Parallel Quadric Error Simplification: Milestone Report

Hyojae Park, Eugene Lee

April 2026

Summary

We set up a bare-bones single-threaded implementation of Quadric Error Simplification by starting from the existing mesh-simplification repo, stripping the workflow down to an OBJ-in/OBJ-out CLI, and adding timing plus an optional thread-count parameter for later parallelization experiments. We tested it on bunny, buddha, and dragon, then fixed several issues needed for the more complex meshes, including negative OBJ face indices, degenerate/duplicate faces, stale half-edge topology references, and low-target simplification crashes. After that, we built sweep scripts to run different target vertex percentages (50%, 25%, 10%, 1%, and 0.1%) across thread counts and logged all timings on the GHC machine. We also implemented a simple renderer that renders the simplified OBJ outputs so the results can be visually compared.

The following table shows the three obj files that we have selected for testing due to their range in difficulty (ex: bunny has a small number of vertices and has genus 0, while dragon has a lot more vertices with more holes, and buddha has the most amount of vertices with numerous concave regions and holes).

Model	Vertices	Faces
Bunny	72,027	144,046
Dragon	435,545	871,306
Buddha	543,524	1,087,474

Table 1: Mesh Statistics

SIMD Parallelization Update

To begin parallelizing the Quadric Error Metric (QEM) simplification, we implemented data-level parallelism targeting the compute-heavy 4×4 matrix operations. Using AVX2 intrinsics, we vectorized three core components: the initial face quadric setup (outer products), the accumulation of error quadrics during edge collapse ($Q_0 + Q_1$), and the evaluation of edge collapse costs using Fused Multiply-Add instructions ($v^T Q v$). High-resolution profiling timers were injected around these operations to measure the exact compute speedup across

multiple target vertex percentages.

Results and Analysis
Contrary to theoretical expectations, micro-benchmarking revealed that the explicit SIMD math performed worse than the sequential baseline, with SIMD matrix additions taking over twice as long. This performance regression highlighted a classic architectural bottleneck: the Data Marshalling Tax. Because the underlying mesh data structure utilizes an Array of Structures (AoS) memory layout, the CPU cycles spent packing unaligned, scattered data into 256-bit AVX registers, and unpacking it afterward, completely overshadowed the cycles saved by the parallel math. Ultimately, despite the localized math slowdown, the overall simplification time remained nearly identical to the sequential baseline.

Goals and Deliverables

It took more time than expected to set up the sequential implementation as existing implementations online were buggy and did not have good ways to time and visualize the results. This resulted in requiring much more initial debugging time than expected.

We are still on track to meet all deliverables, but we may not be able to extend these optimization strategies to beyond the GHC machines.

More concretely, our plan is to continue investigating non-trivial parallelization opportunities across threads, which involves:

- efficient mesh partitioning and updating the partition over time
- dealing with contention on edges/vertices on boundaries
- efficiently selecting edges to collapse without sacrificing noticable output fidelity

Concerns

Primary Technical Concerns
Amdahl's Law and Memory Bandwidth: Profiling shows matrix math accounts for only 3% to 6% of total runtime. Over 90% is spent on memory-bound tasks like pointer-chasing and priority queue updates, which may limit OpenMP scalability.
Boundary Synchronization: Partitioning the mesh requires careful coordination for edges on partition boundaries to maintain topological consistency.
Dynamic Load Imbalance: As the mesh collapses irregularly, static partitions may become unbalanced, leaving threads idle.
Mesh Quality: Moving from a global to multiple thread-local priority queues may cause the algorithm to miss globally optimal collapses, potentially degrading output fidelity.

Remaining Unknowns
Partitioning Strategy: The most efficient method (spatial, vertex-based, or connectivity-based) to balance workload versus communication remains to be determined.
Boundary Mechanism: The exact implementation for efficient boundary updates—whether semi-static or dynamic—is still an open challenge.
High Core-Count Scalability: While functional on GHC machines, scalability on larger systems like PSC remains an unknown.

Poster Session

We plan to prepare a video that shows how the meshes simplify from 100% to 0.5%, along with graphs the show the performance differences between the sequential and parallel implementations.

Updated Schedule

Completed Work

- Understood the QEM/QES algorithm and submitted the proposal
- Cloned the existing mesh simplification repository and converted it into a bare-bones CLI workflow
- Set up the C++ build and command-line interface for OBJ input/output
- Tested baseline sequential QEM on bunny, buddha, and dragon meshes
- Fixed support for more complex OBJ files, including negative face indices and degenerate/duplicate triangles
- Added timing output and sweep scripts for different target simplification percentages
- Added simple mesh rendering for visual comparison of output quality
- Completed initial SIMD investigation/implementation

Remaining Tasks

Period	Dates	Planned Tasks
Half-Week 1	4.14 – 4.17	• **Person A:** Identify preprocessing steps that are safely parallelizable, such as initial quadric computation and initial edge-cost evaluation • **Person B:** Investigate which parts of edge collapse and priority queue updates create correctness or data-dependency constraints • **Both:** Define the main parallelization risks: stale edge costs, invalid collapses, vertex/edge contention, and output quality degradation
Half-Week 2	4.18 – 4.20	• **Person A:** Implement initial OpenMP parallelization for preprocessing and benchmark the isolated speedup • **Person B:** Prototype a simple mesh partitioning strategy and record how many edges/vertices lie on partition boundaries • **Both:** Compare the cost of partitioning against the benefit of parallel work
Half-Week 3	4.21 – 4.24	• **Person A:** Prototype thread-local edge selection queues and evaluate whether they preserve reasonable collapse choices • **Person B:** Investigate boundary contention and design rules for avoiding conflicting collapses on shared vertices or neighboring edges • **Both:** Test whether local edge selection changes visual fidelity compared to the sequential global priority queue
Half-Week 4	4.25 – 4.27	• **Person A:** Improve queue update logic after collapses and measure how much time is spent maintaining candidate edges • **Person B:** Investigate how partitions should be updated over time as the mesh changes • **Both:** Compare alternative strategies for handling boundary edges, including locking, skipping, or periodically repartitioning
Final Week	4.28 – 5.1	• **Person A:** Run final timing sweeps across target vertex percentages and thread counts • **Person B:** Generate final mesh renderings for bunny, buddha, and dragon to compare output quality • **Both:** Summarize which parallelization approaches worked, which did not, and what tradeoffs were observed • **Both:** Prepare final report figures, tables, discussion, and poster materials by the 5.1 deadline

Preliminary Results

The following table and figures show the initial results that we got from running our simplification algorithm, as well as the time it took to run them.

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 2: Overall simplification runtime in seconds for Sequential vs. SIMD modes.

Mesh	Mode	25%	10%	1%	0.5%
bunny	Seq	0.215	0.257	0.278	0.271
bunny	SIMD	0.197	0.230	0.251	0.246
buddha	Seq	1.533	1.840	1.964	2.001
buddha	SIMD	1.435	1.686	4.515	4.487
dragon	Seq	1.230	1.440	1.551	1.553
dragon	SIMD	1.228	1.420	1.534	1.540

Table 3: Total time spent (in seconds) performing Matrix Additions ($Q_0 + Q_1$) during edge collapse.

Mesh	Mode	25%	10%	1%	0.5%
bunny	Seq	0.043	0.051	0.056	0.056
bunny	SIMD	0.045	0.053	0.058	0.058
buddha	Seq	0.320	0.379	0.415	0.418
buddha	SIMD	0.334	0.395	0.434	0.436
dragon	Seq	0.256	0.303	0.332	0.334
dragon	SIMD	0.267	0.317	0.347	0.348

Table 4: Total time spent (in seconds) evaluating Quadric Error costs ($v^T Q v$).

Mesh	Mode	25%	10%	1%	0.5%
bunny	Seq	0.022	0.023	0.023	0.023
bunny	SIMD	0.022	0.022	0.023	0.023
buddha	Seq	0.193	0.197	0.196	0.195
buddha	SIMD	0.195	0.193	0.199	0.193
dragon	Seq	0.149	0.150	0.150	0.149
dragon	SIMD	0.150	0.149	0.149	0.149

Table 5: Total time spent (in seconds) calculating fundamental error matrices for initial faces.

Analysis of SIMD Optimization Results

Our initial parallelization effort focused on accelerating the compute-heavy matrix operations of the Quadric Error Metric (QEM) using AVX2 SIMD intrinsics. We targeted three specific areas: face quadric setup, matrix addition ($Q_0 + Q_1$), and quadric error evaluation ($v^T Q v$).Contrary to theoretical expectations, the SIMD implementation did not yield a tangible speedup. In fact, micro-benchmarking revealed that the explicit SIMD math performed worse than the sequential baseline.

The Data Marshalling Tax (AoS vs. SoA)

The most significant slowdown occurred during Matrix Additions (Table 3), where the SIMD implementation was 2x slower than the sequential version (e.g., jumping from 1.96s to 4.51s on the Buddha mesh). This is a direct consequence of the starter code's Array of Structures (AoS) memory layout. Because the matrices are tied to individual, scattered Vertex and Edge objects, we could not leverage contiguous memory loads. To perform a SIMD matrix addition, the CPU had to fetch unaligned memory, pack 32 individual double values into the 256-bit AVX registers, perform the addition, and unpack the results back to memory. The CPU cycles spent shuffling memory (marshalling) completely overwhelmed the cycles saved by the parallel addition.

Amdahl's Law and Memory-Bound Limitations

Despite the SIMD math taking 2 to 3 seconds longer overall, the Overall Simplification Time (Table 1) remained nearly identical (and occasionally marginally faster due to standard system jitter and cache luck). This exposes the true bottleneck of the algorithm: Amdahl's Law. Our profiling indicates that the local matrix operations account for only 3% to 6% of the total execution time.

Conclusion and Next Steps

This proves that local mathematical computation is not the critical path in QEM simplification. Optimizing the math via SIMD is neutralized by memory latency and the inherent efficiency of compiler-unrolled 4×4 scalar math.

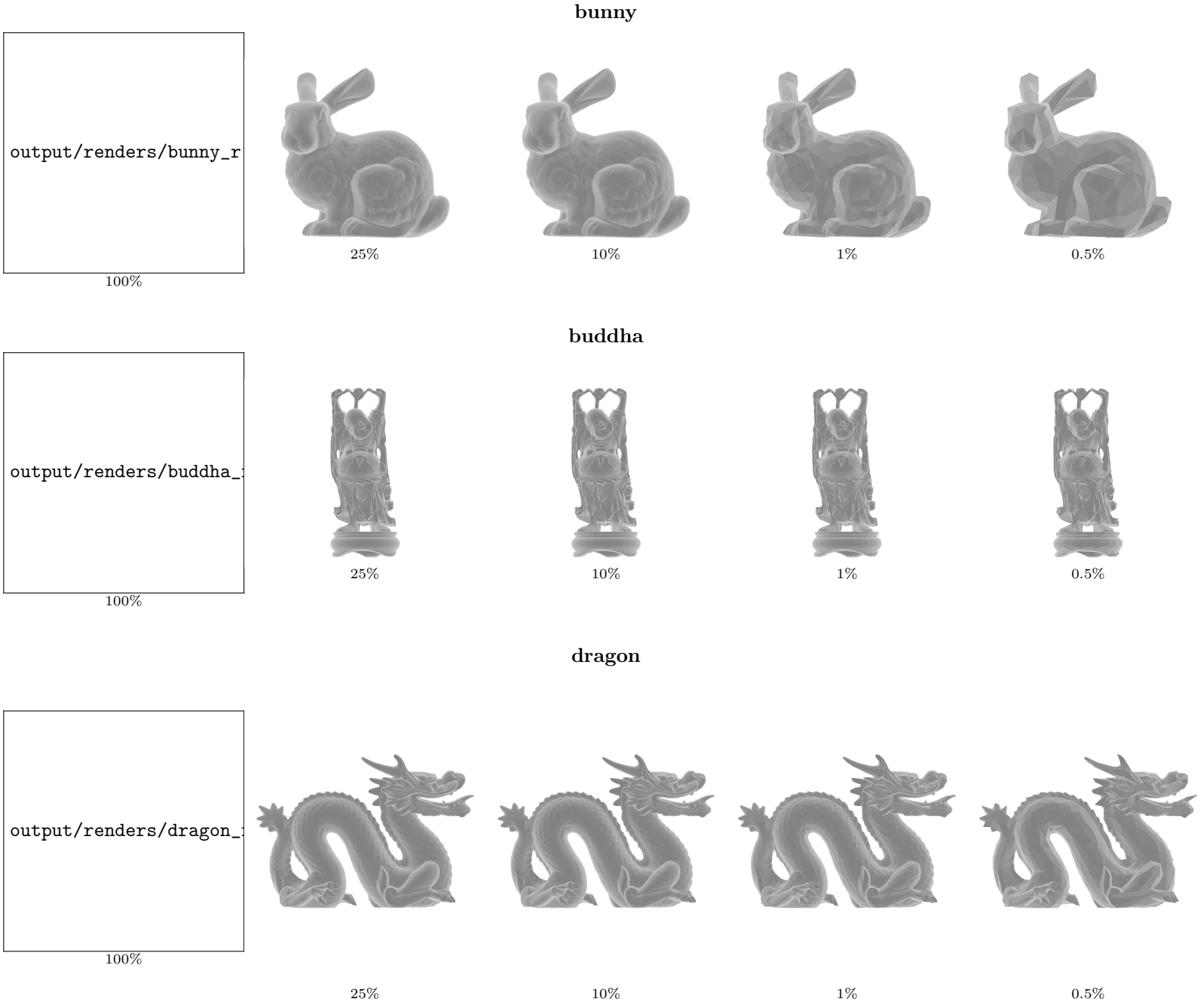

Figure 1: Side-view renderings of simplified meshes for target vertex percentages 100, 25, 10, 1, 0.5.