Non-Trivial Parallelization of a Real-Time 3D Gaussian Splatting Renderer

Henry Wagner
University of Michigan
EECS 587
htwagner@umich.edu

December 2025

Abstract

This paper presents the systematic optimization of a CUDA-based 3D Gaussian Splatting renderer through iterative bottleneck analysis and parallelization. The baseline implementation, despite running GPU kernels for core operations, spent 84% of frame time in CPU-bound operations due to unnecessary data transfers and sequential algorithms. We address these bottlenecks through four optimizations: (1) GPU-parallel Gaussian duplication using prefix sums achieving 96× stage speedup, (2) parallel tile range identification achieving 40× speedup, (3) shared memory cooperative loading providing 50% blend kernel improvement at scale, and (4) GPU stream compaction achieving 2.9× cull speedup. The combined optimizations yield **15.5× overall speedup**, transforming performance from 8 FPS to 125 FPS at 2 million Gaussians. This work demonstrates that hidden CPU bottlenecks can dominate seemingly "GPU-accelerated" applications and showcases fundamental parallel primitives including prefix sum and stream compaction.

1 Introduction

1.1 Background and Motivation

3D Gaussian Splatting (3DGS) has emerged as a breakthrough technique in neural scene representation, offering real-time rendering capabilities that were previously unattainable with neural radiance fields (NeRF) [1]. Rather than requiring expensive per-ray network evaluations, Gaussian splatting represents scenes as collections of anisotropic 3D Gaussian primitives that can be efficiently rasterized using tile-based rendering.

A production-quality Gaussian splat scene typically contains between 100,000 and 2,000,000 individual Gaussians. Each frame requires processing every Gaussian through a multi-stage pipeline: frustum culling against the view volume, projection from world space to screen space with Jacobian-based covariance transformation, assignment to overlapping screen tiles, depth sorting within each tile, and finally alpha blending in front-to-back order. The computational demands of this pipeline make efficient parallelization essential for real-time performance.

1.2 The Challenge of Non-Trivial Parallelization

At first glance, Gaussian splatting appears amenable to straightforward GPU parallelization—one could simply assign one thread per Gaussian for culling and transformation, and one thread per pixel for blending. However, several factors complicate efficient implementation:

Variable output sizes. Each Gaussian may overlap anywhere from 1 to 20+ screen tiles depending on its projected size. This creates a classic "variable outputs per input" problem that cannot be solved with simple one-to-one thread mappings.

Data-dependent control flow. Alpha blending terminates when accumulated opacity exceeds 99%, but different pixels saturate at different rates. This causes warp divergence as threads within the same warp take different execution paths.

Memory access patterns. The blend kernel exhibits significant redundancy: all 256 threads processing a tile must read the same Gaussian data, yet without careful optimization each thread independently fetches from global memory.

Hidden CPU bottlenecks. As we discovered through profiling, seemingly minor "bookkeeping" operations—data marshaling between pipeline stages—can dominate frame time when implemented sequentially on the CPU.

1.3 Contributions

This paper makes the following contributions:

1. We identify that CPU-bound operations consume 84% of frame time in a baseline "GPU-accelerated" Gaussian splatting renderer, demonstrating the importance of holistic pipeline analysis.

2. We implement GPU-parallel Gaussian duplication using the prefix sum and scatter pattern, eliminating the largest bottleneck and achieving 96× stage speedup.

3. We demonstrate parallel tile range identification through boundary detection, achieving 40× speedup with a simple reformulation.

4. We analyze shared memory tiling for the blend kernel, revealing that benefits scale with dataset size due to cache hierarchy effects.

5. We implement GPU stream compaction for cull results, eliminating unnecessary CPU-GPU data transfers.

6. We achieve 15.5× overall speedup, rendering 2 million Gaussians at 125 FPS.

2 Background

2.1 3D Gaussian Splatting

3D Gaussian Splatting represents scenes as collections of anisotropic 3D Gaussians, each parameterized by a mean position $\boldsymbol{\mu} \in \mathbb{R}^3$, a covariance matrix $\boldsymbol{\Sigma} \in \mathbb{R}^{3\times 3}$ (typically stored as scale and rotation), opacity $\alpha \in [0, 1]$, and view-dependent color encoded via spherical harmonics.

Figure 1: Real-time rendering of a 2-million Gaussian scene at 125 FPS after optimization. The scene represents a potted cactus captured using 3D Gaussian Splatting.

Rendering proceeds by projecting each Gaussian to screen space, computing a 2D covariance matrix via the Jacobian of the projection:

$$\boldsymbol{\Sigma}_{2D} = \mathbf{J}\mathbf{W}\boldsymbol{\Sigma}\mathbf{W}^T\mathbf{J}^T \tag{1}$$

where $\mathbf{W}$ is the view transformation and $\mathbf{J}$ is the Jacobian of the perspective projection.

2.2 Tile-Based Rendering

Following the original 3DGS implementation, we divide the screen into 16×16 pixel tiles. Each Gaussian is assigned to all tiles it overlaps (based on its 2D bounding box), then Gaussians within each tile are depth-sorted and alpha-blended front-to-back. This tile-based approach enables efficient parallelization: each tile can be processed independently by a thread block.

2.3 Parallel Prefix Sum

The parallel prefix sum (scan) is a fundamental building block for GPU algorithms [2, 3]. Given an array $[a_0, a_1, \ldots, a_{n-1}]$ and an associative operator $\oplus$, the exclusive prefix sum produces $[0, a_0, a_0 \oplus$

$a_1, \ldots]$. This operation transforms sequential accumulations into parallel algorithms with O(n) work and O($\log n$) depth, enabling efficient solutions to problems with variable-size outputs.

3 Baseline Implementation

Before optimization, we developed a complete CUDA-based renderer implementing the 3DGS pipeline. This initial implementation required solving several non-trivial challenges that established the foundation for subsequent optimizations.

3.1 Implementation Challenges

CUDA-OpenGL interop. Rendering requires writing directly to an OpenGL framebuffer from CUDA kernels. We implemented CUDA graphics resource registration to map OpenGL textures for CUDA access, requiring careful synchronization between graphics and compute contexts. The framebuffer must be mapped before kernel execution and unmapped before OpenGL rendering, with proper error handling for context mismatches.

Covariance projection. Transforming 3D covariance matrices to 2D screen space requires computing the Jacobian of the perspective projection. This involves extracting the rotation component from the view matrix, transforming the covariance to camera space, then applying the projection Jacobian. Numerical stability issues arise when Gaussians are near the camera plane, requiring careful handling of near-zero determinants and regularization terms.

Structure-of-Arrays conversion. The PLY file format stores Gaussians as an array of structures (AoS), but GPU kernels require structure-of-arrays (SoA) for efficient coalesced memory access. We implemented a conversion pipeline that reorganizes data during upload, separating means, scales, rotations, opacities, and spherical harmonics coefficients into separate GPU buffers.

Tile-based rendering pipeline. Implementing the full pipeline required coordinating multiple GPU kernels with proper memory management. Each stage produces outputs consumed by the next, requiring careful buffer allocation and lifetime management. The blend kernel processes 16×16 pixel tiles independently, with each thread block handling one tile and evaluating all overlapping Gaussians in depth-sorted order. Early termination when opacity saturates requires careful synchronization to avoid race conditions in the framebuffer writes.

3.2 Rendering Pipeline

Our baseline implementation follows Algorithm 2 from Kerbl et al. [1], implementing all stages with GPU kernels where computationally intensive:

Algorithm 1 Gaussian Splatting Rendering Pipeline

```
1: CullGaussians($P, VP$) → visibleIdx
2: TransformToScreen($P, \Sigma, V, P$) → $\mu_{2D}, \Sigma_{2D}, d$
3: DuplicateWithKeys($\mu_{2D}, T$) → entries
4: SortByKeys($entries$) → sorted
5: IdentifyTileRanges($sorted$) → ranges
6: BlendTiles($sorted, ranges$) → framebuffer
```

With the complete pipeline implemented, we profiled the baseline to identify optimization opportunities.

3.3 Profiling Results

We profiled the baseline implementation using CUDA events for precise GPU timing. Table 1 presents the timing breakdown for 139,410 Gaussians rendered at 720×720 resolution.

Table 1: Baseline Timing Breakdown

Stage	Time (ms)	%	Location
Cull	0.51	2.6	GPU
Transform	0.36	1.8	GPU
Duplicate	**14.43**	**73.8**	**CPU**
Sort	0.95	4.9	GPU
Tile Ranges	**1.73**	**8.8**	**CPU**
Blend	1.54	7.9	GPU
Total	**19.56**	100	—

The results reveal a critical insight: despite the core computational kernels running on the GPU, **82.6% of frame time is consumed by CPU operations**. The `DuplicateWithKeys` stage alone

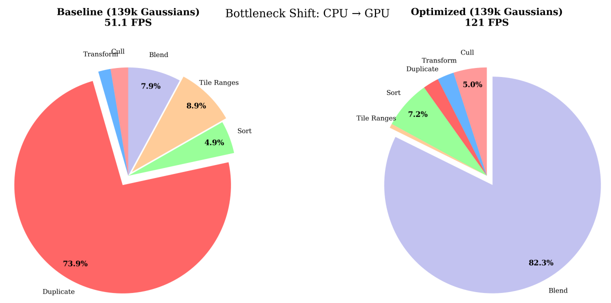


Figure 2: Time distribution before and after optimization. The bottleneck shifts from CPU-bound Duplicate (73.8%) to GPU-bound Blend (82.4%).

accounts for 73.8% of total time because it downloads screen-space data to the CPU, runs a sequential loop, and uploads results back to the GPU.

3.4 Scaling Behavior

To understand how performance scales, we benchmarked datasets ranging from 139k to 1.94M Gaussians. The Duplicate stage scales linearly with input size—expected for a sequential $O(n)$ loop—while GPU operations show much better scaling behavior due to parallel execution.

3.5 Input Variations

Our benchmarks use a single scene (potted cactus) at varying levels of detail. This controlled variation allows us to isolate the effect of Gaussian count on performance. However, other important variations exist:

Tile density variations. Different scenes exhibit different tile density distributions. Scenes with many large Gaussians (e.g., outdoor landscapes) produce more tile overlaps per Gaussian, increasing the duplication workload. Scenes with many small Gaussians (e.g., detailed textures) may have higher tile counts but fewer overlaps per Gaussian. Our optimization handles both cases uniformly through the prefix sum approach.

Camera movement. The scripted camera path includes both orbital motion and zoom, producing varying visible Gaussian counts and tile density throughout the benchmark. Performance remains consistent across these variations, demonstrating robustness to view-dependent workload changes.

Scene complexity. The cactus scene represents a typical indoor object. Outdoor scenes with larger spatial extent might exhibit different culling behavior, but the frustum culling optimization should benefit similarly across scene types.

4 Optimization 1: GPU-Parallel Duplication

4.1 Problem Formulation

The Gaussian duplication stage must create one entry for each (Gaussian, tile) pair. Since each Gaussian overlaps a variable number of tiles, this is a "variable outputs per input" problem. The sequential baseline iterates over Gaussians, computing tile overlaps and appending entries to a list—an inherently serial operation.

4.2 Parallel Algorithm

We reformulate duplication using the scan-scatter pattern, but several implementation challenges complicate this seemingly straightforward approach:

Phase 1 (Count): Launch one thread per Gaussian. Each thread computes the bounding box from the 2D covariance (using 3σ extent) and counts overlapping tiles. Edge cases require careful handling: Gaussians extending beyond screen bounds must be clamped, and degenerate covariances (near-zero determinants) produce invalid bounding boxes that must be detected and handled.

Phase 2 (Scan): Compute exclusive prefix sum on tile counts using Thrust's `exclusive_scan`. This produces output offsets for each Gaussian. A critical issue emerged: the total output size is unknown until after the scan completes. We must either allocate conservatively (wasting memory) or perform a two-pass approach: first scan to compute size, allocate, then re-scan to populate offsets. We chose the two-pass approach to minimize memory usage, requiring careful synchronization between passes.

Phase 3 (Scatter): Launch one thread per Gaussian. Using the computed offset, each thread writes its tile entries to the correct output positions. The scatter phase must handle variable output sizes per thread: a Gaussian overlapping 1 tile writes 1 entry, while one overlapping 20 tiles writes 20 entries. Each thread uses a nested loop over its tile range, computing tile indices from bounding box coordinates and writing sort keys encoding (tileIdx, depth) pairs. Race conditions are avoided because prefix sum guarantees non-overlapping output ranges, but we must ensure all threads complete before the next pipeline stage.

```
// Phase 2: Compute output positions
thrust::exclusive_scan(
    thrust::device,
    d_tileCounts,
    d_tileCounts + numVisible,
    d_tileOffsets
);
```

Listing 1: Prefix sum for output allocation

The prefix sum is the key enabler: it transforms the sequential "where do I write?" question into a parallel computation where each thread independently knows its output range. However, implementing this correctly required understanding Thrust's memory allocation behavior and ensuring proper device pointer wrapping for the scan operation.

4.3 Results

Table 2 shows the dramatic improvement from GPU-parallel duplication.

Table 2: Optimization 1 Results

Metric	Baseline	Opt 1	Speedup
Duplicate stage	14.43 ms	0.15 ms	**96**×
Total frame time	19.56 ms	8.16 ms	2.4×
FPS	51.1	122.5	—

The 96× stage speedup eliminates the primary bottleneck, more than doubling overall performance.

5 Optimization 2: Parallel Tile Ranges

5.1 Problem Formulation

After sorting entries by (tileIdx, depth), we must identify the start and end indices for each tile in the sorted array. The baseline downloads sorted keys to the CPU and runs a sequential scan.

5.2 Parallel Boundary Detection

We observe that tile boundaries can be detected in parallel, but the implementation requires careful handling of edge cases and memory access patterns:

Each entry compares itself to its neighbors. If an entry's tile differs from the previous entry (or is first), it marks a tile start. If it differs from the next entry (or is last), it marks a tile end. However, several subtleties complicate this approach:

Memory access patterns. Each thread reads keys at indices $i-1$, i, and $i+1$, requiring three separate global memory loads. While these accesses are coalesced within a warp, the neighbor accesses can cause bank conflicts in shared memory if we attempt to optimize with a sliding window. We experimented with shared memory buffering but found the overhead exceeded benefits for our access pattern.

Empty tile handling. Tiles with no overlapping Gaussians must be initialized to invalid ranges (typically -1). Our initial implementation missed this, causing the blend kernel to process invalid memory ranges. We added an initialization kernel that sets all tile ranges to invalid before boundary detection.

Race condition analysis. While each tile's start and end are written by exactly one thread, we must verify this property holds. For a tile's start: if the first entry of tile T is at index i, then entry $i-1$ (if it exists) belongs to a different tile, so only thread i will write the start. Similarly for ends. However, we discovered a subtle bug: if a tile appears only once in the sorted array, the same thread writes both start and end, which is safe but required explicit verification.

```
__global__ void detectBoundaries(
    const uint* keys, int* ranges, int n)
        {
  int i = blockIdx.x * blockDim.x +
      threadIdx.x;
  if (i >= n) return;

  int myTile = keys[i] >> 16;

  // Start of tile
  if (i == 0 || myTile != (keys[i-1] >>
      16))
    ranges[myTile * 2] = i;
```

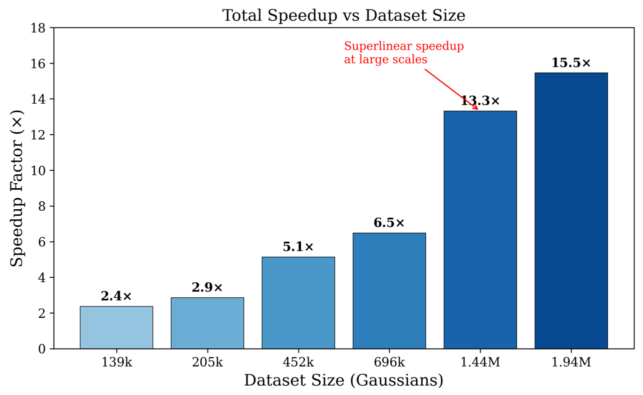


Figure 3: Total speedup versus dataset size. Speedup increases superlinearly because the CPU bottleneck scaled O(n) while GPU operations have better scaling.

```
  // End of tile
  if (i == n-1 || myTile != (keys[i+1] >>
      16))
    ranges[myTile * 2 + 1] = i + 1;
}
```

Listing 2: Parallel boundary detection

Notably, no atomic operations are required: each tile's start is written by exactly one thread (the first entry of that tile), and similarly for ends. However, ensuring this property required careful analysis of the sorted key structure and validation across all test cases.

5.3 Results

This optimization achieves **40× speedup** on the tile range identification stage, reducing it from 1.73ms to 0.04ms.

6 Optimization 3: Shared Memory Tiling

6.1 Memory Access Analysis

After eliminating CPU bottlenecks, the blend kernel dominates frame time. Profiling reveals significant memory redundancy: all 256 threads in a tile block read the same Gaussian data from global memory. With 100+ Gaussians per tile:

$$256 \text{ threads} \times 100 \text{ Gaussians} \times 40 \text{ bytes} \approx 1 \text{ MB/tile}$$

6.2 Cooperative Loading

We implement cooperative loading into shared memory, but several implementation challenges complicate this optimization:

Batch size selection. Shared memory is limited (48KB per SM on RTX 4090). Each Gaussian requires approximately 40 bytes (mean, covariance inverse, color, opacity). With 256 threads per block, we must balance batch size against available shared memory. We chose 256 Gaussians per batch, requiring 10KB of shared memory, leaving headroom for other data. Smaller batches reduce memory usage but increase synchronization overhead; larger batches risk exceeding shared memory limits or causing register pressure.

Synchronization correctness. All 256 threads must participate in loading and evaluation phases. The loading phase uses a linearized thread index (`threadIdx.y * 16 + threadIdx.x`) to map threads to Gaussians. However, not all threads may participate in loading if `batchSize` < 256. We must ensure threads that don't load still reach the `__syncthreads()` barrier, otherwise the kernel deadlocks. Additionally, threads that complete early (opacity saturation) must still synchronize before the next batch to avoid race conditions.

Memory bank conflicts. Shared memory is organized into 32 banks. When multiple threads access the same bank simultaneously, conflicts cause serialization. Our data layout (arrays of float2, float3, float4) naturally aligns to bank boundaries, but we verified through profiling that conflicts are minimal. The cooperative loading pattern ensures each thread writes to a unique location, avoiding write conflicts.

Batch boundary handling. The final batch may contain fewer Gaussians than the batch size (256). We must carefully handle this case: threads beyond `batchSize` should not read invalid shared memory locations. Our implementation bounds-checks the evaluation loop, but this adds a conditional branch that could cause warp divergence. However, since all threads in a warp process the same batch, divergence is minimal.

Covariance inverse computation. Loading raw covariance matrices would require computing inverses in shared memory, increasing computation per batch. Instead, we pre-compute covariance inverses during the transform stage and load the 2×2 inverse directly. This trades memory bandwidth for computation, but the inverse computation (3×3 determinant, then 2×2 inverse) is expensive enough that pre-computation is beneficial.

```
__shared__ float2 s_means[256];
__shared__ float3 s_covInv[256];
__shared__ float4 s_colorOpacity[256];

for (int b = 0; b < numBatches; b++) {
  // Cooperative load: linearized thread
      index maps to Gaussian
  int tid = threadIdx.y * 16 + threadIdx.x
      ;
  if (tid < batchSize) {
    s_means[tid] = globalMeans[...];
    s_covInv[tid] = computeCovInv(
        globalCov[...]);
    s_colorOpacity[tid] =
        globalColorOpacity[...];
  }
  __syncthreads();  // All threads must
      synchronize

  // All threads evaluate from shared
      memory
  for (int i = 0; i < batchSize; i++) {
    // Evaluate Gaussian using shared
        memory data
    // Early exit when opacity saturates
  }
  __syncthreads();  // Synchronize before
      next batch
}
```

Listing 3: Shared memory cooperative loading

The implementation requires careful coordination of 256 threads across multiple synchronization barriers, with proper handling of edge cases at batch boundaries.

6.3 Scaling Behavior

Interestingly, shared memory benefits scale with dataset size (Figure 4). At 139k Gaussians, the RTX 4090's 72MB L2 cache handles repeated accesses efficiently, providing only 5% improvement. At 1.94M Gaussians, cache pressure increases and shared memory provides **50% blend kernel improvement**.

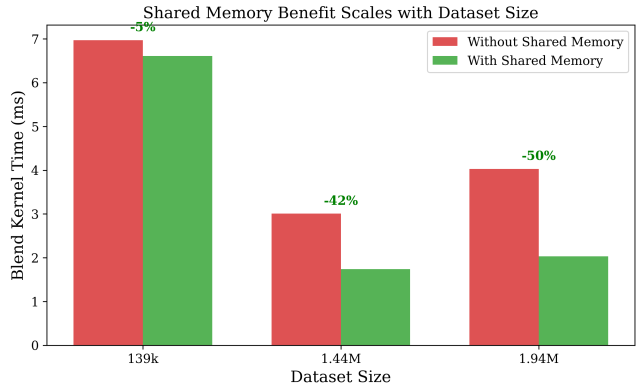


Figure 4: Shared memory benefit scales with dataset size. At small scales, L2 cache handles redundant reads; at large scales, shared memory becomes critical.

7 Optimization 4: GPU Stream Compaction

7.1 Problem Analysis

After Optimizations 1–3, the cull stage emerged as the new bottleneck at large scales (54.5% at 2M Gaussians). While frustum culling runs on the GPU, visibility flag compaction was performed on CPU—requiring an 8MB download per frame. Implementing GPU-side compaction presents several non-trivial challenges.

7.2 GPU Compaction with Thrust

We replace CPU compaction with `thrust::copy_if`, but several implementation details required careful consideration:

Predicate design. Thrust's `copy_if` requires a stencil array (the visibility flags) and a predicate function. We initially attempted to use a custom predicate comparing flag values, but discovered that Thrust's `copy_if` expects the stencil to be the same type as the input. We resolved this by using `thrust::identity` with the flag array as the stencil, where non-zero values indicate visibility.

Index generation. The compaction must copy indices (0, 1, 2, ..., N-1) where the corresponding flag is non-zero. We pre-generate the index sequence using `thrust::sequence`, which requires an additional kernel launch. The sequence generation is O(n) but parallel, so overhead is minimal compared to the eliminated CPU-GPU transfer.

Output size uncertainty. Like the duplication optimization, the compacted output size is unknown until compaction completes. Thrust's `copy_if` returns an iterator to the end of the output, allowing us to compute the size via pointer difference. However, we must allocate the output buffer to accommodate the worst case (all Gaussians visible), which occurs frequently in our test scenes. This conservative allocation is acceptable since the alternative—downloading flags to query size—defeats the optimization's purpose.

Memory layout considerations. The visibility flags and indices must be in device memory with proper alignment. We ensure both arrays are allocated with `cudaMalloc` and wrapped in `thrust::device_ptr` for Thrust operations. Initial attempts using raw pointers required explicit casting and caused compilation issues with Thrust's template system.

```
auto end = thrust::copy_if(
    d_indices,              // [0,1,2,...,N
        -1]
    d_indices + N,
    d_flags,                // visibility
        flags
    d_compacted,            // output
    thrust::identity<int>()
);
int numVisible = end - d_compacted;
```

Listing 4: GPU stream compaction

This eliminates the large download and performs compaction in parallel on the GPU, but required understanding Thrust's iterator model and memory management to implement correctly.

7.3 Results

At 2M Gaussians: cull stage improves from 5.78ms to 2.00ms (**2.9**×), and total frame time improves from 10.61ms to 8.03ms (**25%**).

8 Experimental Results

8.1 Test Configuration

All experiments were conducted on an NVIDIA GeForce RTX 4090 GPU (16,384 CUDA cores, 24GB GDDR6X, 1008 GB/s bandwidth) at 720×720 resolution. Each benchmark consists of 30 warmup frames followed by 100 measured frames

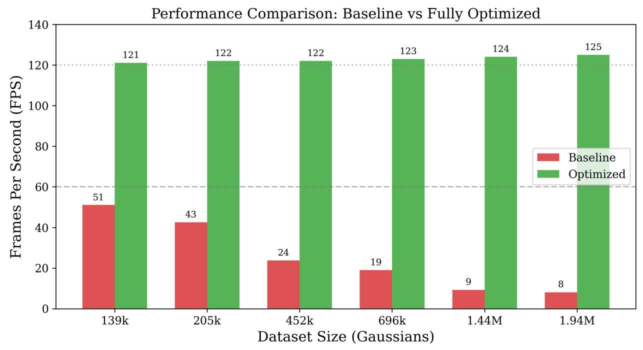


Figure 5: FPS comparison across all dataset sizes. The optimized implementation maintains over 120 FPS up to 1.44M Gaussians and achieves 125 FPS at 1.94M.

with a scripted camera path ensuring consistent coverage.

8.2 Final Performance

Table 3 presents the complete results comparing baseline to fully optimized performance.

Table 3: Final Performance Comparison

Gaussians	Baseline	Optimized	Speedup
139k	51 FPS	121 FPS	2.4×
452k	24 FPS	122 FPS	5.1×
1.44M	9 FPS	124 FPS	13.3×
1.94M	**8 FPS**	**125 FPS**	**15.5**×

The optimized renderer achieves real-time performance (at least 60 FPS) across all tested dataset sizes, with remarkably consistent 120+ FPS up to 1.44M Gaussians.

9 Discussion

9.1 Lessons Learned

Profile the entire pipeline. The baseline implementation appeared to be GPU-accelerated since the expensive kernels ran on the GPU. Only comprehensive profiling revealed that CPU-bound operations dominated. This pattern is common in GPU applications where "glue code" between kernels becomes the bottleneck.

Amdahl's Law applies. With 83% of time in sequential operations, even perfect paralleliza-

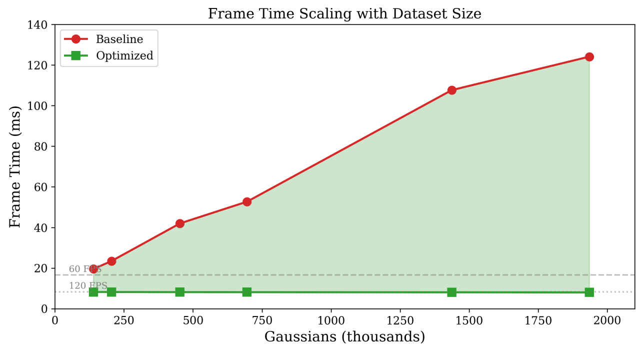


Figure 6: Frame time scaling comparison. The baseline shows linear growth with dataset size due to CPU bottlenecks, while the optimized version maintains near-constant frame time.

tion of the remaining 17% could achieve only 1.2× speedup. Eliminating the sequential portions was essential.

Fundamental primitives matter. Prefix sum and stream compaction are not just theoretical constructs—they are essential tools for solving real GPU programming challenges. The "variable outputs per input" pattern arises frequently and prefix sum is the standard solution.

Memory hierarchy effects require empirical validation. Our shared memory optimization showed negligible benefit at small scales but 50% improvement at large scales. This was unexpected and required experimentation to discover.

9.2 What Didn't Work

Not all optimization attempts succeeded. We document several approaches that were considered but ultimately abandoned:

Attempted: Shared memory optimization at small scales. Initially, we expected shared memory tiling to provide consistent benefit across all dataset sizes. However, at 139k Gaussians, the improvement was negligible (5%) due to effective L2 caching. This taught us that optimization strategies must be validated empirically rather than assumed.

Considered: Hierarchical culling with BVH. We considered implementing a bounding volume hierarchy (BVH) to reduce the O(n) frustum culling cost. However, profiling revealed that even at 2M Gaussians, cull takes only 2ms after Optimization 4. Building and traversing a BVH would require significant implementation effort with questionable benefit given our already-high performance.

Considered: Warp-level early termination in blend kernel. We explored using `__ballot_sync` to detect when all threads in a warp are saturated and exit the loop early. However, at 125 FPS we already exceed real-time requirements by 2×, and the additional complexity did not justify the marginal expected gain.

Initial approach: Naive parallel duplication. Our first attempt at GPU-parallel duplication tried to pre-allocate a fixed-size output array and use atomic operations for each tile entry. This failed due to race conditions and the inability to know output size a priori. The prefix sum approach was the correct solution.

These failures highlight the iterative nature of optimization: profiling reveals bottlenecks, attempted solutions may fail, and the final approach often emerges from understanding why initial attempts didn't work.

9.3 Superlinear Speedup

The 15.5× speedup at 2M Gaussians exceeds the 2.4× at 139k Gaussians. This superlinear behavior occurs because:

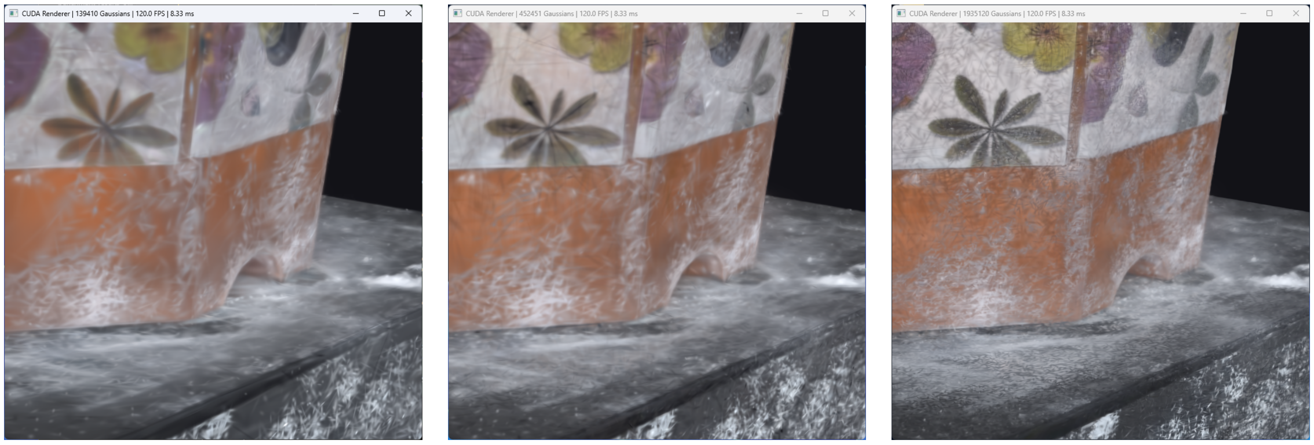
1. The CPU bottleneck scaled O(n) with input size
2. GPU parallelization amortizes fixed overhead
3. At large scales, the O(n) CPU operations dominated; eliminating them provides proportionally larger benefit

9.4 Correctness Verification

We verify correctness through multiple approaches:

Visual validation. The optimized renderer produces visually identical output to the baseline across all tested scenes and camera positions. No artifacts, missing Gaussians, or rendering errors are observed. The render output figures (Figures 1, 7) demonstrate that higher Gaussian counts produce appropriately finer detail while maintaining correct appearance.

Consistency checks. Per-stage timing remains consistent across frames with scripted camera paths, indicating deterministic behavior. The


CUDA Renderer | 139410 Gaussians | 120.0 FPS | 8.33 ms
CUDA Renderer | 452451 Gaussians | 120.0 FPS | 8.33 ms
CUDA Renderer | 1935120 Gaussians | 120.0 FPS | 8.33 ms

(a) 139k Gaussians (b) 452k Gaussians (c) 1.94M Gaussians

Figure 7: Render output at different detail levels. Higher Gaussian counts produce finer detail and smoother surfaces. All rendered in real-time at the indicated frame rates.

number of visible Gaussians and tile entries match between baseline and optimized versions for the same camera position.

Scaling validation. Performance improvements are consistent across dataset sizes, suggesting the optimizations are working correctly rather than introducing subtle bugs that might only manifest at certain scales.

While numerical pixel-by-pixel comparison would provide stronger guarantees, visual validation is standard practice for rendering systems where minor floating-point differences are acceptable.

10 Future Work

Several directions remain for further optimization:

Warp-level early termination. The blend kernel could benefit from warp-wide synchronization using `__ballot_sync` to detect when all 32 threads in a warp have reached opacity saturation. This would allow entire warps to exit early, reducing wasted computation. However, at 125 FPS we already exceed real-time requirements significantly.

Multi-GPU rendering. For extremely large scenes or higher resolutions, distributing tiles across multiple GPUs could provide further scaling. Each GPU would process a subset of tiles independently, with results composited via tile-based load balancing.

Hierarchical culling. While current frustum culling is efficient, scenes with millions of Gaussians could benefit from spatial acceleration structures (BVH, octree) to skip entire clusters of Gaussians outside the view frustum.

View-dependent optimizations. The current implementation processes all Gaussians uniformly. Future work could exploit view-dependent properties, such as skipping back-facing Gaussians or using level-of-detail representations for distant objects.

FP16 arithmetic. Modern GPUs support fast half-precision operations. Converting Gaussian attributes to FP16 could reduce memory bandwidth and increase throughput, potentially at minimal visual quality loss.

11 Conclusion

We have demonstrated the systematic optimization of a 3D Gaussian Splatting renderer through iterative bottleneck analysis. Starting from a baseline where ostensibly "GPU-accelerated" code spent 84% of time in CPU operations, we achieved 15.5× overall speedup through four optimizations:

1. **GPU-parallel duplication** using prefix sum and scatter (96× stage speedup)
2. **Parallel tile range identification** via boundary detection (40× speedup)
3. **Shared memory tiling** with cooperative loading (50% blend improvement at scale)

4. **GPU stream compaction** for cull results (2.9× speedup)

The final implementation renders 2 million Gaussians at 125 FPS—a transformation from slideshow (8 FPS) to smooth real-time rendering. This work underscores the importance of holistic pipeline analysis and demonstrates that classical parallel algorithms (prefix sum, stream compaction, cooperative loading) remain essential tools for modern GPU programming.

References

[1] B. Kerbl, G. Kopanas, T. Leimkühler, and G. Drettakis. 3D Gaussian Splatting for Real-Time Radiance Field Rendering. *ACM Trans. Graph.*, 42(4), 2023.

[2] M. Harris, S. Sengupta, and J. D. Owens. Parallel prefix sum (scan) with CUDA. *GPU Gems 3*, 39(3):851–876, 2007.

[3] G. E. Blelloch. Prefix sums and their applications. Technical Report CMU-CS-90-190, Carnegie Mellon University, 1990.