LoBE-GS: **Lo**ad-**B**alanced and **E**fficient 3D Gaussian Splatting for Large-Scale Scene Reconstruction

Sheng-Hsiang Hung[1], Jeng Wen Joshua Lean[1], Ting-Yu Yen[1], Wei-Fang Sun[2], Simon See[2], Shih-Hsuan Hung[1], Chih-Yuan Yao[3], Hung-Kuo Chu[1]

[1] National Tsing Hua University [2]NVIDIA AI Technology Center (NVAITC) [3]National Taiwan University of Science and Technology

Abstract

We present **LoBE-GS**, a load-balanced and efficient framework for large-scale 3D Gaussian Splatting. Unlike prior large-scale 3DGS methods such as CityGS, DOGS and VastGS, which focus on reconstruction and partitioning heuristics but ignore runtime imbalance, LoBE-GS uses a lightweight runtime-correlated proxy to steer partitioning. Guided by the proxy, we apply (i) load balance-aware scene partitioning to produce blocks with similar predicted runtime, (ii) fast camera selection to minimize partition overhead, and (iii) visibility cropping and selective densification to reduce per-block optimization. Across large-scale scenes, LoBE-GS achieves up to **2x** end-to-end training speedup over previous methods while preserving rendering quality. After training, we prune and merge the block outputs, export the unified model into USDZ format, and stream it via NVIDIA Omniverse Kit for real-time visualization, collaborative review, and interactive playback.

Motivation

Runtime goal. Partition–merge with parallel fine stage:

$$T_{E2E} = T_{coarse} + T_{partition} + \max_b T_{fine}.$$

⇒ **Minimize the slowest block**.

What predicts runtime? Correlation study across 5 datasets shows:

- Weak: block area $A^{(b)}$, in-block Gaussian count $G^{(b)}_{blk}$, camera count $C^{(b)}$, and average visible per camera $G^{(b)}_{avg\_blk}$.
- **Strong :** total **visible Gaussians** per block $G^{(b)}_{vis}$ **before** fine optimization.

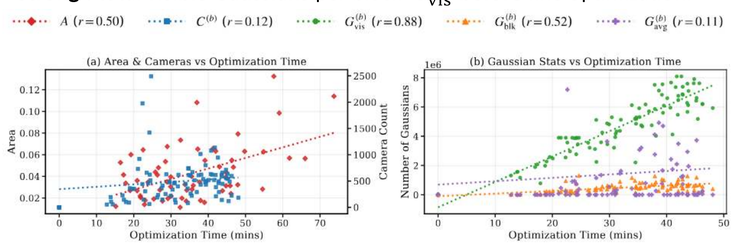

Figure 1. Correlation between per-block training time and block-level statistics under CityGS's partitioning.

Methodology

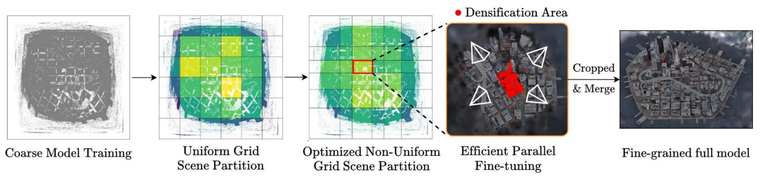

Figure 2. Overview of our pipeline

We train a coarse 3DGS model, optimize grid cuts with load-balance-aware partitioning, apply visibility cropping and selective densification during parallel fine-tuning, then prune outside regions and merge the blocks into a single high-quality model.

Load Balance-Aware Scene Partition (LB-SP)

Goal: Minimize the slowest block's cost by balancing initial visible Gaussians per block (a strong runtime proxy).
How: Bayesian Optimization (BO) over vertical/horizontal cuts with ordering constraints; evaluate maximum visible Gaussians each round; pick best cuts.
Implementation note: visible Gaussians per block computed in **NVIDIA Warp** for near-CUDA speed across BO iterations.

Fast Camera Selection (FCS)

Key idea: Only **N** projections: render per-view depth once from the coarse model, back-project to a dense point cloud, then count points inside each block to get a visibility ratio
Assign views: For each block, select cameras whose visibility of that block exceeds a chosen threshold, and reuse those per-view point clouds across BO iterations.

Visibility Cropping (VC) & Selective Densification (SD)

VC: Before fine-training block b, keep only Gaussians visible from its assigned **cameras** instead of the full coarse model → big cut in optimizer updates and memory.
SD: During fine-training, densify only inside the block; visible-but-out-of-block Gaussians are optimized but not densified → fewer new points, faster training.

Experiments

All Experiments ran on 10 nodes on **OVX server** (8 **NVIDIA L40** GPUs/node, 80 GPUs total) via **Run:ai** platform.

Methods	MatrixCity-Aerial			Mill19			UrbanScene3D		
	PSNR	SSIM	LPIPS	PSNR	SSIM	LPIPS	PSNR	SSIM	LPIPS
3DGS†	23.67	0.735	0.384	22.97	0.749	0.291	21.25	0.814	0.239
CityGS	27.46	0.865	0.204	23.66	0.796	**0.237**	**21.70**	0.825	0.221
Ours	**27.74**	**0.875**	**0.186**	**23.87**	**0.797**	0.240	21.33	**0.826**	**0.213**

Table 1. Quantitative Comparison

Methods	MatrixCity-Aerial			Mill19			UrbanScene3D		
	PSNR	SSIM	LPIPS	PSNR	SSIM	LPIPS	PSNR	SIM	LPIPS
VastGS†	28.33	0.835	0.220	23.50	0.735	0.245	21.83	0.730	0.261
DOGS	28.58	0.847	0.219	24.26	0.762	**0.231**	23.18	0.772	0.232
Ours	**28.91**	**0.879**	**0.187**	**24.68**	**0.795**	0.241	**23.55**	**0.832**	**0.213**

Table 2. Quantitative Comparison on color-correct metrics

Methods	MatrixCity-Aerial				Mill19				UrbanScene3D			
	T_{coarse}	$T_{partition}$	$\max T_{fine}$	T_{E2E}	T_{coarse}	$T_{partition}$	$\max T_{fine}$	T_{E2E}	T_{coarse}	$T_{partition}$	$\max T_{fine}$	T_{E2E}
3DGS†	01:50	–	–	01:50	01:20	–	–	01:20	01:01	–	–	01:01
VastGS†	–	00:48	01:13	02:01	–	**00:05**	00:42	**00:47**	–	00:17	00:40	00:57
CityGS	00:52	01:39	01:00	03:31	01:03	00:15	01:10	02:28	00:43	00:20	01:04	02:07
Ours	**00:38**	**00:16**	**00:30**	**01:24**	**00:24**	00:07	**00:36**	01:07	**00:21**	**00:07**	**00:28**	**00:55**

Table 3. End-to-End time comparison

Interactive 3DGS Map via Omniverse Streaming

Challenge: Large Scale 3DGS (e.g., 50M Gaussians, 8 GB) cannot run on phones/laptops — client rendering is 1–4 FPS or unusable.
How: Browser-based rendering using **RTX** and **NuRec** via **Omniverse Streaming** on an RTX 5090, with USDZ assets converted by **3DGRUT**. WebRTC streams 30 FPS video to a Three.js frontend. UE5/OSM metadata (building boxes, paths) and invisible Three.js overlay objects enable clickable scene interactions and road-level navigation. Low-latency camera transform sync keeps frontend/backend views aligned.
Result: Stable **30 FPS** on laptops & phones (≈**7–30**x speedup vs client-side Three.js).

Figure 3, 4. Interactive 3DGS map of NTHU campus