Optimizing 3D Gaussian Splattering for Mobile GPUs

Md Musfiqur Rahman Sanim
University of Georgia
Athens, GA, USA
musfiqur.sanim@uga.edu

Zhihao Shu
University of Georgia
Athens, GA, USA
Zhihao.Shu@uga.edu

Bahram Afsharmanesh
University of Georgia
Athens, GA, USA
bahram.afsharmanesh@uga.edu

AmirAli Mirian
University of Georgia
Athens, GA, USA
AmirAli.mirian@uga.edu

Jiexiong Guan
William & Mary
Williamsburg, VA, USA
jguan@wm.edu

Wei Niu
University of Georgia
Athens, GA, USA
wniu@uga.edu

Bin Ren
William & Mary
Williamsburg, VA, USA
bren@wm.edu

Gagan Agrawal
University of Georgia
Athens, GA, USA
gagrawal@uga.edu

***Abstract*—Image-based 3D scene reconstruction, which transforms multi-view images into a structured 3D representation of the surrounding environment, is a common task across many modern applications. 3D Gaussian Splatting (3DGS) is a new paradigm to address this problem and offers considerable efficiency as compared to the previous methods. Motivated by this, and considering various benefits of mobile device deployment (data privacy, operating without internet connectivity, and potentially faster responses), this paper develops `Texture3dgs`, an optimized mapping of 3DGS for a mobile GPU. A critical challenge in this area turns out to be optimizing for the two-dimensional (2D) texture cache, which needs to be exploited for faster executions on mobile GPUs. As a sorting method dominates the computations in 3DGS on mobile platforms, the core of `Texture3dgs` is a novel sorting algorithm where the processing, data movement, and placement are highly optimized for 2D memory. The properties of this algorithm are analyzed in view of a cost model for the texture cache. In addition, we accelerate other steps of the 3DGS algorithm through improved variable layout design and other optimizations. End-to-end evaluation shows that `Texture3dgs` delivers up to 4.1× and 1.7× speedup for the sorting and overall 3D scene reconstruction, respectively – while also reducing memory usage by up to 1.6× – demonstrating the effectiveness of our design for efficient mobile 3D scene reconstruction.**

***Index Terms*—3D Gaussian Splatting, Mobile Computing, GPU Processing, Texture Memory, Parallel Sorting**

I. Introduction

Image-based 3D scene reconstruction transforms multi-view images into a structured 3D representation of the surrounding environment. This has emerged as a cornerstone technology with a wide range of applications, ranging from robotics [1], [2], augmented reality (AR) [3], [4], to autonomous systems [5], [5], [6]. This has been a very active area of research – the success of deep learning models in image-based 3D scene reconstruction led to the development of novel view synthesis (NVS) models, which can predict novel views of a scene from a set of input images. Approaches following this direction, such as multi-view stereo (MVS) [7]–[10] and implicit methods (e.g., Neural Radiance Fields or NeRF [11]–[15]) have shown impressive results in generating high-quality 3D reconstructions. However, their computational inefficiency leads to high resource requirements. When these methods need to support an application on a mobile, edge, or other resource-constrained platform, the only practical choice is to use it as a server or cloud for execution [16].

3D Gaussian Splatting (3DGS) represents a paradigm shift in scene representation. This approach involves structuring the surrounding environment with adaptive and learnable 3D Gaussian primitives, parameterized by spatial position, covariance (defining anisotropic spread), opacity, and view-dependent spherical harmonic color coefficients [17]. Compared to the methods listed earlier like Neural Radiance Fields (NeRF) [18], 3DGS leverages explicit splatting operations to project Gaussian kernels onto the image plane through differentiable affine transformations. This formulation not only has reduced computational requirements, but also inherently supports parallelization, enabling faster rendering speeds while maintaining photorealistic fidelity [19].

The efficiency of 3DGS naturally leads to its consideration for platforms such as the mobile devices. On (mobile) device processing of deep learning tasks has been a popular direction in recent years [20]–[22] – besides advantages such as data privacy and support for operations with low or even no internet connectivity [23], [24], mobile device processing can help support latency-critical tasks [25], [26]. The use of scene reconstruction in applications such as dynamic obstacle avoidance for autonomous drones or responsive AR applications [27] has latency requirements as low as 20 ms [28]. Such a requirement cannot be met typically by sending data/query to a server and receiving a response.

The deployment of 3DGS on mobile architectures introduces fundamental challenges due to hardware-software asymmetries. Splatting operations such as sorting are inherently memory-intensive, and in comparison, mobile memory subsystems are constrained by narrow LPDDR5/X buses (¡ 50 GB/s bandwidth) [29]. Particularly, unlike desktop-level GPUs that typically rely on scratch buffer or shared memory to achieve

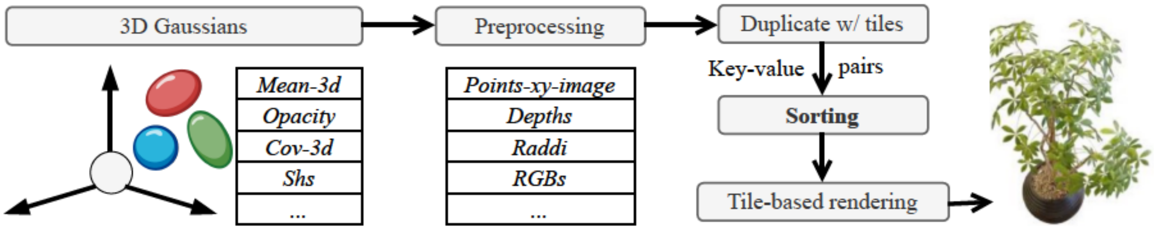


Fig. 1: 3DGS rendering pipelines.

high performance, mobile GPU applications achieve better performance by exploiting texture memory (and associated cache) [29], [30]. However, the two-dimensional nature of the cache requires new approaches to designing algorithms and tuning the implementations.

There is a line of work on optimizing 3DGS for desktop GPUs, focusing on topics like efficient Gaussian point pruning [31], optimizing memory access patterns [16], and efficient rendering [32]. These approaches are either not directly applicable, and/or still leave significant challenges, for the following two reasons: 1) Mobile GPUs are better suited for tile-based rendering, i.e., have higher memory locality requirements; and 2) the limited unified memory hierarchy in mobile GPUs intensifies contention during rasterization of overlapping Gaussians.

Addressing the challenges that we have discussed, this paper makes the following contributions.

- We introduce a novel sorting algorithm optimized for GPUs for 2D texture memory. Our method improves on the limited previous work on sorting with texture memory [33] and achieves significantly better cache reuse by careful index transformation, ensuring that pair of values compared (and potentially swapped) in each step are adjacent.
- We also design schemes for variable packing and layout organization, in view of the use of different data structures in the entire application, GPU-based parallel processing, and properties of texture memory.
- Further adding a number of optimizations, we implement a complete mobile-optimized 3DGS pipeline, namely `Texture3dgs`.

`Texture3dgs` has been extensively evaluated on different off-the-shelf mobile platforms, covering representative 3D Gaussian Splatting (3DGS) workloads across various scenes and model complexities. Compared to state-of-the-art baseline implementations, our optimized sorting algorithm achieves up to 4.1× performance improvements by effectively utilizing 2D texture caches. Furthermore, the full implementation of `Texture3dgs` demonstrates up to 1.7× end-to-end speedup, alongside memory savings of up to 1.6×, achieved through efficient variable packing and data layout organization, highlighting the practical potential of our techniques to enable efficient, real-time 3D scene reconstruction applications on resource-constrained mobile platforms.

II. Background

A. 3D Gaussian Splatting Rendering Pipeline

As stated earlier, explicit representations like 3D Gaussian Splatting (3DGS) [19] have emerged as efficient alternatives to prior methods, particularly the Neural Radiance Fields (NeRF) introduced by Mildenhall et al. [18]. For scene reconstruction, this method efficiently converts a given camera viewpoint and 3D Gaussian primitives into a rendered 2D image through four main pipeline stages (shown in Figure 1):

Preprocessing. In this stage, the camera parameters and 3D Gaussian properties, i.e., position (mean), opacity, covariance matrix, and spherical harmonics, are processed to calculate the visual attributes of each Gaussian. A step called *Frustum culling* eliminates Gaussians outside the camera view. The remaining Gaussians are projected onto the 2D image plane, forming elliptical footprints with updated attributes necessary for rasterization.

Duplication with Tiles. The image plane is subdivided into 16×16 tiles. Each projected 2D Gaussian ellipse is represented by an axis-aligned bounding box (AABB). Gaussians are duplicated across all tiles that intersect their AABB. These duplicates form *key-value* pairs, indexed by a combination of the tile identifier and Gaussian depth (i.e., distance from the camera).

Sorting. The duplicated 2D Gaussians are first sorted based on tile indices to group Gaussians belonging to the same spatial region. Within each tile, Gaussians are further sorted by their depth values to ensure correct rendering order for transparency and occlusion.

Rendering. During the final rendering step, the color and opacity values from sorted 2D Gaussians are composited through a step called *alpha blending*, determining the final color of each pixel within a tile. This structured pipeline leverages GPU rasterization for efficient, high-quality, real-time rendering of complex scenes.

B. Mobile GPUs and Texture Memory

Modern mobile GPUs utilize a specialized memory hierarchy optimized for energy efficiency and reduced memory bandwidth usage [34]. Unlike desktop GPUs, mobile GPUs primarily employ a 2.5D texture memory, with a corresponding read-only cache (Figure 2). The 2D part of the memory signifies support for spatial locality along two dimensions, while the .5 part implies that each *texture element* (also called a *texture point*) is actually a vector of length 4. Two

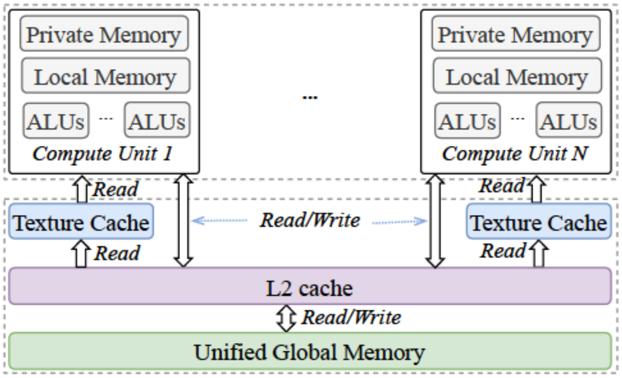


Fig. 2: Mobile GPU memory hierarchy.

key design aspects critically affect texture memory performance: texture representation and cache organization [35]–[37]. Unfortunately, both are often patented and not completely understood for mobile platforms [38]. Texture representation refers to how 2D texture data is stored in memory, including the *data unit* (e.g., *2D blocks*), their *layout* (e.g., hierarchical blocking), and the *storage order* (e.g., row-major, column-major, zigzag, or Hilbert). The cache organization covers how data is fetched between the texture cache and main memory, including mapping, replacement policies, and other hardware-specific strategies. The support for 2.5D texture memory significantly reduces memory bandwidth requirements but imposes constraints on how memory accesses should be optimized. More specifically, adapting algorithms such that their data access patterns match the properties of the texture cache can be a significant challenge.

C. Sorting on GPUs

Sorting is a fundamental and one of the most widely studied problems in computer science. Traditional CPU-based sorting algorithms suffer from significant performance limitations on GPUs due to inadequate parallelism and frequent cache misses. Researchers have explored GPU-based sorting, leveraging its massive parallel processing capabilities by adapting sequential sorting algorithms into suitable parallel implementations [39]–[42]. Early GPU-based sorting algorithms were predominantly derived from sorting networks, such as Batcher's bitonic and odd-even merge sort [43] and Dowd's periodic balanced sorting networks [44], the latter being inspired by the bitonic sort network. While the bitonic sort network is well-suited for parallel processing, it does have a high memory access complexity of $O(n \log^2 n)$.

Most recent research in GPU-based sorting has primarily focused on NVIDIA GPUs and CUDA-based implementations (and largely targeting integer arrays) [42], [45], [46]. With advancements in GPU architectures and the introduction of shared memory, sorting strategies evolved. Instead of applying a global sorting network, modern approaches first partition the sequence into sub-sequences, which are independently sorted in parallel. These sorted sub-sequences are then merged in parallel to construct the final sorted sequence [47], [48]. For a comprehensive overview of advancements in parallel sorting techniques, readers can refer to the surveys in [49], [50].

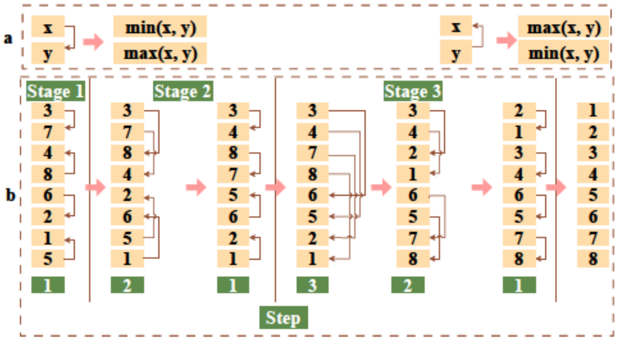


Fig. 3: Bitonic sorting network illustration: (a) comparator operations, where two elements are compared and swapped based on their order, and (b) sorting process across multiple steps, where at each stage, the array is conceptually partitioned into sorted segments (length of 2^{step}). The arrows indicate the comparisons and exchanges at each step.

III. Texture Cache Friendly Sorting

A. Sorting on Texture memory

There is very little work on optimizing sorting for mobile texture memory. The original approach by Govindaraju et al. [51] targeted texture memory/cache on desktop GPUs. This work was later extended into GPUTeraSort [41], which we will carefully examine below. Since then, there has been minimal development of GPU sorting algorithms specifically optimized for mobile architectures. Notably, popular mobile deep learning frameworks, including MNN [52] and NCNN [53], do not offer specialized sorting operations for texture memory.

GPUTeraSort leveraged Dowd's Periodic Balanced Sorting Network (PBSN) [44] to achieve improved memory usage while sorting large key-value datasets. As background for presenting the existing work and its improvements toward our algorithm, we review the well-known bitonic sorting method. The key concept here is a *bitonic sequence*, which is a sequence that is either entirely non-decreasing or non-increasing. The algorithm starts with the input array $a = (a_0, a_1, \ldots, a_{n-1})$ and works from the bottom up, gradually merging smaller bitonic sequences of equal size. Initially, pairs of adjacent elements, like (a_{2i}, a_{2i+1}), are merged to form bitonic sequences of size 2. In the next stage, these smaller sequences are merged into larger ones of size 4, and this continues with the size doubling at each stage. To obtain a fully sorted list, we require $\log(n)$ stages. Figure 3 illustrates the full bitonic sort network process.

GPUTeraSort, which maps this process to GPUs with texture memory, is summarized as Algorithm 1. If n is the number of values to be sorted, the algorithm proceeds in $log(n)$ *stages*. In the i^{th} stage, there are i *steps*, executed in reverse order from i down to 1. Each step is responsible for building and merging bitonic sequences of size 2^i, progressively combining smaller

Algorithm 1 GPUTerasort's Bitonic sort process

```
procedure BitonicSort($texture$, $W$, $H$)
    $n \leftarrow$ numValues to be sorted $\leftarrow W \times H$    ▷ single array representation
    for $i = 1 \rightarrow \log n$ do    ▷ for each stage
        for $j = i \rightarrow 1$ do
            Quad size $B = 2^{j-1}$
            Compare and swap within the Quads of size $B$
            Copy from frame buffer to $texture$
        end for
    end for
end procedure
```

TABLE I: Weights in machine learning model developed for relating two-dimensional strides to latency.

Block Size	2	4	8	16	32
Cross Horizontal Block	0.64	0.03	0.26	0.55	0.40
Cross Vertical Block	0.81	0.62	0.87	0.29	0.4

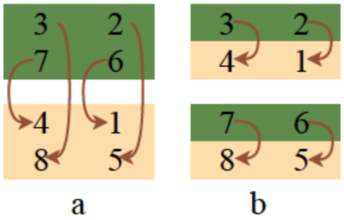


Fig. 4: GPUTeraSort's sort a) stage 2, step 2 and b) stage 2, step 1 – the quad sizes are 4 and 2, respectively. During each comparison step, a value from the green region is paired with its corresponding value from the yellow region. After comparison, the minimum and maximum values are placed in the corresponding green and yellow positions, respectively.

sorted sequences into larger ones until the entire array is sorted. The algorithm operates on texture memory by reading from an input texture and writing to an output texture. The output texture of one stage becomes the input texture for the next stage, while the former input texture is now free to store the new output. The key concept here is the *quad size*–specifically, in *compare and swap* using a quad of size B, a location from one quad is compared (and potentially swapped) with a corresponding location in the next quad (with specific pairing chosen to minimize divergence). This process is illustrated in Figure 4. Each of these quads is processed in a separate kernel to achieve parallelism.

To understand how these quads are physically allocated in 2D memory with optimal layouts, with n being the number of values that need to be sorted, the texture dimensions W and H are set to powers of 2 that are each closest to $\sqrt{n}$ (and such that $W \times H$ is closest to n). When working with a quad size of B, the texture is segmented as follows. Each quad occupies a continuous dimension when $B < H$; otherwise, the quad forms a block with dimensions H and B/H. The implications of this allocation will be discussed after we present a cost model for texture memory.

Several details of texture cache architecture, such as specific replacement policies or exact block mappings, are typically proprietary. Our algorithm relies on a precise offline profiling (as detailed in the next section) involving micro-benchmarking and modeling of cache performance to accurately capture mobile GPU characteristics. This approach enables us to approximate hardware behavior effectively ensuring robustness across multiple mobile GPUs (as validated in our Section V-E).

B. Cost Model for Texture Memory Based Processing

For predicting the performance of an algorithm that accesses the texture cache, we refer to a recent study [30]. As background, in a 2D texture cache, data is typically organized in *2D blocks*, enabling data locality along both the width and height dimensions [37]. However, because the exact architectural details are not publicly known, this empirical study leveraged a set of micro-benchmarks involving random accesses to establish the relationship between data accesses and memory latency for a single thread.

In this process, inspired by pointer-chasing benchmarking [54], two-dimensional random access indices were generated offline. This step takes a list of possible strides as input and constructs a multinomial distribution, with each benchmark execution assuming a different multinomial distribution to ensure randomness. Next, a micro-benchmark kernel was used to measure memory latency values for a set of random strides. This micro-benchmark kernel operates in a pointer-chasing style, i.e., it fetches a pixel and uses its value as 2D strides for subsequent accesses.

Next, the concept of *cross-block stride* is introduced – it is defined as a function of the shape and size of the data block, and is *the stride that goes across distinct data blocks* (under the assumed shape and size of the block). The benchmarking process collects the cross-block strides for various assumed data block shapes and sizes, along with the execution latency for each run. This information is used as the input feature for the subsequent machine learning model. This leads to the training data (H, L), where H is the histogram of cross-block strides for the assumed data block shapes and sizes, and L is the profiled latency for each run. The collected training data are fed into a machine learning regression model based on the least squares method.

The results have shown that the performance can be effectively captured in terms of cross-block stride-based summarization of data accesses [30]. In one of the experiments, the values obtained are shown in Table I. These results show that, on average, there is not a large difference between horizontal and vertical accesses. Moreover, the latency can be fully captured by considering horizontal or vertical strides of up to 32. For the analysis of cache performance of algorithms on texture memory, we can assume an abstract two-dimensional block size b, such that "cache misses" occur when a stride crosses the two-dimensional block.

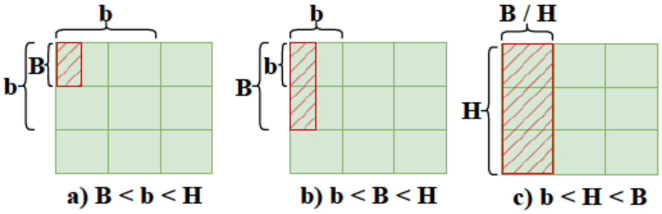


Fig. 5: GPUTeraSort's texture memory allocation for different quad sizes: b is the texture cache block size, B is the quad size, W and H are texture dimensions. This figure shows three cases: (a) quad is along a single dimension, with length less than b, (b) the quad is along a single dimension, with length greater than b, and (c) the quad is two-dimensional with length H, and width B/H.

C. Optimizing Texture Memory Based Sorting

It is easy to see that the original TeraSort algorithm is not optimized for the modern texture cache. If the quad size is B, the distance between two compared pixels turns out to be an average of B, as shown in Figure 4. While GPUTeraSort optimizes data access when $B < b$, since all comparisons happen within a single texture cache block (Figure 5*a*), this advantage diminishes when $B \geq b$ (Figure 5*b* and *c*), because the values being compared are in different cache blocks. This leads to a large number of misses in the texture cache. In addition, it should be noted that, because of the read-only nature of the texture cache, the initial reading of a quad always causes cache misses.

In view of this analysis, we now present a sorting algorithm optimized for texture memory on modern mobile GPUs. The key idea in this work is a *layout transformation*, which we explain first.

Layout Transformation. Our sorting algorithm is designed to take full advantage of the texture cache's features by keeping the comparing pairs for each stage physically close in memory. This concept is illustrated in Figures 6. The core idea is as follows: if a texture point $\boldsymbol{a}$ at coordinate (x, y) is to be compared with another point $\boldsymbol{b}$, then $\boldsymbol{b}$ is placed at either $(x, y+1)$ or $(x+1, y)$. To make this possible, we apply a layout transformation at every sorting step. This layout transformation is applied while the output tensors are written. As a result, the layout transformation does not cause any additional data movement, though there can be a cost associated with index transformation-related computations. This layout transformation is feasible because of the predefined structure of the bitonic sorting network.

The concept behind the index transformation can be shown through four steps in Figure 7, i.e., `Slice`, `Concat`, `Segment Swap`, and `Reshape`. Initially, the texture is *sliced* by grouping every two consecutive rows and concatenating each set of (odd or even) rows. As a result, all even-numbered rows are *concatenated* into the first group, and all odd-numbered rows into the second group. Next, each of these groups is divided into *segments*, where the segment size is determined by the quad size at the next step, specifically, $k = 2^{\text{step}-2}$. Following this, a *segment swap* is performed between the two groups – the odd-numbered segments from the first group are swapped with the even-numbered segments from the second group, i.e., we perform the operation:

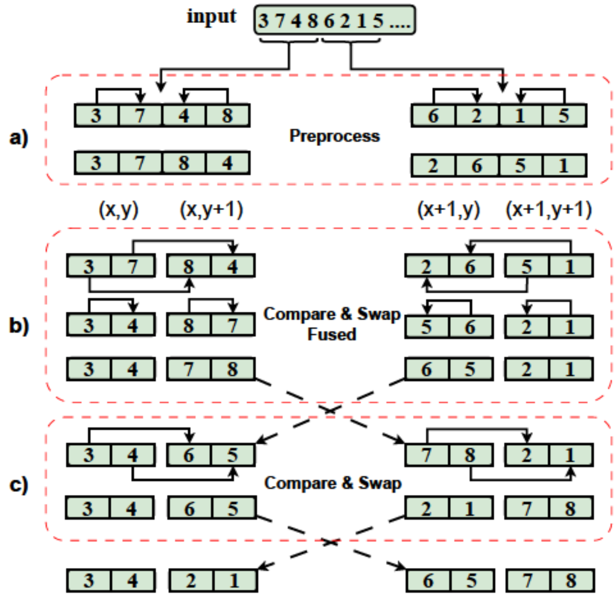


Fig. 6: Sorting pipeline: (a) Stage 1 comparison and swap operations; (b) Stage 2, steps 2 and 1 comparison and swap operations—all comparisons are with immediate vertical (step 2) or horizontal (step 1) neighbors only (the same process occurs in Stage 3, steps 2 and 1 as well); and (c) Stage 3, step 3 – all comparisons and swaps are with immediate vertical neighbors only. Solid arrows represent compare-and-swap operations, and dotted arrows represent data movement.

$$\text{swap}(first\_group\_segment_i, second\_group\_segment_{i-1}) \\ i \in \{1, 3, 5, \ldots\} \qquad (1)$$

After the segment swap, the elements of the group are reshaped back to their original texture width and height. This reshaping ensures that the comparing pairs are positioned correctly for the next sorting step. In the resulting layout, every element at coordinate (x, y) in an even-numbered row is directly aligned with its comparison partner at $(x, y+1)$, aligning perfectly with the comparisons needed at this step in the original bitonic sort network. It is important to emphasize that the above four steps are conceptual, intended to present the method. For actual implementation, we apply an *index transformation* along with the compare-and-swap operation of the previous step. This results in the placement of each value into its correct position for the next sorting step. Although this introduces some additional index computation costs, the overhead is minimal compared to the performance gain from improved cache efficiency, which we analyze later. This trade-

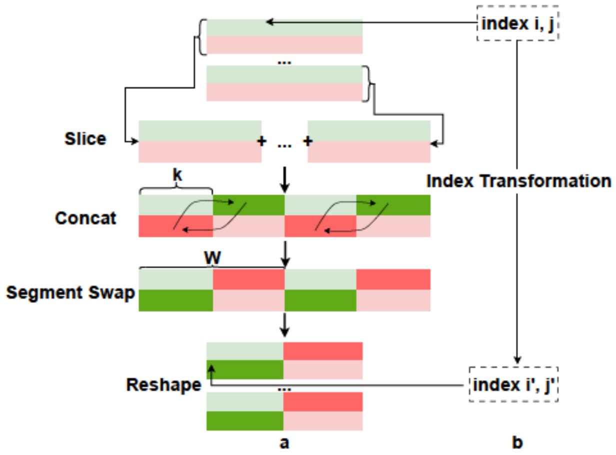


Fig. 7: Layout transformation (logical view). k is segment length and W is the original texture width dimension. Even and odd parts are shown in green and red, respectively, with swapped segments highlighted.

off is especially favorable in the context of GPU sorting, where performance is typically bound by memory accesses rather than computations.

Stage Fusion. Another optimization targets the fact that each texture point has 4 values. For each stage of the bitonic sort, each kernel thread reads two textures, each containing two key-value pairs, for a total of four key-value pairs. In the last two stages, only these four elements are required for that particular step. Therefore, instead of using two separate kernels for the last two stages, we can combine them into a single kernel. Given that the four elements share the same vector, this reduces the total memory traffic.

Algorithm 2 Sorting Pipeline

```
1:  procedure SortingPipeline($keys, values$)
2:      $n \leftarrow$ array size of keys
3:      $x \leftarrow ceil(log_2(n))$
4:      Choose dimensions $H$ and $W$
5:      $tex[2][W][H] \leftarrow init$
6:      $texPoint \leftarrow 0$
7:      $PreprocessKernel(keys, values, tex[texPoint])$
8:      for $i = 2 \rightarrow x$ do                                   ▷ stage
9:          for $j = i \rightarrow 3$ do                               ▷ step
10:             $CompareSwap(tex[texPoint], tex[1 - texPoint], i, j)$
11:             $texPoint \leftarrow 1 - texPoint$
12:         end for
13:         $CompareSwapFused(tex[texPoint], tex[1 - texPoint], i)$
14:         $texPoint \leftarrow 1 - texPoint$
15:     end for
16: end procedure
```

Overall Algorithm. With these two key novel optimizations, the overall sorting method, presented as Algorithm 2 operates as follows. The process begins with the *PreprocessKernel* (Line 7), which prepares the input data by organizing it into a newly structured texture layout and performing the first stage of bitonic sorting. This initial setup is detailed in Figure 6(a). Each *PreprocessKernel* thread reads 4 keys and 4 values as vectors of 4 – this size supports SIMD processing. The pipeline then continues through the remaining $(x - 1)$ sorting stages (Line 8). In the i^{th} stage, a total of i steps are performed: the first $(i - 2)$ steps are executed using the *CompareSwap* while the final two steps are fused into a single pass using the *CompareSwapFused* (Algorithm 2, Line 13). The method *CompareSwap* (Algorithm 3) reads two row-adjacent texture points, determines their sorting order according to the bitonic sorting network, swaps them if necessary, computes the updated indices, and writes them back to the output texture. As stated previously, the last two stages are merged together – the primary difference between ***CompareSwap*** and ***CompareSwapFused*** is that the latter performs an additional *intra-texture point* comparison step.

Algorithm 3 Kernel for Sorting with Texture Memory

```
procedure ComapareSwap($inTex, outTex, stage, step$)
    $upper\_point \leftarrow inTex[x][y << 1]$
    $lower\_point \leftarrow inTex[x][y << 1 + 1]$
    $dir \leftarrow get\_direction(x, y)$
    $compare\_and\_swap(upper\_point, lower\_point, dir)$
    Calculate index for next stage and write to $outTex$
end procedure
```

Cost Analysis. Our advantages arise because of both layout transformation and stage fusion. Due to this fusion, our approach significantly reduces the number of memory accesses compared to the baseline texture-based sorting algorithm, GPUTeraSort [41]. The total number of memory accesses required by our algorithm for sorting is given by $n\log(n) \times \frac{\log(n)-1}{2}$. In contrast, the total memory accesses required by Govindaraju et al. [51]:5 $\times\, n\log(n) \times \frac{\log(n)+1}{2}$ because they do not have any stage fusion. In terms of texture cache hits and misses – as discussed earlier, the L1 texture cache is optimized for 2D spatial locality, meaning that data stored in a square block of dimension size b can be efficiently accessed with minimal cache misses. Given a texture of width W and height H, the theoretical minimum number of cache misses is for any computation that traverses all the data is $\frac{W \times H}{b^2}$. In our sorting kernel, each comparison requires two data reads, and as explained, these two data points should be close. Thus, unlike the original algorithm, we achieve a cost close to the minimum.

IV. Design of `Texture3dgs`

This section outlines the key optimizations in our `Texture3dgs` implementation. Building on the sorting algorithm discussed earlier, we detail how it has been further optimized for use within `Texture3dgs`.

A. Variable Packing to Exploit Texture Memory

As we discussed while presenting our sorting algorithm, designing an appropriate data layout that adheres to the

TABLE II: Grouped parameter input of preprocessing.

Group	Parameter Name	Number of data points	Number of Texture
Group 1	Mean3d	3	1
	Opacity	1	
Group 2	Cov3Ds	6	2
Group 3	Shs	48	12

TABLE III: Comparison of parameters across operations.

	PrefixSum	DuplicateWithTile	Render
Points_xy_image		Yes	Yes
Depths		Yes	Yes
Raddi		Yes	Yes
Conic_Opacity			Yes
RGBs			Yes
Tile_touched	Yes		

TABLE IV: Grouped parameter output of preprocessing step.

Group	Parameter	No. of data points	Texture Points
Group 1	Points_xy_image	2	1
	Depths	1	
	Raddi	1	
Group 2	Conic_opacity	4	1
Group 3	Rgbs	3	1
Group 4	Tiles_touched	1	n/a

TABLE V: Texture layout for different input groups. n represents the number of Gaussians.

Group	Size	Width	Height	Block
Group 1	n	$\lceil\sqrt{n}\rceil$	$\lceil \text{size}/\lceil\sqrt{n}\rceil\rceil$	(x, y) to (x, y)
Group 2	$2n$	$\lceil\sqrt{n}\rceil \times 2$	$\lceil \text{size}/\lceil\sqrt{n}\rceil\rceil$	(x, y) to $(x, y+1)$
Group 3	$12n$	$\lceil\sqrt{n}\rceil \times 3$	$\lceil \text{size}/\lceil\sqrt{n}\rceil\rceil \times 4$	(x, y) to $(x+3, y+4)$

dimensional limits of texture memory is crucial to ensure efficiency during texture read operations. To this end, the width and height of a texture should be configured to approximate a square shape. The key data structure in the code is the set of Gaussians – each Gaussian is represented by 58 data points, as detailed in Table II. During the preprocessing stage of 3D Gaussian rasterization, these data points must be read and processed, resulting in 12 data points being output for subsequent kernel operations (please see Table IV).

Input Data Organization. A number of possibilities can be considered for the layout of Gaussian-related parameters in texture memory. One possibility is storing parameters for different Gaussians sequentially across the four texture channels. However, with these alignments, as each thread is assigned to process a single Gaussian, a significant amount of unused data is read. Another possibility is assigning a single thread to process multiple Gaussians to fully utilize the texture data being read – however, this would reduce the total amount of parallelism in the implementation. As a result, we utilize the grouping strategy outlined in Table II, which reduces the total number of read operations from 58 to 15. As Group 1 shown in Table II contains all the necessary information for the parameters of a single Gaussian, the texture array size should match the number of Gaussians. For Group 2, each Gaussian requires 2 texture points to represent its parameters, so the texture size must be twice the number of Gaussians.

Output Data Grouping. Output data grouping presents additional challenges due to its usage across multiple kernels. Table III outlines the specific usage of each parameter across future kernels. Based on this analysis, `Points_XY_Image`, `Depth`, and `Radii` are grouped together as they are predominantly accessed during the *DuplicateWithTile* operation. This grouping minimizes the read operations required for that kernel. Similar considerations inform the rest of the design of the packing strategy.

Overall Layout. The overall layout is shown through Table V, where the size column indicates the total number of pixels required for each group, while the width and height columns define the dimensions of the texture dimension. The block column specifies the rectangular region (lower-left to upper-right) that stores all the information for a single Gaussian. The texture dimension for Group 1 is chosen to be close to the square root of the number of Gaussians to maximize the width and height while staying within texture memory limits. For Group 2, its size is twice that of Group 1, and both texture points are stored adjacent to each other to simplify index calculations. The adjacency can be either row-wise or column-wise, but in our implementation, we use row adjacency, making the width of Group 2 double that of Group 1.

Group 3, however, is more critical, as its block layout directly impacts texture dimensions. To determine the optimal block layout, two key factors must be considered: (1) The layout should be as square as possible, and (2) it should efficiently utilize the L1 cache. Based on these criteria, we select a 3×4 block dimension for Group 3. Overall, this configuration has two advantages: 1) Group 3 is three times wider than Group 1 and four times taller than Group 1. Since Group 1 is already square-like, this ensures that Group 3 maintains a near-square aspect ratio, making index calculations efficient with minimal arithmetic operations; and 2) With a 3×4 block dimension, an 8×8 cache block can hold two row-adjacent Gaussians along with their corresponding column-adjacent Gaussians, totaling four Gaussians, allowing for better intra-warp and local-group spatial locality.

B. Sorting Optimizations in Context of 3DGS

The original 3D Gaussian Splatting (3DGS) method employs 64-bit integer sorting, but since our texture memory-based sorting operates on floating-point numbers, it does not support native 64-bit sorting. To address this limitation, we implement *key normalization*, which converts the 64-bit integer key for 3DGS to a 32-bit floating point number. Two important factors guide the decision here: 1) The key is created using the *tile* number and the *depth*, and the keys are sorted initially by tile number and then depth; 2) Only the tile number is used in the later kernel to get the range information for each tile. So, to create a 32-bit representation, we keep 20 bits for the tile number and normalize the depth value (which is still needed during sorting) to float within the value 2^{10} (Figure 8).

Since the sorted output is subsequently used for range identification and rendering, it is crucial to store the data in a way that maximizes L1 cache efficiency. Noting that: 1)

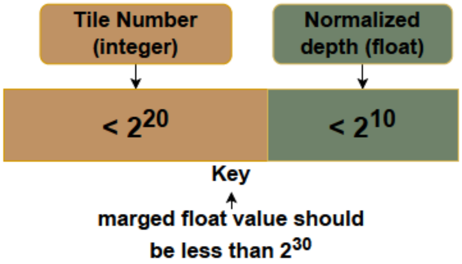


Fig. 8: Key normalization illustration.

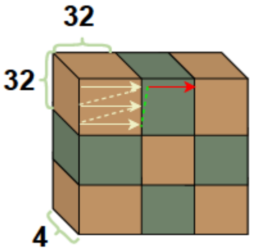


Fig. 9: Block-based data storage.

in the *identify range step*, each local work group accesses consecutive key-value pairs in the sorted dataset, and 2) during rendering, each local work group processes a 16× 16 tile, and all threads within the group access the same range of key-value pairs. In view of this information and the properties of texture cache, we store the sorted data in a block-wise layout (Figure 9), ensuring that memory access patterns align with the 2D spatial locality of the L1 texture cache. Based on empirical analysis, we select a 32×32 block for storing the sorted data.

C. Efficient Tile-Based Rendering

In the original 3D Gaussian Splatting (3DGS) approach, each local work group processes a 16×16 tile, with each thread responsible for computing a single pixel. After analyzing the computations involved, we observe that the operations within a tile exhibit significant data redundancy and can be optimized using Single Instruction, Multiple Data (SIMD) execution. Since all pixels within a 16 × 16 tile share the same set of information to compute their respective colors, we can restructure the computation so that each thread processes four pixels instead of one, utilizing SIMD operations. Additionally, we also utilized loop unrolling. In the rendering stage, there are two loops, both of which are responsible for calculating the color for each channel. Since the number of channels is very limited (e.g., 3 channels for color images), these loops can be unrolled to eliminate loop overhead and improve efficiency.

V. Evaluation

This section systematically assesses the performance and effectiveness of `Texture3dgs` and our custom sorting pipeline. In all, we have the following objectives: (1) Show that our GPU-based sorting significantly outperforms existing mobile sorting techniques (or available implementations) in both latency and resource efficiency; (2) Demonstrate that the full pipeline in `Texture3dgs` outperforms the only other existing mobile implementation, and approaches near-real-time 3D Gaussian Splatting on mobile devices without requiring additional hardware support; (3) Quantify how our key optimizations contribute to different components in 3DGS pipelines; and (4) Illustrate the portability of `Texture3dgs` across other mobile platforms.

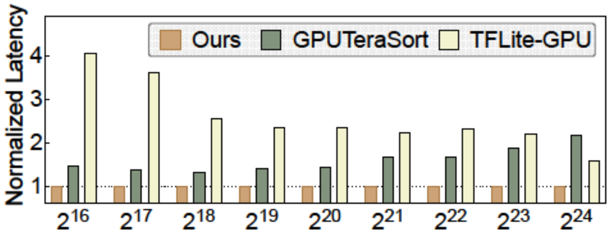


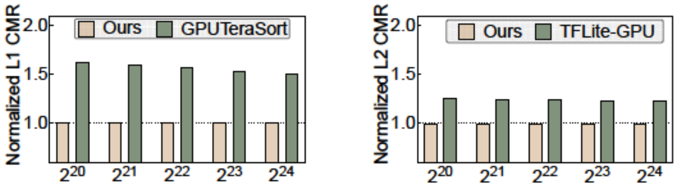
Fig. 10: Latency comparison for sorting routines with different sizes of key-value pairs (as indicated in X-axis). The results are normalized by Ours for readability.

Specifically, the evaluation includes comparisons with the state-of-the-art framework, 3dgs.cpp [55], which supports end-to-end 3DGS pipeline, featuring efficient radix sorting and rendering. Additionally, since sorting is a major component of our application, we also compare `Texture3dgs` with sorting baselines including TFLite [56] and GPUTeraSort [41].

A. Evaluation Setup

Testbed. All evaluations are conducted using the off-the-shelf mobile platforms – the Snapdragon 8 Gen 2 mobile platform, featuring an octa-core Kryo CPU and an Adreno GPU, equipped with 12 GB unified memory. For portability testing, we have used Xiaomi MI 6 (Adreno 540) and Redmi Note 10 (Mali-G57 MC2) with 8 GB and 4 GB unified memory size, respectively. The baselines used for comparison include product-level framework TensorFlow Lite (TFLite 2.19.0) [57], which is a widely adopted general-purpose GPU framework for mobile devices, and the end-to-end framework 3dgs.cpp [55] we referred to earlier. We have also compared our sorting with another GPUTeraSort [41], chosen as it specifically targeted GPU texture-cache-aware sorting. This comparison is carried out is through our own implementation of GPUTerasort as the implementation from the original paper is not available. Specifically, we re-engineered its core optimizations – such as parallel partitioning and bucket-based merging – for compatibility with mobile hardware. In addition, we report sorting comparisons against the modern radix-sort implementation used internally by 3dgs.cpp [55]. Each experiment is executed 50 times, and only the average numbers are reported, as the variance is negligible.

Datasets. Experiments involved multiple datasets of varying complexity: specifically, the point cloud data from the Tanks and Temples [58], DeepBlending [59] and Synthetic-Nerf [18]. Particularly, `Train` and `Truck` are from the Tanks and Temples dataset, `Playroom` is from DeepBlending, and `Chair`,



(a) L1 cache miss rate (b) L2 cache miss rate

Fig. 11: Cache Miss Rate (CMR) comparison for sorting. The results are normalized by Ours for readability.

Drums, and Lego are from the Synthetic-NeRF dataset. Point cloud datasets are generated by training with the original 3DGS [19] framework using Nvidia GPUs with 7k iterations. Additionally, we generate random datasets of diverse sizes (0.5 million to 17 million key-values) to benchmark sorting efficiency. We omit the accuracy results since they remain the consistent among different frameworks on the same hardware.

B. Sorting Performance Comparison

This part evaluates the performance improvements from our sorting and analyzes the underlying causes using cache profiling results.

Latency Comparison. Figure 10 illustrates the latency impact of our proposed sorting optimizations compared to two baseline methods (GPUTeraSort [41] and TFLite [57]) on varied sizes of key-value pairs. For clarity, all results are normalized to our sorting approach. Across all evaluated scenes, our sorting algorithm consistently outperforms both baselines, achieving speedups ranging from 1.5× to 4×. These gains highlight the effectiveness of our hardware-aware optimizations, which reduce computational overhead and improve memory throughput.

Cache Miss Analysis. Figure 11 presents the cache miss rates normalized to our sorting algorithm for improved readability. To ensure a fair and meaningful comparison, we evaluate L1 cache misses against GPUTeraSort [41], as both our method and TeraSort leverage the texture cache backed by the dedicated L1 cache on mobile GPUs. In contrast, we compare L2 cache misses with TFLite-GPU [57], which does not utilize the L1 texture cache and instead relies on the unified memory hierarchy. Our sorting algorithm achieves substantial improvements in cache efficiency, reducing L1 cache misses by up to 60% and L2 cache misses by up to 25% over the compared baselines. These improvements are primarily attributed to our hardware-aware memory access design, which promotes coalesced accesses and spatial locality.

C. End-to-End Performance Comparison

We report results from the full pipeline, first presenting overall latency, followed by memory analysis.

Latency Comparison. We benchmark the end-to-end latency of our optimized 3DGS implementation against the baseline 3dgs.cpp across six scenes listed earlier. Different scenes have different complexities and the absolute latency of our implementations ranges from 60 ms to 600 ms, approaching near

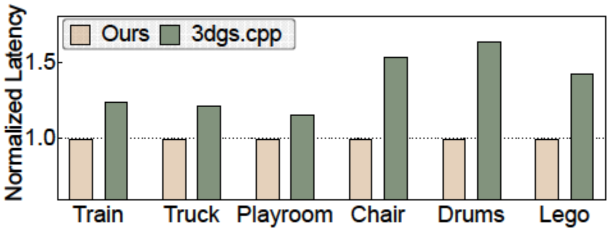
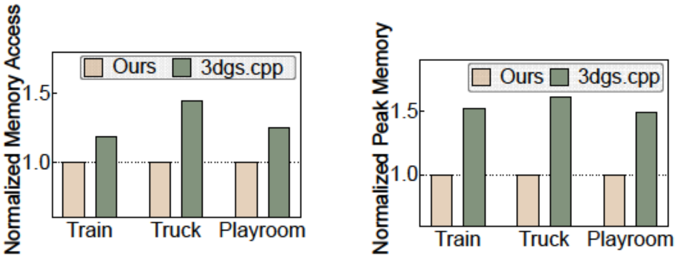


Fig. 12: Overall latency (average) comparison for 6 datasets against 3dgs.cpp. The results are normalized by Ours.



(a) Memory access count. (b) Peak memory consumption.

Fig. 13: Overall memory usage comparison. The results are normalized by ours for readability.

real-time for less complex scenes. The normalized latencies, shown in Figure 12, indicate consistent latency reductions across all cases, with our implementation achieved an average speedup of 1.25×. Notably, the Drums scene exhibits the highest speedup among the real-world scenes. This is attributed to the increased average number of visible Gaussians per tile, which amplifies the benefits of our optimized tile-based renderer by improving batch execution efficiency and reducing per-tile overhead. Similarly, the Chair and Lego yield greater speedups than other three – (Train,Truck, and Playroom). This turns out to be primarily due to their smaller model sizes, combined with larger rendered image resolutions, where our parallelism and locality benefits are higher. In contrast, with Train, Playroom, and Lego scenes we show comparably modest improvements. This is because they involve either fewer Gaussians or are larger models, leading to limited room for memory and execution overlap optimization.

Memory Efficiency Comparison. We evaluate memory efficiency through detailed runtime profiling, focusing on both memory access volume and peak memory usage. The results for Chair, Drums, and Lego show similar trends and are omitted. As shown in Figure 13a and Figure 13b, our optimized implementation reduces total memory accesses by 25% and peak memory usage by 20% on average, compared to the baseline. These improvements stem from our our optimizations involving variable packing, block-wise data layout, – they all reduce both the data movement and memory footprint. Among the evaluated scenes, Truck exhibits the highest reduction in memory accesses. This is because the number of tile-Gaussian pairs in Truck is relatively small compared to the total number of Gaussians, enabling more

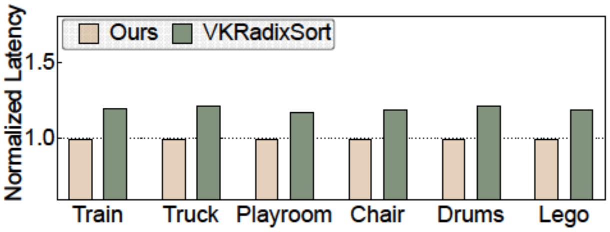
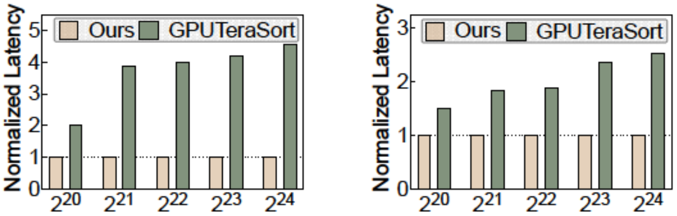


Fig. 14: Comparison of (average) end-to-end latency of our implementation against VKRadixSort for 6 datasets. The results are normalized by Ours for readability.

efficient indexing and pruning during rendering. In contrast, Playroom demonstrates a smaller memory reduction. This is due to inefficiencies in the underlying base sorting scheme, which pads the number of key-value pairs to the nearest power-of-two. For Playroom, the actual number of pairs is substantially lower than the next power-of-two threshold, resulting in underutilized texture memory.

D. Benefits of Other Optimizations

Table VI compares the performance of our full pipeline ("Ours") with its ablated variants across three representative scenes: Train and Truck from the Tanks and Temples dataset, and Playroom from Deep Blending. We evaluate the impact of disabling three key optimizations: optimal data layout (dl), variable packing (vp), and execution fusion (ec). All versions include our optimized sorting implementation, as we have already studied benefits from sorting. Our full version achieves the best end-to-end latency, with up to a 1.27× improvement over the least optimized variant (but still with optimized sorting). The performance gains are most prominent in the rendering and preprocessing stages because these kernels heavily depend on a huge number of data read/write operations. Despite Truck and Playroom having similar numbers of 3D Gaussians, the sorting cost in Playroom is nearly double that of Truck. This is attributed to the significantly larger number of Gaussian-tile pairs in Playroom, which directly increases the sorting workload. We further observe that data layout optimizations influence multiple stages: preprocess, identify range, and render. This is because preprocessing handles spherical harmonics (Shs) and writes to a blockwise layout, which is then consumed downstream. The layout also improves spatial locality during rendering and range identification. Data layout selection (dl) contributes 1.05× to 1.07× speedup on the selected datasets. Variable packing, applied to both inputs and outputs of the preprocessing stage, also impacts Duplicate with Tiles and render, since packed buffers are reused across these stages. When disabled, the memory footprint and processing time increase due to less efficient data movement and reuse. This optimization adds 1.03× to 1.05× speedup. Lastly, the execution fusion (ec) optimization is specific to the render stage. Disabling it only affects rendering time, since this optimization is limited to kernel-level fusion and instruction scheduling within that stage. It offers 1.06× to 1.1× speedup.



(a) XiaoMi 6

(b) Redmi Note 10

Fig. 15: Sorting time comparison on 2 older devices.

Apart from the above optimizations, we also compare our sorting implementation with VKRadixSort [60], which is a widely used radix-sort implementation available for mobile GPUs. As shown in Figure 14, our approach consistently achieves lower latency than VKRadixSort. Specifically, Figure 14 shows that the sorting algorithm alone achieves a performance improvement of 1.10× to 1.15× over VKRadixSort.

E. Portability

Figure 15 reports the execution latency of our sorting implementation and compares to the GPUTeraSort [41] on two additional mobile devices–Xiaomi MI 6 (Adreno 540) and Redmi Note 10 (Mali-G57 MC2). This experiment focuses on demonstrating portability by selecting the sorting implementation, as the baseline for the full 3DGS could not be run on older platforms. Our implementation consistently delivers stable performance across both devices, particularly on the older-generation MI 6. Our optimizations greatly reduce intermediate memory usage and computational complexity, ensuring robustness – especially important for devices with limited memory bandwidth and processing power.

VI. Related Work

End-to-end 3DGS Acceleration. To address efficiency issues for 3DGS, a number of end-to-end acceleration strategies have been proposed. Mini-Splatting [61] represents one early effort to constrain model size by reorganizing Gaussians in the scene. LightGaussian [62] takes a complementary approach by aggressively compressing the 3DGS representation: it prunes away globally least significant Gaussians and quantizes remaining parameters. EfficientGS [63] focuses on reducing per-Gaussian complexity, for instance by retaining only low-order spherical harmonic coefficients for most Gaussians and increasing the coefficient order or density only when needed. EagleS [31] introduces a novel Gaussian pruning strategy that leverages a learned saliency map to identify and remove redundant Gaussians. Morgenstern et al. propose a compressed 3D scene that [64] primarily addresses compact data representation through structured 2D grids rather than runtime sorting efficiency. These methods collectively demonstrate that end-to-end performance improvements are possible by simplifying or restructuring the entire 3DGS pipeline. However, they are primarily designed for desktop GPUs and rely on high memory bandwidth, shared memory, and large-scale parallelism. Such assumptions do not hold on mobile GPUs, where memory

TABLE VI: Latency comparison (ms) across different 3DGS pipelines on different datasets and scenes. “w/o” refers to without that optimization. “sz” refers to image size and “n” refers to the number of 3d gaussians. “dl” shorts for optimal data layout. “vp” and “ec” stand for variable packing and execution fusion, respectively. All versions in this table employ our optimized texture-memory-aware sorting approach, hence sorting latency is almost unchanged among all versions.

Scene	Approach	Speedup	End to End	Preprocess	Scan	Duplicate with tiles	Sorting	Identify Range	Render
Train sz : 980×545 n : 741,295	**Ours**	**1.21×**	**299**	**4.60**	**2.70**	**3.19**	**267**	**0.47**	**20.9**
	Ours w/o dl		310	5.40	2.67	3.19	267	0.69	30.6
	Ours w/o vp + dl		326	8.60	2.67	5.60	267	0.69	40.8
	Ours w/o vp + dl + ec		361	8.60	2.67	5.60	267	0.69	76.0
Truck sz : 979×546 n : 1,689,804	**Ours**	**1.27×**	**314**	**6.80**	**6.94**	**3.04**	**270**	**0.46**	**25.8**
	Ours w/o dl		342	10.7	6.94	3.04	271	0.64	49.4
	Ours w/o vp + dl		370	25.4	6.94	5.49	271	0.64	60.6
	Ours w/o vp + dl + ec		400	25.4	6.94	5.49	271	0.64	90.5
Playroom sz : 1264×832 n : 1,491,851	**Ours**	**1.18×**	**584**	**5.60**	**6.10**	**5.30**	**506**	**0.83**	**59.5**
	Ours w/o dl		619	9.20	6.10	5.30	506	1.83	90.3
	Ours w/o vp + dl		648	20.2	6.10	7.90	506	1.83	105
	Ours w/o vp + dl + ec		690	20.2	6.10	7.90	506	1.83	148

and compute resources are significantly constrained. The sort-free methods, OIT rendering [65] and StochasticSplats [66], effectively remove sorting overhead but introduce significant computational complexities. StochasticSplats relies on expensive multi-sample Monte Carlo estimation, and OIT rendering adds considerable per-fragment computation overhead. In our experiments, these methods typically require substantial computational resources and are not able to directly apply to resource-constrained mobile devices used in our experiments.

Memory Optimizations for 3DGS. Another line of work focuses on algorithm-level optimizations to reduce 3DGS memory usage and runtime cost. Taming 3DGS [67] adopts a saliency-based approach by selectively densifying or pruning Gaussians using image-space gradient metrics. Most recently, GaussianSpa [68] formulates a sparsity-constrained optimization problem, applying an alternating minimization framework to jointly optimize reconstruction and sparsification. 3DGS-LM [69] replaces the Adam optimizer with a tailored Levenberg–Marquardt solver, achieving faster convergence during reconstruction through second-order updates and view-consistent gradient fusion. Orthogonal to pruning, other methods leverage quantization and compression to reduce per-Gaussian storage overhead. CompGS [70] applies vector quantization with codebooks for spatial and color attributes. RDO-Gaussian [71] combines pruning and entropy-constrained quantization in a rate-distortion framework, achieving efficient storage with bounded reconstruction loss. EfficientGS [63] reduces spherical harmonic (SH) order adaptively, compressing color representation without compromising rendering quality. While these works have made meaningful progress in simplifying Gaussian representations and accelerating training, our optimization is orthogonal to these approaches. Combining our sorting and memory optimizations with these compression techniques can be promosing directions for future work.

GPU Sorting Optimization. While extensively studied on desktop GPUs, sorting remains a bottleneck on mobile GPUs due to its irregular memory access pattern. Classical GPU sorting methods —- such as parallel radix sort [72], bitonic sort [73], and sample sort – leverage warp-level parallelism and shared memory, achieving high throughput in large-scale settings. These methods exemplify how cache-aware GPU designs can scale sorting throughput with large input sizes. However, these designs assume access to large, shared memory regions and high-bandwidth memory interfaces. Mobile GPUs, by contrast, employ tile-based deferred rendering with highly localized memory and lower peak throughput. As a result, GPU sorting algorithms optimized for desktop architectures often underperform or become infeasible on mobile platforms.

VII. Conclusions and Future Work

In this paper, we presented a novel optimization framework for accelerating 3DGS applications on mobile GPUs. Our approach specifically targets efficient computation utilizing the 2D texture cache on mobile GPU architectures. The primary innovation is a novel sorting algorithm that significantly reduces cache misses by optimizing data movement and layout, along with enhanced data placement strategies for greater overall efficiency. Extensive evaluations show that our end-to-end implementation achieves up to 1.6× performance improvement compared to baseline implementations, with particularly significant gains in sorting kernel operations by up to 4.1×. Compared to the alternative 3dgs frameworks and sorting solutions, our solution provides novel insights into exploiting 2D memory characteristics. The work presented here can be extended in multiple directions. One area can be revisiting the proposed optimizations to support adaptive resolution models. Another area can be exploring sorting and full application enhancements for other mobile architectures, e.g., mobile NPUs or DSP chips.

Acknowledgment

The authors want to extend their appreciation to the anonymous reviewers for their valuable and thorough feedback, which helped improve the paper. This work was supported in part by the National Science Foundation (NSF) under the awards of CCF-2428108, OAC-2403090, CNS-2341378, and CCF-2333895. Any errors and opinions are not those of the NSF and are attributable solely to the author(s).

REFERENCES

[1] Y. Wang, Y. Long, S. H. Fan, and Q. Dou, "Neural rendering for stereo 3d reconstruction of deformable tissues in robotic surgery," in *International conference on medical image computing and computer-assisted intervention*, pp. 431–441, Springer, 2022.

[2] T. Zhang, K. Huang, W. Zhi, and M. Johnson-Roberson, "Darkgs: Learning neural illumination and 3d gaussians relighting for robotic exploration in the dark," *arXiv preprint arXiv:2403.10814*, 2024.

[3] D. Kalkofen, E. Mendez, and D. Schmalstieg, "Comprehensible visualization for augmented reality," *IEEE transactions on visualization and computer graphics*, vol. 15, no. 2, pp. 193–204, 2008.

[4] X. Yang, H. Li, H. Zhai, Y. Ming, Y. Liu, and G. Zhang, "Vox-fusion: Dense tracking and mapping with voxel-based neural implicit representation," in *2022 IEEE International Symposium on Mixed and Augmented Reality (ISMAR)*, pp. 499–507, IEEE, 2022.

[5] X. Zhou, Z. Lin, X. Shan, Y. Wang, D. Sun, and M.-H. Yang, "Drivinggaussian: Composite gaussian splatting for surrounding dynamic autonomous driving scenes," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 21634–21643, 2024.

[6] H. Zhou, J. Shao, L. Xu, D. Bai, W. Qiu, B. Liu, Y. Wang, A. Geiger, and Y. Liao, "Hugs: Holistic urban 3d scene understanding via gaussian splatting," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 21336–21345, 2024.

[7] Y. Furukawa, C. Hernández, *et al.*, "Multi-view stereo: A tutorial," *Foundations and Trends® in Computer Graphics and Vision*, vol. 9, no. 1-2, pp. 1–148, 2015.

[8] A. Kar, C. Häne, and J. Malik, "Learning a multi-view stereo machine," *Advances in neural information processing systems*, vol. 30, 2017.

[9] Y. Yao, Z. Luo, S. Li, T. Fang, and L. Quan, "Mvsnet: Depth inference for unstructured multi-view stereo," in *Proceedings of the European conference on computer vision (ECCV)*, pp. 767–783, 2018.

[10] R. Chen, S. Han, J. Xu, and H. Su, "Point-based multi-view stereo network," in *Proceedings of the IEEE/CVF international conference on computer vision*, pp. 1538–1547, 2019.

[11] B. Mildenhall, P. P. Srinivasan, M. Tancik, J. T. Barron, R. Ramamoorthi, and R. Ng, "Nerf: Representing scenes as neural radiance fields for view synthesis," *Communications of the ACM*, vol. 65, no. 1, pp. 99–106, 2021.

[12] J. T. Barron, B. Mildenhall, D. Verbin, P. P. Srinivasan, and P. Hedman, "Mip-nerf 360: Unbounded anti-aliased neural radiance fields," in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pp. 5470–5479, 2022.

[13] S. Fridovich-Keil, A. Yu, M. Tancik, Q. Chen, B. Recht, and A. Kanazawa, "Plenoxels: Radiance fields without neural networks," in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pp. 5501–5510, 2022.

[14] T. Müller, A. Evans, C. Schied, and A. Keller, "Instant neural graphics primitives with a multiresolution hash encoding," *ACM transactions on graphics (TOG)*, vol. 41, no. 4, pp. 1–15, 2022.

[15] B. Kerbl, G. Kopanas, T. Leimkühler, and G. Drettakis, "3d gaussian splatting for real-time radiance field rendering.," *ACM Trans. Graph.*, vol. 42, no. 4, pp. 139–1, 2023.

[16] J. Lee, S. Lee, J. Lee, J. Park, and J. Sim, "Gscore: Efficient radiance field rendering via architectural support for 3d gaussian splatting," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS '24, (New York, NY, USA), p. 497–511, Association for Computing Machinery, 2024.

[17] S. Zhou, H. Chang, S. Jiang, Z. Fan, Z. Zhu, D. Xu, P. Chari, S. You, Z. Wang, and A. Kadambi, "Feature 3dgs: Supercharging 3d gaussian splatting to enable distilled feature fields," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 21676–21685, 2024.

[18] B. Mildenhall, P. P. Srinivasan, M. Tancik, J. T. Barron, R. Ramamoorthi, and R. Ng, "Nerf: representing scenes as neural radiance fields for view synthesis," *Commun. ACM*, vol. 65, p. 99–106, Dec. 2021.

[19] B. Kerbl, G. Kopanas, T. Leimkühler, and G. Drettakis, "3d gaussian splatting for real-time radiance field rendering," *ACM Transactions on Graphics*, vol. 42, July 2023.

[20] H. Cai, J. Lin, Y. Lin, Z. Liu, H. Tang, H. Wang, L. Zhu, and S. Han, "Enable deep learning on mobile devices: Methods, systems, and applications," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 27, no. 3, pp. 1–50, 2022.

[21] W. Niu, X. Ma, S. Lin, S. Wang, X. Qian, X. Lin, Y. Wang, and B. Ren, "Patdnn: Achieving real-time dnn execution on mobile devices with pattern-based weight pruning," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 907–922, 2020.

[22] W. Niu, J. Guan, X. Shen, Y. Wang, G. Agrawal, and B. Ren, "Gcd 2: A globally optimizing compiler for mapping dnns to mobile dsps," in *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 512–529, IEEE, 2022.

[23] L. N. Huynh, Y. Lee, and R. K. Balan, "Deepmon: Mobile gpu-based deep learning framework for continuous vision applications," in *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys)*, pp. 82–95, ACM, 2017.

[24] W. Niu, G. Agrawal, and B. Ren, "Sod 2: Statically optimizing dynamic deep neural network execution," in *Proceedings of the 2024 International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2024.

[25] R. Lee, S. I. Venieris, L. Dudziak, S. Bhattacharya, and N. D. Lane, "Mobisr: Efficient on-device super-resolution through heterogeneous mobile processors," in *The 25th Annual International Conference on Mobile Computing and Networking*, pp. 1–16, 2019.

[26] W. Niu, J. Guan, Y. Wang, G. Agrawal, and B. Ren, "Dnnfusion: accelerating deep neural networks execution with advanced operator fusion," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pp. 883–898, 2021.

[27] C. U. Press, *Virtual reality*. Cambridge, United Kingdom: Addison-Wesley, 2023.

[28] L. Yariv, P. Hedman, C. Reiser, D. Verbin, P. P. Srinivasan, R. Szeliski, J. T. Barron, and B. Mildenhall, "Bakedsdf: Meshing neural sdfs for real-time view synthesis," in *ACM SIGGRAPH 2023 Conference Proceedings*, pp. 1–9, 2023.

[29] R. Liang, T. Cao, J. Wen, M. Wang, Y. Wang, J. Zou, and Y. Liu, "Romou: Rapidly generate high-performance tensor kernels for mobile gpus," in *The 28th Annual International Conference On Mobile Computing And Networking (MobiCom 2022)*, ACM, February 2022.

[30] J. Guan, Z. Hu, C. D. Antonopoulos, N. Bellas, S. Lalis, E. Smirni, G. Zhou, G. Agrawal, and B. Ren, "Tmmodel: Modeling texture memory and mobile gpu performance to accelerate dnn computations," in *Proceedings of the ACM International Conference on Supercomputing (ICS)*, 2025.

[31] S. Girish, K. Gupta, and A. Shrivastava, "Eagles: Efficient accelerated 3d gaussians with lightweight encodings," in *European Conference on Computer Vision*, pp. 54–71, Springer, 2025.

[32] M. Niemeyer, F. Manhardt, M.-J. Rakotosaona, M. Oechsle, D. Duckworth, R. Gosula, K. Tateno, J. Bates, D. Kaeser, and F. Tombari, "Radsplat: Radiance field-informed gaussian splatting for robust real-time rendering with 900+ fps," *arXiv preprint arXiv:2403.13806*, 2024.

[33] N. K. Govindaraju, J. Gray, R. Kumar, and D. Manocha, "Gputerasort: High performance graphics co-processor sorting for large database management," *Proceedings of the 2006 ACM SIGMOD*, 2006.

[34] W. Niu, M. M. R. Sanim, Z. Shu, J. Guan, X. Shen, M. Yin, G. Agrawal, and B. Ren, "Smartmem: Layout transformation elimination and adaptation for efficient dnn execution on mobile," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS '24, (New York, NY, USA), p. 916–931, Association for Computing Machinery, 2024.

[35] Z. S. Hakura and A. Gupta, "The design and analysis of a cache architecture for texture mapping," in *Proceedings of the 24th annual international symposium on Computer architecture*, pp. 108–120, 1997.

[36] H. Igehy, M. Eldridge, and K. Proudfoot, "Prefetching in a texture cache architecture," in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, pp. 133–ff, 1998.

[37] M. Doggett, "Texture caches," *IEEE Micro*, vol. 32, p. 136–141, May 2012.

[38] Qualcomm., "Qualcomm snapdragon mobile platform opencl general programming and optimization.." 2023.

[39] H. Shi and J. Schaeffer, "Parallel sorting by regular sampling," *Journal of Parallel and Distributed Computing*, vol. 14, no. 4, pp. 361–372, 1992.

[40] P. Kipfer, M. Segal, and R. Westermann, "Uberflow: a gpu-based particle engine," in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS*

Conference on Graphics Hardware, HWWS '04, (New York, NY, USA), p. 115–122, Association for Computing Machinery, 2004.

[41] N. Govindaraju, J. Gray, R. Kumar, and D. Manocha, "Gputerasort: high performance graphics co-processor sorting for large database management," in *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, SIGMOD '06, (New York, NY, USA), p. 325–336, Association for Computing Machinery, 2006.

[42] E. Manca, A. Manconi, A. Orro, G. Armano, and L. Milanesi, "Cuda-quicksort: an improved gpu-based implementation of quicksort," *Concurr. Comput.: Pract. Exper.*, vol. 28, p. 21–43, Jan. 2016.

[43] K. E. Batcher, "Sorting networks and their applications," in *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*, AFIPS '68 (Spring), (New York, NY, USA), p. 307–314, Association for Computing Machinery, 1968.

[44] M. Dowd, Y. Perl, L. Rudolph, and M. Saks, "The periodic balanced sorting network," *J. ACM*, vol. 36, p. 738–757, Oct. 1989.

[45] B. R, C. G, N. F. M, and S. F, "Sorting using bitonic network with cuda," 2009.

[46] E. Stehle and H.-A. Jacobsen, "A memory bandwidth-efficient hybrid radix sort on gpus," in *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, (New York, NY, USA), p. 417–432, Association for Computing Machinery, 2017.

[47] D. Cederman and P. Tsigas, "Gpu-quicksort: A practical quicksort algorithm for graphics processors," *ACM J. Exp. Algorithmics*, vol. 14, Jan. 2010.

[48] A. Davidson, D. Tarjan, M. Garland, and J. D. Owens, "Efficient parallel merge sort for fixed and variable length keys," in *2012 Innovative Parallel Computing (InPar)*, pp. 1–9, 2012.

[49] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell, "A survey of general-purpose computation on graphics hardware," in *Computer graphics forum*, vol. 26, pp. 80–113, Wiley Online Library, 2007.

[50] D. P. Singh, I. Joshi, and J. Choudhary, "Survey of gpu based sorting algorithms," *International Journal of Parallel Programming*, vol. 46, pp. 1017–1034, 2018.

[51] N. K. Govindaraju, N. Raghuvanshi, M. Henson, and D. Manocha, "A cache-efficient sorting algorithm for database and data mining computations using graphics processors," *University of North Carolina, Tech. Rep*, 2005.

[52] X. Jiang, H. Wang, Y. Chen, Z. Wu, L. Wang, B. Zou, Y. Yang, Z. Cui, Y. Cai, T. Yu, C. Lv, and Z. Wu, "Mnn: A universal and efficient inference engine," in *MLSys*, 2020.

[53] H. Ni and The ncnn contributors, "ncnn," June 2017.

[54] V. Volkov and J. W. Demmel, "Benchmarking gpus to tune dense linear algebra," in *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, SC '08, IEEE Press, 2008.

[55] Steven, "3dgs.cpp." https://github.com/shg8/3DGS.cpp, 2024. GitHub commit `8fe4b2fba09306e9fcb0308fa11c6aa7b8d0ac41`, accessed 13 Nov. 2024.

[56] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015. Software available from tensorflow.org.

[57] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "Tensorflow: A system for large-scale machine learning," in *OSDI 2016*, (USA), pp. 265–283, USENIX Association, 2016.

[58] A. Knapitsch, J. Park, Q.-Y. Zhou, and V. Koltun, "Tanks and temples: benchmarking large-scale scene reconstruction," *ACM Trans. Graph.*, vol. 36, July 2017.

[59] P. Hedman, J. Philip, T. Price, J.-M. Frahm, G. Drettakis, and G. Brostow, "Deep blending for free-viewpoint image-based rendering," vol. 37, no. 6, pp. 257:1–257:15, 2018.

[60] V. Contributors, "Vkradixsort." https://github.com/MircoWerner/VkRadixSort, 2024.

[61] G. Fang and B. Wang, "Mini-splatting: Representing scenes with a constrained number of gaussians," in *European Conference on Computer Vision*, pp. 165–181, Springer, 2024.

[62] Z. Fan, K. Wang, K. Wen, Z. Zhu, D. Xu, Z. Wang, *et al.*, "Light-gaussian: Unbounded 3d gaussian compression with 15x reduction and 200+ fps," *Advances in neural information processing systems*, vol. 37, pp. 140138–140158, 2024.

[63] W. Liu, T. Guan, B. Zhu, L. Xu, Z. Song, D. Li, Y. Wang, and W. Yang, "Efficientgs: Streamlining gaussian splatting for large-scale high-resolution scene representation," *IEEE MultiMedia*, 2025.

[64] W. Morgenstern, F. Barthel, A. Hilsmann, and P. Eisert, "Compact 3d scene representation via self-organizing gaussian grids," in *European Conference on Computer Vision*, pp. 18–34, Springer, 2024.

[65] Q. Hou, R. Rauwendaal, Z. Li, H. Le, F. Farhadzadeh, F. Porikli, A. Bourd, and A. Said, "Sort-free gaussian splatting via weighted sum rendering," *arXiv preprint arXiv:2410.18931*, 2024.

[66] S. Kheradmand, D. Vicini, G. Kopanas, D. Lagun, K. M. Yi, M. Matthews, and A. Tagliasacchi, "Stochasticsplats: Stochastic rasterization for sorting-free 3d gaussian splatting," *arXiv preprint arXiv:2503.24366*, 2025.

[67] S. S. Mallick, R. Goel, B. Kerbl, M. Steinberger, F. V. Carrasco, and F. De La Torre, "Taming 3dgs: High-quality radiance fields with limited resources," in *SIGGRAPH Asia 2024 Conference Papers*, pp. 1–11, 2024.

[68] Y. Zhang, W. Jia, W. Niu, and M. Yin, "Gaussianspa: An "optimizing-sparsifying" simplification framework for compact and high-quality 3d gaussian splatting," *arXiv preprint arXiv:2411.06019*, 2024.

[69] A. Hoel *et al.*, "3dgs-lm: Fast 3d gaussian splatting with levenberg–marquardt optimization," *arXiv preprint arXiv:2409.12892*, 2024.

[70] K. Navaneet, K. P. Meibodi, S. A. Koohpayegani, and H. Pirsiavash, "Compgs: Smaller and faster gaussian splatting with vector quantization," *ECCV*, 2024.

[71] P. Papantonakis, G. Kopanas, B. Kerbl, A. Lanvin, and G. Drettakis, "Rdo-gaussian: End-to-end rate-distortion optimized 3d gaussian representation," *arXiv preprint arXiv:2406.01597*, 2024.

[72] N. Satish, M. Harris, and M. Garland, "Designing efficient sorting algorithms for manycore gpus," *Proceedings of the IEEE International Parallel and Distributed Processing Symposium*, 2009.

[73] K. E. Batcher, "Sorting networks and their applications," *Proceedings of the AFIPS Spring Joint Computer Conference*, 1968.