NeuRex: A Case for Neural Rendering Acceleration

Junseo Lee Kwanseok Choi Jungi Lee Seokwon Lee Joonho Whangbo Jaewoong Sim
Seoul National University
{junseo.lee, kwanseok.choi, jungi.lee, seokwon.lee, joonho0320, jaewoong}@snu.ac.kr

ABSTRACT

This paper presents NeuRex, an accelerator architecture that efficiently performs the modern neural rendering pipeline with an algorithmic enhancement and supporting hardware. NeuRex leverages the insights from an in-depth analysis of the state-of-the-art neural scene representation to make the multi-resolution hash encoding, which is the *key* operational primitive in modern neural renderings, more *hardware-friendly* and features a specialized hash encoding engine that enables us to effectively perform the primitive and the overall rendering pipeline. We implement and synthesize NeuRex using a commercial 28nm process technology and evaluate two versions of NeuRex (NeuRex-Edge, NeuRex-Server) on a range of scenes with different image resolutions for mobile and high-end computing platforms. Our evaluation shows that NeuRex achieves up to 9.88× and 3.11× speedups against the mobile and high-end consumer GPUs with a substantially small area overhead and lower energy consumption.

CCS CONCEPTS

• **Computer systems organization → Neural networks**; • **Computing methodologies → Rendering**.

KEYWORDS

Neural rendering, NeRF, neural networks, machine learning, accelerators

ACM Reference Format:
Junseo Lee, Kwanseok Choi, Jungi Lee, Seokwon Lee, Joonho Whangbo, Jaewoong Sim. 2023. NeuRex: A Case for Neural Rendering Acceleration. In *Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA '23), June 17–21, 2023, Orlando, FL, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3579371.3589056

1 INTRODUCTION

Neural rendering is a new and rapidly emerging approach that synthesizes photo-realistic images or videos in a controllable way using deep neural networks (DNNs) [55]. By encoding scenes and objects in the weights of deep neural networks, neural rendering *implicitly* maps input coordinates into some numeric values such as colors or radiance. Compared to traditional *explicit* 3D representations such as polygonal meshes, voxels, or point clouds, *implicit* neural scene representations allow for capturing the fine details of complex surfaces or shapes in a more compact way.

While neural rendering is a promising approach to a variety of tasks in graphics such as image super-resolution [11, 33] and novel view synthesis [35, 39, 51], it requires a significant amount of computation to achieve high-quality renderings. Conventional neural rendering is based on the multi-layer perceptron (MLP) network, which consists of a set of fully-connected layers. The MLP needs to be queried millions of times to render an image because *every* sample point along the ray for *each* pixel needs to run through the neural networks to produce an output value that corresponds to the input coordinate. This makes the neural rendering process extremely slow even on the high-end consumer GPUs.

As such, there has been a plethora of recent works that aim to reduce the training and rendering time of neural representations via algorithmic enhancements [10, 17, 18, 23, 30, 37, 38, 48, 52, 58]. Despite the active research in the graphics community and the importance of neural scene representations, however, there has been little to no work that systematically evaluates the performance of the workload on today's hardware systems and helps understand its architectural implications from the hardware perspective.

In this work, we start by investigating the characteristics of modern neural rendering algorithms and present an in-depth characterization of several representative models to understand their architectural implications along with compute and memory requirements. In particular, we performed a detailed characterization of the state-of-the-art neural scene representation [37] that substantially reduces the training and rendering time while also improving the quality of rendered views compared to others. To do so, instead of using a large MLP with simple input encodings, the state-of-the-art exploits the direction of using a smaller MLP with multiple *hash encoding tables* that contain *trainable* feature vectors (i.e., input encoding parameters), each of which captures different grid resolutions.

Although it performs significantly better than prior works in both rendering time and quality, we observe that the *multi-resolution hash encoding* primitive used in the state-of-the-art model is *not* hardware-friendly and leads to several challenges and inefficiencies in executing the neural rendering pipeline on general-purpose computing platforms. Our profiling results on commodity GPUs reveal that it takes more time to perform multi-resolution hash encodings than MLP computation, and these two operations are *serialized* in execution. In addition, due to the irregular access nature of hash tables, the *large* encoding table needs to fit into the on-chip cache; otherwise, the time spent on encoding lookups significantly increases, and so does the overall training and rendering time. Furthermore, each hash entry access only uses four out of 64 bytes of data from a cacheline or off-chip memory, leading to a substantial waste of the memory bandwidth. The compute cores

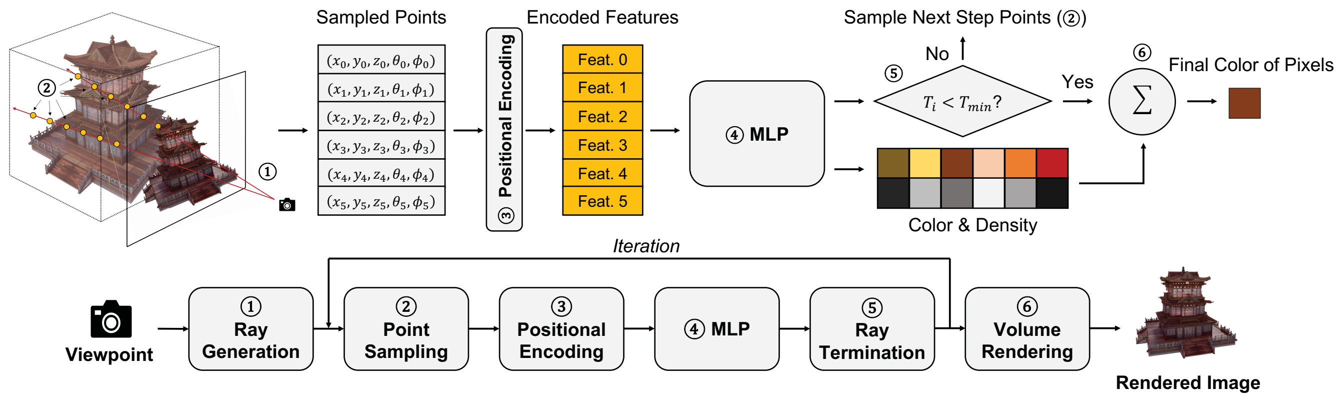

Figure 1: A volume rendering pipeline with Neural Radiance Fields (NeRF).

are also underutilized because the MLP is small in size, and they are mostly idle when performing input encodings and hash table lookups. In short, general-purpose GPUs are an imbalanced design point and are inefficient in running the state-of-the-art neural rendering models.

In this paper, we present NeuRex, a neural rendering accelerator that efficiently performs modern neural graphics computation by making changes in the execution flow of the rendering pipeline with algorithmic enhancements and supporting hardware. NeuRex builds on the key observations from our in-depth analysis to make multi-resolution hash encodings more *hardware-friendly* and features a specialized hash encoding engine that enables us to effectively perform the primitive and the overall neural rendering pipeline.

The key idea behind our algorithmic enhancement is to partition the input coordinate grid into *several subgrids*, each of which owns a *portion* of a large hash encoding table. We then arrange the processing of input coordinates such that we complete processing one subgrid for all resolutions before moving onto another. This restricts hash table access to a range of consecutive entries, thereby allowing the hardware accelerator to load *only* a part of the hash table to the on-chip memory at a time; thus, hardware accelerators do not need to employ a multi-megabyte on-chip memory to perform the multi-resolution hash encoding primitive efficiently. This also enables the opportunities to break the serialized execution of input encodings and MLP computation and overlap these two operations effectively with supporting hardware, thereby leading to better utilization of overall compute and memory resources in the accelerator.

We implement the hardware components of NeuRex in RTL and synthesize them using a commercial 28nm process node. For performance evaluation with a detailed off-chip memory timing model, we build a cycle-level simulator that models the NeuRex architecture and evaluate it on a set of popular tasks and datasets in graphics. Our evaluation shows that two variants of NeuRex achieve up to 9.88× and 3.11× speedups compared to the representative mobile (Jetson Xavier NX; Volta GPU; 12nm) and high-end consumer (RTX 3070; Ampere GPU; 8nm) computing platforms, with a small area budget of 3.14mm^2 and 21.37mm^2. In summary, this paper makes the following contributions:

- To our knowledge, this is the *first* work to comprehensively analyze the performance bottlenecks of the modern neural scene representation on today's computing platforms and identify the root causes of the performance inefficiencies.
- We propose an algorithmic enhancement that makes multi-resolution hash encodings more *hardware-friendly* to efficiently perform the primitive *without* the need for a multi-megabyte on-chip memory.
- We present NeuRex, a hardware accelerator that effectively performs neural graphics computation by *minimally* extending the existing DNN accelerators. It features a specialized hash encoding engine tailored to the needs of modern neural renderings.

2 BACKGROUND

In this section, we briefly introduce neural scene representations and discuss the characteristics of the ML-based rendering method.

2.1 Neural Rendering

Neural rendering combines the ideas from classical computer graphics with the recent advances in deep neural networks to render images or videos. At a high level, a neural rendering pipeline learns a representation of a mathematical function that parameterizes a 2D or 3D scene using a multi-layer perceptron (MLP) as a function approximator. Depending on the tasks and objectives, the MLP learns different *implicit* representations such as the mapping from 2D coordinates to RGB colors of an image [32] or the mapping from 3D coordinates to the distance to a surface [54]. Note that although the specific tasks may differ, they share the common idea of using MLPs as function approximators. In the following, we take as a representative task a recent breakthrough of neural representations for volume rendering, called Neural Radiance Fields (NeRF), to discuss state-of-the-art algorithms in this field.

2.2 Neural Radiance Fields (NeRF)

A Neural Radiance Field (NeRF) is a method that generates *novel* views of complex 3D scenes from a *partial* set of 2D images. In NeRF, scenes are represented *implicitly* in the weights of an MLP

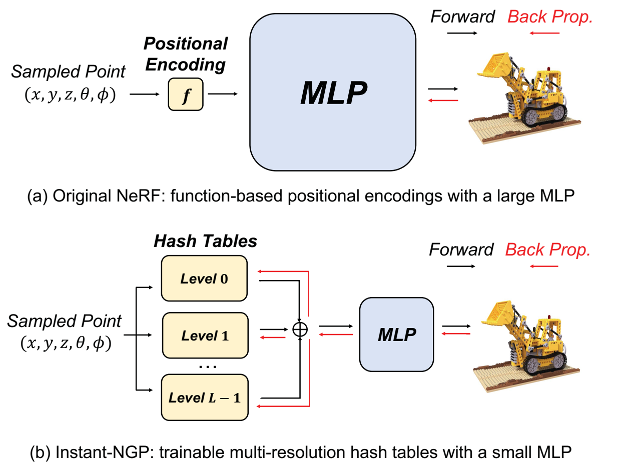
Positional Encoding
Sampled Point (x, y, z, θ, ϕ)
f
MLP
Forward
Back Prop.
(a) Original NeRF: function-based positional encodings with a large MLP
Hash Tables
Level 0
Level 1
Level $L-1$
Sampled Point (x, y, z, θ, ϕ)
MLP
Forward
Back Prop.
(b) Instant-NGP: trainable multi-resolution hash tables with a small MLP

Figure 2: Model comparison between the original NeRF [35] and Instant-NGP [37].

using radiance fields.[1] The MLP weights are trained with the partial set of 2D images, and the trained weights are used for rendering (inference) from a specific viewpoint.

Figure 1 shows how NeRF renders an image from a novel viewpoint. First, it generates and shoots a ray for each pixel of the image from the camera viewpoint. It then takes K samples ($\mathbf{s}_1$, $\mathbf{s}_2$, ..., $\mathbf{s}_k$) along the ray, each of which ($\mathbf{s}_i$) is a five-dimensional vector that consists of a 3D position (x_i, y_i, z_i) and a 2D viewing direction (θ_i, ϕ_i) of the i$^{\text{th}}$ sample point. The five-dimensional input vector is mapped to a higher dimensional space through the stage called *positional encodings*. We feed this *encoded feature* after the positional encoding into the neural network (MLP) to obtain the color and density ($\mathbf{c}_i$, σ_i) of the sample point. After obtaining all the color and density values from the sampled points, the final pixel color $\hat{C}$ is computed by alpha-blending the set of color and density values along the ray, as shown in Equation 1. The transmittance T_i, which is the probability of the ray reaching a point without colliding with other objects, is computed by using the density (σ_i) and the distance between adjacent samples (δ_i).

$$\hat{C} = \sum_{i=1}^{K} T_i \alpha_i \mathbf{c}_i, \tag{1}$$

where $T_i = \exp(-\sum_{j=1}^{i-1} \sigma_j \delta_j)$ and $\alpha_i = 1 - \exp(-\sigma_i \delta_i)$.

Computation Cost. To render an image, this process needs to be repeated for *every* pixel in the image, which leads to a large number of MLP evaluations. For example, rendering a $W \times H$ image requires $W \times H \times K$ times MLP evaluations. To reduce the computation cost, NeRF models may adopt algorithm-level optimization techniques such as early ray termination (ERT) and empty space skipping (ESS) [30, 48]. When the ray meets the surface, we can skip the computation for the points behind the surface. It is the idea of the ERT, and we can detect the solid surface when the accumulated transmittance (T_i) gets lower than the pre-defined threshold value. The ESS is another optimization technique that ignores the computation of the sample points in an empty space.

[1]The radiance field consists of all light rays that flow through every point in every direction in a 3D space.

2.3 NeRF Model Architectures

The original NeRF model [35] is a pioneering work that demonstrates the benefit of using positional encodings and radiance fields, which inspires a large number of subsequent works that build upon the original NeRF. To understand the performance characteristics and rendering quality of different NeRF model structures, we choose four representative models for comparisons, NeRF [35], mip-NeRF [6], NSVF [30], and Instant-NGP [37], which we classify into two categories.

Original NeRF-based Models. In the original NeRF, the fully-connected (FC) layers in the MLP are separated into two parts: one for density (σ_i) computation and the other for color ($\mathbf{c}_i$) computation. The first part consists of 8 FC layers (with 256 channels per layer) that produce the density value and a 256-dimensional feature vector. The feature vector is then concatenated with the encoded viewing direction ($F(\theta_i, \phi_i)$), and the resulting vector is fed into one additional FC layer with 128 channels to produce the color value. It is the forerunner in NeRF, but it takes prohibitively long training and inference time due to the *large* and *deep* FC layers.

`mip-NeRF` attempts to address the issue of the original NeRF that the rendering quality is significantly degraded when it renders a *different* resolution from the trained images. To mitigate the problem, `mip-NeRF` uses information from multiple points in a circular region instead of a single critical point. However, the main model architecture is similar to the original NeRF, so it is still bottlenecked by the long latency of MLP computation. It also does not noticeably improve the rendering quality when rendering images of the same resolution. `Neural Sparse Voxel Fields (NSVF)` exploits a sparse voxel representation to train the structure that captures the emptiness of a scene along with MLP weights. It skips computation for empty voxels to accelerate training and rendering time, but it is still slow due to large MLPs.

Parametric Encoding-based Models. Figure 2 compares the original NeRF-based models with `Instant-NGP`. The key difference of `Instant-NGP` compared to the previous NeRF models is the use of a *parametric encoding* with multi-resolution hash tables rather than using a *fixed* input encoding. All the models previously mentioned use an *untrainable* input encoding, such as frequency encodings [35]. Although it is useful to extract high-dimensional features from input position vectors, it is unavoidable to use a large MLP to achieve reasonable rendering quality. Instead of using a fixed input encoding function, `Instant-NGP` employs several *trainable* hash tables for input encodings.[2] This enables the use of a much smaller MLP and reduces the computation cost, thereby improving the training and rendering speed while also achieving high-quality renderings.

2.4 Performance and Rendering Quality

To understand the performance and rendering quality of the representative NeRF models, we compare them using four different datasets: two synthetic (Syn-NeRF [35], Syn-NSVF [30]) and two real-world (BlendedMVS [57], Tanks&Temples [27]) datasets. We also choose the scenes with varying image resolutions to have more generalized results.

[2]The encoding parameters are also learned along with the MLP weights during training.

Table 1: Peak signal-to-noise ratios (PSNR) comparison.

Dataset Scene	**Syn-NeRF** Mic (800×800)	**Syn-NSVF** Palace (800×800)	**BlendedMVS** Fountain (768×576)	**Tanks&Temples** Family (1920×1080)
NeRF [35]	16.10	15.96	14.73	14.44
mip-NeRF [6]	15.82	16.04	15.14	14.64
NSVF [30]	30.84	28.58	23.27	26.73
Instant-NGP [37]	**40.33**	**35.86**	**30.70**	**33.42**

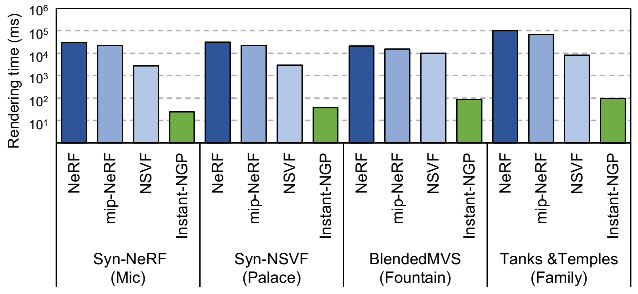

Figure 3: Rendering time of four representative NeRF models across various datasets (log scale).

Table 1 compares the peak signal-to-noise ratio (PSNR) of the four representative models after training.[3] We train each NeRF model for 100K iterations (which takes a few to tens of hours) except for Instant-NGP. Although we train Instant-NGP for less than 10 minutes (31K iterations), it reaches a significantly higher PSNR compared to others across all the datasets.

Figure 3 shows the rendering (inference) time of each trained model.[4] With the best quality of a rendered image, the rendering time of Instant-NGP is also significantly lower than others. Note that the original NeRF-based models can *hardly* be used in real-time or on-device renderings as they render images at less than one frame per second (FPS). In particular, for the real-world scene with 1920×1080 FHD resolution (Family), it takes about 8∼100 seconds to render a *single* image. In contrast, we see that Instant-NGP renders a single image *significantly* faster than others.

In short, the original NeRF-based models require a significant amount of computation as every sample point needs to run through the *large* and *deep* MLP. Considering the points are sampled from a single ray, and each ray is sampled from every pixel, they are *not* likely viable solutions for real-time or on-device rendering tasks. To alleviate the problem, state-of-the-art algorithms focus on reducing the size of compute-intensive MLPs without losing the quality of rendered images. The parametric encoding is one promising way to achieve this, which effectively reduces the amount of computation while maintaining or even increasing the image quality over the original NeRF-based models. In the following section, we further investigate the state-of-the-art neural representation that employs trainable input encoding parameters (i.e., feature vectors) [37], which is our target for acceleration.

3 MOTIVATION

In this section, we first explain the parametric encodings and the neural rendering pipeline used in Instant-NGP (Sections 3.1 and 3.2).

[3]The peak signal-to-noise ratio (PSNR) is one of the standard metrics to measure model quality. Higher is better.

[4]We run the experiments on Titan RTX due to the large memory requirement of NSVF.

Table 2: Parameters for multi-resolution hash encodings (default).

Parameter	**Symbol**	**Value**
Num. of resolution levels (num. of hash tables)	L	16
Num. of entries per level (hash table size)	T	2^{19}
Feature dimensions per entry	F	2
Each feature size		2 bytes

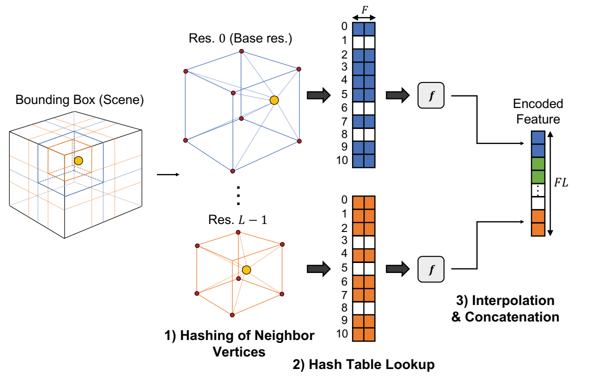

Figure 4: Multi-resolution hash encodings.

We then identify the key operations that contribute to the overall rendering time (Section 3.3) and discuss our observations and bottlenecks of the execution flow on GPUs (Section 3.4).

3.1 Multi-resolution Hash Encoding

Instant-NGP [37] introduces a new primitive called *multi-resolution hash encoding*. Figure 4 shows how a hash table-based input encoding maps the input positions to the encoded feature. First, for a sample point s, we find the voxel that surrounds the point and obtain an F-dimensional feature vector for *each vertex* of the voxel by indexing into the hash table. The hash index is computed using the hash function in Equation 2. Each of x_v, y_v, z_v corresponds to the *vertex coordinate* of the voxel grid. P_1 and P_2 are unique, large *prime* numbers, and ⊕ is the bit-wise XOR operator.

$$h(x_v, y_v, z_v) = (x_v \cdot 1) \oplus (y_v \cdot P_1) \oplus (z_v \cdot P_2) \bmod T \quad (2)$$

We then perform linear interpolation of the eight F-dimensional feature vectors to obtain an F-dimensional feature vector for the sample input point at resolution level L. We repeat these steps L times, each with a different grid resolution (i.e., a different hash table), and concatenate the L feature vectors from all levels, which results in an F×L-sized input vector for the MLP.

The multi-resolution hash encoding uses L as the number of resolution levels. The base (i.e., coarsest) grid resolution is set to 16, so there are 16^3 voxels in the base resolution (i.e., L=0). The resolution is scaled by a constant factor for finer levels (e.g., L=1,2,3,...), thereby increasing the total number of voxels in a *cubic* fashion. Each resolution level is assigned to an *independent* hash table, each of which has up to T hash entries. Each entry contains an F-dimensional feature vector, so the total number of *trainable* parameters for the multi-resolution hash encoding is L×T×F. Table 2 shows the default parameters for multi-resolution hash encodings in [37]. We use the same values for our discussions in the following sections.

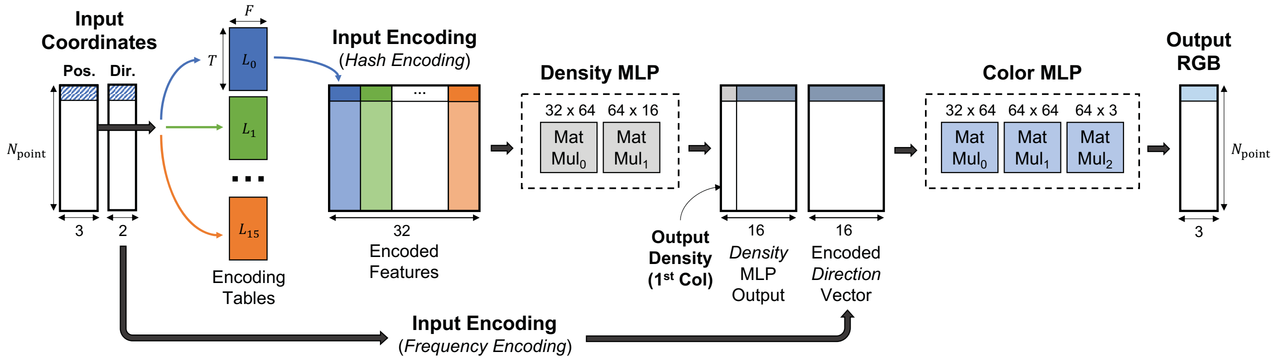

Figure 5: The model architecture in [37]. All parameters are the default values used for NeRF in the original paper.

3.2 GPU Execution Flow

Figure 5 shows the high-level execution flow of the rendering pipeline with the multi-resolution hash encoding. Initially, there are N_{point} input positions, and they go through 16 hash tables and produce an $N_{\text{point}}\times 32$ input feature matrix. The density MLP takes as input the feature matrix and produces an $N_{\text{point}}\times 16$ matrix, which is then concatenated with an $N_{\text{point}}\times 16$ encoded direction matrix. The resulting $N_{\text{point}}\times 32$ matrix is fed into the color MLP to produce 3-D RGB values (i.e., $\mathbf{c}_i$) for each input position. The number of input positions (N_{point}) can be from hundreds of thousands to tens of millions depending on the image resolution; for instance, an FHD image has two million pixels. Note that we need to perform *eight* encoding lookups *per level* for each input position, which leads to a significant number of hash table lookups in total.

3.3 Latency Breakdown

Figure 6 decomposes the rendering time into five major operations: Hash Encoding (ENC), Feature Computation (MLP), Ray Compaction (Compaction),[5] Empty Space Skipping (ESS), and Early Ray Termination (ERT). For the experiments, we run Instant-NGP with the large Fox dataset (1920×1080 FHD resolution) on a range of GPUs including the edge device (Jetson Xavier NX).

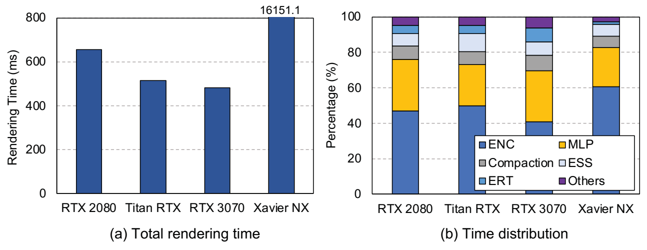

Figure 6: Latency breakdown on GPUs (Instant-NGP/Fox dataset).

The results show that ENC and MLP are the major performance bottlenecks among the operations. Note that Feature Computation (MLP) takes less than half of the rendering time, which is quite different from the original NeRF-based models where MLP computation dominates the overall rendering time. At the same time, Hash Encoding (ENC, which includes hash table lookups and some computation for interpolation) takes more than 40% of the rendering time. Note that this ENC operation does not fit well into the contemporary DNN accelerators.

[5]Ray compaction is the process to compact the rays into a dense data structure after the ERT.

3.4 Observations and Inefficiencies

We further investigate the multi-resolution hash encoding primitive and make the following key observations.

Observation I: Performance portability of multi-resolution hash encodings. Although the time complexity of a hash table lookup is $O(1)$, it is *not* a hardware-friendly operation. A well-designed hash function outputs *seemingly random* hash indexes, which lead to irregular accesses to the hash table. As previously mentioned, the state-of-the-art neural representation model trains 16 (L) hash tables (along with MLP weights), each of which is a *multi-megabyte* in size (e.g., sixteen 2MB hash tables in [37]). As such, they do not *all* fit in the on-chip memory of most of today's mobile or consumer GPUs/accelerators, thus the hash table access can lead to frequent off-chip memory accesses if we *naïvely* perform the operation. Furthermore, each hash entry access only takes four bytes (F=2) out of 64 bytes of data from the off-chip memory, leading to a substantial waste of off-chip memory bandwidth.

For the high-end consumer GPUs where the on-chip memory capacity is larger than a single hash table, one solution is to load the hash table into the on-chip memory *level by level* and stream through all sample points to obtain an $N_{\text{point}}\times 2$ *partial* input feature matrix for the corresponding level (L) before moving onto another (L+1) to avoid the costly off-chip memory access (which is the operation flow on GPUs); note that even in this case, the hash encoding occupies more than 40% of the rendering time, as shown in Figure 6. For the mobile and low-end/mid-range consumer GPUs, however, the trained model does not efficiently run because even a single hash table *does not* fit in the small on-chip cache, thereby leading to frequent off-chip memory accesses.

Observation II: Serialized execution of rendering pipeline. As previously discussed, two major operations that spend the most of the rendering time are hash encodings (ENC) and feature computation (MLP). In the execution flow with the multi-resolution hash encoding, these two main operations are *serialized* in execution although they have different compute and memory requirements; ENC is memory bandwidth-intensive, whereas MLP is more compute-intensive.

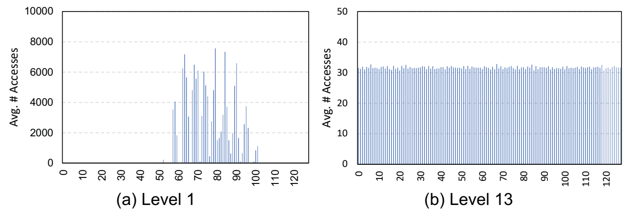

Figure 7: Average number of accesses to hash entry groups. For clarity, hash entries are sequentially grouped as the number of bins (bin size: $L1$=112, $L13$=4096) starting from entry 0 (Fox/Iteration 0).

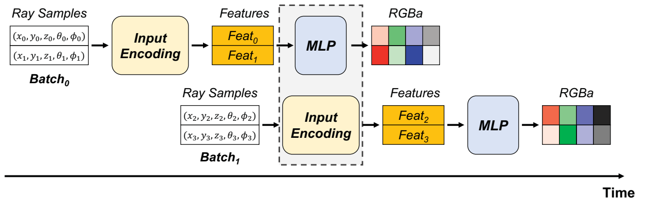

Figure 8: Execution pipeline in NeuRex.

Note that because the $N_{\text{point}}\times 32$ input feature matrix is constructed *level by level* (i.e., *column-wise*), we cannot perform MLP computation for the rest of the pipeline *until* we finish hash table lookups for the finest level (L=15). Ideally, if we overlap the execution of ENC and MLP in parallel, we can speed up the rendering pipeline. NeuRex enables overlapping the executions of these two operations and also better utilizes hardware resources by making changes in the execution flow with algorithmic optimizations and supporting hardware.

Observation III: Difference in access characteristics across different levels of hash tables. Figure 7 shows the distribution of hash entry accesses for the coarse level (L=1) and fine level (L=13) tables. For the resolution levels at which a hash collision does not occur (e.g., L=1), a hash table entry is *solely* assigned to a single vertex point of the voxel grid. Also, there is a large number of sample positions in a voxel, which share the same vertex points. Consequently, the accesses are somewhat *localized* to a few entries, and the number of accesses for each entry is high. On the other hand, for finer resolution levels (e.g., L=13), the accesses are more evenly (and randomly) distributed across the hash entries, while the number of accesses for each entry is quite low. Based on this observation, NeuRex features two different types of specialized on-chip memories to effectively serve the encoding lookups of coarser and finer levels.

4 NEUREX: NEURAL GRAPHICS ENGINE

In this section, we present NeuRex, a neural graphics engine that leverages the insights from Section 3 to efficiently perform neural graphics computation.

4.1 Execution Flow in NeuRex

Figure 8 shows the high-level execution flow of the neural rendering pipeline in NeuRex. The main difference between the NeuRex execution flow and the original one is the *pipelining* and *overlapping* of the hash encoding (ENC) and MLP operations. For example, as previously discussed, these two main operations are serialized in the original flow that both ENC and MLP contribute to the critical path latency. However, NeuRex *breaks* the serialization and executes them *in parallel* by processing the input positions at the granularity of a set of positions, which we refer to as a *batch.* For instance, we first load a batch of input positions (B) and perform multi-resolution hash encodings for the batch. We process the batch *level by level* to exploit the locality of hash entries *within* a batch. Once the ENC is done, we obtain a $B \times 32$ *partial* input feature matrix, which we can feed into the MLP. Then, while the previous batch goes through the FC layers, we fetch a *new batch* and perform the ENC operation. By doing so, NeuRex better utilizes the compute units and memory bandwidth.

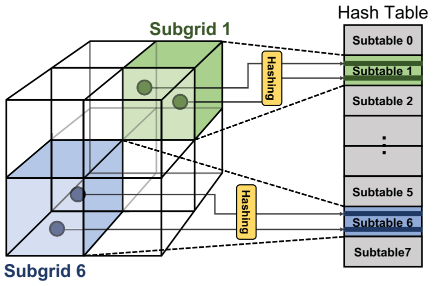

Figure 9: Restricted hashing.

4.2 Restricted Hashing

We propose a *hardware-friendly* multi-resolution hash encoding, which effectively enables the NeuRex execution flow. The key idea of our enhancement is to partition the input coordinate grid into *several subgrids*, each of which owns a *portion* of a large hash table for each level. We then arrange the processing of input points in a way that we finish processing a subgrid for *all* resolutions before processing another. In this way, we effectively *restrict the hash table access* for the vertex feature lookups to a range of consecutive hash entries, rather than being randomly distributed across the table.

This provides us with two key benefits. First, it allows the acceleration devices with small on-chip memories (e.g., mobile/edge devices or low-end GPUs) to avoid the costly off-chip memory accesses and perform encoding lookups more efficiently by loading *only* a portion of the hash table into the on-chip memory at a time, thus enabling *performance portability* of multi-resolution hash encodings across a range of compute platforms. Second, it offers opportunities for a batch of inputs within a subgrid to efficiently perform hash encodings in parallel with the MLP computation of another batch.

Figure 9 shows the restricted hashing mechanism, in which we arrange the input positions that belong to the same subgrid to be clustered in a set of batches. By doing so, the region of the hash table accessed from the same batch is restricted to a small subset of the table, which we refer to as a *subtable.* In the figure, for example, the sample positions in Subgrid 6 only access the hash entries in Subtable 6, while being indexed by a *new* hash function. When we divide the 3D scene (i.e., 3D grid) into R subdivisions for each dimension, the number of subgrids becomes R^3, and the hash table is also equally divided into R^3 subtables. We refer to R as the subgrid resolution (sugbrid_res). Then, for each input position ($\mathbf{p}$),

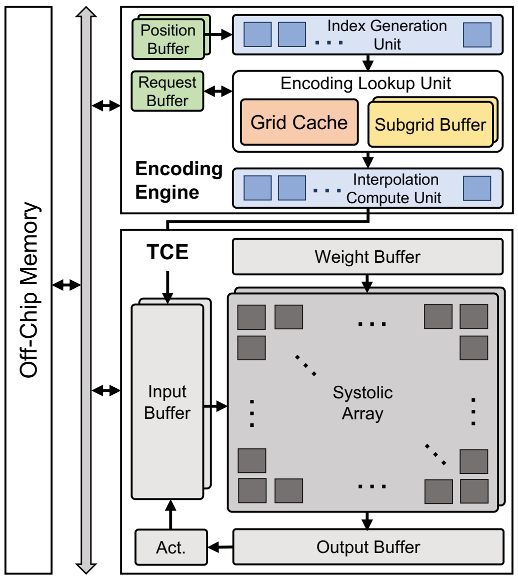

Figure 10: Overview of NeuRex architecture.

we can compute to which subgrid (subgrid_id) the input belongs using Equation 3.

$$\text{subgrid\_id} = \sum_{k=0}^{2} \lfloor p_k \cdot \text{subgrid\_res} \rfloor \cdot \text{subgrid\_res}^k, \quad (3)$$

where $\mathbf{p} = (p_0, p_1, p_2) = (x, y, z)$ and $x, y, z \in [0, 1)$.

By using the subgrid index for a batch, we load the corresponding subtable onto the on-chip buffer. This allows us to perform encoding lookups *solely* from the on-chip memory without any further access to the off-chip memory. Note that the *new* hash index used for accessing the entries in the subtable can be computed using Equation 2 with a minor modification for the modulo operation (i.e., using the subtable size instead of the table size).

4.3 Architecture Overview

Figure 10 shows the overview of our accelerator design, which consists of two main modules: Encoding Engine (EE) and Tensor Compute Engine (TCE). The TCE module is similar to the conventional DNN accelerators that employ a TPU [24]-like systolic array with memory buffers for data movement. NeuRex *minimally* extends the existing DNN accelerator design with a *dedicated* hardware module (EE) that efficiently performs multi-resolution hash encodings.

At a high level, the encoding engine is responsible for performing hash table lookups and interpolating the feature vectors obtained from the lookups to produce an input feature vector. To do so, a *batch* of input positions is first streamed into the *position buffer* from the off-chip memory. The positions in each batch are processed for all *L* levels of encoding lookups before we process the next batch.

The Index Generation Unit (IGU) in the EE generates the hash indexes and interpolation weights of the neighbor vertices for each input position. With the hash indexes, the Encoding Lookup Unit (ELU) fetches the encoded vertex features from the on-chip buffers (Grid Cache or Subgrid Buffer). After that, the final input feature

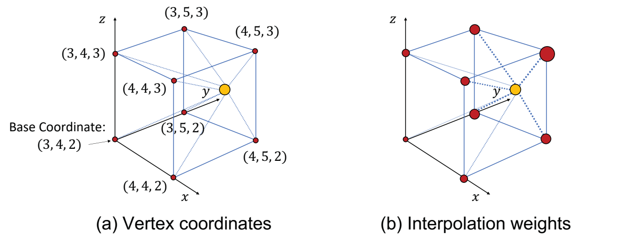

Figure 11: The process of trilinear interpolation.

vector is computed by the Interpolation Compute Unit (ICU) and is sent to the input buffer of the TCE. The TCE performs MLP using the systolic array. Because the weights are small in size and are heavily reused, we choose a TPU-like *weight stationary* dataflow for the systolic array. In the following, we describe the key hardware components of the encoding engine in detail.

4.4 Index Generation Unit

The Index Generation Unit (IGU) consists of *N* compute units (64/8 units in our NeuRex-Server/NeuRex-Edge designs) that perform computation in parallel. The IGU is composed of three main parts: position scaling, hash index computation, and interpolation weight computation.

The IGU first scales up the input coordinate because it is in the bounding box in which it is normalized to the coordinate between 0 and 1. Based on the resolution level, which corresponds to a specific resolution, we scale up the input to the coordinate system of the target level. This can be done by simple floating-point multiply-and-add operations. After scaling, we obtain two useful pieces of information. The integer part of the coordinate indicates the grid index, and the fractional part indicates the *relative* position of the point within its voxel. The grid index is used for locating the neighbor vertices of the voxel for hashing, and the relative position is used for computing interpolation weights for the vertices.

Given the grid index, which corresponds to the base coordinate of the voxel, the IGU computes all the coordinates of neighbor vertices by adding one or zero to each coordinate value. Figure 11(a) shows how to obtain the coordinates of seven neighbor vertices when the base coordinate is (3, 4, 2). These coordinates (including the base) are the inputs of the hash function in Equation 2. Each *Hash Index Compute Unit* in the IGU is responsible for computing the hash indexes of the vertices in parallel. In our design, the compute units are fully pipelined, and the IGU produces $N \times 8$ hash indexes per cycle.

To aggregate the features of the vertices, we need to compute an interpolation weight for each vertex. The weight is determined by the distance from the input position to each vertex of the voxel, as shown in Figure 11(b). Note that a *larger weight* is assigned to the vertex that is *closer to the sample position*, which implies that the feature vector of the vertex is *more representative* of the position. Equation 4 shows how to compute the interpolation weight. The *Interpolation Weight Compute Unit* in the IGU performs this operation with multipliers and subtractors, and the IGU also produces $N \times 8$ weights per cycle. Note that the hash index and interpolation weight compute units are not shown in Figure 10 for brevity.

$$w_{\text{interp}} = (1 - |x_s - x_v|) \cdot (1 - |y_s - y_v|) \cdot (1 - |z_s - z_v|), \quad (4)$$

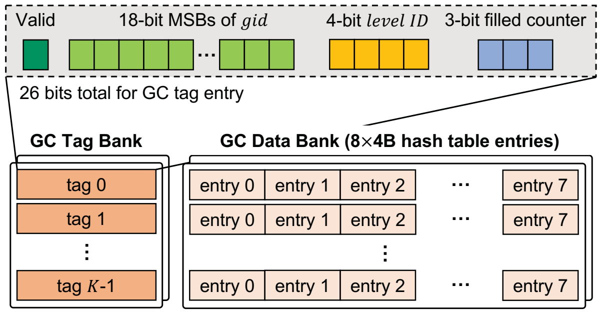

Figure 12: Grid cache (GC) structure.

where (x_v, y_v, z_v) and (x_s, y_s, z_s) are the vertex and scaled position coordinates, respectively.

4.5 Encoding Lookup Unit

The Encoding Lookup Unit (ELU) is responsible for performing hash table lookups of the *vertex* points. Note that for a sample input point, we fetch $8 \times L$ hash table entries. As we discuss in Section 3.4, we observe that the access characteristics of hash tables are different across the resolution levels. We can divide the resolution levels into two categories: coarse and fine levels. For coarse levels, the accesses show high locality to a relatively small number of hash entries. In contrast, the accesses to the finer levels are evenly distributed across the entries in the hash table. Based on the observation, we deal with two types of hash table lookups in different ways. We use a *grid cache* for coarse levels and a *subgrid buffer* for fine levels.

Grid Cache. For coarse resolution levels (e.g., L=0,1,2,...), the number of input positions that are included in the same voxel is large enough. At the same time, the granularity of hash table access for an input position is *not* a single hash entry but is a set of entries for eight vertices of the voxel. We exploit the observation by *coalescing* the eight entries into a single data block with additional information about the voxel grid; i.e., level index (lid) and grid index (gid). Then, for an input position, we fetch the *coalesced* eight vertex features using its gid with a *single* access.

Figure 12 shows the grid cache (GC) structure. The GC consists of heavily-banked SRAMs, each with an independent address decoding logic, in order to sustain high on-chip memory bandwidth. Each data block in the GC contains the feature vectors of eight vertices for a voxel (8×4B=32B). The gid computed by the IGU is used to index into the GC. The tag contains four fields: a 1-bit valid, 18-bit msbs of gid, a 4-bit level index lid, and a 3-bit counter. The GC is a direct-mapped cache style buffer, and the lsbs of gid are used to index the bank and data block.

Note that if the GC does not contain the vertex features for a gid request, it sends the memory requests for *eight vertex entries* to off-chip memory while recording the request addresses and metadata in the request buffer. When the data is returned from off-chip memory one by one, we find the matching address in the request buffer and fill the data block entry while incrementing the counter. Note that it generates multiple 64B requests, and we only take 4B out of 64B for each returned data. The data block becomes *only* valid once all the eight entries are filled from the off-chip memory.

Subgrid Buffer. For finer levels (e.g., L=...,13,14,15), we load each partitioned hash table into the subgrid buffer for encoding lookups. Note that the subgrid buffer contains *all* the hash entries required to process the input positions for a resolution level in a subgrid; thus, there is no further off-chip memory access until we move on to another level unlike the GC. As in the grid cache, the subgrid buffer is also heavily-banked to sustain high memory bandwidth (32 banks in our implementation). Unlike the grid cache, however, each bank of the subgrid buffer provides a *single* hash entry (i.e., 4 bytes) for a lookup. So, a bank conflict occurs when any of the eight vertex lookups falls onto the same bank. For 32 banks, however, we empirically find that the overall rendering time does not noticeably increase as the hash encoding operation is overlapped with MLP computation in NeuRex. We use the subgrid buffer from Level 8 for our evaluation as it shows the best overall rendering time. Note that NeuRex supports an arbitrary value if needed.

4.6 Interpolation Compute Unit

Once the encoding lookups are finished, we aggregate the vertex features from the lookups with the corresponding weights. The Interpolation Compute Unit (ICU) performs this operation in four stages. In the first stage, the eight features are multiplied by the corresponding weights. The other three stages are consumed by an adder tree. The ICU has 64/8 fully pipelined compute units (in NeuRex-Server and NeuRex-Edge, respectively), and it sends the aggregated feature vectors to the input buffer in the TCE.

4.7 Tensor Compute Engine

The MLP in neural rendering comprises only a few *small* FC layers. The number of sampled inputs, on the other hand, is orders of magnitude larger than the width of FC layers. Small MLP weights and a large input dimension lead to a huge opportunity for *layer fusion*, as observed in other work [3]. We also adopt fusion-based MLP computation for our accelerator design. Given a batch of input features, the Tensor Compute Engine (TCE) works on a series of FC and activation layers and generates the final outputs without storing the intermediate features back to the off-chip memory. The TCE has large enough input and output buffers to store them.

5 EXPERIMENTAL METHODOLOGY

Hardware Implementation. We implement the hardware components of NeuRex in RTL using SystemVerilog. The functionality of each component is verified via RTL simulations with synthetic data. We synthesize the NeuRex components using a commercial 28nm technology node with Synopsys Design Compiler [53]. On-chip SRAMs are also generated from a commercial memory compiler with the same technology. The position/subgrid buffers in the EE and the input/output buffers in the TCE are double-buffered. The subgrid buffer is sized at 128KB with 32 banks for the case where the per-level table size becomes significantly larger in the future, but it can be as small as 32KB for our evaluation. The grid cache is sized at 64KB with 32 banks, and the request buffer can handle up to 64 addresses and 64 merged requests per address. We design our architecture to run at a 1GHz clock frequency for most components except for the on-chip memory that runs double-pumped at 2GHz to provide high on-chip memory bandwidth, as similar to [50].

To evaluate the system-level performance of NeuRex with off-chip memory, we also implement a cycle-level simulator that models

Table 3: Evaluated workloads.

Task	Dataset	Scene (Resolution)	Type
NeRF	Synthetic-NeRF [35]	Mic (800×800)	Synthetic
	Synthetic-NSVF [30]	Palace (800×800)	Synthetic
	BlendedMVS [57]	Fountain (768×576)	Real world
	Tanks&Temples [27]	Family (1920×1080)	Real world
	Instant-NGP [37]	Fox (1080×1920)	Real world
Neural SDF	The Stanford Models [1]	Bunny (1920×1920)	-
		Armadillo (1920×1920)	-
Image Approximation	-	Tokyo (6144×2324)	-
		Albert (3250×4333)	-

the NeuRex architecture with Ramulator [26] for DRAM timing. We collect position traces by running the workloads on GPUs and use them as input for the simulator. The timing parameters of the simulator are determined based on the RTL synthesis results. We measure the accelerator performance using the cycles reported by the simulator. The simulator also outputs the number of SRAM accesses, which we use to obtain the energy numbers of on-chip buffers. The energy numbers of the off-chip memory are computed using the DRAM statistics from the memory simulator [26].

We evaluate two variants of NeuRex: NeuRex-Edge and NeuRex-Server. NeuRex-Edge is a design point when there are strict area and power constraints, which is a typical case for mobile and edge computing platforms. NeuRex-Server is a scaled-up architecture for high-end computing platforms. The batch size is set to 1024 and 8192 for NeuRex-Edge and NeuRex-Server. We configure the off-chip memory of NeuRex-Edge as LPDDR4-3200 [4] and analyze the statistics using DRAMPower [9, 34]. HBM2 [5] is used for NeuRex-Server with the energy model from FGDRAM [44]. We use a multiple of a 32×32 systolic array instead of using a larger one; this improves the utilization of the compute units. The TCE consists of one and sixteen 32×32 systolic arrays for the NeuRex-Edge and NeuRex-Server, respectively. Section 6.5 discusses the hardware configurations and energy efficiency of two variants of NeuRex.

Baselines. We compare our accelerator designs with two different classes of computing platforms. We choose NVIDIA Jetson Xavier NX [42] as a representative of edge devices. Also, RTX 3070 [43] is selected as high-end consumer-level rendering acceleration hardware. We use and modify the author-released code that includes heavily-optimized CUDA kernels (e.g., fused MLP and other optimizations for better tensor core utilization). We measure the performance and power consumption of each GPU by using the built-in hardware counters. Note that RTX 3070 is fabricated using the Samsung 8nm process node, which is a couple of generations advanced compared to the technology node used for NeuRex (28nm).

Workloads. Table 3 shows the workloads that we use to evaluate our design. We carefully select a range of synthetic and real-world datasets from several prior works to cover the scenes with varying resolutions and complexity. The number of initial rays is proportional to the resolution of a rendering image, while the number of ray sampling iterations depends on how realistic the scene is. In addition to NeRF, we also evaluate our design with other graphics tasks, such as neural signed distance functions (SDF) and 2D image approximation, to demonstrate the general applicability of the parametric encoding-based neural scene representations in Section 6.6.

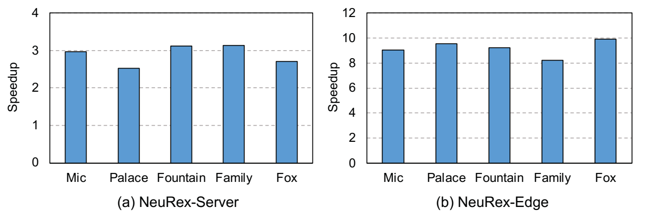

Figure 13: Speedup of NeuRex over GPUs.

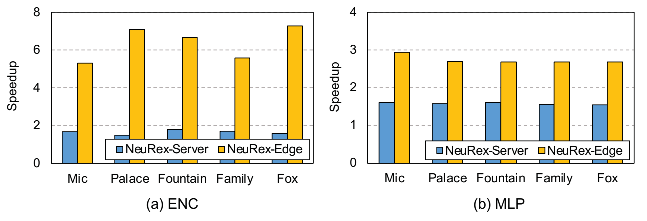

Figure 14: Speedup on hash encodings (ENC) and feature computation (MLP).

6 EVALUATION

6.1 NeuRex Performance

Figure 13 shows the performance of NeuRex over RTX 3070 and Xavier NX. On average, NeuRex-Server and NeuRex-Edge achieve 2.88× and 9.17× speedups over the baseline GPUs. It is worth noting that NeuRex-Edge exhibits a higher speedup over the baseline than NeuRex-Server. This is because irregular accesses to large encoding tables quickly become a performance bottleneck in the GPU execution when the GPU has a small on-chip memory capacity (e.g., a 256KB L2 cache in Xavier NX). By employing restricted hashing and loading a portion of the encoding table at a time, NeuRex enhances the *performance portability* of the multi-resolution hash encoding.[6]

Figure 14 compares the performance of NeuRex and GPUs for two key operations in modern neural renderings: hash encodings (ENC) and feature computation (MLP). Note that the speedup shown in Figure 14(a) comes from the restricted hashing algorithm and specialized on-chip memory design. Figure 14(b) shows that NeuRex performs MLP computation *faster* despite the lower peak compute throughput compared to the GPUs. This is because the GPU tensor cores are underutilized due to small FC layers, whereas the TCE in NeuRex achieves higher compute utilization. Also, the overall speedup of NeuRex (Figure 13) is higher than the individual speedups for ENC and MLP because these two operations are serialized in the original execution flow, whereas NeuRex enables them to be overlapped thanks to the restricted hashing.

6.2 Rendering Quality

Restricted hashing slightly modifies the multi-resolution hash encoding to make it hardware-friendly. To demonstrate that our proposed scheme does not degrade the quality of rendered images, we compare the average PSNR between the original hash encoding primitive (Org-Hash) and the restricted one (Ours). For each scene, we obtain PSNRs from 10 different camera views and average them to generalize the result.

[6]We use 64 subgrids for restricted hashing in our evaluation.

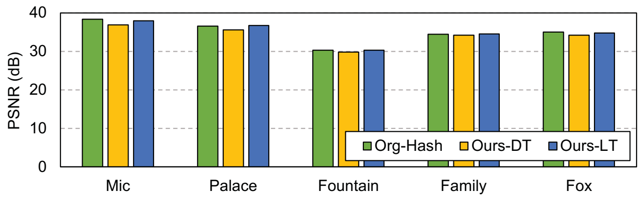

Figure 15: Rendering quality (PSNR) of the original hash encoding (Org-Hash) and restricted hashing (Ours).

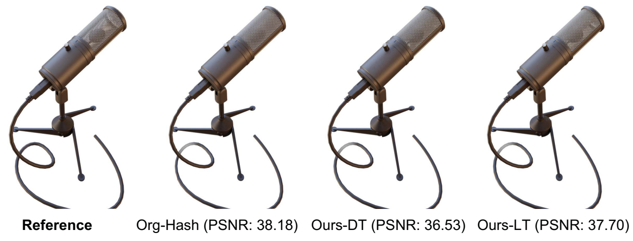

Figure 16: Rendered images. Models are trained for 31K iterations.

Figure 15 shows that there is a *negligible* PSNR drop (0.7%∼3.9%) over the baseline when restricted hashing is applied to the default table size in [37] (Ours-DT; 2MB per level). Note that Ours-DT is already *superior* to the original NeRF-based models that do not use hash encodings. As discussed in Section 4, restricted hashing limits each batch to accessing input encodings only within a single subgrid buffer. Consequently, increasing the hash table size has less impact on performance as only a portion of the table needs to be loaded on-chip at a time. Based on this observation, we configure our model with a 4× larger hash table (Ours-LT; 8MB per level) to further improve PSNRs without compromising performance. The results show that Ours-LT leads to only a minor 1.1% decrease in PSNR for the worst case, and for several other scenes, it even produces higher PSNRs than Org-Hash.

Figure 16 compares the reference image to the rendered ones using the original primitive (Org-Hash) and restricted hashing (Ours) for the scene that exhibits the *highest* PSNR drop. We see that Ours-DT/LT does not degrade the rendering quality, and interestingly, in some parts, the images produced using restricted hashing look closer to the reference image than Org-Hash even though the PSNRs are lower. This could be because some parts experience fewer hash collisions than the case with a single hash table in Org-Hash. Note that the off-chip memory is large enough to accommodate the increased hash tables, which makes restricted hashing an attractive solution for edge and mobile platforms.

6.3 Source of Performance Gain

Figure 17 shows the speedup from each component in NeuRex. We can divide NeuRex into two key components: grid cache (GC) and restricted hashing (RH) with the subgrid buffer. By accumulating each optimization from the baseline, we evaluate three variants of NeuRex: Baseline, GC, and GC+RH. The baseline is the batch-based execution model with no optimization. Note that GC and RH are exclusively used for encoding lookups of the coarse and fine levels, respectively. The level threshold is set to eight. Meanwhile, for the variants that do not employ the GC or RH, we use a 2MB cache,

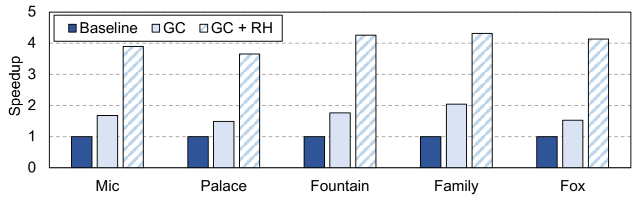

Figure 17: Source of gain. Speedup of variants isolating each optimization in NeuRex.

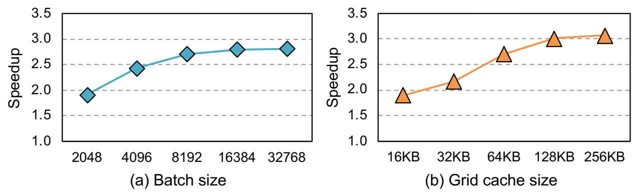

Figure 18: Speedup across the batch and grid cache sizes.

which is the size of a single hash table, to model a conventional cache in GPU. All the configurations are adopted for NeuRex-Server.

The speedup of GC over the baseline comes from utilizing the on-chip memory bandwidth more effectively than the conventional cache for coarse levels. In the baseline, we need to access the cache eight times to obtain all the vertex features for a single sample point. Also, each access only takes 4B out of a 64B cacheline, which leads to a waste of on-chip bandwidth. In contrast, the grid cache provides the coalesced eight vertex features in a single access, thus effectively serving the hash encoding lookups with a small cache capacity for coarse levels. By maximizing the hash entry reuse in the subgrid buffer for fine levels, NeuRex (GC+RH) further improves performance over the baseline with GC.

6.4 Sensitivity Study

This section evaluates how NeuRex performance varies with different hardware resource configurations. In particular, we focus on two configurations: batch size and grid cache size. Figure 18 shows the speedup over the GPU across the batch and grid cache sizes. We show the results of NeuRex-Server only with the Fox dataset as the general trend holds.

Batch Size. The batch size affects the performance because a larger batch can increase the temporal locality of the grid cache and improve the compute utilization of TCE. However, a larger batch leads to an increase in the size of some on-chip buffers, such as the position buffer. We increase the batch size from 2048 to 32768 and observe that the speedup does not noticeably increase after a size of 8192. This is because the streaming latency from off-chip memory to fill the double-buffered subgrid buffer is mostly hidden by the on-chip memory access latency from encoding lookups for fine levels. We choose a batch size of 8192 and 1024 to balance the performance and area overhead for NeuRex-Server and NeuRex-Edge.

Grid Cache Size. We observe that a small grid cache is sufficient to achieve the full benefit of the grid cache. We perform a sweep of grid cache sizes from 16KB to 256KB and see that the speedup noticeably increases until 64KB but not much for larger caches. The grid cache size is closely related to the number of unique voxels in coarse levels because each cacheline has the eight vertex features of a voxel. Since most input batches belong to less than 2048 unique

Table 4: Area and power of NeuRex (NeuRex-Server / NeuRex-Edge).

Component	Remarks	Area [mm^2]	Power [W]
Systolic Array	16×(32×32) / 1×(32×32)	7.68 / 0.48	3.04 / 0.19
Weight Buffer	20KB / 20KB	0.04 / 0.04	0.02 / 0.02
Input Buffer	2×1MB / 2×128KB	3.39 / 0.57	0.36 / 0.26
Output Buffer	2×1MB / 2×128KB	3.39 / 0.57	0.36 / 0.26
Total TCE		**14.50 / 1.66**	**3.78 / 0.73**
Position Buffer	2×96KB / 2×12KB	0.43 / 0.05	0.20 / 0.02
Grid Cache	64KB / 64KB	0.14 / 0.14	0.06 / 0.06
Subgrid Buffer	2×128KB / 2×128KB	0.57 / 0.57	0.26 / 0.26
Request Buffer	16KB / 8KB	0.03 / 0.01	0.08 / 0.04
Index Generation Unit	64 units / 8 units	4.80 / 0.60	1.34 / 0.16
Interpolation Compute Unit	64 units / 8 units	0.90 / 0.11	0.38 / 0.05
Total EE		**6.87 / 1.48**	**2.32 / 0.58**
Total		**21.37 / 3.14**	**6.10 / 1.31**

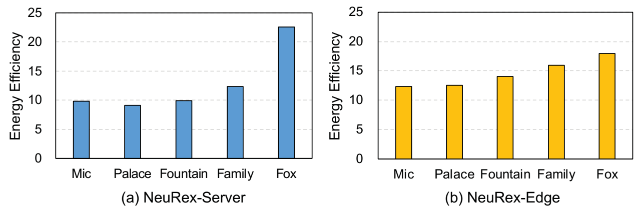

Figure 19: Energy efficiency of NeuRex over GPUs.

voxels in coarse levels, we can cover most of the voxels with a small grid cache. We use a 64KB grid cache in our design.

6.5 Area and Energy Efficiency

Table 4 shows the area and power numbers of NeuRex, which indicate that NeuRex can be implemented with small areas and powers. Note that we implement a TPU-like TCE, but one can choose other designs. For the encoding engine, which is the key component in NeuRex, the largest area overhead is the IGU as it consists of multiple FP and integer MAC units. However, the areas of NeuRex-Server and NeuRex-Edge are both negligible compared to the baseline GPU SoCs (i.e., 392mm^2 for RTX 3070 and 350mm^2 for Xavier NX).

Figure 19 shows the energy comparisons between NeuRex and GPUs in which NeuRex shows significantly higher energy efficiency than RTX 3070 and Xavier NX. Note that NeuRex uses a 28nm technology node, whereas the GPUs are fabricated with more advanced nodes (8nm/12nm for RTX 3070/Xavier NX). Therefore, instead of directly comparing the numbers, it is more appropriate to infer that NeuRex would become even more attractive if it were fabricated with more advanced technology.

6.6 Discussion

Restricted Hashing and Pipelining on GPUs. While restricted hashing (RH) and software pipelining (PP) can be applied to GPUs, we observe that the GPUs do not benefit much from these. In Figure 20, RH shows the case where we *solely* apply restricted hashing to the GPUs. For Xavier NX, restricted hashing helps reduce off-chip memory accesses as we only load a portion of the hash table for a subgrid to process on-chip. However, we now feed multiple smaller input matrices into the MLP, instead of a single large one, for feature computation of all sampled points. This effectively increases the total execution time of feature computation due to *lower*

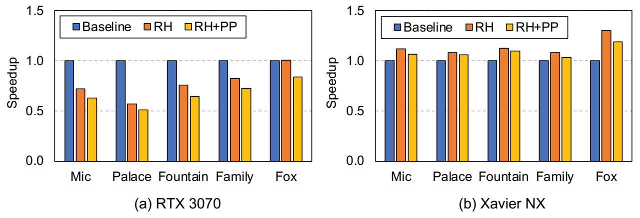

Figure 20: Restricted hashing and pipelining on GPUs.

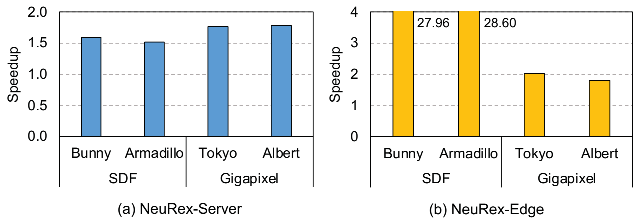

Figure 21: Speedup on other graphics tasks beyond NeRF.

core utilization for each MLP. Overall, restricted hashing improves performance, but the increase is limited. RTX 3070 already has a large L2 cache, so restricted hashing does not help reduce off-chip memory accesses. Furthermore, due to the same reason for the edge GPU, the core utilization becomes lower than the case without restricted hashing. Thus, restricted hashing in fact reduces performance for most of the scenes except for the Fox dataset.

RH+PP refers to the case where we also apply software pipelining on top of restricted hashing. Although CUDA now supports concurrent kernel execution, we have observed that it is challenging to nicely overlap the execution of *complex and optimized* kernels, as also noted in prior work [60], unlike the case of overlapping the kernel execution with data transfers (e.g., cudaMemcpyAsync). This is because hardware resources (e.g., registers, shared memory) are limited, and the CUDA runtime does not effectively schedule thread blocks from multiple complex kernels; users have limited control over it. We observe that only small portions of execution between the hash encoding and MLP kernels can be overlapped. On the other hand, the overheads to enable overlapping (e.g., synchronization) are higher, so RH+PP even decreases the performance compared to RH. We also observe that the overlapped portion can be increased by reducing the resource usage of each kernel, but this leads to an increase in execution time for each kernel.

Long-Term Viability of NeuRex. As discussed in Section 2.3, Instant-NGP does not significantly alter the NeRF model architecture but introduces a simple change in the way the input positions are encoded by employing several *learnable* hash tables. This input encoding approach bears some resemblance to the positional encoding and word embedding used in *Transformer*-based models [7, 14, 56], which have now become essential elements in natural language processing (NLP). Similarly, the hash encoding primitive can be applied to other graphics workloads or tasks, such as neural signed distance functions (SDF) [46] and image approximation (Gigapixel). Figure 21 shows that NeuRex also helps improve performance for SDF and image approximation over GPUs. We envision that this new input encoding technique will be widely adopted in the future whenever applicable, and NeuRex can help improve the performance of these workloads beyond NeRF.

7 RELATED WORK

Deep Learning for Graphics. Graphics applications are widely adopting deep learning approaches to improve their quality. One of the recent works, NeRF [35], demonstrated promising results for volume renderings through a neural network that comprises a positional encoding and a large MLP. Subsequent studies [6, 30, 48, 58] improve on the NeRF work by focusing on the MLP. Although these works improve the training/rendering time or rendering quality, Instant-NGP [37] substantially outperforms other works by exploiting the idea of multi-resolution hash encodings. NeuRex focuses on the parametric encoding-based models that are captivating.

Domain-Specific Acceleration. Both hardware and software optimization techniques have been developed for the traditional ray tracing-based rendering pipeline [8, 49]. Commercial GPUs now feature acceleration modules [8] to speed up the tree traversal in ray tracing, and some recent work improves the ray tracing performance on GPUs with architectural support [29, 31]. In contrast, our work focuses on the modern neural rendering pipeline, and NeuRex is the *first* work that accelerates neural rendering models with parametric encodings. Hardware accelerators for DNNs have also been extensively explored [3, 12, 13, 15, 16, 21, 24, 25, 28, 36, 40, 41, 47]. DNN models can be pruned with minimal accuracy loss, providing architects with an opportunity to skip unnecessary computation [20, 22]. Popular neural network primitives (e.g., ReLU) also introduce zeros during computation. Sparse accelerators exploit these opportunities to efficiently perform DNN inference [2, 19, 21, 45, 59]. NeuRex can take advantage of some of the optimizations in these works, but the existing DNN accelerators are unlikely to be used for modern neural renderings out of the box.

8 CONCLUSION

Neural rendering gains significant traction as a promising method for synthesizing complex scenes from novel viewpoints. This work takes a careful look at the modern neural rendering model, which significantly enhances rendering performance and quality over others by employing multi-resolution hash encodings. We observe that the hash encoding operation now becomes the performance bottleneck in conventional hardware and needs to be more hardware-friendly to achieve its full potential. Based on our analysis, we present NeuRex, a specialized accelerator that features a novel hash encoding engine for modern neural renderings. NeuRex exploits our proposed restricted hashing to mitigate irregular access to off-chip memory and enable concurrent execution of key operations in the neural rendering pipeline. With the algorithm-hardware co-design, NeuRex greatly improves rendering performance over conventional hardware with substantially smaller area and power budgets.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their insightful comments. This work was supported by the New Faculty Startup Fund from Seoul National University and the Ministry of Science and ICT under the ITRC support program (IITP-2023-RS-2022-00156295) supervised by the IITP (Institute for Information & Communications Technology Planning & Evaluation). The Institute of Engineering Research at Seoul National University provided research facilities for this work. Jaewoong Sim is the corresponding author.

REFERENCES

[1] 1994. *The Stanford 3D Scanning Repository.* https://graphics.stanford.edu/data/3Dscanrep/

[2] Jorge Albericio, Patrick Judd, Tayler Hetherington, Tor Aamodt, Natalie Enright Jerger, and Andreas Moshovos. 2016. Cnvlutin: Ineffectual-Neuron-Free Deep Neural Network Computing. In *ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA).*

[3] Manoj Alwani, Han Chen, Michael Ferdman, and Peter Milder. 2016. Fused-Layer CNN Accelerators. In *49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO).*

[4] JEDEC Solid State Technology Association. 2014. *JEDEC Standard JESD209-4: Low Power Double Data Rate 4 (LPDDR4).* JEDEC, Virginia, USA.

[5] JEDEC Solid State Technology Association. 2015. *JEDEC Standard JESD235A: High Bandwidth Memory (HBM) DRAM.* JEDEC, Virginia, USA.

[6] Jonathan T. Barron, Ben Mildenhall, Matthew Tancik, Peter Hedman, Ricardo Martin-Brualla, and Pratul P. Srinivasan. 2021. Mip-NeRF: A Multiscale Representation for Anti-Aliasing Neural Radiance Fields. In *IEEE/CVF International Conference on Computer Vision (ICCV).*

[7] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models Are Few-Shot Learners. In *Conference on Neural Information Processing Systems (NeurIPS).*

[8] John Burgess. 2020. RTX on—The NVIDIA Turing GPU. *IEEE Micro* (2020).

[9] Karthik Chandrasekar, Christian Weis, Yonghui Li, Sven Goossens, Matthias Jung, Omar Naji, Benny Akesson, Norbert Wehn, and Kees Goossens. 2012. *DRAMPower: Open-source DRAM power & Energy Estimation Tool.* http://www.drampower.info

[10] Anpei Chen, Zexiang Xu, Andreas Geiger, Jingyi Yu, and Hao Su. 2022. TensoRF: Tensorial Radiance Fields. In *European Conference on Computer Vision (ECCV).*

[11] Yinbo Chen, Sifei Liu, and Xiaolong Wang. 2021. Learning Continuous Image Representation with Local Implicit Image Function. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR).*

[12] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, and Olivier Temam. 2014. DaDianNao: A Machine-Learning Supercomputer. In *47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO).*

[13] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. 2016. Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks. In *ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA).*

[14] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT).*

[15] Yu Feng, Boyuan Tian, Tiancheng Xu, Paul Whatmough, and Yuhao Zhu. 2020. Mesorasi: Architecture Support for Point Cloud Analytics via Delayed-Aggregation. In *53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO).*

[16] Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Logan Adams, Mahdi Ghandi, Stephen Heil, Prerak Patel, Adam Sapek, Gabriel Weisz, Lisa Woods, Sitaram Lanka, Steven K. Reinhardt, Adrian M. Caulfield, Eric S. Chung, and Doug Burger. 2018. A Configurable Cloud-Scale DNN Processor for Real-Time AI. In *ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA).*

[17] Sara Fridovich-Keil, Alex Yu, Matthew Tancik, Qinhong Chen, Benjamin Recht, and Angjoo Kanazawa. 2022. Plenoxels: Radiance Fields Without Neural Networks. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR).*

[18] Stephan J. Garbin, Marek Kowalski, Matthew Johnson, Jamie Shotton, and Julien P. C. Valentin. 2021. FastNeRF: High-Fidelity Neural Rendering at 200FPS. In *IEEE/CVF International Conference on Computer Vision (ICCV).*

[19] Ashish Gondimalla, Noah Chesnut, Mithuna Thottethodi, and T. N. Vijaykumar. 2019. SparTen: A Sparse Tensor Accelerator for Convolutional Neural Networks. In *52nd Annual IEEE/ACM International Symposium on Microarchitecture (MIRCO).*

[20] Tae Jun Ham, Yejin Lee, Seong Hoon Seo, Soosung Kim, Hyunji Choi, Sung Jun Jung, and Jae W. Lee. 2021. ELSA: Hardware-Software Co-Design for Efficient, Lightweight Self-Attention Mechanism in Neural Networks. In *ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA).*

[21] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A. Horowitz, and William J. Dally. 2016. EIE: Efficient Inference Engine on Compressed Deep Neural Network. In *ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA).*

[22] Song Han, Jeff Pool, John Tran, and William J. Dally. 2015. Learning Both Weights and Connections for Efficient Neural Networks. In *Conference on Neural*

Information Processing Systems (NeurIPS).

[23] Peter Hedman, Pratul P. Srinivasan, Ben Mildenhall, Jonathan T. Barron, and Paul Debevec. 2021. Baking Neural Radiance Fields for Real-Time View Synthesis. In *IEEE/CVF International Conference on Computer Vision (ICCV).*

[24] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. 2017. In-datacenter performance analysis of a tensor processing unit. In *ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA).*

[25] Liu Ke, Udit Gupta, Benjamin Youngjae Cho, David Brooks, Vikas Chandra, Utku Diril, Amin Firoozshahian, Kim Hazelwood, Bill Jia, Hsien-Hsin S. Lee, Meng Li, Bert Maher, Dheevatsa Mudigere, Maxim Naumov, Martin Schatz, Mikhail Smelyanskiy, Xiaodong Wang, Brandon Reagen, Carole-Jean Wu, Mark Hempstead, and Xuan Zhang. 2020. RecNMP: Accelerating Personalized Recommendation with near-Memory Processing. In *ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA).*

[26] Yoongu Kim, Weikun Yang, and Onur Mutlu. 2016. Ramulator: A Fast and Extensible DRAM Simulator. *IEEE Computer Architecture Letters (CAL)* (2016).

[27] Arno Knapitsch, Jaesik Park, Qian-Yi Zhou, and Vladlen Koltun. 2017. Tanks and Temples: Benchmarking Large-Scale Scene Reconstruction. *ACM Transactions on Graphics (SIGGRAPH)* (2017).

[28] Youngeun Kwon, Yunjae Lee, and Minsoo Rhu. 2019. TensorDIMM: A Practical Near-Memory Processing Architecture for Embeddings and Tensor Operations in Deep Learning. In *52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO).*

[29] Lufei Liu, Wesley Chang, Francois Demoullin, Yuan Hsi Chou, Mohammadreza Saed, David Pankratz, Tyler Nowicki, and Tor M. Aamodt. 2021. Intersection Prediction for Accelerated GPU Ray Tracing. In *54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO).*

[30] Lingjie Liu, Jiatao Gu, Kyaw Zaw Lin, Tat-Seng Chua, and Christian Theobalt. 2020. Neural Sparse Voxel Fields. In *Conference on Neural Information Processing Systems (NeurIPS).*

[31] Yashuai Lü, Libo Huang, Li Shen, and Zhiying Wang. 2017. Unleashing the Power of GPU for Physically-Based Rendering via Dynamic Ray Shuffling. In *50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO).*

[32] Julien N. P. Martel, David B. Lindell, Connor Z. Lin, Eric R. Chan, Marco Monteiro, and Gordon Wetzstein. 2021. ACORN: Adaptive Coordinate Networks for Neural Scene Representation. *ACM Transactions on Graphics (SIGGRAPH)* (2021).

[33] Yiqun Mei, Yuchen Fan, and Yuqian Zhou. 2021. Image Super-Resolution With Non-Local Sparse Attention. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR).*

[34] Micron. 2016. *Automotive LPDDR4/LPDDR4X SDRAM.* Micron Technology, Inc, Boise, USA.

[35] Ben Mildenhall, Pratul P. Srinivasan, Matthew Tancik, Jonathan T. Barron, Ravi Ramamoorthi, and Ren Ng. 2020. NeRF: Representing Scenes as Neural Radiance Fields for View Synthesis. In *Proceedings of the European Conference on Computer Vision (ECCV).*

[36] Duncan J.M Moss, Srivatsan Krishnan, Eriko Nurvitadhi, Piotr Ratuszniak, Chris Johnson, Jaewoong Sim, Asit Mishra, Debbie Marr, Suchit Subhaschandra, and Philip H.W. Leong. 2018. A Customizable Matrix Multiplication Framework for the Intel HARPv2 Xeon+FPGA Platform: A Deep Learning Case Study. In *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA).*

[37] Thomas Müller, Alex Evans, Christoph Schied, and Alexander Keller. 2022. Instant Neural Graphics Primitives with a Multiresolution Hash Encoding. *ACM Transactions on Graphics (SIGGRAPH)* (2022).

[38] Thomas Neff, Pascal Stadlbauer, Mathias Parger, Andreas Kurz, Joerg H. Mueller, Chakravarty R. Alla Chaitanya, Anton S. Kaplanyan, and Markus Steinberger. 2021. DONeRF: Towards Real-Time Rendering of Compact Neural Radiance Fields using Depth Oracle Networks. *Computer Graphics Forum (EGSR)* (2021).

[39] Michael Niemeyer, Lars Mescheder, Michael Oechsle, and Andreas Geiger. 2020. Differentiable Volumetric Rendering: Learning Implicit 3D Representations without 3D Supervision. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR).*

[40] Eriko Nurvitadhi, David Sheffield, Jaewoong Sim, Asit Mishra, Ganesh Venkatesh, and Debbie Marr. 2016. Accelerating Binarized Neural Networks: Comparison of FPGA, CPU, GPU, and ASIC. In *International Conference on Field-Programmable Technology (FPT).*

[41] Eriko Nurvitadhi, Ganesh Venkatesh, Jaewoong Sim, Debbie Marr, Randy Huang, Jason Ong Gee Hock, Yeong Tat Liew, Krishnan Srivatsan, Duncan Moss, Suchit Subhaschandra, and Guy Boudoukh. 2017. Can FPGAs Beat GPUs in Accelerating Next-Generation Deep Neural Networks?. In *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA).*

[42] NVIDIA. 2018. NVIDIA Xavier System-on-Chip, HotChips 30.

[43] NVIDIA. 2020. *GeForce RTX 3070 Family.* Retrieved April 10, 2023 from https://www.nvidia.com/en-us/geforce/graphics-cards/30-series/rtx-3070-3070ti/

[44] Mike O'Connor, Niladrish Chatterjee, Donghyuk Lee, John Wilson, Aditya Agrawal, Stephen W. Keckler, and William J. Dally. 2017. Fine-Grained DRAM: Energy-Efficient DRAM for Extreme Bandwidth Systems. In *50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO).*

[45] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel Emer, Stephen W. Keckler, and William J. Dally. 2017. SCNN: An accelerator for compressed-sparse convolutional neural networks. In *ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA).*

[46] Jeong Joon Park, Peter Florence, Julian Straub, Richard Newcombe, and Steven Lovegrove. 2019. DeepSDF: Learning Continuous Signed Distance Functions for Shape Representation. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR).*

[47] Brandon Reagen, Paul Whatmough, Robert Adolf, Saketh Rama, Hyunkwang Lee, Sae Kyu Lee, José Miguel Hernández-Lobato, Gu-Yeon Wei, and David Brooks. 2016. Minerva: Enabling Low-Power, Highly-Accurate Deep Neural Network Accelerators. In *ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA).*

[48] C. Reiser, S. Peng, Y. Liao, and A. Geiger. 2021. KiloNeRF: Speeding up Neural Radiance Fields with Thousands of Tiny MLPs. In *IEEE/CVF International Conference on Computer Vision (ICCV).*

[49] Alexander Reshetov, Alexei Soupikov, and Jim Hurley. 2005. Multi-Level Ray Tracing Algorithm. *ACM Transactions on Graphics (SIGGRAPH)* (2005).

[50] Nikola Samardzic, Axel Feldmann, Aleksandar Krastev, Srinivas Devadas, Ronald Dreslinski, Christopher Peikert, and Daniel Sanchez. 2021. F1: A Fast and Programmable Accelerator for Fully Homomorphic Encryption. In *54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO).*

[51] Vincent Sitzmann, Michael Zollhoefer, and Gordon Wetzstein. 2019. Scene Representation Networks: Continuous 3D-Structure-Aware Neural Scene Representations. In *Conference on Neural Information Processing Systems (NeurIPS).*

[52] Cheng Sun, Min Sun, and Hwann-Tzong Chen. 2022. Direct Voxel Grid Optimization: Super-fast Convergence for Radiance Fields Reconstruction. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR).*

[53] Synopsys. 2023. *Design Compiler - Synopsys.* Retrieved April 10, 2023 from https://www.synopsys.com/implementation-and-signoff/rtl-synthesis-test/dc-ultra.html

[54] Towaki Takikawa, Joey Litalien, Kangxue Yin, Karsten Kreis, Charles Loop, Derek Nowrouzezahrai, Alec Jacobson, Morgan McGuire, and Sanja Fidler. 2021. Neural Geometric Level of Detail: Real-time Rendering with Implicit 3D Shapes. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR).*

[55] Ayush Tewari, Justus Thies, Ben Mildenhall, Pratul Srinivasan, Edgar Tretschk, Yifan Wang, Christoph Lassner, Vincent Sitzmann, Ricardo Martin-Brualla, Stephen Lombardi, Tomas Simon, Christian Theobalt, Matthias Niessner, Jonathan T. Barron, Gordon Wetzstein, Michael Zollhoefer, and Vladislav Golyanik. 2021. Advances in Neural Rendering. https://doi.org/10.48550/ARXIV.2111.05849

[56] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is All You Need. In *Conference on Neural Information Processing Systems (NeurIPS).*

[57] Y. Yao, Z. Luo, S. Li, J. Zhang, Y. Ren, L. Zhou, T. Fang, and L. Quan. 2020. BlendedMVS: A Large-Scale Dataset for Generalized Multi-View Stereo Networks. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR).*

[58] Alex Yu, Ruilong Li, Matthew Tancik, Hao Li, Ren Ng, and Angjoo Kanazawa. 2021. PlenOctrees for Real-time Rendering of Neural Radiance Fields. In *IEEE/CVF International Conference on Computer Vision (ICCV).*

[59] Shijin Zhang, Zidong Du, Lei Zhang, Huiying Lan, Shaoli Liu, Ling Li, Qi Guo, Tianshi Chen, and Yunji Chen. 2016. Cambricon-X: An accelerator for sparse neural networks. In *49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO).*

[60] Han Zhao, Weihao Cui, Quan Chen, Jieru Zhao, Jingwen Leng, and Minyi Guo. 2021. Exploiting Intra-SM Parallelism in GPUs via Persistent and Elastic Blocks. In *IEEE 39th International Conference on Computer Design (ICCD).*