NeRF-PIM: PIM Hardware-Software Co-design of Neural Rendering Networks

Jaeyoung Heo and Sungjoo Yoo

***Abstract*—Neural Radiance Field (NeRF) has emerged as a state-of-the-art technique, offering unprecedented realism in rendering. Despite its advancements, the adoption of NeRF is constrained by high computational cost, leading to slow rendering speed. Voxel-based optimization of NeRF addresses this by reducing the computational cost, but it introduces substantial memory overheads. To address this problem, we propose NeRF-PIM, a hardware-software co-design approach. In order to address the problem of memory accesses to large model (of voxel grid) with poor locality and low compute density, we propose exploiting processing-in-memory (PIM) together with PIM-aware software optimizations in terms of data layout, redundancy removal, and computation reuse.**

Our PIM hardware aims to accelerate tri-linear interpolation and dot product operations. Specifically, to address the low utilization of internal bandwidth due to random accesses to voxels, we propose a data layout that judiciously exploits the characteristics of interpolation operation on voxel grid, which helps remove bank conflicts in voxel accesses and also improves the efficiency of PIM command issue by exploiting all-bank mode in the existing PIM device. As PIM-aware software optimizations, we also propose occupancy-grid-aware pruning and one-voxel two-sampling (1V2S) methods, which contribute to compute efficiency improvement (by avoiding redundant computation on empty space) and memory traffic reduction (by reusing per-voxel dot product results). We conduct experiments using an actual baseline HBM-PIM device. Our NeRF-PIM demonstrates a speedup of 7.4x and 5.0x compared to the baseline on two datasets, Synthetic-NeRF and Tanks and Temples, respectively.

***Index Terms*—accelerator, hardware/software co-design, neural radiance fields (NeRF), processing in memory, voxel grid**

I. Introduction

Neural Radiance Fields (NeRF), which produces highly realistic images using deep neural networks, has attracted considerable attention, especially in augmented and virtual reality (AR/VR) applications. Despite its revolutionary potential, the practical application of NeRF is substantially hindered by its high computational cost, resulting in prohibitively slow rendering speed.

To address the compute-cost problem, several efficient algorithms have been proposed [1]–[3]. Among them, voxel grid-based ones stand out as a promising candidate [4]–[6]. Unlike traditional NeRF, which relies mostly on multilayer perceptron (MLP) computations, voxel grid-based NeRF divides 3D space into a grid of voxels and exploits pre-computation, i.e., utilizes stored values on these voxels, thereby reducing the computation cost per sampling point down to that of a tiny MLP or simple dot products.

For fast rendering, a minimum compute-cost solution is desirable in the voxel grid-based NeRF models. Plenoxels [4] and FastNeRF [7] are two representative examples since they perform only dot product operations at each sampling point.[1] Such minimum compute-cost models trade model size for fast rendering. However, they often fail to realize fast rendering due to the overhead of accessing large models.

To investigate the limitations of the minimum compute-cost voxel grid-based NeRF, we performed performance profiling and found that the primary operations consist of tri-linear interpolations and dot products, both characterized by low computational intensity (Ops/Byte) due to the characteristics of storing voxel grid in the memory (all neighbor voxels cannot be stored contiguously) and volume rendering (sampling points are distributed across voxels). To address these issues coming from large model size, heavy memory traffic, and low computational intensity, we propose to judiciously exploit processing-in-memory (PIM) as a solution to improve performance.

In this paper, we propose a PIM hardware and software co-design for accelerating the minimum compute-cost voxel grid-based NeRF models. Our contributions are as follows:

1) Our hardware-software co-design comprises two key aspects: hardware-software partitioning and hardware-aware software optimization. Through detailed profiling, we identify that the performance bottleneck of NeRF model execution is in voxel grid accesses and interpolation, and propose a co-processor for hardware acceleration by enhancing the existing HBM-PIM with a new data layout for efficient voxel grid accesses and a new interpolation unit for reducing memory traffic. We also propose modifying the NeRF model software to make the best use of the underlying hardware, as will be introduced below.
2) Executing tri-linear interpolation within PIM poses architectural challenges, specifically bank conflicts due to random accesses. We address this issue by proposing a novel data layout that distributes, in an aligned manner, eight voxel data across different banks, which enables simultaneous accesses to eight voxels with a single all-bank PIM command, thereby avoiding bank conflicts. Additionally, we introduce an ML-based channel assign-

J. Heo is with the Interdisciplinary Program in Artificial Intelligence, Seoul National University, Seoul 08826, South Korea (e-mail: wodud7721@snu.ac.kr).

S. Yoo is with the Department of Computer Science and Engineering, Seoul National University, Seoul 08826, South Korea (e-mail: sungjoo.yoo@gmail.com).

[1] Voxel grid-based models typically execute a small MLP on each sampling point. The minimum compute-cost models minimize the per-sampling point compute-cost down to a few dot products.

ment to improve load balancing across memory channels. Our proposed NeRF-PIM architecture augments the existing HBM-PIM architecture [8] by introducing an interpolation unit at a small additional area cost.

3) The PIM-aware software optimization focuses on voxel grid-aware redundancy removal and intra-PIM computation reuse. While adopting an occupancy grid for voxel pruning, we propose a novel pruning method that further prunes the sparse voxel grid by exploiting the fact that a vertex pruning can have a different impact on its voxel and its neighboring voxels under the occupancy grid. In addition, based on the fact that when a sampling point is already selected in a voxel, adding an additional sampling point in the same voxel does not incur additional memory accesses inside the PIM device, we also propose one-voxel two-sampling (1V2S) method, which contributes to performance improvement by effectively reducing memory accesses inside the PIM device.
4) We evaluated the NeRF-PIM with a representative minimum compute-cost voxel grid-based NeRF model, namely Plenoxels [4], using two datasets, Synthetic-NeRF dataset [9] and Tanks and Temples [10], employing an actual HBM-PIM device [8].

II. Background and Motivation

A. Preliminaries of NeRF

NeRF uses deep neural networks to generate highly realistic images from 2D images. The basic idea is that the color of each pixel is obtained via volume rendering where we first shoot a ray (corresponding to a pixel on the image) into the space and the color is calculated by a weighted summation of colors at sampling points on the ray. The neural network, upon the coordinate of a sampling point and the ray direction, predicts the weight and color of the sampling point.

Figure 1 illustrates the rendering pipeline of NeRF [9], divided into four steps. In Step 1 (Ray generation), given a target view (i.e., camera pose) for the image to be rendered, we generate rays connecting the camera origin to pixels, expressed as **r** = **o** + t**d**, where **o** represents the camera center coordinate and **d** represents the direction. In the subsequent steps, we calculate the color of each ray corresponding to each pixel of the image.

In Step 2 (Generate sampling points with ray marching), we locate sampling points along the ray (e.g., at regular intervals). In this process, known as ray marching, we check to see whether an object is present at each location or the sample point is in empty space (via hierarchical sampling or occupancy grid). If the sampling point is not in empty space, we generate the sample point and move to Step 3.

In Step 3 (Calculate density/color data of each sampling point), we execute a trained multilayer perceptron (MLP) with the information of each generated sampling point including its coordinates (x, y, z) and the ray direction (**d**) as inputs. The MLP outputs the sampling point's density (σ) and color (R, G, B). As multiple sampling points are generated and evaluated for each ray through ray marching, Steps 2 and 3 are repeated multiple times before moving to Step 4.

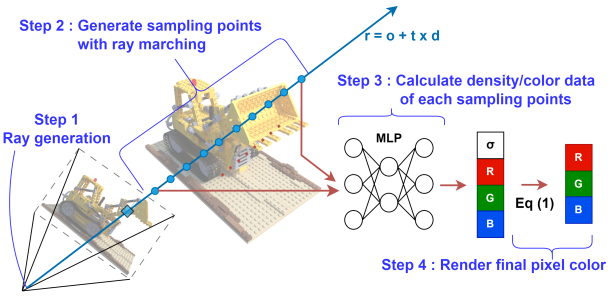


Fig. 1. Rendering pipeline of NeRF [9]

In Step 4 (Render final pixel color), we utilize the results of Step 3 and calculate the pixel color of the current ray ($\widehat{C}(r)$), using the volume rendering equation as follows.

$$\widehat{C}(r) = \sum_{i=1}^{N} T_i(1 - exp(-\sigma_i\delta_i))c_i$$
$$\text{where } T_i = exp\left(-\sum_{j=1}^{i-1}\sigma_j\delta_j\right) \quad (1)$$

Here, N is the number of sampling points along the ray, σ_i and c_i are the density and color at the i-th sample point, respectively, δ_i is the distance between sampling points, and T_i represents the transmittance, which indicates the presence of object from the camera center to a sampling point i. We apply Steps 1 to 4 to all the rays to generate an image.

The rendering pipeline requires executing MLP as many times as there are sampling points. For instance, rendering an 800x800 image would result in 800x800xN sampling points, where N, the number of sampling points per ray, is 192 in [9]. Experiments on an NVIDIA RTX 2080ti show that the rendering speed is only 0.03 fps.

B. Voxel Grid-based NeRF Algorithm

Figure 2 illustrates the structure and rendering pipeline of a minimum compute-cost voxel grid-based NeRF model, Plenoxels [4]. This model employs a sparse voxel grid structure, where each unpruned grid vertex stores a density scalar and three feature (or color) vectors for RGB computation.

As shown in the figure, the rendering pipeline comprises four steps, where Steps 1, 2, and 4 align with those of the vanilla NeRF, while Step 3 replaces the MLP with a minimal computation using dot product operations. To be specific, Step 3 involves a density stage and a color stage. In the density stage, density scalar values are retrieved from the eight vertices of a voxel containing the sampling point, and tri-linear interpolation is applied to compute a density value. Similarly, in the color stage, tri-linear interpolation is performed on the color vectors from the eight vertices, which gives an output color vector. To render view-dependent color, a dot product operation is carried out between the basis vector (for spherical harmonics in [4]) and the output color vector for each RGB color. In Plenoxels [4], the basis vector, representing the ray direction, is generated using spherical harmonics (SH) functions and the ray direction. Thus, it is shared by all the

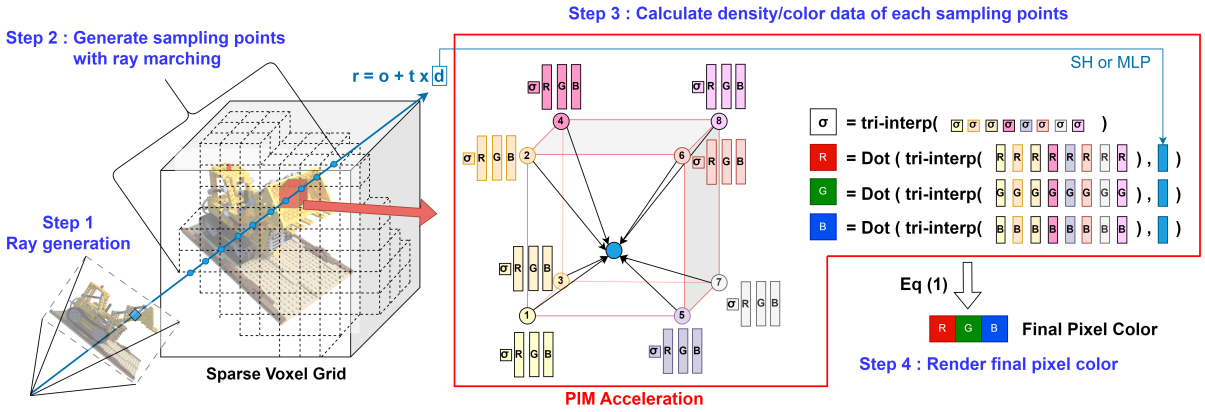


Fig. 2. Rendering pipeline of voxel grid-based NeRF [4]

sampling points on the same ray. The dimensions of the basis and color vectors are set to 9 in our experiments as in [4].

Note that we utilize Plenoxels to showcase the potential of minimum compute-cost models, hence our proposed NeRF-PIM is not limited to Plenoxels.

III. Problem

A. Minimum Compute-Cost Voxel Grid-based NeRF

Minimum compute-cost voxel grid-based NeRF represents a trade-off between computational and memory overhead, leading to its unique characteristics. First, **the model size is considerably large**. As shown in Figure 3, accessing density/color data requires link data that stores the vertex indices. The size of the link data depends on the grid resolution. For instance, a [512,512,512] grid gives a link data size of approximately 537MB. Additionally, each unpruned vertex holds a density scalar and color vectors, which leads to the overall model size ranging from several hundred megabytes to gigabytes, e.g., average 816MB and maximum 1.3GB in Synthetic-NeRF dataset [9].

Second, the model exhibits the characteristics of **heavy memory traffic with weak spatial locality**, especially in Step 3 in Figure 2 where we need to fetch density/color data from eight vertices on each sampling point. Furthermore, sampling points are generated along the ray, i.e., irrespective of the voxel's storage format. Thus, the voxels referenced by sampling points on the same ray are scattered within the memory, resulting in poor spatial locality in memory accesses thereby giving low chances of row buffer hits.

Figure 2 exemplifies the weak spatial locality behavior under heavy memory traffic. During the density stage, eight vertices must be accessed to fetch the density scalar data for tri-linear interpolation, requiring link data of the eight vertex indices first (Figure 3), followed by the fetch of density data

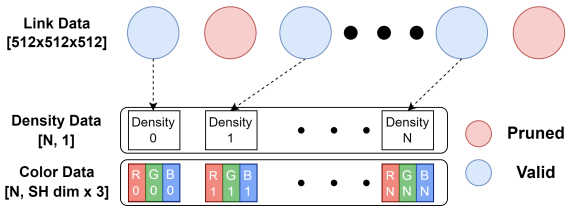


Fig. 3. An example of data structure of sparse voxel grid

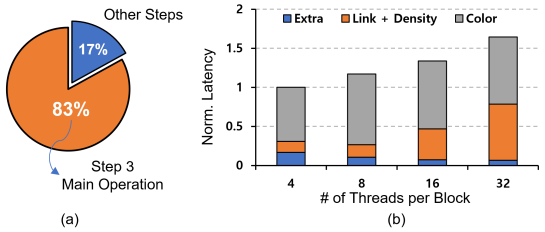


Fig. 4. Profiling results of [4] on RTX 2080ti (a) Latency breakdown at the Step level (b) Latency breakdown according to block dimension of GPU

using the link values. The key issue here is that the (link or density) data of the eight vertices cannot be sequentially read in a single memory access. Only a pair of two adjacent vertices can be stored continuously, while four pairs from eight vertices are scattered throughout the memory. This tends to result in four memory requests across different memory rows on the same or different banks for the link data and another four for the density data across different memory rows. In the color stage, the required data volume is even larger. Interpolation for RGB color requires fetching three color vectors from each of the eight vertices, totaling 24 vectors. Assuming a color vector dimension of 9 and half-precision [4], each vertex requires, for RGB color, fetching 3x9x2 bytes (three 9-dim fp16 vectors), i.e., 2 memory accesses (of 32 bytes each). Thus, for all the eight vertices, this amounts to 16 memory requests, mostly, across different rows in the memory.

Last but not least, both operations of tri-linear interpolation and dot product are characterized by a **low compute intensity (Ops/Byte)**. Specifically, the vertex data fetched for tri-linear interpolation are not reused unless the adjacent voxel has a sample point. Even in such a case, only half of the fetched vertex data can be reused since only four vertices are shared by two adjacent voxels.

Figure 4 shows the profiling results of running Plenoxels [4] on an RTX 2080ti. The utilized kernel was custom-implemented to calculate one ray per thread with the capability of profiling memory overhead, and this setup shows no performance difference from the official code. Since Plenoxels operates on a single CUDA kernel, we conducted detailed profiling by measuring latency while removing each step in Figure 2 and overlaying these results onto the total latency. Figure 4 (a) shows that a significant portion (83%) of the processing time is taken up by Step 3. Figure 4 (b) illustrates

the latency breakdown relative to GPU block size. The color stage accounts for the largest portion (52∼77%) of the latency across different numbers of threads/block. For link and density data, their latency share sharply increases with the number of threads/block, ranging from 13% to 44%. According to Nsight Compute [11], as the number of threads/block increases from 4 to 32, the L1 cache hit ratio decreases by 53%, and global memory traffic increases by 4.3 times. This indicates that voxel grid-based NeRF's memory traffic is significantly large with weak spatial locality, causing cache trashing that hampers GPU utilization.

B. Problems and Opportunities of PIM

PIM aims to overcome the CPU-memory bottleneck by performing computations within memory chips, leveraging high internal bandwidth.

The minimum compute-cost voxel grid-based NeRF model exhibits the characteristics that make it an ideal candidate for PIM application: large model size, heavy memory traffic, and low compute intensity. First, heavy memory traffic due to large model size can be better served with high internal memory bandwidth available inside of the PIM memory chip. The interpolation and dot product operations, which finally output scalar values, align well with the PIM architecture which specializes in vector operations including dot products. Additionally, the small amount of scalar outputs also helps reduce memory traffic to the host.

However, merely applying PIM to minimum compute-cost voxel grid-based NeRF models does not inherently leverage these advantages. Typically, PIM utilizes high internal bandwidth with an all-bank operation, which issues computation on all PIM banks with a single memory command with identical row and column addresses. If the data of eight vertices required for tri-linear interpolation are not aligned in the same row and column address on each bank, a single all-bank read operation cannot fetch all of them. In such a case, multiple memory requests will be necessary, potentially requiring as many requests as would be needed without using PIM. To address this, we propose a novel data layout and its management in Section IV which enables full exploitation of internal bandwidth in all-bank mode.

Minimum compute-cost voxel grid-based NeRF offers new opportunities to better exploit the memory bandwidth and compute capability on PIM architecture. Specifically, having multiple sample points in the same voxel enables us to reduce PIM-internal memory traffic by reusing fetched vectors.

IV. NeRF-PIM: Hardware Architecture

We provide an overview of the architecture and its key components (Section IV-A, D) followed by a detailed explanation of the data layout (Section IV-B) and PIM operations (Section IV-C). Finally, we explore optimization possibilities and discuss aspects of system integration (Section IV-E).

A. Architecture Overview

Figure 5 shows the structure of a channel in the NeRF-PIM architecture. The overall architecture is based on the HBM-PIM [8]. Thus, the PIM unit (PU) is allocated to every two banks and is directly connected to the banks, enabling efficient dot product operations by exploiting high internal bandwidth. Its architecture is based on the functional unit in AiM [12], comprising 16 FP16 multipliers and an adder tree that enables dot product computation. The adder tree incorporates a bank-wide MA shift (BWMS) scheme [12]. The interpolation unit (IU) is located in the peripheral logic area. It receives results from each PU and performs the interpolation. The detailed architecture of the IU is discussed in Section IV-D.

The NeRF-PIM architecture executes a NeRF model as follows. Initially, the host GPU processes Steps 1 and 2 (Figure 2) and sends the inputs of PIM operations to the memory in memory write commands. Each input consists of per-ray shared values such as the SH basis vector and per-sample coordinates within its voxel. As will be explained in Section IV-D, each input (of 32 bytes) accommodates up to 24 sampling points. Upon receiving these inputs, the NeRF-PIM hardware calculates the density and color results for each sampling point through NeRF-PIM operations (Step 3) and sends the results back to the host. Subsequently, the host performs alpha composition (Equation 1) to produce the pixel color (Step 4). The remainder of Section IV primarily discusses how NeRF-PIM efficiently accelerates Step 3.

Tri-linear interpolation operations are typically performed before dot product computation. However, as both operations are linear, altering their execution order does not change the result. Utilizing this characteristic, our design performs dot product operations on eight PIM units in parallel, reducing 8 pairs of vector inputs to eight scalar results, which are then sent to the IU for tri-linear interpolation. This approach, using PU and IU, reduces eight scalar (density) / vector (color) pairs to a single scalar density/color result, each in fp16. Thus, for each sampling point, we obtain 8 bytes (four fp16 data for density and three colors) of PIM result in total. The host reads, from the IU, the results of multiple (usually four) sampling points which are packed into a memory transfer of 32 bytes.

B. Data Layout

1) Inter-Bank Data Layout: We propose an all-bank conscious data layout that allows for full utilization of PIM's internal bandwidth. Figure 6 illustrates our data layout for color vectors of two voxels. When processing a sampling point within voxel 1, the data from its vertices (0, 1, 2, 3, 4, 5, 6, 7) become the input operands for dot product and tri-linear operations. Consequently, we place these eight data in the same location across all eight banks as shown in the figure. This arrangement is also applied to voxel 2, with vertices (1, 8, 3, 9, 5, 10, 7, 11) applying the same pattern of data placement.

Without our proposed data layout, scattered storage of the eight color vectors within memory necessitates multiple, possibly random, access requests. However, our layout allows the eight color vectors to be read in a single all-bank operation. This layout also reduces the memory traffic of link data. Without the proposed layout, it would be necessary to fetch link data for all eight vertices. However, as shown in Figure 6, only the link data of vertex 0 is required for reading the eight vertices' data of voxel 1. Knowing only the index of vertex 0's

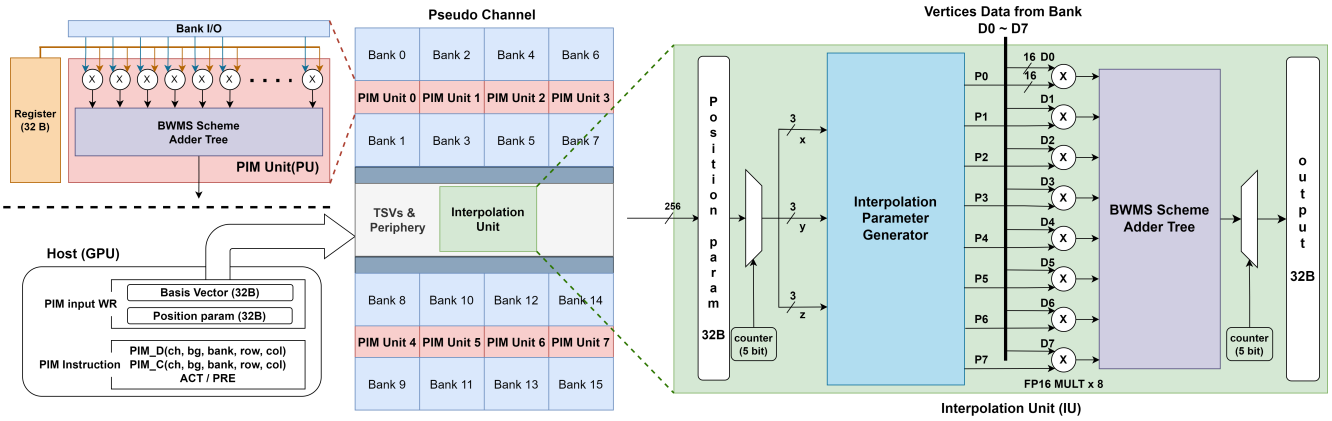


Fig. 5. NeRF-PIM architecture overview (single memory channel)

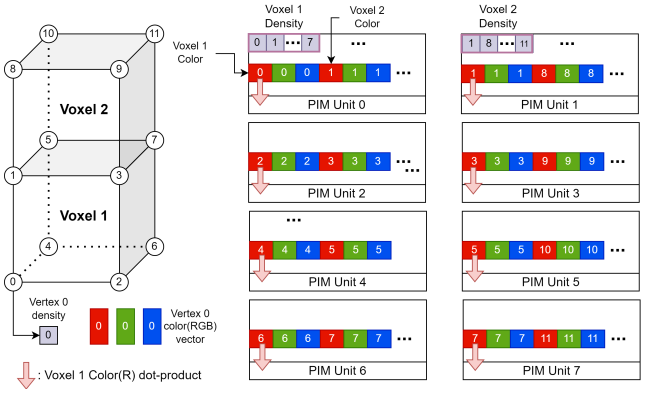


Fig. 6. A 2-voxel example of data layout for NeRF-PIM: For density data, the data from 8 vertices are sequentially stored. For color data, the data from 8 vertices are distributed across 8 banks. The data comprising one voxel are stored in the same row and column but in different banks.

data, the density data can be directly read, and the color data of all eight vertices can be read with a single all-bank read command. Hence, our data layout scheme not only enables efficient utilization of internal bandwidth but also significantly reduces memory requests for link data.

Implementing this data layout involves data duplications, as seen with vertices (1, 3, 5, 7) in Figure 6, which incurs overhead in runtime memory capacity and setup time. Only the density/color data of valid vertices are duplicated, neither the link data nor the pruned vertices. As a result, the overall impact on runtime memory size is significantly reduced, yielding an increase of approximately 3.34x on average for the Synthetic-NeRF [9] dataset. The increase in runtime memory size gets further reduced when adopting our proposed pruning method in Section V-B, which brings the average runtime memory overhead down to 2.62x. Note that the original model structure remains unchanged implying no increase in storage usage despite a larger runtime memory requirement.

The setup time duplication process also incurs overhead in setup time latency. However, its impact on overall frames per second (FPS) is negligible in real-time NeRF scenarios as it takes an initial single-time latency of 233 ms.

Note that the data layout approach differs on density data. Considering that dot product is not computed in the density stage and its smaller data size, leveraging internal bandwidth becomes unnecessary for density data. To avoid power consumption from all-bank operations, the eight density data of a voxel are stored sequentially, allowing a single burst read transmission to the IU.

2) Inter-Channel Data Distribution: The data layout of Figure 6 is applied to a channel. We also need to distribute voxel data across channels to exploit channel-level compute parallelism. To do that, we divide the entire grid into sub-grids of 8x8x8 voxels and distribute sub-grids across channels.

When determining the size of the sub-grid, the efficiency of PIM input transfer needs to be considered. For a PIM operation, it is necessary for the host to write to the memory both per-ray shared values (shared by all sampling points on the same ray), e.g., the ray's basis vector, and the coordinates of the sampling points. The more sampling points are packed into a single write, the fewer memory writes are needed, thereby contributing to faster PIM operations. In addition, packing multiple sampling points can increase the row buffer hit rate in the density stage. As shown in Figure 6, density and color data are stored on different rows. Consequently, performing the density stage consecutively for multiple sampling points, packed together, can lead to row buffer hits, whereas alternating between density and color stages for every sampling point gives no row buffer hit for the density stage.

The size of the sub-grids, set to 8x8x8, is chosen to contain as many sampling points as possible on a single DRAM row thereby improving the row buffer hit rate. A sub-grid can contain up to 512 valid data points. With data duplication, all density data within the same sub-grid can be stored in a single row. It means that during the density stage, row buffer misses are eliminated inside the sub-grid.

3) Channel Assignment for Load Balance: The performance of NeRF-PIM is bound by the channel with the most

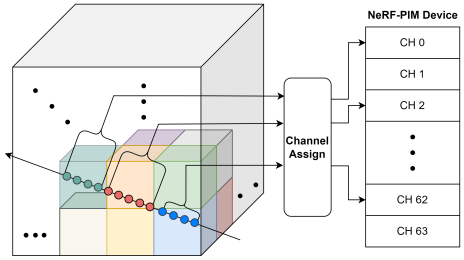


Fig. 7. Sub-grid distribution across channels using ML-based channel assignment

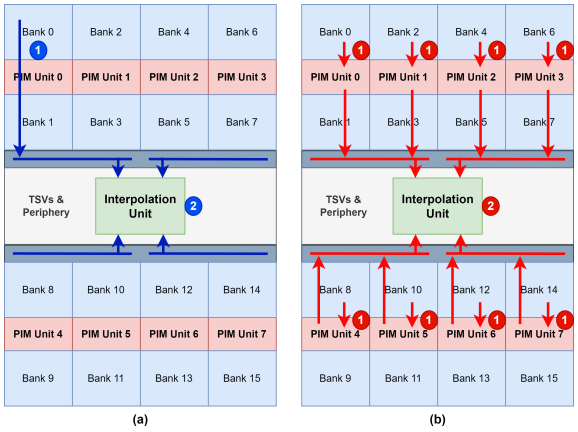


Fig. 8. (a) Density stage dataflow. (b) Color stage dataflow

computation. Therefore, when distributing multiple sub-grids across channels, load balance is essential. The amount of PIM operations per channel is not determined by the number of allocated sub-grids but by the amount of valid (i.e., unpruned) data contained in sub-grids. Consequently, we propose an ML-based channel assignment module that allocates sub-grids to channels considering the amount of valid data they contain.

We train a channel assignment module, i.e., a linear layer for optimal channel assignment for each target scene/object. It can be obtained with an additional one-time training of approximately 30 seconds on a trained NeRF model and, after training, provides a channel assignment as illustrated in Figure 7. Note that the channel assignment, once obtained by training, remains fixed for different camera positions.

C. NeRF-PIM Operation

Operation Overview The initial step in operating NeRF-PIM, as depicted in Figure 5, involves writing PIM inputs, i.e., packed data of basis vector and coordinates, from the host to the memory. Note that, as mentioned before, multiple sampling points (maximum 24 points as will be explained in Section IV-D) are packed into a single PIM input thereby reducing host-to-PIM traffic. Since the basis vector remains constant for each ray, it is shared across all sampling points on the ray. Thus, once the PIM inputs are transferred to the memory, the basis vector is broadcast and written into the PU's registers, while the coordinates are stored in the IU's input buffer.

Figure 8 illustrates each of the density and color stages. A key distinction between the two stages is that while both stages require tri-linear interpolation, the color stage also necessitates dot product operations between the basis and color vectors.

Table I shows the NeRF-PIM commands for these operations. Note that both commands have the same timing constraints as the existing HBM-PIM commands [8] as will be explained in follows.

Density Stage The density result is obtained through tri-linear interpolation. This operation, conducted by the PIM_D command, can be divided into two steps, as illustrated in Figure 8 (a). In step 1, the density data of eight vertices are fetched and sent to the IU. As mentioned in Section IV-B, these eight vertices' density data are sequentially stored

TABLE I
NERF-PIM COMMANDS

Name	Description
PIM_D	Fetch 8 density scalar data from bank, and then performs a tri-linear interpolation
PIM_C	Fetch 8 color vector data from even/odd banks, performs dot product in PIM unit, and then performs tri-linear interpolation

in a single row. At the same time, the IU generates interpolation parameters in the interpolation parameter generator (IPG) using the coordinates of the sampling points stored as PIM input. In step 2, using the fetched data and parameters generated by the IPG, IU performs tri-linear interpolation to obtain the density result. As steps 1 and 2 can be performed in parallel, two PIM_D commands can be pipelined. From the accessed bank's perspective, the PIM_D operation is identical to a standard RD command and thus shares the same timing constraints (tCCD_L, tCCD_S). As discussed in Section III-A, in a GPU-only system, generating one density result requires four random access memory requests. However, in NeRF-PIM, a single PIM_D operation suffices.

Color Stage To determine the color values, we perform dot product operations and tri-linear interpolation for each of the R, G, and B color vectors. This process, executed by the PIM_C command for each RGB color, can be divided into two steps, as shown in Figure 8 (b). In the first step, the color vectors of the eight vertices are fetched in parallel using an all-bank read operation and then delivered to each PU. Each PU performs a dot product operation with the basis vector stored in its registers. Note that our proposed data layout, explained in Section IV-B, enables us to perform only a single all-bank read operation to fetch the color vectors of eight vertices and perform dot product operations on them. The IPG generates interpolation parameters as in PIM_D operation for density. The second step involves each of the eight PUs sending its dot product result to the IU for tri-linear interpolation. Since the PIM_C command must be performed separately for R, G, and B, three PIM_C commands are required to compute the color values for a single sampling point. The PIM_C command also operates in a pipelined manner, similar to PIM_D, but the intervals between commands are tCCD_L [8], as the PIM_C command utilizes an all-bank operation. As explained in Section III-A, in the baseline GPU-only system, completing the operation for one sampling point incurs 16 memory read commands since each of eight vertices requires, for 3x9 fp16 data, two memory commands of 32 bytes each. In contrast, NeRF-PIM requires only three PIM_C commands.

D. Interpolation Unit

Figure 5 shows the interpolation unit (IU) which comprises an interpolation parameter generator (IPG), FP16 multipliers, and an adder tree. The adder tree employs the BWMS scheme used in the PU. The IU stores the coordinates of sampling points as input and sends the corresponding coordinates to the IPG upon receiving NeRF-PIM commands.

Figure 9 depicts the internal structure of the IPG. The IPG generates interpolation parameters using the x, y, and z

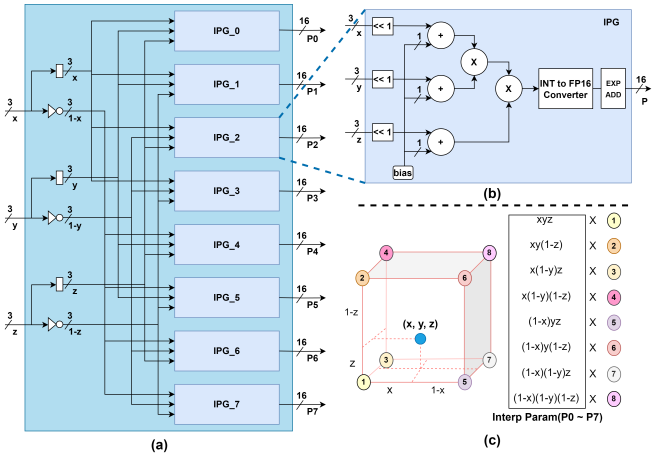


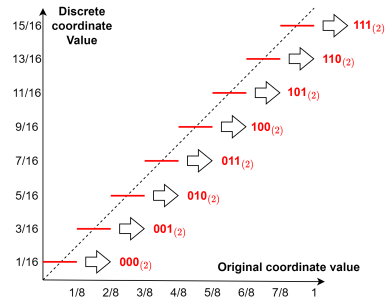
Fig. 9. (a), (b) Architecture of interpolation parameter generator (IPG). (c) Tri-linear interpolation

coordinates of a sampling point as input. Figure 9 (c) shows the equations for the interpolation operation and parameter generation. As illustrated in Figure 10, coordinate values ranging from 0 to 1 are discretized and represented with 3 bits for efficiency. This reduces the size of coordinates from 48 bits (FP16x3) to 9 bits (3bitx3) for each sampling point. Considering the size of data transfer (32 bytes) between GPU and memory, the coordinates of up to 24 sampling points can be packed into a single PIM input WR command of 32 bytes. The 3-bit representation reduces the number of PIM input WR commands and the area cost of IPG design at a negligible PSNR degradation of 0.02 dB for the Synthetic-NeRF dataset.

Upon receiving the 3-bit inputs for x, y, and z coordinates, the IPG generates interpolation parameters (P0 to P7, as shown in Figures 5 and 9). Binary inputs from $000_{(2)}$ to $111_{(2)}$, corresponding to float values from 1/16 to 15/16, are converted to the numerator of their float values (1 to 15) through shifting and bias addition, followed by multiplication. The omitted value of the denominator (1/16), being a power of 2, is taken into account in the final exponent adder. After being converted to fp16, the interpolation parameters are multiplied with the eight data fetched from the bank for interpolation operation.

E. System Integration

This section discusses how NeRF-PIM integrates with the baseline HBM-PIM system and operates when running an application end-to-end. NeRF-PIM is implemented based on the HBM-PIM system [8] and is compatible with the JEDEC



PIM Input Table		
Original Coordinate	PIM Input	Discrete Coordinate
0 ~ 1/8	$000_{(2)}$	1/16
1/8 ~ 2/8	$001_{(2)}$	3/16
2/8 ~ 3/8	$010_{(2)}$	5/16
3/8 ~ 4/8	$011_{(2)}$	7/16
4/8 ~ 5/8	$100_{(2)}$	9/16
5/8 ~ 6/8	$101_{(2)}$	11/16
6/8 ~ 7/8	$110_{(2)}$	13/16
7/8 ~ 1	$111_{(2)}$	15/16

Fig. 10. We propose discrete coordinate sampling. It converts real-value coordinate (0.0∼1.0) to discrete value in 3 bits as shown in the table

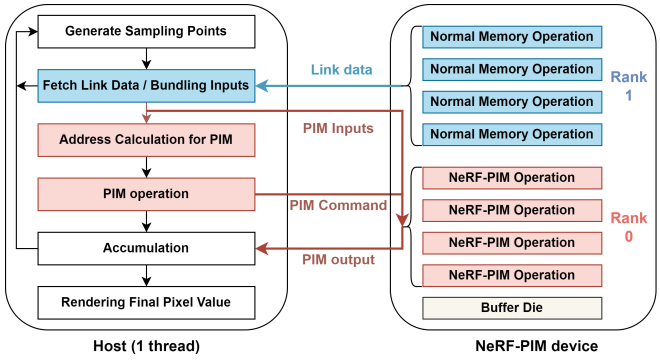


Fig. 11. Methodology of the NeRF-PIM system in executing the end-to-end NeRF application.

specification for HBM2 interface [13]. It is fully compatible with the HBM-PIM software stack and programming model [8], as its PIM memory manager seamlessly supports our data layout, and its PIM executor runs our NeRF-PIM kernel. For NeRF-PIM operation, the memory controller needs to support the NeRF-PIM commands listed in Table I. As explained in Section IV-C, the PIM_D and PIM_C commands share the same timing constraints as the HBM-PIM commands. NeRF-PIM is capable of performing both standard HBM2 and NeRF-PIM operations. It satisfies backward compatibility with the host memory controller managing and scheduling the memory state. Our NeRF-PIM architecture offers the practical advantage of being implementable with minimal changes on the host GPU and actual PIM devices [8].

Figure 11 illustrates an end-to-end NeRF model execution scenario. Initially, the host fetches link data and writes PIM inputs to the memory by normal memory operation. Using the link data, it calculates density/color data addresses and issues PIM commands to compute results. Each PIM command can generate one scalar output (density or one of RGB color). For every 16 PIM commands, the IU returns a packet of results for four sampling points (32 bytes = 4 sets of 8 bytes per-sample point density and RGB) as the read data of the last PIM command of the 16 ones. This procedure is repeated until all sampling point results for the current ray are received, after which the host computes the ray's color. The host runs multiple threads (each for a ray) concurrently. It results in a situation where normal memory operation and PIM operation can access the same memory channel. During the execution of PIM commands, the memory channel does not consume external memory bandwidth [8]. Utilizing this, we overlap memory accesses between link data (on a rank) and PIM operation (on another rank on the same memory channel).

V. NeRF-PIM: Software Optimization

We first describe the inefficiency issue of voxel grid-based NeRF algorithms (Section V-A) and propose optimizations to address it (Section V-B). Additionally, we suggest further optimizations to increase data reuse (Section V-C).

A. Motivation

Voxel grid-based NeRF algorithms first calculate density data for each sampling point and skip the color stage if the

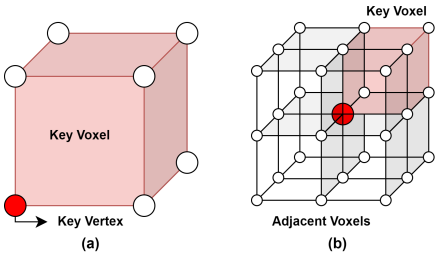


Fig. 12. Occupancy grid implementation with link data: (a) Key vertex indicating the pruning status of a voxel. (b) One key voxel and seven adjacent voxels containing the key vertex.

density is below a threshold, indicating a low probability of object surface. Profiling Plenoxels [4] reveals that only 19% of sampling points in the Synthetic-NeRF dataset proceed to the color stage. While density thresholding effectively skips 81% of unnecessary color stages, it also implies that 81% of the sampling points for fetching density data are redundant. Hence, we call this problem "density sampling inefficiency."

The problem of density sampling inefficiency is two-fold. First, it loses computational efficiency by spending memory bandwidth and compute resources for to-be-discarded density data. Second, it is unfriendly to PIM, possibly increasing its implementation cost. When density thresholding is done by the host, PIM operations cannot proceed until the density results are retrieved from memory, reducing PIM utilization. Implementing density thresholding within the memory, as a part of PIM function, requires a separate internal PIM controller to handle density-dependent color computation, increasing logic overhead. To address these issues, we propose optimizations for higher sampling efficiency and PIM-friendly sampling.

B. Occupancy Grid and Occupancy-Grid-aware Pruning

Applying Occupancy Grid To address the sampling inefficiency problem, we adopt an occupancy grid as in [1]. An occupancy grid divides 3D space into a grid, storing information about the presence of objects at each location. When sampling, the occupancy grid can reduce the number of sampling points by avoiding areas, i.e., voxels where no objects are present. Instead of creating a separate occupancy grid, we propose utilizing the link data from the sparse voxel grid structure in Figure 3 to implement an occupancy grid.

As shown in Figure 12 (a), among the eight vertices forming a voxel, the bottom-left vertex is designated as the key vertex. If the key vertex is invalid (i.e., pruned), we do not choose sampling points in the associated voxel (called key voxel) for the subsequent density/color computation. Thus, after determining the location of a sample point, we first fetch the link data of the key vertex (of the voxel where the sampling point resides). If it is pruned (i.e., the voxel is empty), then we skip subsequent computation for the sampling point. If it is valid, we proceed directly to the density/color stage.

Occupancy-Grid-Aware Pruning In the conventional vertex pruning for sparse voxel grid [4], we prune a vertex when two conditions are met. First, its density needs to be below a threshold. Even if this condition is met, if it has at least one valid neighbor vertex (among 26 vertices), then we do not prune it. We found that this approach is too conservative, thereby incurring the sampling inefficiency problem. To further enhance sampling efficiency, we propose a more aggressive pruning while minimizing the degradation of rendering quality.

Figure 12 (b) illustrates a key vertex and its associated key voxel. As the figure shows, the pruning of the key vertex can make a larger impact on its key voxel than on the other seven neighbor voxels (sharing the key vertex). Thus, when determining whether a vertex is pruned, it would be desirable to consider the impact of vertex pruning on its key voxel as well. Our proposed pruning method, which we call occupancy-grid-aware pruning, prunes a vertex when the following three conditions are all met: (1) its density is less than a threshold, (2) there is no adjacent voxel which has more than h_adj (=6 in our experiments) valid vertices, and (3) there are no more than h_key (=3) valid vertices in its key voxel.

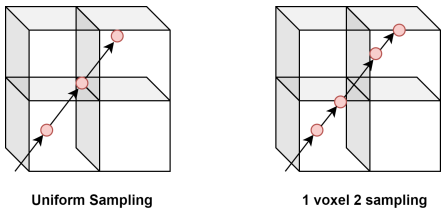


Fig. 13. One-voxel two-sampling method: enforcing two sampling points within a single voxel.

C. One-Voxel Two-Sampling Method

Traditional sampling methods sample points without considering voxel granularity, e.g., sampling at a fixed step size in Figure 13. We propose the one-voxel two-sampling (1V2S) method, which samples two points per voxel, allowing reuse of the dot product results at the eight vertices of the voxel.

Figure 13 illustrates how our proposed method samples points. First, assume that we sampled a point in a voxel. Then, as illustrated in the figure, we sample another point at the intersection between the ray and a face of the current voxel. The spacing between the two sample points in the same voxel can be different from the step size. We choose the third sampling point one step size away along the ray as the figure demonstrates. We sample another point at the intersection of a ray and a face of the new voxel. We repeat this process until we reach the far limit of the ray. As will be reported in the experiments, our proposed 1V2S method is effective in reducing memory accesses by reusing dot product results, which leads to significant performance improvement. Note that the location of sampling points is determined by the host GPU.

VI. Methodology

Workloads We conducted our experiments using the Synthetic-NeRF [9] and (a subset of) Tanks and Temples [10] datasets. Synthetic-NeRF is comprised of eight synthetic scenes. Tanks and Temples is comprised of real, unbounded, 360° scenes. We used four scenes based on the provided official checkpoint.[2]

[2]The checkpoints of the Tanks and Temples dataset are not reproducible. We could train four scenes, M60, Playground, Train, and Truck, in the same environment provided by the official checkpoints and applied our optimizations to them.

TABLE II
HBM-PIM DEVICE SPECIFICATION [8]

External clocking frequency	1 ∼1.2GHz
Timing parameters	Same as HBM2
# of pCH	16
# of PCU per pCH	8
# of banks per pCH	16
On-chip (Compute) bandwidth	1TB/s ∼ 1.229TB/s
Off-chip (I/O) bandwidth	256GB ∼ 307.2GB/s
Capacity	6GB

TABLE III
IMPLEMENTATION RESULTS OF NERF-PIM MODULES

Units	Area (um^2)	Power (mW)
Interpolation unit	5274	0.82
PIM unit	9345	1.50

Software baselines The proposed NeRF-PIM is implemented based on the official custom Pytorch CUDA extension library of Plenoxels [4], which supports volume rendering on sparse voxel grid, and we integrated it into HBM-PIM software stack [8]. However, the CUDA kernel of this library is not designed to profile memory traffic. Consequently, we developed a new custom kernel. In our experiments, we evaluated both kernels and reported the superior one.

Hardware baselines and system setup To evaluate the performance of NeRF-PIM, we used two hardware baselines: HBM-PIM and GPU-only systems. We employed the evaluation system used in [8], called HBM-PIM system, which comprises four integrated HBM-PIM devices in an AMD MI100-PIM GPU. The detailed specifications of the HBM-PIM device are presented in Table II. The GPU-only system adopts the same evaluation system without utilizing PIM capability. Note that NeRF-PIM hardware, as well as HBM-PIM hardware, is an in-memory co-processor to the GPU.

Our NeRF-PIM architecture is based on the HBM-PIM architecture and adheres to the same timing constraints as the HBM-PIM commands [8]. The additional NeRF-PIM commands share identical latency with HBM-PIM commands as explained in Section IV-C. Thus, by exploiting the characteristics of deterministic latency on DRAM commands, we can reliably measure the performance of the NeRF-PIM system on the HBM-PIM system. Specifically, to measure the runtime of NeRF-PIM, we execute the PyTorch NeRF model on the GPU and the model issues the existing PIM commands (via the memory controller on the GPU) that have the same latency as our proposed PIM commands, e.g., RD command on behalf of PIM_D. Considering the pipelined operation, we inserted a dummy PIM command after the final PIM_C command to ensure the completion of interpolation.

From the perspective of software-side code running on the GPU, the implementations of the software-only and our proposed hardware-software co-design differ slightly. This is because the PIM design requires additional operations, such as buffering for PIM inputs and inter-thread communication, to efficiently issue PIM commands. However, these additional operations adversely affect only the performance of our PIM design. Furthermore, as discussed in Section IV-E, the parallel operation of normal memory operation rank and PIM operation rank is also implemented in the HBM-PIM system.

TABLE IV
COMPARISON BETWEEN THE BASELINE AND NERF-PIM SOFTWARE-ONLY IMPLEMENTATION

Dataset	Methods	PSNR	Valid Data	Sampling Eff.
Synthetic-NeRF [9]	**Plenoxels [4]**	31.71	4811341	19%
	NeRF-PIM	31.66	2876115	100%
Tanks&Temples [10]	**Plenoxels [4]**	20.12	16701993	23%
	NeRF-PIM	20.14	7927691	100%

The hardware modules of NeRF-PIM were synthesized in Verilog to estimate the silicon area, timing, and power consumption of each module. We used the Synopsys Design Compiler with TSMC's 45nm technology. In line with prior works [14], [15], we scaled our implementation results to 22nm technology using the scaling factor from [16]. To account for the differences between DRAM and logic processes, we adopted the method from [17]. The implementation results of our module are shown in Table III.

VII. EVALUATION

A. NeRF-PIM Ideas on Software Only Implementation

Table IV compares the baseline [4] and the SW-only implementation of our NeRF-PIM ideas. Our occupancy-grid-aware pruning results in a substantial 40% reduction in the amount of valid data while only incurring a minor PSNR degradation of 0.05 dB, alleviating the runtime memory overhead.

Figure 14 shows the performance improvements from our optimization methods. The occupancy grid (OG) alone enhances the performance by 1.52 times and increases sampling efficiency by 35%. Occupancy-grid-aware pruning (OGPR) further reduces valid data, thereby offering an additional 1.4 times speedup and a 26% increase in sampling efficiency. Thus, these two methods collectively achieve a speedup of 2.14 times and an 80% improvement in sampling efficiency. Lastly, the application of the one-voxel two-sampling (1V2S) method exploits data reuse while contributing to a further speedup (of NeRF-PIM in the figure).

Sensitivity analysis on the occupancy-grid-aware pruning Figure 15 shows the benefits of the proposed occupancy-grid-aware pruning and its sensitivity analysis based on hyperparameter variations. When both hyperparameters (h_key, h_adj) are set to values below or equal to 3, there is limited pruning, resulting in negligible performance differences. However, transitioning from (3,3) configuration to our approach of (3,6) setting, which applies stricter criteria for adjacent voxels, we observe a minor PSNR degradation of only 0.05 dB, while

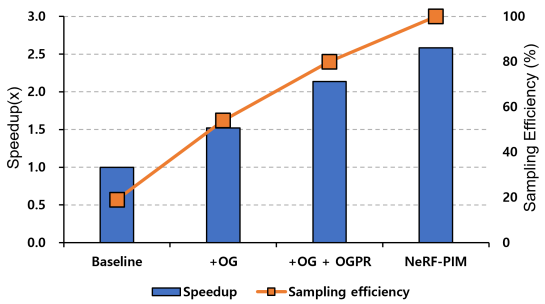


Fig. 14. Analysis of speedup and sampling efficiency for each NeRF-PIM's software optimization method on the Synthetic-NeRF Dataset [9]

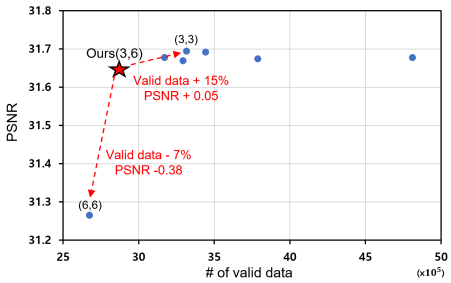


Fig. 15. Sensitivity analysis on occupancy-grid-aware pruning.

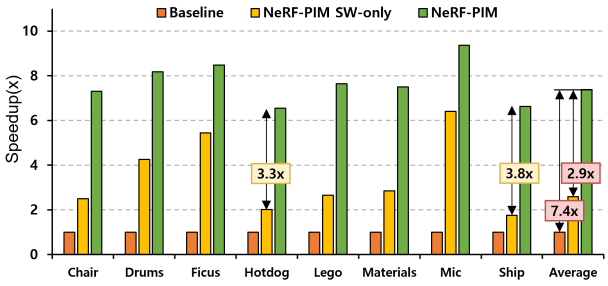


Fig. 16. Speedup of our NeRF-PIM software and hardware-software co-design on the Synthetic-NeRF dataset [9]

the number of valid data decreases by 14%. Changing the parameters from our (3,6) to a (6,6) setting leads to a further reduction in valid data by 7%, but with a significant PSNR degradation of 0.38 dB. These results highlight the different sensitivities of vertex pruning on the key and adjacent voxels. Considering the trade-off of pruning and PSNR degradation, we choose the (h_key, h_adj) configuration as (3,6).

B. NeRF-PIM Hardware Evaluation

Figure 16 shows the speedup enabled by our proposed NeRF-PIM ideas on the SW-only (NeRF-PIM SW-only) design and hardware-software (NeRF-PIM) co-design. NeRF-PIM offers a substantial average speedup of 7.4x over the case of running the original NeRF model on the GPU only (Baseline). Furthermore, when comparing the NeRF-PIM SW-only design (in Figure 14), we observe an additional 2.9x speedup on the NeRF-PIM hardware-software co-design.

The success of NeRF-PIM hardware in accelerating voxel grid-based NeRF can be attributed to three key factors. First, by leveraging large internal bandwidth, NeRF-PIM reduces memory traffic. For example, in the color stage, the baseline and NeRF-PIM SW-only designs, which use only a GPU without PIM hardware, require 16 memory requests to fetch the color vectors of 8 vertices (each vertex requires two memory requests of 32 bytes for 3*9 fp16 data). In contrast, NeRF-PIM needs only three PIM commands. Secondly, the proposed data layout for all-bank mode enables us to minimize PIM command overhead. In addition, the sub-grid level data placement across channels enables us to perform the density stage by accessing only a single row on each sub-grid, without incurring any bank conflicts. Finally, the hardware-software co-design helps enhance the cache utilization of the host. Unlike GPU systems suffering from cache thrashing as exemplified in Figure 4, NeRF-PIM enables the host to utilize the cache mostly for link data while density and color data

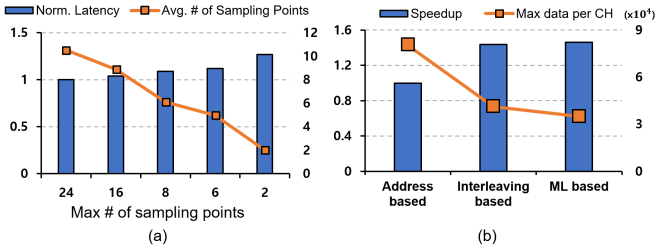


Fig. 17. (a) Normalized latency and average number of sampling points per PIM operation w.r.t. maximum number of sampling points (b) Speedup and maximum number of valid data per channel with channel assignment method.

are accessed inside the PIM chip. Furthermore, as explained in Section IV-B, NeRF-PIM also reduces memory traffic for link data by eight times.

Effectiveness of the discrete coordinate sampling Figure 17 (a) illustrates the normalized latency and average number of sampling points per PIM command with respect to the maximum number of sampling points per PIM command. Our discrete coordinate sampling approach, which represents the coordinates of sampling points with 3 bits, allows us to pack up to 24 sampling points per PIM input. As the figure shows, when the maximum number of packed sample points per PIM input gets reduced, the normalized latency of the entire model gets increased, which demonstrates the effects of packing, a reduced number of PIM input WR commands, and enhanced row buffer hit rate for density data, enabled by the discrete coordinate sampling, on the latency of the entire model.

Load balance Figure 17 (b) shows the effect of load balancing. In a typical address-based assignment where the channel assignment follows the address of data, the channel with the most accesses determines the overall performance. In the Interleaving-based method where the channel-level interleaving is applied at sub-grid granularity, a more balanced workload is assigned across channels as shown in the figure (Max data per CH). Our ML-based approach reduces the data volume in the most loaded channel by 15% compared to the Interleaving-based method. However, as the figure shows, our method offers an overall performance gain of only about 2%. We will discuss the weak correlation of (per-channel) workload and total runtime in Section VIII.

In the experiments rendering images across various viewpoints on the Lego dataset, our method significantly reduced the maximum load (the number of sampling points each channel is responsible for), outperforming the Interleaving-based method by 1.26x (1.22x to 1.32x), demonstrating its superior capability in load balancing across different viewpoints.

Sensitivity analysis of the hardware-software co-design Figure 18 demonstrates the effectiveness of our co-design approach. The configuration of NeRF-PIM SW-only represents the case where GPU runs the NeRF, applying our software optimizations. The configurations of PU (+DL) and IU show the effect of hardware-software co-design. PU (executing dot products on PIM hardware) alone gives a marginal (average 1.15x) speedup due to inefficient utilization of internal memory bandwidth. However, DL (our data layout) boosts performance (2.49x speedup) by better utilizing PUs, i.e., internal memory bandwidth via all-bank mode operations.

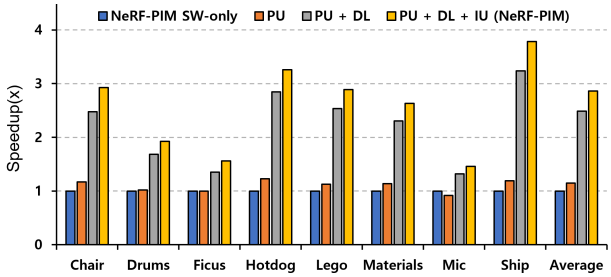


Fig. 18. The speedup achieved by our proposed NeRF-PIM's hardware modules and co-design on the Synthetic-NeRF dataset [9]

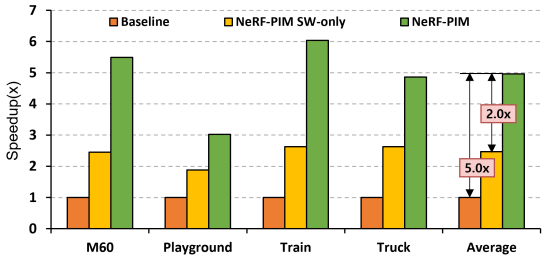


Fig. 19. Speedup of our NeRF-PIM software and hardware-software co-design on the Tanks and Temples dataset [10]

IU further increases speedup (2.86x speedup) by moving interpolation from GPU to PIM hardware, thereby reducing memory traffic otherwise incurred for interpolation on GPU.

Considering that the NeRF-PIM SW-only configuration also corresponds to the baseline HBM-PIM, which does not accelerate dot product and interpolation operations, Figure 18 demonstrates that NeRF-PIM improves the existing HBM-PIM by equipping it with a new PU, the proposed data layout, and memory-side interpolation.

C. Evaluation on Real 360° Scenes

We additionally evaluated NeRF-PIM on the Tanks and Temples dataset [10] to capture its performance characteristics in real, unbounded, and 360° scenes. Table IV illustrates that our occupancy-grid-aware pruning results in a 53% reduction of valid data without any image quality loss. Figure 19 shows a total 5x speedup enabled by our hardware-software (NeRF-PIM) co-design. Our software optimizations achieved a 2.5x speedup, and our hardware-software co-design further increased performance by 2x. The results still show a tendency for larger models to achieve greater speedups. For instance, the scene 'Train' gives a 4 times larger model than 'Playground.'

NeRF-PIM still demonstrates robust performance with a 5x speedup on the real-world dataset. However, the speedup is slightly lower than that of the Synthetic-NeRF dataset [9]. The difference is due to the presence of a background model, which is not accelerated by PIM. As the background model occupies 14% of the total latency, it diminishes the impact of our NeRF-PIM. Without the background model, NeRF-PIM achieves a total speedup of 7.9x.

VIII. Discussion & Ablation

Architectural benefits of NeRF-PIM w.r.t. software-only design The performance of the NeRF-PIM SW-only design (yellow bar of Figure 16) did not include our data layout

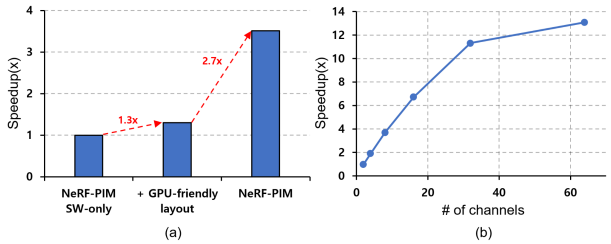


Fig. 20. (a) NeRF-PIM SW-only vs. NeRF-PIM hardware-software co-design and (b) speedup vs. the number of PIM channels

because the key benefit of our data layout comes from all-bank mode which is not available on the GPU. To fairly assess our architecture compared to GPU, we further implemented a GPU-friendly data layout, ensuring that the data for eight vertices within a single voxel are stored sequentially through data duplication, to avoid irregular memory accesses.

Figure 20 (a) shows a 1.3x speedup when the NeRF-PIM SW-only design adopts the GPU-friendly layout. As the figure shows, NeRF-PIM acceleration gives a further 2.7x speedup to the SW-only design adopting the GPU-friendly layout. Given that the NeRF-PIM architecture's internal bandwidth is 4x larger than the external bandwidth, an end-to-end runtime speedup of 2.7x indicates that NeRF-PIM efficiently accelerates the NeRF model. In this comparison, we chose not to employ the 1V2S method to ensure a fair assessment, as it yields varying degrees of speedup for the GPU and PIM systems.

Performance scalability of NeRF-PIM Figure 20 (b) presents performance scalability with respect to the number of channels. As the number of channels increases, the performance of NeRF-PIM increases almost linearly up to 16 channels. This growth is attributed to each channel independently executing PIM operations, enhancing the overall compute capability. However, performance improvements start to weaken after 16 channels, indicating that the entire set of channels is not fully utilized. This weak scaling may stem from the limited issue capabilities of PIM commands in the baseline HBM-PIM. Primarily, this is due to our host ISA's requirement of utilizing two threads to generate a single PIM command [8], which necessitates inter-thread communication and effectively halves the command issue capabilities.

Trade-offs and limitations As a summary, compared with the baseline, which runs the original NeRF model on the GPU only, NeRF-PIM provides a 7.4x speedup at the one-time cost of setup time (233ms) and training time (30 seconds), with an unnoticeable loss of image quality (0.07dB), a runtime-only overhead of 2.62x larger memory usage, and a very small area cost of additional circuits. The total speedup of 7.4x is decomposed into 2.6x by our software optimizations, 2.5x by our data layout and PIM unit that accelerate the dot product, and 1.2x by interpolation unit. For higher performance, the current NeRF-PIM would need to address the issues of low ratio of internal to external memory bandwidth (currently, 4) and limited issue capabilities of PIM commands (as explained above). Addressing those issues is left for future work.

Regarding the experimental methodology, our results are also susceptible to the approximations (e.g., the usage of 45nm

logic and scaling to estimate 20nm DRAM circuit cost) and the mismatches between the codes for the NeRF-PIM SW-only design and hardware-software (NeRF-PIM) co-design (mentioned in Section VI).

Generality of NeRF-PIM Since our NeRF-PIM architecture supports dot product and tri-linear interpolation, it is also compatible with other minimum compute-cost voxel grid models like [7] sharing the same characteristics of minimizing compute-cost down to a few dot products per sampling point. Our proposed NeRF-PIM can also be used to accelerate the previously trained NeRF models, e.g., based on MLP or Gaussian splatting, by training Plenoxels with images generated from the pre-trained model to leverage the high-performance rendering of our NeRF-PIM.

IX. Related Work

Voxel grid-based NeRF To address the limitations of NeRF [9], various studies [1], [3]–[6] have used 3D grid structures to reduce computational demands. [1] factorizes the grid into low-rank tensor components, while [3] uses a voxel grid with a hash table, both improving efficiency but still requiring MLP computations. [5] adopts octree structures, replacing MLP with SH operations for faster rendering but leading to slow training and larger models. In [4], the authors propose a sparse voxel grid without neural components, which offers faster training and inference as well as smaller model sizes.

NeRF accelerators For efficient NeRF processing, several NeRF-specific accelerators have been presented. [18] employs a hardware-software co-design to expedite the rendering of [1]. [19] enhances the reconstruction speed of [3], while [20] focuses on boosting its rendering performance. However, despite these advancements [18]–[20], accelerating larger models without MLP remains challenging due to their extremely low computational intensity and high memory demands.

X. Conclusion

In this study, we propose NeRF-PIM, a hardware-software co-design aimed at efficiently accelerating voxel grid-based NeRF using PIM. The NeRF-PIM ideas adopted in our software-only implementation significantly enhance performance over conventional algorithms through optimizations that improve sampling efficiency and leverage data reuse. Moreover, the NeRF-PIM hardware further improves performance by efficiently utilizing internal memory bandwidth. We evaluated the proposed NeRF-PIM based on real system execution and obtained significant (up to 7.4x) speedup over the baseline software-only design.

Acknowledgments

This work was supported by Samsung Advanced Institute of Technology, and Memory Division, Samsung Electronics Co., Ltd., and Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) [NO. RS-2021-II211343, Artificial Intelligence Graduate School Program (Seoul National University)]. Additionally, we would like to thank Euntae Choi for his insightful discussions and contributions to the manuscript revision.

References

[1] A. Chen, Z. Xu, A. Geiger, J. Yu, and H. Su, "Tensorf: Tensorial radiance fields," in *European Conference on Computer Vision*. Springer, 2022, pp. 333–350.

[2] C. Reiser, S. Peng, Y. Liao, and A. Geiger, "Kilonerf: Speeding up neural radiance fields with thousands of tiny mlps," in *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 2021, pp. 14 335–14 345.

[3] 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.

[4] 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*, 2022, pp. 5501–5510.

[5] A. Yu, R. Li, M. Tancik, H. Li, R. Ng, and A. Kanazawa, "Plenoctrees for real-time rendering of neural radiance fields," in *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 2021, pp. 5752–5761.

[6] L. Liu, J. Gu, K. Zaw Lin, T.-S. Chua, and C. Theobalt, "Neural sparse voxel fields," *Advances in Neural Information Processing Systems*, vol. 33, pp. 15 651–15 663, 2020.

[7] S. J. Garbin, M. Kowalski, M. Johnson, J. Shotton, and J. Valentin, "Fastnerf: High-fidelity neural rendering at 200fps," in *Proceedings of the IEEE/CVF international conference on computer vision*, 2021, pp. 14 346–14 355.

[8] S. Lee, S.-h. Kang, J. Lee, H. Kim, E. Lee, S. Seo, H. Yoon, S. Lee, K. Lim, H. Shin *et al.*, "Hardware architecture and software stack for pim based on commercial dram technology: Industrial product," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2021, pp. 43–56.

[9] 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.

[10] A. Knapitsch, J. Park, Q.-Y. Zhou, and V. Koltun, "Tanks and temples: Benchmarking large-scale scene reconstruction," *ACM Transactions on Graphics (ToG)*, vol. 36, no. 4, pp. 1–13, 2017.

[11] NVIDIA, "Nvidia nsight compute," 2024. [Online]. Available: https://developer.nvidia.com/nsight-compute

[12] S. Lee, K. Kim, S. Oh, J. Park, G. Hong, D. Ka, K. Hwang, J. Park, K. Kang, J. Kim *et al.*, "A 1ynm 1.25 v 8gb, 16gb/s/pin gddr6-based accelerator-in-memory supporting 1tflops mac operation and various activation functions for deep-learning applications," in *2022 IEEE International Solid-State Circuits Conference (ISSCC)*, vol. 65. IEEE, 2022, pp. 1–3.

[13] S. JEDEC, "High bandwidth memory (hbm) dram," *JESD235*, 2013.

[14] L. Wu, R. Sharifi, M. Lenjani, K. Skadron, and A. Venkat, "Sieve: Scalable in-situ dram-based accelerator designs for massively parallel k-mer matching," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2021, pp. 251–264.

[15] M. Zhou, W. Xu, J. Kang, and T. Rosing, "Transpim: A memory-based acceleration via software-hardware co-design for transformer," in *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2022, pp. 1071–1085.

[16] A. Stillmaker, Z. Xiao, and B. Baas, "Toward more accurate scaling estimates of cmos circuits from 180 nm to 22 nm," *VLSI Computation Lab, ECE Department, University of California, Davis, Tech. Rep. ECE-VCL-2011-4*, vol. 4, p. m8, 2011.

[17] A. Yazdanbakhsh, C. Song, J. Sacks, P. Lotfi-Kamran, H. Esmaeilzadeh, and N. S. Kim, "In-dram near-data approximate acceleration for gpus," in *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*, 2018, pp. 1–14.

[18] C. Li, S. Li, Y. Zhao, W. Zhu, and Y. Lin, "Rt-nerf: Real-time on-device neural radiance fields towards immersive ar/vr rendering," in *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design*, 2022, pp. 1–9.

[19] S. Li, C. Li, W. Zhu, B. Yu, Y. Zhao, C. Wan, H. You, H. Shi, and Y. Lin, "Instant-3d: Instant neural radiance field training towards on-device ar/vr 3d reconstruction," in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, 2023, pp. 1–13.

[20] J. Lee, K. Choi, J. Lee, S. Lee, J. Whangbo, and J. Sim, "Neurex: A case for neural rendering acceleration," in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, 2023, pp. 1–13.