm.

Simplification using the inverse reduction algorithm does not affect the volume of the test cases significantly. It does, however, increase the average area of each element, which is to be expected. Additionally, the simplification rate is approximately 9% to 10% after one iteration of the algorithm, which is much lower than that of the naïve algorithm. This demonstrates the tradeoff between speed and rate of compression.

We also consider the simplification rate after performing 10 iterations of the CPU-GPU inverse reduction algorithm on the test meshes. The simplifi-

cation rates for performing 10 iterations of the algorithm on the bunny and gargoyle meshes can be seen in Tables 5 and 6, respectively.

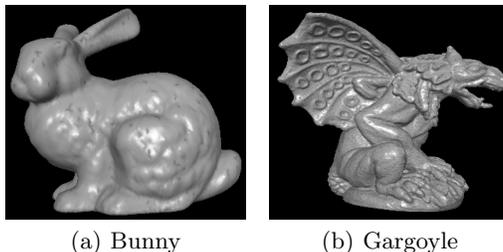


Fig. 8. The resulting meshes after three iterations of the CPU-GPU inverse reduction algorithm.

The simplification rate for both vertices and faces after 10 iterations of the algorithm hovers around 57% to 63%. After three iterations, there are visible areas where repeated simplification occurred on the bunny mesh as shown in Figure 8, but this is not the case on the larger meshes. After 10 iterations, the bunny, horse, and kitten meshes exhibit an extreme loss of detail, as shown in Figure 9. The armadillo lost some detail in the head; the other meshes do not show too much loss of detail. This suggests that the larger a mesh is initially, the more that it can be simplified without any visible negative effects.

Table 7 shows the simplification percentages as a function of the number of iterations for the vertices and faces in each mesh. The simplification percentages increase at a decreasing rate. This suggests that as a mesh increases in size, the decrease in running time achieved by using the GPU simplification algorithm instead of the CPU simplification algorithm increases as well.

4 Conclusions and Future Work

In this paper, we have proposed two lossless CPU-GPU surface mesh simplification algorithms. Our algorithms are based on the GPU marking algorithm, which uses multiple GPU threads to concurrently mark elements as affected, and the GPU inverse reduction algorithm, which attempts to perform the edge-collapse operation on twice as many edges at each step compared to GPU marking algorithm. Both algorithms were tested on numerous triangular surface meshes from computer graphics. The combinations of the CPU algorithm with the two GPU algorithms are novel. In particular, the CPU-GPU algorithms leverage the concurrency of the GPU to accelerate the simplification of our surface meshes. The algorithms can be used in various areas of computer graphics, such as video games and computer imaging, where there is a clear benefit in creating a series of meshes from a single source mesh or a simplified

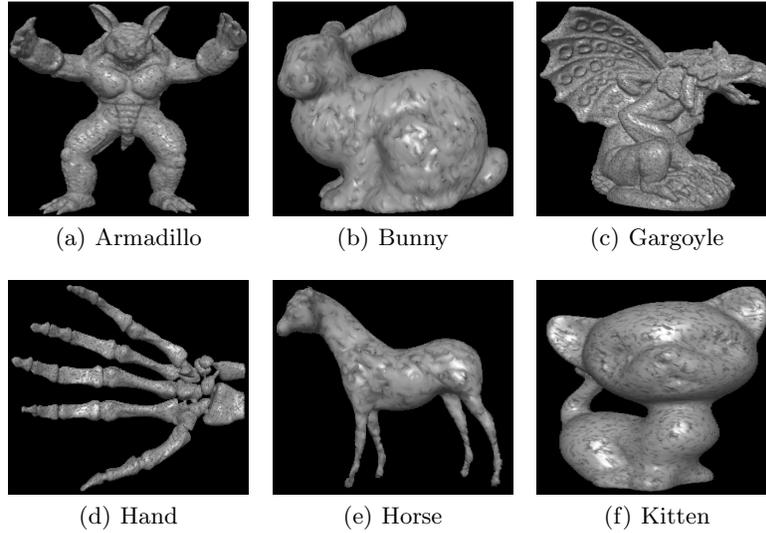


Fig. 9. The resulting meshes after 10 iterations of the CPU-GPU inverse reduction algorithm.

version of an original mesh. They may also be useful for real-time visualization of triangular surface meshes used in scientific computing applications.

The simplification rate and running time of the CPU-GPU algorithms depended on multiple factors, including the GPU algorithm selected. Both CPU-GPU algorithms used the same amount of GPU memory for any specific CPU-GPU split, as they both needed to keep track of the same numbers of vertices and elements on the GPU. As the GPU workload was increased, the amount of GPU memory used increased as well. For both algorithms, as the size of a mesh increased, the decrease in simplification time increased, as the GPU workload was increased. The simplification rate of approximately 68% that the CPU-GPU marking algorithm achieved was higher than the simplification rate of approximately 59% that was achieved by the inverse reduction algorithm, over ten iterations. This slight increase in simplification rate provided by the marking algorithm was counterbalanced by an increase of 5.55% to 11.48% in the running time when the GPU was given the full workload. Further simplification is possible if the simplification algorithm were run for a larger number of iterations. This suggests that the GPU marking algorithm should be used when a larger simplification rate per iteration is needed and running time is not a limiting factor. However, the GPU inverse reduction algorithm should be used when time is a limiting factor or when a lesser simplification rate per iteration is required, such as for smaller meshes.

Our results show that certain areas of the surface mesh were repeatedly simplified over multiple iterations, causing the meshes to look excessively simplified in these areas, since element ordering determines the simplification

order. Reordering the mesh elements as a preprocessing step would likely minimize the repeated simplification of mesh elements in a given area. Mesh optimization may also be useful for improving the quality of the surface mesh elements.

It would also be interesting to implement a hybrid mesh simplification algorithm that takes advantage of the speed of the GPU inverse reduction algorithm and the simplification rate of the GPU marking algorithm. This could potentially increase the simplification rate as well as decrease the running time of the simplification algorithm. Further algorithmic speed may be accomplished through the utilization of additional CPU cores or additional GPUs, which could take full advantage of parallelism. This may prove to be especially useful for real-time graphics and scientific visualization applications.

5 Acknowledgements

The work of the first author is supported in part by NSF grants CNS-0720749 and NSF CAREER Award OCI-1054459.

References

1. Y. Afek, E. Gafni, J. Tromp, and P. Vitany. Wait-free test-and-set. In Adrian Segall and Shmuel Zaks, editors, *Distributed Algorithms*, volume 647 of *Lecture Notes in Computer Science*, pages 85–94. Springer Berlin / Heidelberg, 1992.
2. AIM@SHAPE. Aim@shape shape repository 4.0, February 2012. <http://shapes.aimatshape.net/>.
3. M.-E. Algorri and F. Schmitt. Mesh simplification. *Computer Graphics Forum*, 15(3):77–86, 1996.
4. C.L. Bajaj, V. Pascucci, and G. Zhuang. Progressive compression and transmission of arbitrary triangular meshes. In *Proc. Visualization '99*, Visualization '99, pages 307–537, 1999.
5. D. Brodsky and J.B. Pedersen. A parallel framework for simplification of massive meshes. In *Proc. of the 2003 IEEE Symposium on Parallel and Large-Data Visualization and Graphics*, PVG '03, pages 17–24, Washington, DC, USA, 2003. IEEE Computer Society.
6. P. Cignoni, C. Montani, and R. Scopigno. A comparison of mesh simplification algorithms. *Computers & Graphics*, 22:37–54, 1997.
7. NVIDIA Corporation. CUDA Toolkit, January 2012. <http://developer.nvidia.com/cuda-toolkit-41>.
8. C. DeCoro and N. Tatarchuk. Real-time mesh simplification using the GPU. In *Proc. of the 2007 Symposium on Interactive 3D Graphics and Games*, I3D '07, pages 161–166, New York, NY, USA, 2007. ACM.
9. F. Dehne, C. Langis, and G. Roth. Mesh simplification in parallel. In *ICA3PP '00*, pages 281–290, 2000.
10. C. Dick, J. Schneider, and R. Westermann. Efficient geometry compression for GPU-based decoding in realtime terrain rendering. *Computer Graphics Forum*, 28(1):67–83, 2009.

11. M. Franc and V. Skala. Parallel triangular mesh decimation without sorting. In *Proc. of the 17th Spring Conference on Computer Graphics, SCCG '01*, pages 22–, Washington, DC, USA, 2001. IEEE Computer Society.
12. P. Heckbert and M. Garl. Multiresolution modeling for fast rendering. In *Proceedings of Graphics Interface*, pages 43–50, 1994.
13. J. Hjelmervik and J.-C. Leon. GPU-accelerated shape simplification for mechanical-based applications. In *Proc. of the IEEE International Conference on Shape Modeling and Applications 2007, SMI '07*, pages 91–102, Washington, DC, USA, 2007. IEEE Computer Society.
14. H. Hoppe. Progressive meshes. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH '96*, pages 99–108, New York, NY, USA, 1996. ACM.
15. H. Hoppe, T. DeRose, T. Duchamp, J. McDonald, and W. Stuetzle. Surface reconstruction from unorganized points. *SIGGRAPH Comput. Graph.*, 26(2):71–78, July 1992.
16. H. Hoppe, T. DeRose, T. Duchamp, J. McDonald, and W. Stuetzle. Mesh optimization. In *Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH '93*, pages 19–26, New York, NY, USA, 1993. ACM.
17. M. Isenburg, P. Lindstrom, S. Gumhold, and J. Snoeyink. Large mesh simplification using processing sequences. In G. Turk, J.J. van Wijk, and R.J. Moorhead II, editors, *IEEE Visualization*, pages 465–472. IEEE Computer Society, 2003.
18. A.E. Johnson and M. Herbert. Control of polygonal mesh resolution for 3-D computer vision. *Graph. Models Image Process.*, 60(4):261–285, July 1998.
19. K.-L. Low and T.-S. Tang. Model simplification using vertex-clustering. In *Proc. of the 1997 Symposium on Interactive 3D Graphics, I3D '97*, pages 75–81, New York, NY, USA, 1997. ACM.
20. D. Luebke, B. Watson, J.D. Cohen, M. Reddy, and A. Varshney. *Level of Detail for 3D Graphics*. Elsevier Science Inc., New York, NY, USA, 2002.
21. M.M. Michael and M.L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proc. of the 15th Annual ACM Symposium on Principles of Distributed Computing, PODC '96*, pages 267–275, New York, NY, USA, 1996. ACM.
22. Georgia Institute of Technology. Large geometric models archive, February 2012. http://www.cc.gatech.edu/projects/large_models/.
23. Z. Pan, K. Zhou, and J. Shi. A new mesh simplification algorithm based on triangle collapses. *J. Comput. Sci. Technol.*, 16(1):57–63, 2001.
24. W.J. Schroeder, J.A. Zarge, and W.E. Lorensen. Decimation of triangle meshes. *SIGGRAPH Comput. Graph.*, 26(2):65–70, July 1992.
25. S.N. Sinha, J.-M. Frahm, M. Pollefeys, and Y. Genc. GPU-based video feature tracking and matching. Technical report, In Workshop on Edge Computing Using New Commodity Architectures, 2006.
26. Stanford University. The Stanford 3D scanning repository, January 2012. <http://graphics.stanford.edu/data/3Dscanrep/>.