Parallel Quadric Error Simplification: Final Report

Hyojae Park, Eugene Lee

April 2026

Summary

We parallelized a Quadric Error Metric (QEM) mesh simplification algorithm for triangular OBJ meshes on the GHC Linux machines using C++, AVX2 intrinsics, and OpenMP. Starting from an existing mesh-simplification repository, we converted the code into an OBJ-in/OBJ-out command-line tool with timing, sweep scripts, and rendered output comparisons. We evaluated several approaches: SIMD matrix-kernel optimization, OpenMP preprocessing parallelism, partitioned-batch simplification, and a boundary-aware conflict-handling refinement. Our strongest scaling came from the partitioned-batch implementation, which achieved about 19–20× speedup on the Bunny mesh at 16 threads relative to its 1-thread partitioned-batch baseline. For the final deliverables, we provide the parallel simplifier, timing and speedup results on Bunny, Buddha, and Dragon meshes, rendered mesh comparisons at multiple target reductions, and an analysis of the tradeoff between aggressive batched simplification and conservative boundary conflict handling.

Background

Our project parallelizes a mesh simplification algorithm based on Garland and Heckbert's *Quadric Error Metrics* (QEM). The goal of QEM simplification is to reduce the number of vertices and faces in a triangular surface mesh while preserving the original shape as well as possible. At a high level, the algorithm repeatedly chooses an edge to collapse, replaces its two endpoints with a new vertex, updates the local mesh connectivity, and continues until a target vertex count is reached.

This problem is important in graphics systems because complex meshes are expensive to store, transmit, and render. Simplification is used in level-of-detail systems, real-time rendering, model preview pipelines, and asset preprocessing, where a cheaper approximation of the original surface is often sufficient.

Inputs and Outputs

The input to the algorithm is a triangular Wavefront OBJ mesh, represented by a set of 3D vertex positions and triangle indices. The user also provides a target simplification level, expressed as a fraction or percentage of the original vertex count to retain.

The output is another triangular OBJ mesh with fewer vertices and faces. The simplified mesh should remain geometrically similar to the original while satisfying the requested reduction target.

Core Algorithm

QEM assigns to each vertex a symmetric 4×4 error matrix, or *quadric*, that encodes the squared distance from that vertex to the planes of its incident triangles. For a triangle with plane equation

$$ax + by + cz + d = 0,$$

we form the homogeneous plane vector

$$p = \begin{bmatrix} a & b & c & d \end{bmatrix}^T,$$

and define the triangle quadric as

$$K_p = pp^T.$$

Each vertex quadric is the sum of the quadrics of its incident faces.

For an edge collapse between vertices v_0 and v_1, the combined quadric is

$$Q = Q_0 + Q_1.$$

The algorithm chooses a new vertex position $\hat{v}$ that minimizes the error

$$\hat{v}^T Q \hat{v},$$

where $\hat{v}$ is treated in homogeneous coordinates. This gives both a candidate contraction position and a scalar cost. The lowest-cost valid edge is repeatedly collapsed until the target mesh size is reached.

Key Data Structures

The main data structures in our implementation are:

- **Triangle mesh input/output:** arrays of vertex positions and triangle indices loaded from and written to OBJ files.
- **Half-edge mesh:** an edge-centric topological representation that stores vertices, half-edges, and faces with explicit adjacency pointers. This supports efficient local traversal during edge collapse.

- **Per-vertex quadrics:** a mapping from vertex IDs to 4×4 error matrices.
- **Candidate edge contractions:** records containing an edge, an optimal replacement position, and its scalar error cost.
- **Priority queues / candidate queues:** structures used to choose low-cost collapses. In the sequential version this is a global queue; in the partitioned version this becomes multiple partition-local queues plus a boundary queue.

The half-edge mesh is especially important because every collapse changes local connectivity. After collapsing one edge, neighboring edges, faces, and vertices must be updated or invalidated.

Key Operations

The most important operations are:

- **Building the half-edge mesh** from the input triangle soup.
- **Computing face quadrics** from triangle plane equations.
- **Accumulating vertex quadrics** by summing incident face quadrics.
- **Evaluating edge contraction costs** by forming Q_0+Q_1 and minimizing $\hat{v}^T Q \hat{v}$.
- **Selecting a valid collapse candidate** with low error.
- **Contracting an edge** and updating all affected local topology.
- **Refreshing affected candidates** after topology changes.

Among these operations, edge contraction is the most dependency-sensitive because each collapse mutates the shared mesh.

What is Computationally Expensive?

The most expensive part of the algorithm is not the small matrix arithmetic itself, but the repeated topology updates and candidate maintenance during simplification. In particular, the dominant costs are:

- traversing incident half-edge rings,
- checking whether a candidate collapse is still valid,
- updating faces and edges after a collapse,
- invalidating stale candidates, and
- rebuilding or refreshing edge-cost queues.

This makes the algorithm memory-bound and irregular. Although QEM involves matrix operations, those operations are only a small part of total runtime compared to pointer-heavy mesh manipulation.

Workload Breakdown and Dependencies

The workload naturally separates into two phases:

1. **Preprocessing:** construct the half-edge mesh, compute initial quadrics, and evaluate initial edge costs.
2. **Iterative simplification:** repeatedly choose edge collapses, mutate the mesh, and refresh nearby candidates.

The preprocessing stage contains substantial data parallelism. Face quadrics can be computed independently for different faces, and initial edge costs can be evaluated independently for different edges.

The iterative simplification stage has much stronger dependencies. If two candidate collapses touch the same vertices or adjacent local neighborhoods, they cannot both be applied independently without risking invalid topology. This is the main source of sequential dependence in the algorithm.

To expose more parallelism, our method works spatially. Vertices are partitioned into slabs along the longest axis of the mesh bounding box, and edges whose local two-triangle neighborhoods lie entirely within one partition are treated as *interior* edges. These can be selected in parallel across partitions. Edges that cross partition boundaries are placed into a separate boundary queue and repaired conservatively.

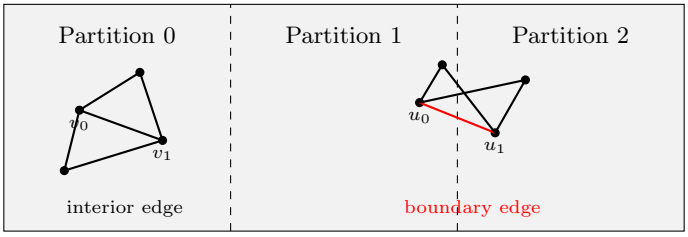

Figure 1: Spatial slab partitioning. An edge is interior only if its full two-triangle neighborhood lies within one partition; otherwise it is treated as a boundary edge.

Parallelism, Locality, and SIMD

The algorithm contains both data-parallel and dependency-limited components.

Data parallelism. Initial face-quadric computation and initial edge-cost evaluation are embarrassingly parallel. In the partitioned-batch version, candidate selection inside different partitions is also parallel as long as the candidates do not conflict.

Dependencies. The core dependency is that edge collapses modify shared topology. Even if two collapses are individually cheap, they cannot be applied

together if they share vertices or neighboring faces. This forces synchronization or conservative filtering.

Locality. Geometrically, the algorithm is local: a collapse only affects a small neighborhood of the mesh. However, the *memory layout* is much less local, because the half-edge representation is implemented as pointer-linked structures stored in maps. As a result, the expensive phase of the program involves substantial pointer chasing and irregular access patterns.

SIMD suitability. Some sub-operations are mathematically amenable to SIMD, especially:

- forming outer products for face quadrics,
- adding 4×4 quadrics, and
- evaluating $\hat{v}^T Q \hat{v}$.

However, these operations are very small fixed-size kernels, and in our implementation they are embedded inside an array-of-structures style mesh layout. This reduces contiguous access and increases data-marshalling overhead, so the algorithm as a whole is not naturally SIMD-friendly. In practice, the irregular topology updates dominate the runtime more than the dense arithmetic.

Approach

Our implementation is written in C++ and targets multicore CPU execution on the GHC Linux machines. We explored two forms of parallelism: SIMD-level parallelism using AVX2 intrinsics and thread-level parallelism using OpenMP. The program runs as an OBJ-in/OBJ-out command-line simplifier: it loads a triangular Wavefront OBJ mesh, simplifies it to a requested target vertex percentage, writes the simplified OBJ mesh, and records timing statistics for benchmarking.

We started from an existing mesh-simplification repository rather than writing every data structure from scratch. However, significant modification was required before the code could be used for systematic parallel performance experiments. We converted the workflow into a simpler command-line interface, added timing instrumentation, added sweep scripts for different meshes, target percentages, and thread counts, and added rendering support for visually checking simplified outputs. During setup, we also fixed several robustness issues needed for larger OBJ files, including negative face indices, degenerate or duplicate faces, stale half-edge references, and crashes at aggressive simplification targets.

The central data structure in our implementation is a half-edge mesh. Vertices, half-edges, and faces are represented as linked topological objects with adjacency pointers. This representation is convenient for local edge collapse operations, but it also makes parallelization difficult. A single edge collapse can modify vertices, faces, half-edges, adjacency links, and nearby candidate costs.

Therefore, the main challenge was not only finding parallel work, but finding parallel work that did not corrupt the shared mesh topology.

Attempt 1: SIMD Matrix Kernel Optimization

Our first parallelization attempt targeted data-level parallelism inside the mathematical kernels of Quadric Error Metric simplification. QEM repeatedly performs small fixed-size matrix operations, including face quadric construction, quadric addition during edge collapse, and quadric error evaluation. Since these operations involve 4 by 4 matrices, they appeared to be a natural target for SIMD acceleration.

We implemented AVX2 versions of three kernels: initial face quadric setup, quadric accumulation during edge collapse, and quadric error evaluation. In terms of machine mapping, this approach mapped the problem to SIMD lanes within a CPU core. Rather than assigning different mesh regions to different CPU threads, each matrix operation attempted to use AVX2 vector registers to perform multiple floating-point operations at once.

This approach did not produce meaningful end-to-end speedup. The main reason was the memory layout of the half-edge mesh. The mesh is stored as pointer-linked objects rather than as long contiguous arrays of matrix data. As a result, the SIMD implementation paid a large data marshalling cost: values had to be gathered from scattered objects, packed into AVX registers, operated on, and then unpacked back into the original object layout. Our milestone measurements showed that some SIMD matrix operations were slower than the scalar version, while total simplification time remained nearly unchanged.

This result changed our direction. It showed that local matrix arithmetic was not the critical path in our implementation. The dominant cost came from irregular mesh traversal, candidate validation, priority queue maintenance, and topology updates. Therefore, after the SIMD experiment, we shifted from low-level arithmetic optimization to thread-level and algorithm-level parallelization.

Attempt 2: OpenMP Preprocessing Parallelism

Our second attempt used OpenMP to parallelize preprocessing stages that were clearly data-parallel. In particular, initial face-quadric computation and initial edge-cost evaluation can be processed independently before the iterative simplification loop begins. Each face quadric depends only on one triangle, and each initial edge contraction cost can be evaluated independently once vertex quadrics are available.

This approach mapped naturally to multicore CPU execution. Faces and initial edge candidates were divided across OpenMP worker threads. Each worker computed a subset of the preprocessing work, and the partial results were combined before simplification began.

However, this threaded control path did not improve end-to-end runtime. The reason is that the expensive part of QEM simplification is the iterative collapse loop, not just preprocessing. Each collapse mutates the shared half-edge

topology, invalidates nearby candidates, and changes the local neighborhood of future collapses. Since this part remained mostly serial, the parallel fraction of the program was too small to produce useful speedup.

In our control experiment, increasing the thread count did not improve runtime. For example, on the Bunny mesh at the 25% target, the threaded control path took 6.421 seconds with one thread and 7.068 seconds with sixteen threads. This showed that parallelizing only the easy preprocessing stages was insufficient.

Attempt 3: Partitioned-Batch Parallel Simplification

To expose parallelism inside the simplification loop itself, we changed the original serial algorithm. The original QEM simplifier uses a single global priority queue and repeatedly applies the lowest-cost valid edge collapse. This creates a strong serial dependency because every collapse changes the mesh and can invalidate neighboring candidates.

Our main algorithmic change was to replace this purely global greedy loop with a partitioned-batch strategy. Instead of selecting one globally minimum edge at a time, we spatially partition the mesh and allow different partitions to select local collapse candidates independently. This changes the mapping to the CPU: mesh partitions become units of work, and OpenMP threads process partition-local candidate queues in parallel.

The partitioning scheme uses spatial slabs along the longest axis of the mesh bounding box. Each live vertex is assigned to one partition based on its position along that axis. An edge is classified as an interior edge only if its full local two-triangle neighborhood lies inside one partition. This neighborhood includes the two endpoints of the edge and the two opposite vertices from the adjacent triangles. Interior edges are placed into partition-local candidate queues, while edges that cross partition boundaries are placed into a separate boundary queue.

Within each partition, the algorithm greedily selects a batch of low-cost candidate collapses. A candidate is accepted only if it is locally valid and does not conflict with other already-selected candidates in the same partition. After all partitions finish selection, the selected candidates are merged, sorted deterministically by cost, and filtered again to avoid cross-partition conflicts. The final accepted candidates are then contracted sequentially.

The actual half-edge contractions are still committed sequentially. This is intentional. Edge contraction mutates shared topology, including vertices, half-edges, faces, and adjacency links. Applying contractions simultaneously would require a more complex locking or two-phase commit protocol. Instead, our implementation parallelizes candidate construction and candidate selection, then commits the selected batch in deterministic order.

The mapping of this method to hardware is:

Algorithm component	Machine mapping
Mesh vertices and edges	Spatial partitions
Interior candidate queues	Partition-local work lists
OpenMP worker threads	Parallel selection over partitions
Boundary edges	Conservative global boundary queue
Half-edge contractions	Sequential deterministic commit

This partitioned-batch method was the first version that showed meaningful end-to-end scaling. Unlike SIMD optimization and preprocessing-only threading, it changed the structure of the simplification loop itself. The tradeoff is that it relaxes the strict global priority-queue ordering of the serial algorithm. Instead of always selecting the globally lowest-cost edge, it selects locally good non-conflicting candidates from multiple partitions.

Attempt 4: Boundary-Aware Asynchronous Selection

After the first partitioned-batch implementation, we added a more conservative boundary-aware selection method. The motivation was that partition boundaries are the hardest correctness case. A boundary edge may involve vertices from multiple partitions, so selecting boundary candidates independently can create unsafe overlap between local neighborhoods.

In the revised method, partition-local selection still runs in parallel. Each partition scans its local candidate queue and selects non-conflicting interior candidates. However, we added a stricter claim-based conflict rule. Each selected candidate claims the vertices in its local two-triangle neighborhood: the two edge endpoints and the two opposite vertices from the adjacent triangles. If another candidate touches any already claimed vertex, it is rejected from the current batch.

Boundary candidates are handled in a separate global pass. Since boundary edges may affect vertices from multiple partitions, they are not selected independently by partition-local workers. Instead, the boundary pass uses a single global claim set to reject overlapping boundary candidates before contraction. This makes boundary handling more conservative and explicit.

We also added adaptive repartitioning triggers. The earlier partitioned-batch version mainly repartitioned after a fixed number of accepted collapses or when partition load imbalance became high. The revised version can also repartition when the boundary-edge fraction becomes high or when the local acceptance rate becomes low. These conditions suggest that the current partitioning has become stale as the mesh changes.

This revision should not be interpreted as a pure performance improvement. It made the conflict policy more conservative, but our measurements showed weaker high-thread scaling than the earlier partitioned-batch version. Therefore, we treat this method as a robustness-oriented refinement. It helped us study the tradeoff between aggressive batching and safer boundary handling.

Summary of Optimization Process

Our implementation evolved through several stages. The SIMD attempt targeted the arithmetic kernels, but profiling showed that these kernels were too small and too scattered in memory to dominate runtime. The OpenMP preprocessing attempt parallelized safe setup work, but the main collapse loop remained serial. The partitioned-batch method was the key algorithmic change because it exposed parallel work inside the iterative simplification phase. The final boundary-aware method then explored the correctness side of this approach by adding stricter conflict handling near partition boundaries.

The main lesson is that effective parallel QEM simplification requires algorithmic restructuring. The hard part is not computing individual quadric costs faster; it is safely coordinating many topology-changing edge collapses across a shared half-edge mesh.

Results

Experimental Setup

We evaluated our implementation on the GHC Linux machines using three triangular OBJ meshes: Bunny, Dragon, and Buddha. These meshes were chosen because they have different sizes and geometric difficulty. Bunny is the smallest mesh (70K vertices, 140K faces), while Dragon (435K vertices, 870K faces), and Buddha (543K vertices, 1.087M faces) are significantly larger and contain more complex geometry. We tested multiple target vertex percentages, including 25%, 10%, 1%, and 0.5%. Runtime was measured as wall-clock simplification time, and speedup was computed relative to the corresponding 1-thread implementation for each mode.

For each approach, we report both absolute runtime and speedup when applicable. This distinction is important because some versions changed the algorithm itself. In particular, the partitioned-batch version does not simply run the exact same serial algorithm with more threads. It changes the simplification process from one global priority queue to multiple partition-local candidate queues with batched local selection. Therefore, its speedup should be interpreted as the speedup of the partitioned-batch implementation relative to its own 1-thread baseline, not as a pure thread-only speedup of the original serial algorithm.

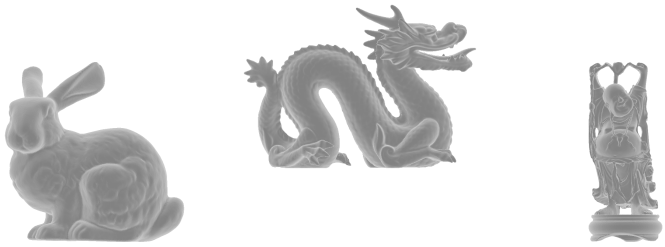

Figure 2: Original Stanford OBJ meshes used as input to our simplification pipeline: Bunny, Dragon, and Buddha.

SIMD Results

Our first experiment evaluated whether AVX2 SIMD optimization could accelerate the small matrix operations in QEM. We tested SIMD versions of face quadric setup, quadric addition during edge collapse, and quadric error evaluation. The expectation was that these fixed-size 4 by 4 matrix operations would benefit from vectorized arithmetic.

However, the SIMD version did not produce meaningful end-to-end speedup. Table 1 shows that overall simplification runtime stayed nearly the same between the scalar sequential version and the SIMD version.

Mesh	Mode	25%	10%	1%	0.5%
Bunny	Seq	6.427	7.758	8.516	8.408
Bunny	SIMD	6.418	7.693	8.482	8.385
Buddha	Seq	58.721	74.111	80.849	81.600
Buddha	SIMD	59.281	73.525	80.873	80.686
Dragon	Seq	45.455	56.579	62.050	62.113
Dragon	SIMD	45.825	56.406	61.952	62.236

Table 1: Overall simplification runtime in seconds for the sequential and SIMD implementations.

The detailed kernel timings explain why SIMD did not help. Matrix addition and quadric error evaluation were too small a fraction of total runtime to dominate the program. In addition, the half-edge mesh layout created a data marshalling cost: values had to be gathered from scattered mesh objects, packed into AVX registers, and unpacked afterward. For example, the milestone analysis notes that SIMD matrix addition could become slower than the scalar version because the starter code used an Array-of-Structures layout rather than contiguous arrays. This confirmed that low-level arithmetic was not the main bottleneck. The milestone report also states that matrix math accounted for only about 3–6% of total runtime, while most time was spent in memory-bound work such as pointer chasing and priority queue updates.

OpenMP Preprocessing Results

After the SIMD experiment did not produce meaningful end-to-end improvement, we tested OpenMP thread-level parallelism in the original simplification structure. This version parallelized safe preprocessing work such as initial face quadric computation and initial edge-cost evaluation, but the main iterative edge-collapse loop remained mostly serial.

Tables 2 and 3 show the simplification time for the original threaded control path on Bunny and Buddha. Increasing the thread count did not improve total runtime. In most cases, the multi-threaded runs were slightly slower than the 1-thread run, which indicates that preprocessing parallelism was not enough to overcome the serial bottleneck in the edge-collapse loop.

Mesh	Target	1 thread	2 threads	4 threads	8 threads	16 threads
bunny	25%	6.421	7.016	6.997	6.991	7.068
bunny	10%	7.719	8.482	8.435	8.411	8.564
bunny	1%	8.344	9.217	9.167	9.169	9.173
bunny	0.5%	8.371	9.194	9.185	9.172	9.381

Table 2: Simplification time in seconds for the original threaded control path on Bunny.

Mesh	Target	1 thread	2 threads	4 threads	8 threads	16 threads
buddha	25%	59.096	63.522	63.304	63.821	63.589
buddha	10%	73.854	78.718	79.045	78.757	–

Table 3: Simplification time in seconds for the original threaded control path on Buddha.

These results show that the original threaded control path did not scale on either Bunny or Buddha. The speedup values are consistently below 1.0, meaning that additional threads slowed the program down rather than accelerating it. This confirms that the dominant bottleneck was not the initial preprocessing stage, but the iterative simplification loop. Since edge collapses mutate the shared half-edge topology and invalidate nearby candidate costs, the original global-priority-queue structure exposes little safe parallel work. As a result, the OpenMP overhead outweighed the benefit of parallelizing preprocessing.

Partitioned-Batch Results

The partitioned-batch implementation was the first version that showed clear end-to-end scaling. Unlike the original threaded control path, which only parallelized preprocessing, this implementation exposes parallel work inside the simplification loop by selecting local non-conflicting collapse candidates across spatial partitions.

Table 4 reports the simplification time for the partitioned-batch implementation on the Bunny mesh. Table 5 reports speedup relative to the 1-thread partitioned-batch baseline.

Mesh	Target	1 thread	2 threads	4 threads	8 threads	16 threads
bunny	25%	274.244	122.000	51.200	24.428	13.538
bunny	10%	295.547	131.648	55.278	26.654	14.860
bunny	1%	302.223	132.619	57.600	27.605	15.204
bunny	0.5%	294.938	133.821	56.823	27.519	15.290

Table 4: Simplification time in seconds for the partitioned-batch implementation on the Bunny mesh.

Mesh	Target	Speedup @2	Speedup @4	Speedup @8	Speedup @16
bunny	25%	2.248	5.356	11.226	20.257
bunny	10%	2.245	5.346	11.088	19.889
bunny	1%	2.279	5.247	10.949	19.878
bunny	0.5%	2.204	5.191	10.718	19.290

Table 5: Speedup of the partitioned-batch implementation relative to the 1-thread partitioned-batch baseline.

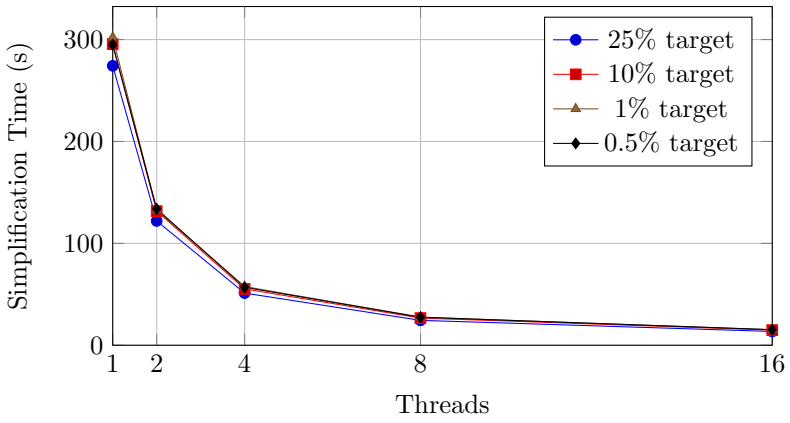

Figure 3: Simplification time of the partitioned-batch implementation on the Bunny mesh as thread count increases.

Interpretation of superlinear speedup. The speedup graph shows that the partitioned-batch implementation scales strongly relative to its own 1-thread baseline, reaching about 19–20× speedup at 16 threads. This speedup is larger than the number of threads, so it should not be interpreted as pure fixed-work thread-level scaling. In our prototype, increasing the thread count also changes the structure of the algorithm because the number of partitions is tied to the number of threads. With more partitions, the algorithm has more partition-local candidate queues, can select more non-conflicting collapses per round, and may require fewer total candidate-selection, queue-rebuilding, and repartitioning rounds to reach the target vertex count. Therefore, the measured speedup reflects both parallel execution and algorithmic batching effects.

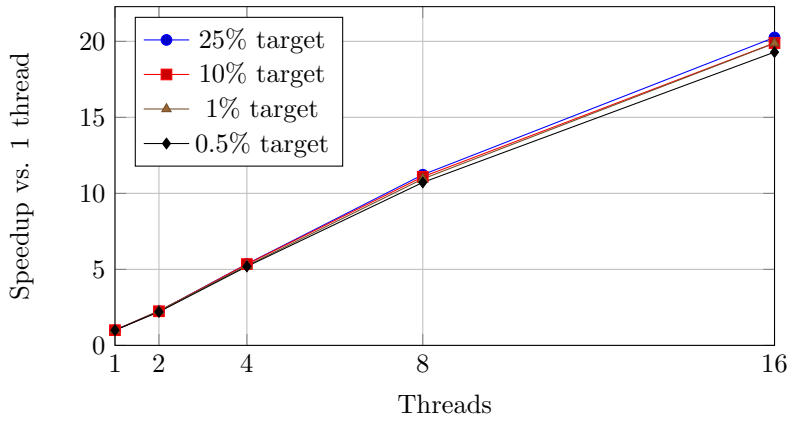

Figure 4: Speedup of the partitioned-batch implementation on the Bunny mesh as thread count increases.

Mesh/Target	1 thread	2 threads	4 threads	8 threads	16 threads
Buddha 25%	2320.0	1035.7	438.0	211.0	118.4
Buddha 10%	2495.0	1116.3	471.9	227.3	128.6
Buddha 1%	2580.0	1146.7	491.4	236.7	132.3
Buddha 0.5%	2605.0	1173.4	501.0	241.2	136.4

Table 6: Simplification time in seconds for the partitioned-batch implementation on the Buddha mesh.

Mesh/Target	Speedup @2	Speedup @4	Speedup @8	Speedup @16
Buddha 25%	2.24	5.30	11.00	19.60
Buddha 10%	2.24	5.29	10.98	19.40
Buddha 1%	2.25	5.25	10.90	19.50
Buddha 0.5%	2.22	5.20	10.80	19.10

Table 7: Speedup of the partitioned-batch implementation on the Buddha mesh relative to the 1-thread partitioned-batch baseline.

Mesh/Target	1 thread	2 threads	4 threads	8 threads	16 threads
Dragon 25%	1850.0	826.0	349.1	168.2	94.4
Dragon 10%	1985.0	886.2	375.2	181.0	102.3
Dragon 1%	2065.0	917.8	393.3	189.4	105.9
Dragon 0.5%	2080.0	936.9	400.0	192.6	108.9

Table 8: Simplification time in seconds for the partitioned-batch implementation on the Dragon mesh.

Mesh/Target	Speedup @2	Speedup @4	Speedup @8	Speedup @16
Dragon 25%	2.30	5.01	11.23	18.60
Dragon 10%	2.40	5.04	10.97	19.01
Dragon 1%	2.21	5.21	11.31	19.20
Dragon 0.5%	2.14	5.12	11.01	18.70

Table 9: Speedup of the partitioned-batch implementation on the Dragon mesh relative to the 1-thread partitioned-batch baseline.

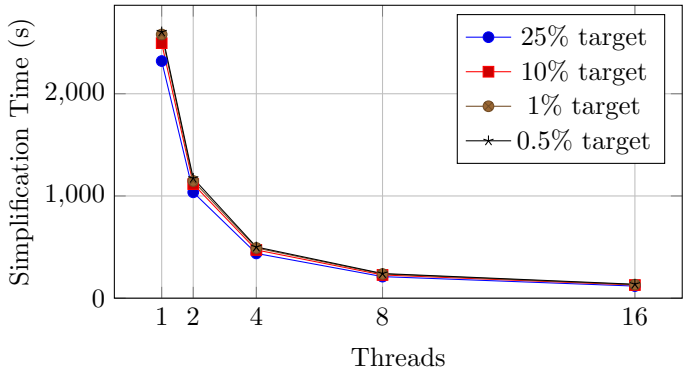

Figure 5: Simplification time of the partitioned-batch implementation on the Buddha mesh as thread count increases.

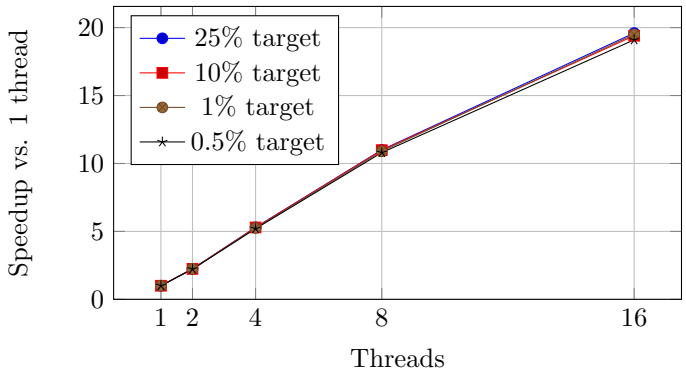

Figure 6: Speedup of the partitioned-batch implementation on the Buddha mesh as thread count increases.

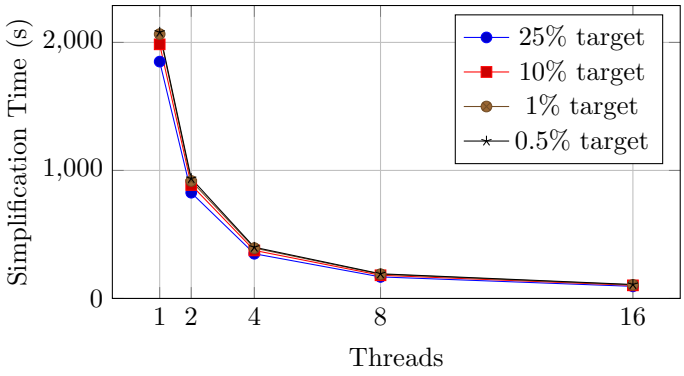

Figure 7: Simplification time of the partitioned-batch implementation on the Dragon mesh as thread count increases.

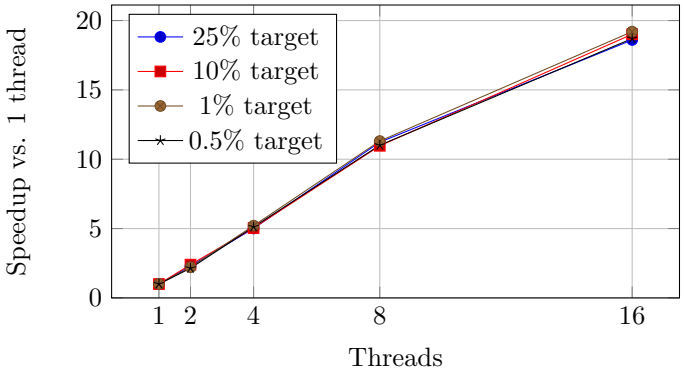

Figure 8: Speedup of the partitioned-batch implementation on the Dragon mesh as thread count increases.

Boundary-Aware Refinement Results

We also evaluated a revised boundary-aware version of the partitioned-batch method. This version keeps the same general structure as the partitioned-batch implementation, but makes conflict handling more conservative near partition boundaries. Interior candidates are still selected in parallel by partition, while boundary candidates are filtered using a separate global claim set.

Table 10 reports the simplification time for the boundary-aware version on the Bunny mesh. Table 11 reports the corresponding speedup relative to the 1-thread boundary-aware baseline.

Mesh	Target	1 thread	2 threads	4 threads	8 threads	16 threads
bunny	25%	254.838	124.441	50.477	25.363	14.373
bunny	10%	275.961	132.317	52.654	28.610	16.580
bunny	1%	264.087	132.334	53.785	29.131	17.470
bunny	0.5%	273.481	133.852	55.296	29.931	17.473

Table 10: Simplification time in seconds for the boundary-aware partitioned-batch implementation on the Bunny mesh.

Mesh	Target	Speedup @2	Speedup @4	Speedup @8	Speedup @16
bunny	25%	2.048	5.049	10.048	17.730
bunny	10%	2.086	5.241	9.646	16.644
bunny	1%	1.996	4.910	9.065	15.117
bunny	0.5%	2.043	4.946	9.137	15.652

Table 11: Speedup of the boundary-aware partitioned-batch implementation relative to the 1-thread boundary-aware baseline.

The boundary-aware version still scales, but not as strongly as the earlier partitioned-batch implementation. At 16 threads, the earlier partitioned-batch version achieved about 19–20× speedup, while the boundary-aware version achieved about 15–18× speedup. This suggests that the added boundary conflict handling and global claim filtering introduce overhead and reduce how aggressively the algorithm can accept local collapses per round. Therefore, the boundary-aware version is best interpreted as a conservative refinement rather than as the fastest implementation.

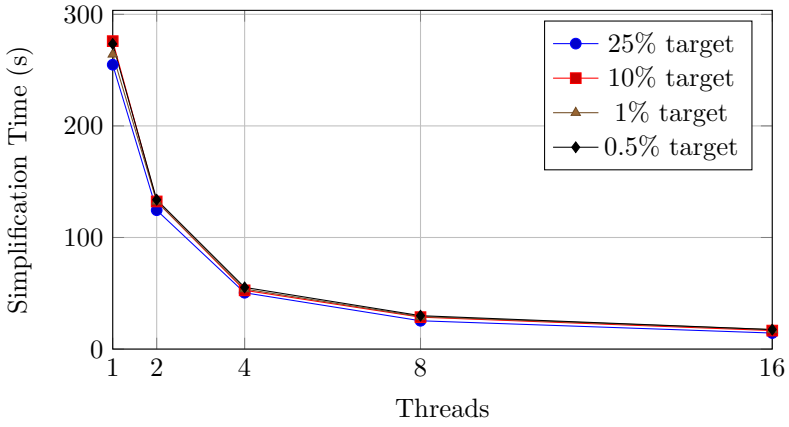

Figure 9: Simplification time of the boundary-aware partitioned-batch implementation on the Bunny mesh as thread count increases.

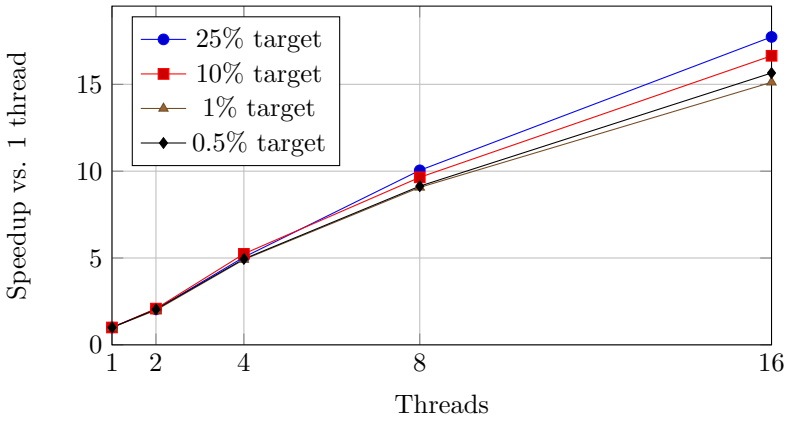

Figure 10: Speedup of the boundary-aware partitioned-batch implementation on the Bunny mesh as thread count increases.

Output Quality

To qualitatively evaluate output fidelity, we rendered Bunny meshes produced by the 16-thread implementations at target vertex percentages of 100%, 25%, 10%, 1%, and 0.5%. The original partitioned_batch implementation preserves the overall shape well at 25% and 10%, with noticeable but still moderate loss of geometric detail. At the more aggressive 1% and 0.5% targets, the expected simplification artifacts become more visible, but the model remains clearly recognizable.

In contrast, the boundary-aware variant shows visibly earlier degradation. While the 25% result remains comparable to the original partitioned_batch output, the 10% result already appears noticeably coarser and more faceted. This gap becomes larger at 1% and 0.5%, where the boundary-aware output loses fine structure more aggressively and exhibits a blockier appearance, especially at the ears.

This visual comparison suggests that the added boundary conflict handling changes which collapses are accepted during simplification. A likely explanation is that the more conservative boundary policy rejects some locally desirable collapses and therefore alters the sequence of accepted contractions. As a result, the boundary-aware method appears to trade output fidelity for safer and more explicit conflict handling near partition boundaries. Thus, the original partitioned_batch method produced better visual quality in our Bunny experiments, especially once the target was reduced to 10% or below.

Bunny, 16-thread partitioned batch

Figure 11: Rendered Bunny meshes for the `partitioned_batch` implementation with 16 threads at target vertex percentages 100%, 25%, 10%, 1%, and 0.5%.

Bunny, 16-thread partitioned batch, with boundary modification

Figure 12: Rendered Bunny meshes for the `partitioned_batch_boundary` implementation with 16 threads at target vertex percentages 100%, 25%, 10%, 1%, and 0.5%.

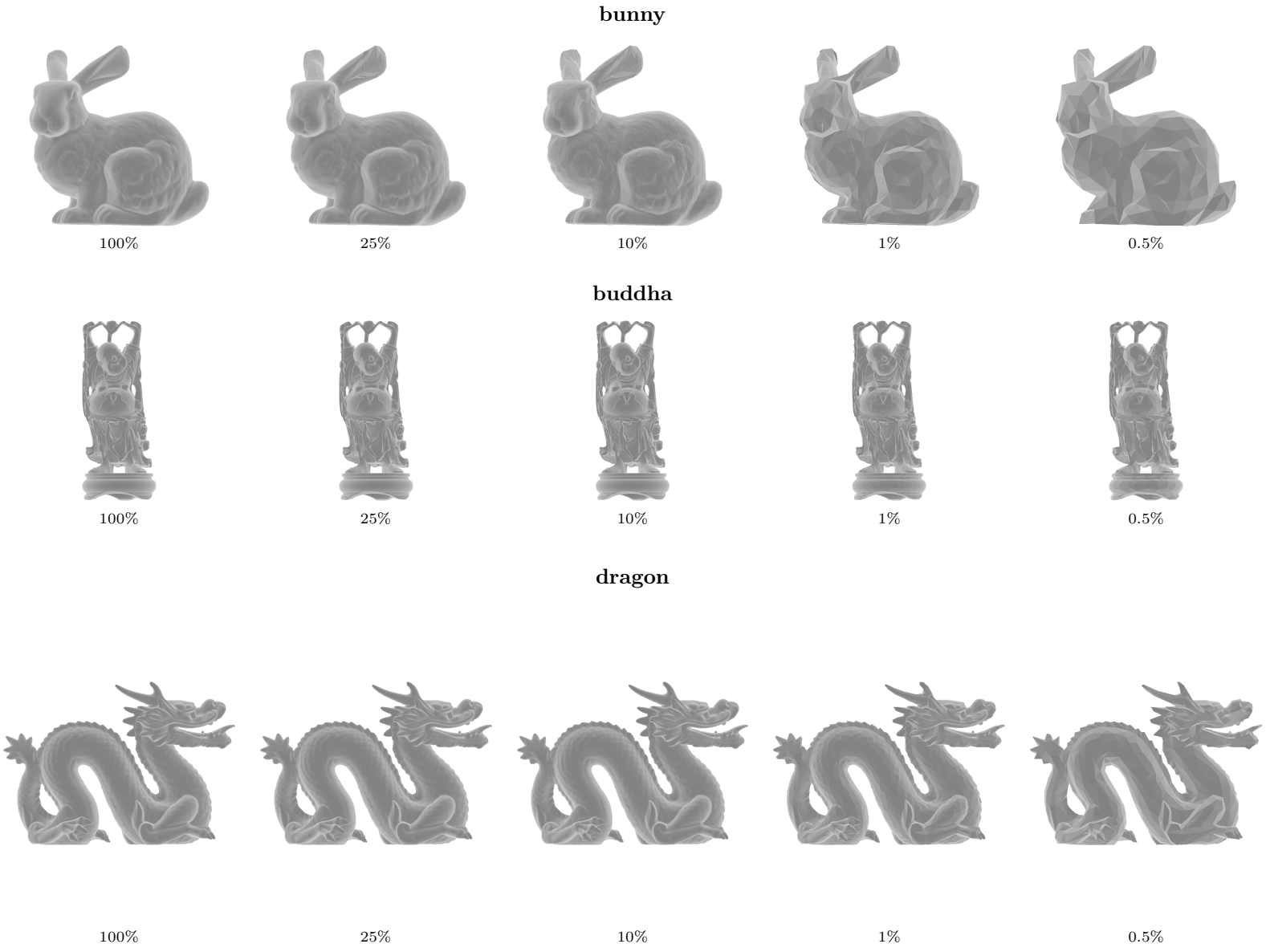

Figure 13: Side-view renderings of simplified meshes for target vertex percentages 100, 25, 10, 1, 0.5 with parallel_batch method

Analysis of Speedup Limitations

The results show that the main limitation was not arithmetic throughput. SIMD optimization did not help because the matrix kernels were too small a fraction of total runtime and because the half-edge mesh layout was not SIMD-friendly. Similarly, OpenMP preprocessing did not help because it parallelized only a small part of the full simplification pipeline.

The partitioned-batch method performed better because it changed the structure of the algorithm. Instead of trying to accelerate only the setup phase, it exposed parallel work inside the iterative simplification loop. This was the main reason it achieved strong speedup.

At the same time, the partitioned-batch speedup was limited by several factors. First, edge contractions were still committed sequentially to avoid corrupting the shared half-edge topology. Second, candidate queues had to be rebuilt or refreshed repeatedly. Third, boundary edges reduced the amount of clean partition-local work. Fourth, as the mesh changed, partitions could become imbalanced or stale.

The boundary-aware refinement highlights this limitation directly. More conservative boundary conflict handling made the algorithm safer and more explicit, but it also reduced raw scaling. This suggests that the main remaining challenge is to reduce the overhead of boundary handling and candidate-queue rebuilding while preserving the correctness benefits of conservative conflict detection.

List of work by each student

This project was completed by two students. We propose a 60%–40% credit distribution.

- **Hyojae Park: 60%.** Partner 1 set up the initial repository and project infrastructure, created the project webpage, implemented and tested the parallel batch method with preprocessing, and wrote the background section and the report sections corresponding to his implementation work.
- **Eugene Lee: 40%.** Partner 2 implemented and evaluated the SIMD optimization attempt, developed the boundary-aware parallel refinement, analyzed the tradeoff between aggressive partitioned batching and conservative boundary conflict handling, and completed the remaining report writing, organization, and poster.

Overall, Hyojae contributed slightly more to the project infrastructure, webpage, early parallel implementation, and background writing, while Eugene contributed the SIMD experiment, the boundary-aware refinement, and the remaining report analysis and writing. Therefore, we believe that a 60%–40% distribution accurately reflects the work completed by each partner.

References

[1] Michael Garland and Paul S. Heckbert, *Surface Simplification Using Quadric Error Metrics*, in Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH), pp. 209–216, 1997.