2026 ACM/IEEE 53rd Annual International Symposium on Computer Architecture (ISCA) | 979-8-3315-5065-3/26/$31.00 ©2026 IEEE | DOI: 10.1109/ISCA66397.2026.00124

NeRArch-Sim: A Unified Simulator for Benchmarking and DSE of Neural Rendering Accelerators

Cheng-Jhih Shih Chaojian Li Chihao Yu† Hsuan-Chen Fang Sixu Li Wei-Po Hsin†
Lexington Whalen Hyewon Suh Greg Eisenhauer Ling Liu Yingyan (Celine) Lin
Georgia Institute of Technology, Atlanta, USA
{cshih63, cli851, hfang87, sli941, lwhalen7, hsuh45, celine.lin}@gatech.edu
{eisen, ling.liu}@cc.gatech.edu †eiclab.gatech@gmail.com

***Abstract*—Recent breakthroughs in neural rendering promise numerous real-time 3D intelligence applications, e.g., AR/VR, robotics, and digital twins. To satisfy real-time requirements, various neural rendering accelerators have been developed, with each employing a different rendering algorithm pipeline and designed for a specific hardware target. Such one-off specialization, representing only a single design point, creates two gaps: (i) fair, extensible cross-accelerator benchmarking, and (ii) design-space exploration (DSE) that retargets designs across pipelines or hardware budgets. We present NeRArch-Sim, an open-source simulator purpose-built for neural rendering accelerators with module-accurate models and validated end-to-end schedules (error $\leq$9.4%). NeRArch-Sim follows two principles: (1) modular abstractions for software workflows and hardware, enabling extensible benchmarking across diverse algorithmic pipelines and accelerator designs; and (2) a dataflow-aware two-level scheduler for rapid, effective DSE. Across eleven accelerators and two datasets, NeRArch-Sim reproduces prior designs with minimal changes (modeling error $\leq$9.4%) and guides new accelerators that achieve up to 1.3× efficiency gains. To our knowledge, NeRArch-Sim is the first open-source simulator that models a wide range of neural rendering accelerators, providing timely infrastructure for this emerging domain. The simulator is publicly available at https://github.com/GATECH-EIC/NeRArch-Sim.**

I. Introduction

Recent advances in neural rendering [56] have enabled immersive 3D intelligence applications, e.g., virtual telepresence, avatar generation, and embodied AI agents [36], [43], [67]. These applications demand both photorealistic quality and real-time performance, driving the development of specialized neural rendering accelerators [9], [12], [13], [23], [25], [27], [44], [51]. Each accelerator employs distinct scene representations, memory hierarchies, and ray sampling strategies to balance rendering quality and computational efficiency. However, such one-off specialization limits each accelerator to a single design point, creating two key challenges: (1) the lack of fair, extensible benchmarking across diverse accelerators, hindering systematic comparison and understanding of their relative merits; and (2) the absence of flexible design-space exploration (DSE) tools capable of adapting existing designs to new algorithmic pipelines or hardware constraints.

These challenges are further exacerbated by the absence of unified simulators tailored to neural rendering pipelines.

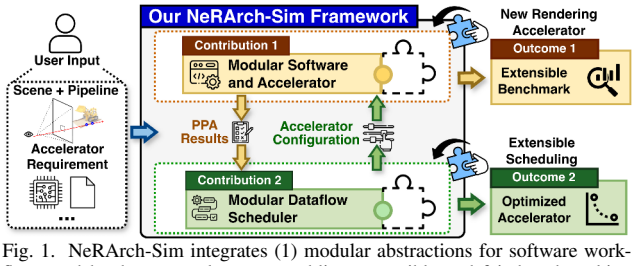

Fig. 1. NeRArch-Sim integrates (1) modular abstractions for software workflows and hardware accelerators, enabling extensible and fair benchmarking across diverse neural rendering designs; and (2) a modular dataflow scheduler for rapid, scalable design-space exploration (DSE), greatly simplifying the evaluation, reproduction, and optimization of neural rendering accelerators.

Existing simulators for neural network accelerators [41], [46] fail to support graphics-specific operations essential to neural rendering, e.g., ray marching [38], spatial sampling [49], and hybrid neural-graphics operators [40], [45]. Hence, researchers face substantial barriers in studying emerging pipelines, incorporating hardware-aware optimizations, and evaluating trade-offs across pipelines and accelerator architectures.

To address these limitations, we present *NeRArch-Sim*, the first unified, open-source simulator purpose-built for neural rendering accelerators. NeRArch-Sim adopts two core principles: (1) modular abstractions of software workflows and hardware accelerators for extensible, fair benchmarking; and (2) a modular dataflow scheduler enabling rapid, scalable, and effective DSE. Leveraging a unified taxonomy of rendering pipeline components and accelerator building blocks, NeRArch-Sim streamlines both the reproduction of existing accelerators and the systematic exploration of new design variants. It can also serve as an educational toolkit, performance profiler, and design automation backend within a

TABLE I
Comparison of existing 3D rendering accelerator simulators and NeRArch-Sim in terms of simulation accuracy, speed, extensibility, and hardware prototyping implementability.

Approach	Accuracy	Speed	Extensibility	Implementability
Analytical Model	✘ Low	✔ High	✔ High	✘ Low
RTL Simulation	✔ High	✘ Low	✘ Low	✔ High
NeRArch-Sim (Ours)	✔ High	✔ High	✔ High	✔ High

979-8-3315-5065-3/26/$31.00 ©2026 IEEE
DOI 10.1109/ISCA66397.2026.00124

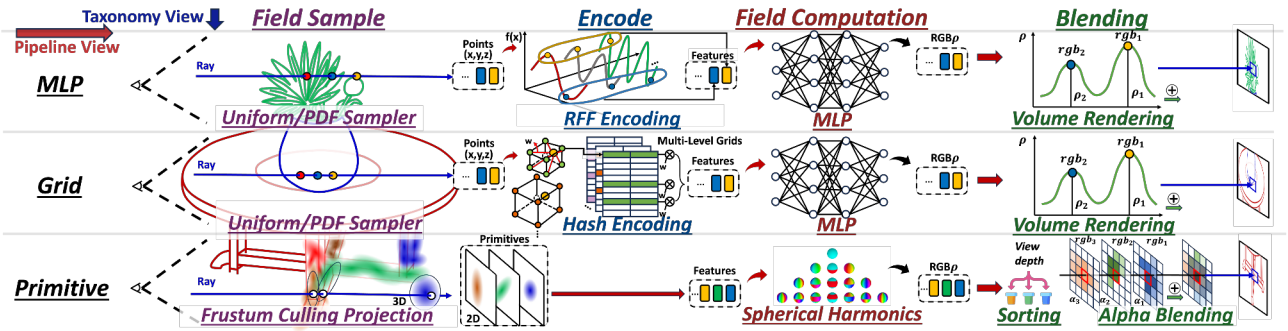

Fig. 2. Overview of different neural rendering pipelines from the perspectives of per-pipeline and the unified taxonomy.

cohesive open-source framework. Fig. 1 provides an overview of NeRArch-Sim, and Tab. I summarizes its advantages over prior simulation approaches. Our main contributions are:

- We present NeRArch-Sim, the first open-source simulator purpose-built for neural rendering accelerators, offering a unified infrastructure for systematic benchmarking and effective design-space exploration (DSE).
- NeRArch-Sim employs modular abstractions of software workflows and hardware accelerators, enabling extensible and fair benchmarking across diverse neural rendering pipelines and architectures.
- A modular dataflow scheduler enables efficient, scalable DSE across varied rendering pipelines and accelerator configurations.
- Experiments on eleven accelerators and two datasets demonstrate NeRArch-Sim's fidelity and flexibility, reproducing prior accelerators with minimal modification (modeling error $\leq$9.4%) and guiding new designs that achieve up to 1.3$\times$ higher efficiency.

II. Background & Motivation

A. Algorithm Trend: Diverse and Evolving Pipelines

As discussed in Sec. I, neural rendering has become a cornerstone of photorealistic 3D intelligence applications. Depending on requirements, e.g., memory footprint, speed, or quality, different algorithm pipelines offer distinct trade-offs. As summarized in Tab. II and illustrated in Fig. 2, representative pipelines include Multi-Layer Perceptron (MLP)-based [3], [39], [57], grid-based [4], [40], [45], [58], and primitive-based [8], [16], [17], [22] approaches.

MLP-based methods [39] sample 3D points along camera rays, encode them via Random Fourier Features (RFF) [39], and query an MLP to predict scene properties (e.g., density and color). Final outputs are produced through volume rendering [38]. These methods typically offer lower memory footprints than other pipelines [3], [4], [8].

Grid-based pipelines use discretized spatial structures, e.g., hash grids [40] or voxel grids [45], to store precomputed scene features. They retain similar sampling and volume rendering steps as MLP-based methods but deliver higher rendering quality on challenging datasets [57], [58].

Primitive-based approaches employ explicit geometric primitives, e.g., triangles [13] or 3D Gaussians (3DGS) [17]. Unlike ray-marching pipelines, they follow a rasterization paradigm, projecting 3D primitives onto 2D pixels and aggregating color and density. Leveraging optimized GPU rasterizers [21], these methods achieve high rendering speed. Although lacking explicit neural networks, they can be viewed as zero-layer networks with parameters learned via gradient descent, and are typically categorized as neural rendering [56].

Hybrid neural rendering [5], [6], [28], [66] is an emerging trend combining multiple pipelines to balance speed, memory, and quality. For example, [6] integrates grid- and 3DGS-based pipelines, achieving high frame rates and compact memory usage. Hence, supporting diverse and evolving pipelines—especially hybrids—has become essential.

TABLE II
Summary of key characteristics of varying neural rendering pipelines and their recent representative algorithms.

Pipeline	Representative Algorithms	Key Characteristic
MLP	[39], [3], [57]	Low Memory Footprint
Grid	[40], [4], [58]	High Rendering Quality
Primitive	[8], [22], [16]	Fast Rendering Speed

B. Accelerator Design Challenge: One-Off Specialization

As shown in Tab. II, each neural rendering pipeline offers unique advantages tailored to different applications. To accelerate rendering and support immersive 3D intelligence, various neural rendering accelerators have been developed [9], [12], [13], [23], [25], [27], [44], [51], summarized in Tab. III. Each

TABLE III
Representative accelerators across different pipelines summarized by hardware specifications. For designs with multiple variants, the server-level versions (NeuRex-Server [23], SRender-Server [51], and CICERO-16 [12]) are reported for fair comparison. Dashes (–) indicate metrics not reported in the original work.

Pipeline	Accelerators	FPS	Freq. (MHz)	Area (mm^2)	Tech. (nm)	Power (mW)
MLP	ICARUS [44]	0.017	400	16.5	40	282.8
	MetaVRain [13]	110	250	20.25	28	899
Grid	NeuRex [23]	19.72	1,000	21.37	28	6,100
	CICERO [12]	-	-	-	16	-
	SRender [51]	56.2	500	20.87	28	-
Primitive	GSCore [25]	190	1,000	3.95	28	870
	GS Processor [9]	373	700	2.43	28	664

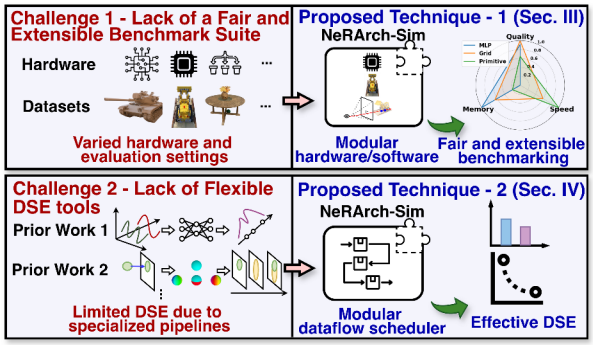

Fig. 3. Overview of key challenges in neural rendering acceleration and our proposed NeRArch-Sim solutions.

is optimized for a specific algorithmic pipeline, but such one-off specialization introduces key challenges (see Fig. 3).

Challenge 1 — Lack of a Fair and Extensible Benchmark Suite. Existing accelerators vary widely in hardware configurations and evaluation settings (Tab. III), hindering systematic comparison and insight into their relative strengths and weaknesses. They are often evaluated on different datasets and resolutions, further complicating fair comparison. A unified framework is needed to enable fair, extensible benchmarking across diverse algorithmic and architectural designs while lowering the barrier for non-expert users.

Challenge 2 — Lack of Flexible DSE Tools. Neural rendering involves large design spaces across hardware and mapping parameters—such as inter-stage buffer sizing, number of functional units per stage, and operator mapping—that impact utilization, bandwidth, and energy. While the accelerators in Tab. III achieve strong performance on specific pipelines, their DSE methodologies are too specialized and cover design objectives tied to their specific scenarios, and are often proprietary, limiting their adaptability to new pipelines or hardware constraints and reducing long-term reusability.

Both challenges stem from the absence of a unified simulator capable of accurately and efficiently evaluating neural rendering accelerators. Existing simulators for general neural network accelerators [41], [46] lack key graphics operations [39], [49] essential to these pipelines, and their operator and dataflow scheduling are tailored for regular tensor workloads. Addressing these benchmarking and DSE challenges requires a unified simulator specifically designed for the unique characteristics of neural rendering workloads.

C. Opportunity: Unified Taxonomy Enables Modular Design

Building a unified simulator for diverse neural rendering pipelines requires supporting not only the accelerators summarized in Tab. III but also future designs for emerging pipelines. Naively implementing each accelerator individually is unsustainable given the rapid evolution of rendering algorithms and hardware. A key opportunity lies in adopting a unified taxonomy that systematically decomposes neural rendering pipelines into modular components, enabling corresponding hardware accelerators to be modularized as well. This modularity allows reusing common modules across accelerators and integrating new ones by modifying only relevant components, without altering the overall simulator.

As summarized in Fig. 2, the taxonomy divides neural rendering pipelines into four key stages:

- **Field Sampler** samples objects (e.g., 3D points) within the scene along camera-emitted rays to define 3D regions of interest. MLP- and grid-based pipelines commonly use uniform or PDF-based sampling [39], while primitive-based pipelines apply frustum culling [17], [21] to discard primitives outside the target 2D region.
- **Encoding** converts each sampled object's position into feature vectors for downstream processing. RFF [39] and hash encoding [40] are widely used in MLP- and grid-based pipelines to enhance spatial discrimination. Primitive-based pipelines like 3DGS bypass this stage since primitives inherently store scene properties.
- **Field Computation** computes scene properties (e.g., color/density) of sampled objects based on encoded features, typically using MLPs or Spherical Harmonics [49].
- **Blending** aggregates scene properties to produce final pixel colors. MLP- and grid-based pipelines employ volume rendering [38], while primitive-based approaches commonly use sorting followed by alpha blending [17].

These four stages form a unidirectional pipeline, producing directed acyclic operator graphs (DAGs). Similar taxonomies adopted in neural rendering frameworks [54], [55] provide a unified view across pipelines and ensure compatibility with existing algorithmic infrastructures. This taxonomy thus enables a unified simulator supporting (1) extensible benchmarking via modular abstractions of software workflows (Sec. III-B) and hardware accelerators (Sec. III-C), and (2) effective DSE through a modular dataflow scheduler (Sec. IV).

III. The Proposed NeRArch-Sim Simulator

A. Overview

Leveraging the modular design opportunities enabled by the unified taxonomy described in Sec. II-C, we develop *NeRArch-Sim*, a unified simulator for modeling a variety of neural rendering accelerators. As illustrated in Fig. 4, NeRArch-Sim consists of three key components: **(1) Modular Software Workflow:** This workflow instruments existing software frameworks [17], [55], [64] to profile and generate runtime traces, extracting workload characteristics from user-defined rendering pipelines and datasets. **(2) Modular Hardware Accelerator:** The hardware component provides an open-source SystemC [1]-based codebase for modeling hardware modules defined by our taxonomy. It supports precise module-level modeling and reports power, performance, and area (PPA) metrics. Additionally, the design supports high-level synthesis (HLS), enabling deployability on real hardware platforms such as FPGAs and ASICs. This modular abstraction of both software and hardware forms the foundation for fair and extensible benchmarking. **(3) Modular Dataflow Scheduler:** The integrated scheduler bridges software-generated operator graphs with the underlying hardware, ensuring standardized

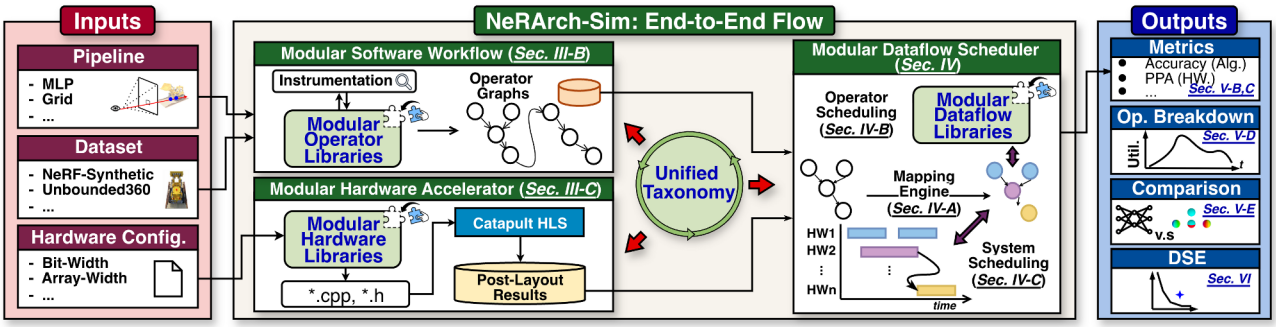

Fig. 4. Overview of the proposed NeRArch-Sim simulator, which comprises three main components: the **modular software workflow**, **modular hardware accelerator**, and **modular dataflow scheduler**. These components process user-specified algorithms and hardware configurations, producing both algorithmic and hardware metrics and enabling effective DSE that results in accelerator designs with better accuracy–efficiency trade-offs than existing solutions.

module mapping based on the taxonomy. It also performs optimal hardware resource allocation, enabling rapid evaluation of PPA metrics and facilitating effective DSE.

B. Modular Software Workflow: Operator Graph Generation

With the overall goal of generating an operator graph for subsequent workload analysis, the modular software workflow leverages taxonomized modules to collect runtime traces of function calls. NeRArch-Sim instruments workloads using lightweight runtime hooks that intercept function calls without modifying the original algorithms. As shown in Fig. 5, the hooks capture function identifiers (specified in `software.json`), call counts, tensor shapes, and tensor values while preserving caller-callee relationships. These traced logs are processed to construct an operator graph that captures function dependencies, input/output data sizes, and call frequencies. Additionally, the modular software workflow enables profiling operator characteristics on GPUs (e.g., roofline analysis) as a developer reference. The instrumented operator graph captures complete computational information from the rendering pipeline. Since all operators are invoked through the taxonomized interfaces, the runtime hooks capture every operator call and its data dependencies by construction. The modular software workflow further ensures no operator calls or algorithmic dependencies are missed by re-executing the extracted graph and comparing its rendered output against the original pipeline to verify no quality degradation.

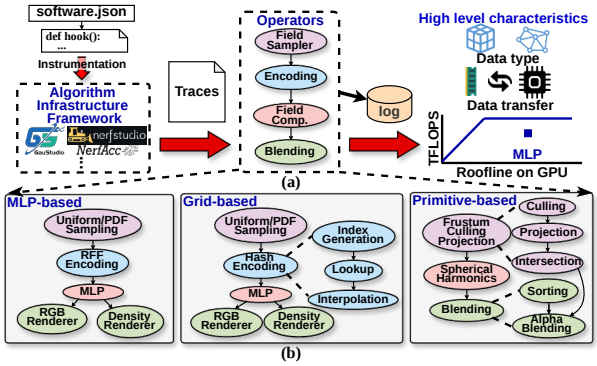

Fig. 5. Overview of NeRArch-Sim's modular software workflow. (a) Runtime instrumentation of widely used algorithm infrastructure frameworks to collect execution traces and extract operator graphs. Operator characteristics on GPU (e.g., roofline analysis) can additionally be profiled as a developer reference. (b) Examples of neural rendering operator graphs extracted by NeRArch-Sim for three different pipelines [17], [39], [40].

At the core of this workflow is the **Modular Operator Libraries**, which encapsulate computation primitives defined by the taxonomy in Sec. II-C. These libraries expose unified and extensible interfaces for constructing rendering pipelines. Each stage follows a taxonomized interface with common attributes while allowing pipeline-specific extensions through detailed parameters. This interface enables the operator library to adapt to algorithm-hardware co-designed variants.

```
def neurex_pipeline(dim):
    g = OperatorGraph()
    s = UniformSampler(dim, graph=g)
    e = HashEncoding(dim, num_levels=16, graph=g)
    m = MLP(dim, in_dim=e.out_dim, num_layers=4,
            layer_width=64, bit_width=8, graph=g)
    r_rgb = RGBRenderer(dim, graph=g)
    r_d   = DensityRenderer(dim, graph=g)
    s.add_child(e); e.add_child(m)
    m.add_child(r_rgb); m.add_child(r_d)
    return g
```

The code block above showcases construction of the NeuRex [23] pipeline using this programming interface. Common attributes are denoted in blue, and extensions in brown. Minor co-design parameters (e.g., precision) are handled through additional attributes (e.g., `bit_width` in `MLP`), while more significant algorithmic changes require new operators like `HashEncoding`, which implements the `Encoding` interface with its own parameters and is recognized by the scheduler in a later stage. Notably, updating these operator attribute parameters does not require re-instrumentation, as the operator graph structure remains unchanged, enabling rapid evaluation of algorithmic variants. In this way, our modular interface allows diverse and evolving rendering algorithms to be continuously supported in NeRArch-Sim.

It is worth noting that, thanks to the similarity between the taxonomy adopted by NeRArch-Sim and those used in existing algorithm frameworks [30], [55], [64], NeRArch-Sim can be seamlessly integrated into their workflows. This compatibility bridges software and hardware implementations and facilitates co-design of neural rendering accelerators.

C. Modular Hardware Accelerator: Implementation and Deployment

To ensure compatibility with the modular software workflow described in Sec. III-B, the modular hardware accelerator in

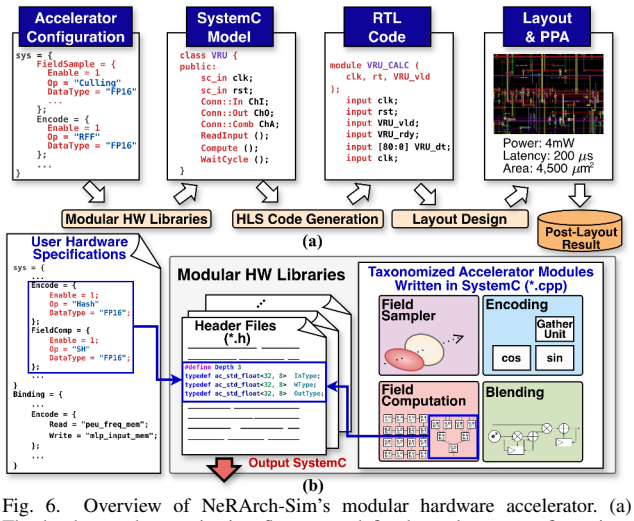

Fig. 6. Overview of NeRArch-Sim's modular hardware accelerator. (a) The hardware characterization flow: user-defined accelerator configurations are transformed into SystemC models, synthesized into RTL via HLS, and processed through layout design to produce post-layout PPA results. (b) The modular hardware libraries contain taxonomized SystemC modules organized by the unified taxonomy, with configurable header files that allow users to customize data types, precision, and memory bindings for each module.

TABLE IV
MODULAR HARDWARE LIBRARY COMPONENTS AND SUPPORTED CONFIGURABLE ARCHITECTURAL PARAMETERS

(a) Modular hardware library components

Category	Modules
Sampling (3)	Culling conversion unit [25], Skipping controller [9], Sampling unit
Encoding (10)	Position encoding unit [44], Address generator [23], Tree reducer, Index generation unit, Index computation unit, Distance computation unit [51], Comparison unit, Sensitivity prediction engine, Feed forward mapper [31], Backward update merger
Field Comp. (5)	MLP engines (4 variants: SSA [44], MONB, SONB, Systolic array [23]), Adder tree [31]
Blending (8)	Volume rendering units (3 variants [9], [14], [25]), Bitonic sort unit, Quick sort unit, Processing element arrays [9] (3 variants)

(b) Supported Configurable Architectural Parameters

Parameter Type	Examples
Synthesis Directives	Pipelining (initiation interval), loop unrolling, array partitioning (inherently supported by HLS)
Precision	Arbitrary integer, floating-point, and fixed-point precision
Implementation	CORDIC, piecewise linear for exponential arithmetic
Module Sizing	Systolic array dimensions, number of PEs, buffer depths
Parallelism Factor	Number of modules integrated
Interface	Channel depths, stream widths, handshaking protocols

our NeRArch-Sim framework is built using High-Level Synthesis (HLS) based on the same unified taxonomy introduced in Sec. II-C. The implementation begins by processing hardware configuration files to generate a SystemC HLS model for individual hardware modules. From this model, the simulator supports two levels of hardware characterization, as illustrated in Fig. 6-(a): (1) HLS synthesis for rapid PPA estimation, and (2) full ASIC post-layout flow for precise PPA analysis and real hardware deployment, capturing physical implementation effects including interconnect and routing overhead. Fig. 6-(b) shows modular hardware libraries, where taxonomized SystemC modules are paired with configurable headers specifying data types, precision, and memory bindings.

The modular design of the hardware accelerator is supported by the **Modular Hardware Libraries** containing 20+ reusable modules organized by the unified taxonomy. Table IV-(a) details the library components across all four categories, while Table IV-(b) presents the hierarchical architectural parameters exposed by each module, spanning from low-level synthesis directives to high-level interface specifications.

This structure allows developers to easily tailor hardware functionality to specific design requirements and plug in custom modules with minimal overhead. By reusing shared modules across different accelerator designs, NeRArch-Sim reduces implementation effort. Developers can quickly integrate new variants of accelerator modules into the framework with minimal modifications, as demonstrated in Sec. VII.

In addition to compute modules, each hardware configuration specifies the memory subsystem: named SRAM blocks with configurable capacity, bank count, port count, and access latency; explicit operator-to-SRAM bindings; and a DRAM backend modeled via Ramulator [18] for DRAM timing. These specifications feed the memory-aware duration modeling described in Sec. IV-B.

D. Modular Dataflow Scheduler: PPA Optimization

To bridge the modular software workflows and hardware accelerators, NeRArch-Sim adopts a dataflow scheduler leveraging the unified taxonomy for systematic hardware–software mapping and scheduling. The scheduler enables rapid PPA-oriented design space exploration, completing end-to-end scheduling within seconds on workstation-class CPUs. Sec. IV details the scheduling algorithms and optimization strategies.

IV. Modular Scheduling Implementation and Design Space Exploration

In this section, we dive into the detailed implementation of the modular dataflow scheduler, as illustrated in Fig. 7. The scheduling challenge in neural rendering is complex due to the *heterogeneous nature of operators* (ranging from matrix multiplications to hash table lookups) and *the diverse hardware modules* available (from systolic arrays to custom hash units). To address this complexity, NeRArch-Sim's scheduler adopts

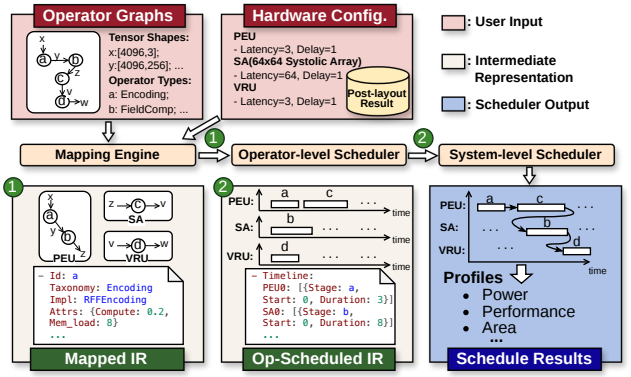

Fig. 7. Illustration of NeRArch-Sim's scheduling process. A domain-specific IR aligned with the unified taxonomy in Sec. II-C is used across the scheduling stages: the mapping engine converts the operator graph into a mapped IR with operator–hardware bindings; the operator-level scheduler schedules tasks per hardware module; and the system-level scheduler finalizes the execution plan and produces end-to-end PPA metrics.

TABLE V

(A) THE MAPPED IR, PRODUCED BY THE MAPPING ENGINE, SERVES AS THE INTERFACE TO OPERATOR-LEVEL SCHEDULING. (B) THE OPERATOR-SCHEDULED IR EXTENDS THE MAPPED IR WITH EXECUTION DETAILS WHILE MAINTAINING OPERATOR-TYPE INDEPENDENCE FOR SYSTEM-LEVEL SCHEDULING. FOR BREVITY, ONLY DETAILS FOR KEY FIELDS ARE SHOWN.

(a) MAPPED IR STRUCTURE	
Field	**Description**
Operator Information	
`op_id`	Unique operator identifier
`op_type`	Operator type from taxonomy (e.g., `HASH_ENCODE`)
`input_tensors`	Input tensor shapes and data types
`output_tensors`	Output tensor shapes and data types
Taxonomy-Specific Attributes	
`attributes/`	Dictionary containing taxonomy-specific parameters:
`Encoding`	`{encoding_type, hash_table_size, feature_dim}`
`Field Comp.`	`{network_depth, hidden_dim, activation}`
`Sampling`	`{num_samples, sampling_strategy}`
`Blending`	`{blend_mode, accumulation_type}`
Hardware Binding Information	
Resource Requirements	
Optimization Techniques	

(b) OPERATOR-SCHEDULED IR STRUCTURE	
Field	**Description**
Inherited from Mapped IR	
`[mapped_ir]`	All fields from mapped IR preserved
Execution Schedule	
`start_cycle`	Scheduled start cycle relative to hardware unit
`duration`	Execution duration in cycles
Resource Allocation	
Data Movement Schedule	
Optimization Metadata	

a systematic and hierarchical approach that decomposes the scheduling problem into three manageable stages: **mapping** (operator-to-hardware assignment), **operator-level scheduling** (local optimization within each hardware module), and **system-level** scheduling (global orchestration across modules), with each stage building upon the results of the previous one.

A. Mapping Engine: Operator-Hardware Binding and IR Generation

Our taxonomy reveals the inherent structure in neural rendering: each operator class aligns with specific hardware architectures. This alignment reflects structural correspondence. A mapping is valid only when an operator's I/O specifications and computational semantics match the hardware's supported operations; otherwise, NeRArch-Sim reports a mismatch. When multiple valid mappings exist, the engine selects the most efficient option based on expected performance.

The mapping engine leverages these relationships to automate implicit design knowledge, enabling rapid prototyping and systematic exploration. Given an operator graph and hardware configuration, the engine (❶ in Fig. 8) matches operators to hardware modules using the unified taxonomy, querying the modular operator and hardware libraries as needed. When several hardware units can execute an operator, it chooses the highest-throughput option; when multiple instances exist, it balances bindings to avoid bottlenecks. As the bridge between modular software workflows and hardware implementation, the mapping process draws from both domains: the **modular operator libraries** (Sec. III-B) provide computational patterns, data-access behavior, and taxonomy attributes, while the **modular hardware libraries** (Sec. III-C) specify compatible hardware units and capabilities.

This automated process produces the **mapped IR** (❷ in Fig. 8), which synthesizes the extracted information into a unified representation. Tab. V-(a) details the five sections of this IR: (1) *Operator Information* captures operator types and tensor specifications; (2) *Taxonomy-Specific Attributes* provides flexible parameters for each operator class; (3) *Hardware Binding Information* specifies the assigned hardware unit for each operator; (4) *Resource Requirements* quantifies computational and memory demands; and (5) *Optimization Techniques* describes the optimizations applied in later stages.

B. Operator-Level Scheduling: Domain-Specific Optimization Strategies

Following the mapping phase, the operator-level scheduler transforms the mapped IR into execution-ready schedules by applying operator-specific optimizations (❸ in Fig. 9) tailored to neural rendering workloads. This scheduling stage leverages domain-specific knowledge to optimize execution. By categorizing these patterns into a structured framework, the operator-level scheduler can select and combine optimization strategies based on operator characteristics and hardware capabilities.

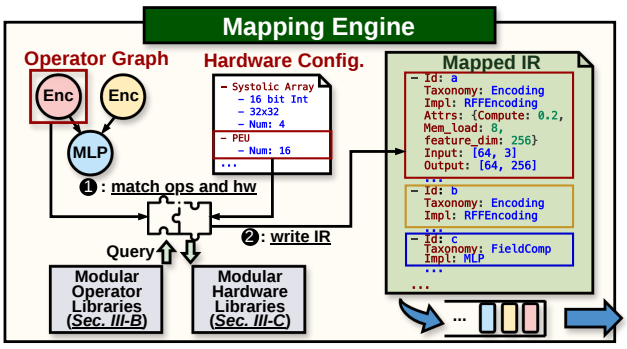

Fig. 8. Illustration of the mapping engine in modular scheduling. Given operator graphs and hardware configurations as inputs, the mapping engine matches operators to hardware using a unified taxonomy. During this process, it may query the modular operator and hardware libraries for necessary information. Finally, it outputs the matched results into a mapped IR for the operator-level scheduler to process.

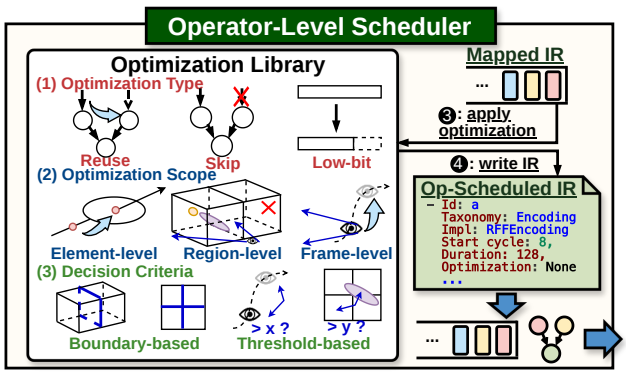

Fig. 9. Overview of the operator-level scheduler. Given the mapped IR input, the operator-level scheduler first queries the implemented optimization library to obtain operator-specific optimizations and execute the corresponding schedule. It generates the detailed scheduling results for each operator–hardware pair in the mapped IR and writes them to the operator-scheduled IR.

The scheduler produces the **operator-scheduled IR** (❹ in Fig. 9) illustrated in Tab. V-(b), augmenting the mapped representation with precise execution details while maintaining an abstract interface for system-level scheduling.

The key to applying these domain-specific optimizations lies in understanding the common patterns employed across existing accelerators. Through systematic analysis, we identified recurring optimization patterns that can be systematically categorized along three orthogonal dimensions, as summarized in Fig. 9. This three-dimensional framework provides a comprehensive categorization for understanding and applying optimizations in neural rendering accelerators.

Optimization type defines the fundamental operation performed by the optimization. **Reuse:** Shares computation results when multiple operations require the same intermediate values, exploiting redundancy in the rendering pipeline. **Skip:** Avoids unnecessary computation and data movement when results are negligible or predictable. **Low bit:** Utilizes low-precision arithmetic [44], [51] for importance computation to reduce memory bandwidth and energy consumption, and to accelerate execution without loss in rendering quality.

Optimization scope determines the granularity at which optimizations are applied. **Element-level:** targets individual rays, points, or primitives, including Gaussian skipping [9] that bypasses rendering based on contribution scores, per-ray early termination [23], [25] using accumulated opacity, and exponential value reuse [9] that shares computations across multiple Gaussians. **Region-level:** operates on spatial groups such as tiles, subgrids, or blocks. Restricted hashing [23] processes rays within subgrids, tile-based Gaussian processing [25] groups primitives by screen tiles, and bitmap culling [25] eliminates entire subtiles. **Frame-level:** spans temporal boundaries between frames. Sparse radiance warping [12] identifies and reuses pixels with similar rays across frames, transforming radiance values to avoid redundant computation.

Decision criteria specify the conditions determining when to apply optimizations. **Boundary-based:** Leverages geometric boundaries and spatial partitions to trigger optimizations. Examples include subgrid boundaries [23] and tile boundaries [25] for reuse decisions, and bounding box tests [25] for skip decisions. **Threshold-based:** Evaluates computed metrics against predefined values to activate optimizations. This includes angular proximity thresholds [12] for temporal reuse, sensitivity scores [51] for precision reduction, and alpha/distance thresholds [9] for Gaussian skipping.

This structured optimization framework, applicable across the unified taxonomy, together with NeRArch-Sim's optimization library implementation, enables developers to readily identify applicable techniques for new accelerator designs.

In NeRArch-Sim's implementation, the operator-level scheduler queries the optimization library and maps the selected strategies to per-operator time and memory traffic recorded in the op-scheduled IR. At a high level, we model each operator's time as in Eq. 1

$$\text{duration}(op) = \max\left(\frac{n_{\text{op}}}{\Theta_{\text{hw}}} \cdot s_{\text{comp}},\ \frac{v_{\text{off}}}{B_{\text{hw}}} \cdot r_{\text{bytes}}\right), \quad (1)$$

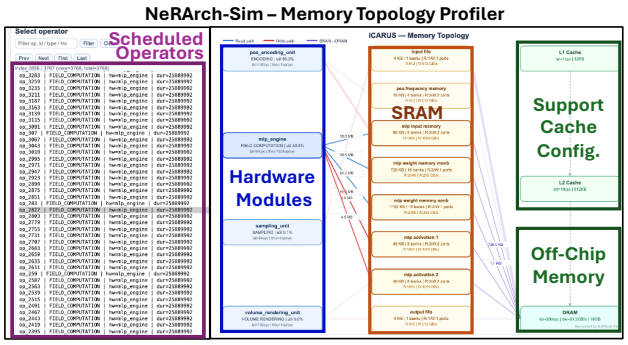

Fig. 10. Per-operator memory topology profiler for the ICARUS accelerator. Left panel: interactive operator selector with filtering. Right pane, left to right: hardware modules with utilization and throughput; SRAM instances with capacity, banks, ports, and bandwidth, connected via read/write bindings with data movement volumes on each edge; and the memory hierarchy with per-level latency and bandwidth.

where n_{op} is the number of elements processed; v_{off} is communication volume; Θ_{hw} is hardware compute throughput; and B_{hw} is effective memory bandwidth. These parameters are extracted from the mapped IR (Tab. V-(a)). The factors $s_{\text{comp}}, r_{\text{bytes}} \in (0, 1]$ interface with new dataflow optimizations, capturing compute and memory-traffic reduction from the optimization library. For instance, with tile culling optimization [25] in 3DGS field compute, we derive s_{comp} from the active Gaussian ratio.

Memory-aware duration modeling. The memory term in Eq. 1 ($v_{\text{off}}/B_{\text{hw}}$) depends on the memory topology. As illustrated in Fig. 10, NeRArch-Sim models a configurable memory hierarchy: hardware modules connect to other modules (direct data passing) or to SRAM instances with configurable capacity, banks, ports, and access latency; SRAM connects to DRAM via Ramulator 2.0 [18]. The updated duration and parameters in Eq. 1 are recorded in the op-scheduled IR and fed into system-level scheduling.

C. System-Level Scheduling: Global Orchestration Through Op-Scheduled IR

The system-level scheduler combines the local decisions into a globally coordinated execution plan, as shown in Fig. 11. This stage consumes the op-scheduled IR from all hardware modules and resolves cross-module dependencies to maximize

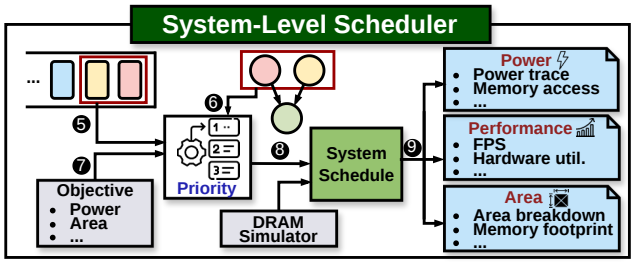

Fig. 11. Overview of the system-level scheduler, which aggregates op-scheduled IR results and analyzes operator dependencies to compute priority scores for each ready operation. Based on the scores, it commits operations to the system schedule while respecting resource constraints. The final schedule generates power, performance, and area metrics.

Algorithm 1 Dependency-Aware Greedy Scheduler (DAGS)

Require: Operator Graph $G = (V, E)$, op-scheduled IR S
Ensure: System-level schedule SS
procedure DAGS(G, S)
  $SS \leftarrow \emptyset$
  $Q \leftarrow$ GETSOURCENODES(G) ▷ all ready ops
  while $Q \neq \emptyset$ **do**
    for all $op \in Q$ **do**
      $d \leftarrow$ COUNTSUCCESSORS(op, G) ▷ number of dependent ops
      $c \leftarrow$ COMPUTECRITICALIMPACT(op, G, S) ▷ resource impact
      $score(op) \leftarrow \alpha\, d \;+\; \beta\, c$
    end for
    $sel \leftarrow \arg\max_{op \in Q} score(op)$
    $st \leftarrow$ FINDEARLIESTSLOT(sel, S, SS)
    $dur \leftarrow S[sel].duration$
    $SS[sel] \leftarrow (st, dur)$
    UPDATEREADYQUEUE(Q, G, SS)
  end while
  return SS
end procedure

system performance. The op-scheduled IR provides all necessary information for global coordination, allowing the system-level scheduler to remain agnostic to operator-specific details while still making informed decisions.

For system-level scheduling, neural rendering workloads exhibit natural parallelism: operators within the same stage (e.g., processing different rays or spatial points) execute independently, creating predictable synchronization points at stage boundaries. Leveraging this, NeRArch-Sim employs a Dependency-Aware Greedy Scheduler (DAGS) that models existing neural rendering accelerators with low error. The DAGS algorithm, shown in Algorithm 1, takes two key inputs: the operator graph $G = (V, E)$ capturing data dependencies and the op-scheduled IR S containing local scheduling decisions, resource allocations, and hardware constraints.

The DAGS algorithm employs two heuristics exploiting neural rendering's computational structure: **Successor Count** (d): Prioritizes operations at stage boundaries to unlock entire stages of independent operators, maximizing parallelism (❺ in Fig. 11). **Critical Resource Impact** (c): Accounts for heterogeneous resource demands across rendering stages (e.g., $n_{\text{op}}, v_{\text{off}}$ in Eq. 1), ensuring efficient utilization during stage transitions (❻ in Fig. 11). The scoring weights (α, β) enable NeRArch-Sim to model existing accelerators and explore alternative scheduling priorities (❼). The `FindEarliestSlot` function resolves resource contention by respecting resource availability, data dependencies, bandwidth constraints, and SRAM bank/port conflicts to determine conflict-free execution. Once the earliest valid slot is found, DAGS commits the operation to the system schedule with its start time and duration from the op-scheduled IR (❽). The resulting schedule drives NeRArch-Sim's performance modeling, generating power traces, memory patterns, and execution timelines that yield latency, throughput, power, and area metrics (❾).

While NeRArch-Sim's operation order may differ from existing accelerator simulators, DAGS targets modeling fidelity rather than scheduling optimality; the critical path remains unchanged under the same data dependencies and hardware constraints; thus, overall latency and energy stay within our reported error bounds. The effectiveness of DAGS is empirically validated in Sec. V-C with **low modeling error**, demonstrating that it captures the system-level scheduling complexity of existing neural rendering accelerators. The evaluated accelerators employ diverse dataflow strategies (weight-stationary in ICARUS, pipelined encoding-MLP overlap in NeuRex, tile-based sorting-rasterization overlap in GSCore), all captured through operator graph structure, hardware configuration, and memory bindings without changes to the scheduling algorithm.

DAGS also offers key advantages: (1) **Fast compilation** – scheduling completes in seconds even for complex workloads, enabling rapid iteration; (2) **Interpretable decisions** – hardware developers understand scheduling choices, facilitating hardware–software co-design; and (3) **Modular integration** – while our greedy approach suffices for current accelerators, the scheduler interface enables exploration of sophisticated algorithms for future architectures as needed.

V. EVALUATION

A. Evaluation Settings

NeRArch-Sim Implementation Details. For the algorithm infrastructure framework illustrated in Fig. 5, we adopt the commonly used NerfStudio [55] in our evaluation. The modular hardware accelerator in NeRArch-Sim is implemented using Catapult HLS [50] with SystemC [1]. The scheduler is implemented in C++. Additionally, we utilize a commercial 28nm HPCP technology node to extract hardware configuration results, and we normalize all baselines to 28nm via DeepScaleTool [48] for fair comparison. SRAM analysis uses the node's SRAM compiler. DRAM timing statistics are obtained using Ramulator [18].

Datasets & Baselines. For datasets, we evaluate small scenes on NeRF-Synthetic [39] (small-scale objects without backgrounds) and large scenes on Unbounded360 [3] (large-scale indoor/outdoor environments with complex backgrounds). To validate NeRArch-Sim's accuracy in modeling existing neural rendering pipelines, we include ICARUS [44], NeuRex [23], CICERO [12], GSCore [25], GBU [65], and Uni-Render [26] as baselines in our evaluation.

B. Individual Hardware Modules Validation

Tab. VI summarizes the modeling results for hardware accelerator modules (introduced in Sec. III-C) across different rendering pipelines, as reported by our modular hardware libraries in NeRArch-Sim and the reference full ASIC design flow, implemented using Fusion Compiler [53] and PrimePower [52] (marked in gray). Both flows start from the same SystemC source: NeRArch-Sim reports HLS-stage estimates, while the full ASIC flow reports post-layout measurements after synthesis and place-and-route. Specifically, the hardware PPA metrics reported by the modular hardware libraries closely align with those from the full ASIC flow. The modular hardware libraries produce identical latency results across all 17 evaluated modules and yield relative errors of 4.72% to 9.33% in area and power across various neural rendering accelerators. These results demonstrate NeRArch-Sim's capability to deliver both fast (minutes vs. hours) and accurate ($\leq$9.4%

TABLE VI
COMPARING THE MODELING RESULTS FROM OUR NERARCH-SIM AND FULL ASIC DESIGN FLOW (MARKED IN GRAY).

Module	Latency (cycle)	Area (μm^2)	Power (μW)	Avg Err.
ICARUS [44]				
Pos Encoding Unit	130/130	6714/5200	305/330	**9.04%**
MLP Engine	64/64	5.9/6.3×10^6	4.0/4.2×10^5	
Volume Rendering Unit	192/192	4755/4960	1917/2110	
NeuRex [23]				
Index Generation Unit	4/4	48563/51563	4836/5000	**9.33%**
Systolic Array (32x32)	37/37	5.4/5.4×10^5	1.1/1.3×10^5	
Interpolation Compute Unit	4/4	17371/15882	2144/2220	
CICERO [12]				
Reducer	8/8	557/628	181/234	**8.28%**
Address Generation	8/8	2745/3188	752/710	
NPU (24x24)	26/26	3.1/3.1×10^5	7.6/7.6×10^4	
GSCore [25]				
Culling & Conversion Unit	128/128	1.7/1.7×10^5	1.4/1.4×10^5	**5.10%**
Bitonic Sorting Unit	4/4	14620/12645	13700/13652	
Quick Sorting Unit	64/64	358/226	130/128	
Volume Rendering Unit	192/192	21690/23841	3270/2610	
GBU [65]				
Row Processing Element	10/10	41253/45345	14227/13750	**4.72%**
Row Generation Engine	8/8	15086/17521	4763/5220	
Decomposition & Binning Engine	16/16	1.1/1.1×10^5	30090/31157	
Uni-Render [26]				
PE Array (16x16)	18/18	13.2/14.2×10^6	5.1/5.4×10^6	**6.71%**

TABLE VII
COMPARISON OF MODELING RESULTS FROM OUR NERARCH-SIM AND CORRESPONDING PRIOR ACCELERATORS IN AN END-TO-END PIPELINE EVALUATION ON THE LEGO SCENE FROM THE NERF-SYNTHETIC DATASET [39]. THE RENDERING QUALITY (PSNR) AND FPS RESULTS FOR NEUREX [23] ARE TAKEN FROM [51]. THE RENDERING RESULTS FOR UNI-RENDER [26] ARE EVALUATED ON THE 3DGS PIPELINE.

Metric	ICARUS [44]	NeuRex [23]	CICERO [12]	GSCore [25]	GBU [65]	Uni-Render [26]
PSNR						
Reported	-	27.0	-	34.4	33.3	33.0
Reproduced	29.4	29.8	23.5	34.4	33.3	33.0
Average Err.			**2.6%**			
Area (mm^2)						
Reported	7.6	21.4	-	3.9	1.8	14.96
Reproduced	6.9	20.3	0.33	3.6	1.9	15.66
Average Err.			**6.5%**			
FPS						
Reported	0.02	19.7	-	190.0	180	65
Reproduced	0.02	18.6	326.4	182.2	172	63
Average Err.			**3.4%**			

relative error) PPA estimations of single modules, thereby enabling extensible benchmarking and effective DSE.

C. End-to-end Pipeline Benchmarking

To demonstrate NeRArch-Sim's ability to reproduce existing neural rendering accelerators for fair and extensible benchmarking, we summarize the comparison between the modeling results generated by NeRArch-Sim and the reported results from prior accelerator designs in Tab. VII. Specifically, the area and FPS of the end-to-end pipelines modeled by NeRArch-Sim closely match the reported metrics, with only 6.5% and 3.4% average error in area and FPS, respectively. Since prior works report end-to-end metrics, we validate at the same granularity. These results highlight NeRArch-Sim's potential as a reliable DSE tool for further exploring improvements to existing accelerators under more challenging design constraints, as detailed in Sec. VI.

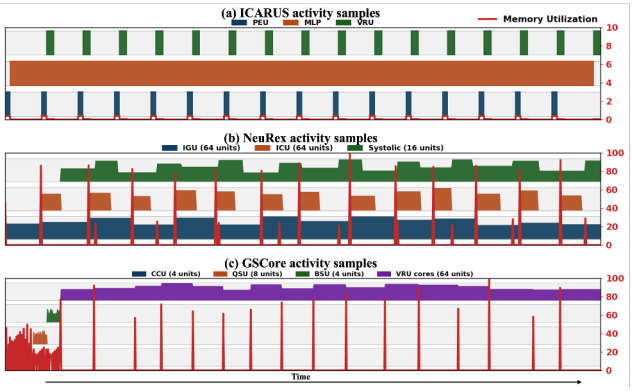

Fig. 12. Resource utilization analysis over time for three different pipelines evaluated on the Bicycle scene of the Unbounded360 dataset [3]. PEU: Position Encoding Unit; IGU: Index Generation Unit; ICU: Interpolation Compute Unit; CCU: Culling & Conversion Unit; QSU: Quick Sorting Unit; BSU: Bitonic Sorting Unit; VRU: Volume Rendering Unit.

D. End-to-end Resource Utilization Breakdown

NeRArch-Sim provides end-to-end resource-utilization analysis (Fig. 12), enabling developers to identify performance bounds and dominant bottlenecks. In Fig. 12-(a), ICARUS exhibits minimal memory activity due to effective on-chip SRAM caching of the MLP's compact model, while the persistently high MLP utilization indicates that the MLP stage is the dominant bottleneck. Fig. 12-(b) shows that NeuRex is primarily driven by IGU and systolic-array operations, with intermittent memory spikes from occasional data misses, suggesting that improving data locality alongside systolic-array balance would boost throughput. Fig. 12-(c) shows that GSCore performs a brief preprocessing stage before becoming dominated by volume-rendering workloads, indicating that accelerating VRU operations or reducing Gaussian count would yield further speedup.

E. Memory Analysis

Using the configurable memory model in Sec. IV-B and visualized in Fig. 10, we analyze the memory behavior of three representative accelerators.

Tab. VIII profiles per SRAM data flow for three accelerators. Both ICARUS and NeuRex perform ray marching at the Field Sampler stage, which streams ray sample coordinates from DRAM through the Input FIFO. For ICARUS, the Input FIFO streams 1.4 GB of sample coordinates per frame; weights (1.9 MB) are loaded once and reused ∼100K times; activations stay on-chip as ping-pong buffers. NeuRex shares the same ray marching stage but its DRAM traffic is dominated by position streaming (469 MB) and hash subtable loading (16 MB) for irregular lookups at fine resolution levels, while on-chip buffers absorb all MLP intermediates. GSCore does not use ray marching; instead it streams Gaussian features from DRAM per tile (79 MB), with sorting and rasterization intermediates fully on-chip.

TABLE VIII
PER SRAM DATA FLOW ANALYSIS ACROSS THREE ACCELERATORS ON THE LEGO SCENE [39].

SRAM Block	Size	SRAM Rd	SRAM Wr	DRAM Rd	DRAM Wr
ICARUS [44]					
Input FIFO	4 KB	1.4 GB	1.4 GB	1.4 GB	–
Frequency Mem.	16 KB	14.4 GB	–	16 KB	–
Input Memory	96 KB	14.4 GB	14.4 GB	–	–
Weight MONB	726 KB	468.8 GB	–	726 KB	–
Weight SONB	1.1 MB	58.6 GB	–	1.1 MB	–
Activation 1	48 KB	234.4 GB	234.4 GB	–	–
Activation 2	48 KB	234.4 GB	234.4 GB	–	–
Output FIFO	4 KB	–	3.7 MB	–	3.7 MB
NeuRex [23]					
Input FIFO	4 KB	469 MB	–	469 MB	–
Position Buffer	96 KB	469 MB	469 MB	–	–
Grid Cache	64 KB	9.8 GB	–	320 KB	–
Subgrid Buffer	128 KB	9.8 GB	–	16 MB	–
Weight Buffer	20 KB	19.5 GB	–	20 KB	–
Input Buffer	1 MB	2.4 GB	2.4 GB	–	–
Output Buffer	1 MB	5.1 GB	5.1 GB	–	–
Output FIFO	4 KB	–	3.7 MB	–	3.7 MB
GSCore [25]					
Gaussian In FIFO	8 KB	78.8 MB	–	78.8 MB	–
Tile List Buffer	32 KB	4.1 MB	4.1 MB	–	–
Sorting Buffer	64 KB	12.4 MB	12.4 MB	–	–
GFeat Buffer	128 KB	6.5 MB	9.5 MB	9.5 MB	–
Pixel Out Buffer	16 KB	–	3.1 MB	–	3.1 MB

TABLE IX
PER SRAM BANK CONFLICTS AND STALLS (ROUNDED). STALL OVERHEAD = STALL CYCLES / TOTAL EXECUTION CYCLES PER FRAME.

ICARUS [44]	Confl.	Stalls	**NeuRex** [23]	Confl.	Stalls	**GSCore** [25]	Confl.	Stalls
Freq. Mem.	9.1M	454K	Pos. Buf.	144K	7.2K	Tile List	2.5K	127
Input Mem.	4.5M	227K	Grid Cache	1.5M	76.8K	Sort. Buf.	3.8K	190
Wt. MONB	73.7M	3.7M	Subgrid Buf.	7.7M	384K	GFeat Buf.	8.8K	438
Wt. SONB	9.2M	461K	Wt. Buf.	12.3M	614K			
Act. 1	73.7M	3.7M	Input Buf.	384K	19.2K			
Act. 2	73.7M	3.7M	Output Buf.	804K	40.2K			
Total	**244M**	**12.2M**	**Total**	**22.8M**	**1.1M**	**Total**	**15.1K**	**755**
Overhead: 0.07%			**Overhead: 2.25%**			**Overhead: 0.01%**		

Tab. IX reports per SRAM bank conflicts and stall overhead. NeuRex has the highest relative overhead because its Subgrid Buffer serves irregular, near-random hash lookups at fine resolution levels. ICARUS has more absolute conflicts from activation ping-pong buffers concurrently read/written by adjacent MLP layers, but the overhead is negligible since MLP compute dominates execution. GSCore's sequential tile processing virtually eliminates contention in this case.

F. Comparison between accelerators

Tab. X compares accelerators across rendering pipelines in area, FPS, and PSNR. The results show that most pipelines except 3DGS fail to achieve 30 FPS as scene complexity increases. It also shows that CICERO uses the least resources to achieve relatively high FPS. Using NeRArch-Sim, we scale CICERO (CICERO+ in Tab. X) to compare with 3DGS pipelines and scale down GSCore (GSCore- in Tab. X) to similar FPS. The results show that CICERO+ requires fewer hardware resources than GSCore-.

TABLE X
COMPARING THE MODELING RESULTS OF NEURAL RENDERING ACCELERATORS ON THE UNBOUNDED360 DATASET [3].

Pipeline	Area (mm^2)	Garden		Bicycle		Stump	
		FPS	PSNR	FPS	PSNR	FPS	PSNR
ICARUS	5.911	7×10^{-4}	22.2	9×10^{-4}	21.2	8×10^{-4}	20.8
NeuRex	20.300	0.68	24.6	0.73	22.2	0.72	23.5
CICERO	0.330	9.57	23.5	10.92	21.8	10.64	23.3
GSCore	3.549	81.77	27.4	61.30	25.6	74.82	26.5
CICERO+	1.320	33.49	23.5	38.22	21.8	35.11	23.3
GSCore-	2.953	43.2	27.4	33.1	25.6	40.5	26.5

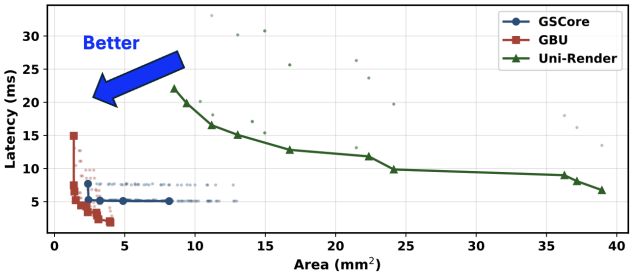

Fig. 13. Comparison of area and latency Pareto fronts for 3DGS accelerators on the Bicycle scene from the Unbounded360 dataset [3].

Fig. 13 further compares 3DGS accelerators—GSCore, GBU, and Uni-Render—in the area–latency space. Each point represents a valid configuration varying core count, on-chip memory, and bandwidth; solid curves highlight the Pareto-efficient operating points. GSCore and GBU lie in the low-area, low-latency region, indicating real-time performance with compact silicon footprints. Uni-Render occupies a higher-area region with higher latency, reflecting overheads from its general, multi-pipeline design.

G. Simulator Latency Breakdown

Tab. XI presents the end-to-end compilation latency breakdown for a single design point. The total latency comprises two components: (1) **software latency**, which includes instrumentation and operator graph construction (Sec. III-B), and (2) **scheduler latency**, consisting of mapping, operator-level scheduling, and system-level scheduling (Sec. IV). Hardware characteristics are precomputed offline and stored as lookup tables, eliminating the need for users to access design tools.

Tab. XI shows that system-level scheduling time correlates with the operator count, while operator-level scheduling latency is determined by both the operator count and inter-operator scheduling complexity. During hardware-only DSE (e.g., varying buffer sizes, core counts, precision), the operator graph structure remains static (Sec. III-B), so instrumentation is performed once and reused across all design points, reducing the amortized per-design-point latency to only the scheduler components (∼1 minute, Tab. XI). For co-design scenarios where the graph structure changes, each variant needs to be instrumented independently (Sec. VII-D).

TABLE XI
MODULAR SCHEDULER LATENCY BREAKDOWN FOR A SINGLE DESIGN POINT, MEASURED ON THE NERF-SYNTHETIC DATASET [39].

	ICARUS [44]	NeuRex [23]	CICERO [12]	GSCore [25]	GBU [65]	Uni-Render [26]
#Operators ($\times10^3$)	**60**	**180**	**180**	**10**	**10**	**10**
Total (s)	**50.7**	**79.2**	**75.4**	**47.7**	**48.1**	**48.4**
Instrumentation	31.0	45.2	43.5	23.2	22.8	22.9
Graph Construction	0.5	0.8	0.7	0.4	0.4	0.4
Mapping	5.3	4.6	4.4	2.1	2.0	2.0
Op-level Sched.	7.1	7.5	7.5	20.7	21.7	24.1
Sys-level Sched.	6.8	21.1	19.3	1.3	1.2	1.0

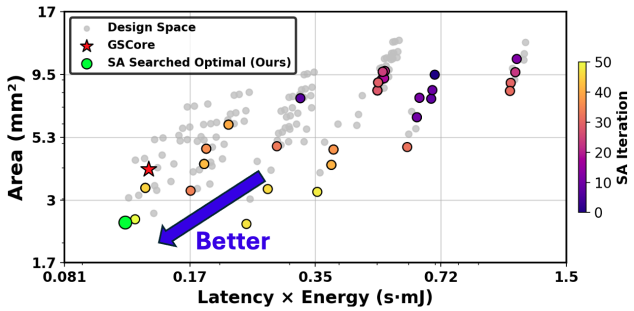

Fig. 14. DSE on GSCore [25] across various hardware module configurations and buffer sizes using the Unbounded360 dataset [3]. The results demonstrate that, in terms of energy-delay product and area, our NeRArch-Sim-based DSE with simulated annealing identifies a design point superior to GSCore.

VI. Design Space Exploration with NeRArch-Sim

To demonstrate NeRArch-Sim's effectiveness in design space exploration (DSE) for optimizing neural rendering accelerator designs, we present a case study in which NeRArch-Sim guides hardware designers toward achieving specific design objectives (e.g., a lower energy-delay product with similar area). As illustrated in Fig. 14, we apply NeRArch-Sim with Simulated Annealing (SA) [19] to perform DSE on GSCore [25] using the large-scale Unbounded360 dataset [3], which represents a more challenging scenario than the commonly used small-scale scenes with simple objects [39].

In this case study, the DSE objective is to identify a hardware configuration that minimizes both energy-delay product and area. NeRArch-Sim leverages SA to efficiently explore the design space, evaluating only a fraction of possible design points while maintaining high solution quality. Each design point corresponds to a specific configuration of five hardware parameters: (Culling Conversion Units, Quick Sorting Units, Bitonic Sorting Units, Volume Rendering Cores (VRCs), and Buffer sizes). As shown in Fig. 14, NeRArch-Sim with SA successfully locates multiple near-optimal design points that differ from the original GSCore configuration yet achieve lower area and improved energy-delay product. This optimization suggests that reducing over-provisioned resources (VRCs) lowers both area and power. In particular, the optimal design point found uses configuration (16, 8, 4, 32, 4) compared to GSCore's original (4, 8, 4, 64, 8), achieving a 1.3× reduction in energy-delay product and a 1.6× reduction in area.

The SA approach achieves a significant speedup of 2.8× compared to exhaustive search by navigating the design space through probabilistic acceptance of neighboring configurations. These results highlight NeRArch-Sim's ability to efficiently navigate complex design spaces using SA, support system-level performance optimization with reduced overhead, and facilitate the development of accelerators tailored to diverse application scenarios and evolving hardware constraints.

VII. Extending NeRArch-Sim to Support Emerging Accelerators and Workloads

We demonstrate NeRArch-Sim's extensibility by incorporating three rendering pipelines, SRender [51] (Sec. VII-A), Lumina (Sec. VII-B), and GS Processor [9] (Sec. VII-C), as well as two training pipelines, Instant-3D [31] and GSArch [14] (Sec. VII-D). We also extend NeRArch-Sim to support 4DGS as a new workload (Sec. VII-E). These case studies highlight the effectiveness of NeRArch-Sim's (1) modular abstractions for the software workflow and hardware accelerator, and (2) modular dataflow scheduler, which together enable seamless integration of new accelerators and new workloads with minimal engineering effort, typically under 300 lines of C++ code.

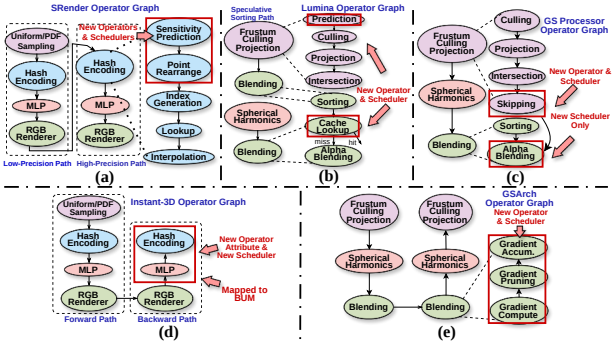

Fig. 15. (a) Illustration of the operator graph in SRender [51]. Two additional operators and their corresponding dataflow schedulers need to be implemented, while all other operators and schedulers can be directly reused. (b) Illustration of the operator graph in Lumina [10]. Two additional operators (Trajectory Prediction and Cache Lookup) and their corresponding dataflow schedulers need to be implemented to support the concurrent speculative sorting and radiance caching execution paths, while all other operators and schedulers can be directly reused. (c) Illustration of the operator graph in GS Processor [9]. One additional operator and two corresponding dataflow schedulers need to be implemented, while all other operators and schedulers can be directly reused. (d) Illustration of the operator graph in Instant-3D [31]. Backward support requires additional operator attributes, as well as a new dataflow scheduler to support it. (e) Illustration of the operator graph in GSArch [14]. New operators and a new scheduler are required to support backward execution.

A. Case Study I: Extending NeRArch-Sim to Support SRender

As summarized in Tab. XII and guided by the taxonomy defined in Sec. II-C, we integrate SRender [51] into our NeRArch-Sim by identifying the required components across modular hardware, software, and scheduler stacks. For the hardware library, three of the six modules required by SRender [51] are directly reused, while the remainder are added.

For the software workflow and dataflow scheduler, Fig. 15-(a) illustrates the operator graph of SRender [51], which adopts a coarse-to-fine rendering strategy based on ray or point sensitivity, followed by recovery [51]. The highlighted box indicates the additional operators and their corresponding schedulers that need to be implemented. All other components, including the system-level scheduler, remain unchanged. Thanks to NeRArch-Sim's high modularity and extensibility, along with its well-structured skeleton, an expert user can complete this integration within just two to three days. The final modeling results for SRender [51] are shown in Tab. XIII, achieving a relative area error of only 5.3%.

B. Case Study II: Lumina Extension in NeRArch-Sim

Case Study II integrates Lumina [10], a primitive-based 3DGS accelerator whose sorting-sharing and radiance caching mechanisms introduce cross-frame temporal state, making the

TABLE XII

TAXONOMY-GUIDED REUSE ANALYSIS OF HARDWARE MODULES IN SRENDER [51], LUMINA [10], GS PROCESSOR [9], INSTANT-3D [31], AND GSARCH [14]. MODULES ARE CATEGORIZED BASED ON THE TAXONOMY IN SEC. II-C, WITH REUSE STATUS AND AREA COMPARISON (REPORTED VS. MODELED BY NERARCH-SIM) SUMMARIZED.

Accelerator	Taxonomy	Hardware Module	Reused	Area (mm^2)	
				Reported	Ours
SRender [51]	Encoding	Hash Index Generators	✔	4.8	4.9
		Interpolation Units	✔	0.9	1.1
		Point Rearrangement Units	✘	0.2	0.17
		Distance Compute Units	✘	0.03	0.03
		Comparison Units	✘	0.01	0.01
	Field Computation	Processing Elements	✔	5.51	5.12
Lumina [10]	Blending	Neural Rendering Unit (NRU)	✔	–	0.010
		LuminCache	✘	–	0.112
		LuminCore	✘	1.05	1.15
GS Processor [9]	Blending	Rasterizing Elements	✔	–	0.465
		Interpolating Elements	✘	–	0.013
		Uni-Interpolating Elements	✘	–	0.025
		Hybrid Array	✔	0.450	0.503
	Field Sampler	Early Skipping Controller	✘	0.028	0.031
Instant-3D [31]	Encoding	Feed-Forward Read Mapper	✘	0.204	0.232
		Back-Propagation Update Merger	✘	1.428	1.503
	Field Computation	MLP Units	✔	1.496	1.721
GSArch [14]	Blending	Sorting Unit	✔	0.060	0.060
		Tile Merging Unit	✘	0.090	0.103
		Feature Computing Unit	✔	0.016	0.021
		Gradient Computing Unit	✘	0.030	0.036
		Gradient Pruning Unit	✘	0.020	0.022
		Rearrangement Unit	✘	0.004	0.003
	Field Sampler	Culling & Conversion Unit	✔	0.100	0.120

operator graph inherently stateful. Tab. XII summarizes the required components: the NRU is directly reused, while LuminCache and LuminCore are newly added. Fig. 15-(b) shows Lumina's operator graph, which extends the standard 3DGS pipeline with two concurrent paths for speculative sorting and cached rasterization. Two new operators and their schedulers are added; all others are reused. As shown in Tab. XIII, NeRArch-Sim achieves a relative FPS modeling error of 7.8%.

C. Case Study III: GS Processor Extension in NeRArch-Sim

Case Study III moves beyond simulation by integrating GS Processor [9], a real 3DGS accelerator, and validating against measured chip results. Following a similar integration process, we first identify the required hardware modules (Tab. XII). For the software workflow and dataflow scheduler, Fig. 15-(c) shows the operator graph of GS Processor, which introduces early-skipping and spatio-temporal fusion strategies in the blending stage. As shown in Tab. XIII, NeRArch-Sim achieves a relative latency modeling error of only 8.0% when compared against the actual fabricated GS Processor chip, validating its flexibility and accuracy across diverse accelerator types.

D. Case Study IV: Training Extension in NeRArch-Sim

Recent neural rendering accelerators increasingly focus on training acceleration. NeRArch-Sim enables training extension by augmenting existing operators with backward attributes, as illustrated in Fig. 15 (d) and (e). The same operators handle both forward and backward computation through different attribute configurations. The mapping engine directs backward-attributed operators to dedicated training hardware when available or reconfigured forward units otherwise. The operator-level scheduler models backward-specific optimizations when present, and the system-level scheduler orchestrates the complete forward-backward pipeline to generate final PPA metrics. Case Study IV demonstrates this capability through two accelerators **focusing on training**, with their required hardware modules and reuse analysis summarized in Tab. XII.

TABLE XIII

COMPARISON OF SIMULATION RESULTS FROM OUR NERARCH-SIM WITH THOSE REPORTED BY THE CORRESPONDING ORIGINAL ACCELERATORS ON THE LEGO SCENE IN THE NERF-SYNTHETIC DATASET [39].

Metric	SRender [51]	Lumina [10]	GS Processor [9]	Instant-3D [31]	GSArch [14]
	FPS			Time (s/scene)	
Reported	55.3	218	373	1.65	180
Reproduced (Ours)	52.2	201	343	1.80	191
Relative Error	5.6%	7.8%	8.0%	9.1%	6.1%
Reported PSNR	28.0	33.1	34.4	26.0	34.5
Reproduced PSNR (Ours)	28.3	33.9	33.9	28.3	33.9
Relative Error	3.5%	2.4%	1.5%	8.8%	1.7%

Instant-3D [31] accelerates grid-based training through asymmetric color-density decomposition, where hash encoding operators utilize feed-forward read mappers during forward passes and back-propagation update mergers during backward passes to consolidate memory accesses. Since the grid structure remains fixed throughout training, the operator graph is static and requires only a single instrumentation pass. The modeling results are shown in Tab. XIII, achieving a relative latency error of 9.1%.

GSArch [14] accelerates primitive-based training by mapping backward-attributed rasterization operators to dedicated backward processing engines and applying informativeness-based pruning and request rearrangement for efficient gradient accumulation. Primitive-based training produces dynamic operator graphs: as Gaussians are densified or pruned across iterations, the number of primitives changes, altering operator dimensions and dependencies. NeRArch-Sim instruments each graph variant independently when structure changes, while reusing all hardware and scheduling infrastructure. Data-dependent effects within a variant (e.g., gradient pruning) are captured via $s_{\text{comp}}/r_{\text{bytes}}$ without re-instrumentation. As shown in Tab. XIII, NeRArch-Sim achieves a relative latency error of 6.1% on GSArch.

E. Case Study V: 4DGS Workload Extension in NeRArch-Sim

Case Studies I–IV demonstrate NeRArch-Sim's extensibility along the architecture axis. Case Study V shows extensibility toward new workloads by integrating 4D Gaussian Splatting (4DGS) [7], [37], [60], [63], which extends 3DGS to dynamic scenes. As illustrated in Fig. 16-(a), 4DGS augments each Gaussian with a temporal dimension (x, y, z, t); rendering a target timestamp requires temporal culling and projection from 4D to 3D, after which the pipeline is identical to static 3DGS.

As shown in Fig. 16-(b), 4DGS requires two additional operators in the Field Sampler stage: Temporal Culling, which filters out Gaussians outside the target timestamp, and Temporal Projection, which extracts standard 3D Gaussians

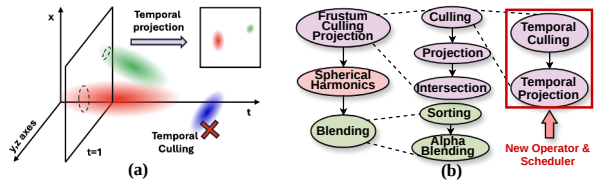

Fig. 16. (a) Illustration of 4DGS [7], [37], [60], [63]: each Gaussian is defined over (x, y, z, t). At a target timestamp t=1, temporal culling filters out Gaussians outside the temporal range (e.g., the blue Gaussian), and temporal projection extracts standard 3D Gaussians from the remaining 4D Gaussians. (b) Operator graph of 4DGS in NeRArch-Sim. Two operators, Temporal Culling and Temporal Projection, are added to the Field Sampler stage; all others are reused from the existing 3DGS pipeline.

TABLE XIV
EVALUATION OF 4DGS WORKLOAD ON GBU [65] USING THE FLAME STEAK SCENE [32].

Metric	FPS	PSNR
Reported	67	33.71
Reproduced (Ours)	63	33.8
Relative Error	5.9%	0.3%

from the remaining 4D Gaussians. All downstream operators and schedulers remain unchanged. We evaluate 4DGS on GBU [65] using the Flame Steak scene from the real-world dynamic scene dataset [32]. Tab. XIV reports a relative FPS error of 5.9%, demonstrating NeRArch-Sim's extensibility to new workloads with minimal additional modeling effort.

VIII. RELATED WORKS

Prior Simulators. As summarized in Tab. XV, traditional neural network simulators [41], [46], while equipped with capable dataflow schedulers for DSE, cannot satisfy the high operator heterogeneity required by neural rendering pipelines. This limitation stems from their lack of support for graphics-specific operators, as discussed in Sec. II-A. Adapting these simulators would require replacing both the operator abstractions and scheduling infrastructure to handle the irregular memory access patterns and data-dependent control flow common in neural rendering (e.g., early termination, Gaussian skipping), effectively amounting to a full redesign. On the other hand, existing in-house neural rendering simulators [12], [23], [25], [44] support a broader set of operators compared to general neural network simulators but do not incorporate dataflow schedulers for DSE.

Single Accelerator for Multiple Rendering Pipelines. To accommodate the diverse and evolving nature of neural rendering research, prior works [11], [26] have proposed general-purpose accelerators that support multiple pipelines within a single hardware accelerator. While sharing the same high-level goal, our work takes an orthogonal approach by developing a unified simulator for specialized neural rendering accelerators that target specific application scenarios. Notably, these general-purpose accelerators can also be supported by NeRArch-Sim, following the process in Sec. VII.

Emerging Neural Rendering Accelerators. Recent efforts extend neural rendering acceleration to dynamic workloads such as real-time SLAM and incremental training [29], [42], [59], while system techniques like efficiency-aware pruning and foveated rendering expose new workload heterogeneity [33]. Concurrently, architectures rethink execution paradigms via sparse pixel processing [15] and ray-tracing for Gaussians [24]. These trends diversify the operators, dataflows, and runtime behaviors in neural rendering acceleration, reinforcing the value of NeRArch-Sim's extensible simulation infrastructure for evaluating future designs.

TABLE XV
COMPARISON OF TRADITIONAL NEURAL NETWORK SIMULATORS, DEDICATED NEURAL RENDERING SIMULATORS, AND OUR NERARCH-SIM BASED ON KEY SIMULATOR CHARACTERISTICS.

Simulator Characteristics	Neural Network Simulators	Dedicated Neural Rendering Simulators	Our NeRArch-Sim
Operator Heterogeneity	Low	Medium	High
Dataflow Scheduler	✔	✘	✔

Neural Rendering Benchmarks. Early benchmarks [2], [20], [35], [39] established quality metrics adopted across pipelines, but primitive-based methods require additional metrics including compression ratios, real-time performance, and memory scaling. Recent datasets evaluate these primitive specific aspects: GGSC [62] for 3DGS compression, DL3DV-10K [34] and GauU-Scene [61] for large-scale scene evaluation. Furthermore, hardware benchmarking [47] reveals rasterization bottlenecks and power efficiency challenges in 3DGS. The development of new benchmarks necessitates simulation frameworks with extensible evaluation capabilities, which NeRArch-Sim addresses through its modular architecture.

IX. CONCLUSION

In this work, we present NeRArch-Sim, the first open-source, modular, and low modeling error simulator specifically designed for neural rendering accelerators, to the best of our knowledge. NeRArch-Sim is built upon modular abstractions of the software workflow, hardware accelerator, and dataflow scheduler. The proposed simulator enables fair and extensible benchmarking across diverse accelerators, supporting efficient DSE to identify architectures that offer improved accuracy–efficiency trade-offs compared to existing solutions.

ACKNOWLEDGEMENTS

This article is based upon work supported by the National Science Foundation (NSF) (Award IDs: 2312758, 2434166, 2048183, and 2016727), the Department of Health and Human Services Advanced Research Projects Agency for Health (ARPA-H) under Award Number AY1AX000003 and Agreement Number 140D042490003, and CoCoSys, one of the seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied of the Advanced Research Projects Agency Health or the U.S. Government.

REFERENCES

[1] Accellera Systems Initiative, "systemc.org," 2024.

[2] H. Baatz, J. Granskog, M. Papas, F. Rousselle, and J. Novák, "Nerf-tex: Neural reflectance field textures," in *Computer Graphics Forum*, vol. 41, no. 6. Wiley Online Library, 2022, pp. 287–301.

[3] J. T. Barron, B. Mildenhall, D. Verbin, P. P. Srinivasan, and P. Hedman, "Mip-nerf 360: Unbounded anti-aliased neural radiance fields," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2022, pp. 5470–5479.

[4] ——, "Zip-nerf: Anti-aliased grid-based neural radiance fields," *arXiv preprint arXiv:2304.06706*, 2023.

[5] J. Cao, V. Goel, C. Wang, A. Kag, J. Hu, S. Korolev, C. Jiang, S. Tulyakov, and J. Ren, "Lightweight predictive 3d gaussian splats," *arXiv preprint arXiv:2406.19434*, 2024.

[6] Y. Chen, Q. Wu, W. Lin, M. Harandi, and J. Cai, "Hac: Hash-grid assisted context for 3d gaussian splatting compression," in *European Conference on Computer Vision*. Springer, 2024, pp. 422–438.

[7] Y. Duan, F. Wei, Q. Dai, Y. He, W. Chen, and B. Chen, "4d-rotor gaussian splatting: towards efficient novel view synthesis for dynamic scenes," in *ACM SIGGRAPH 2024 Conference Papers*, 2024, pp. 1–11.

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

[9] X. Feng, H. Wang, C. Tang, T. Wu, H. Yang, and Y. Liu, "1.78 mj/frame 373fps 3d gs processor based on shape-aware hybrid architecture using earlier computation skipping and gaussian cache scheduler," in *2025 IEEE International Solid-State Circuits Conference (ISSCC)*, vol. 68. IEEE, 2025, pp. 1–3.

[10] Y. Feng, W. Lin, Y. Cheng, Z. Liu, J. Leng, M. Guo, C. Chen, S. Sun, and Y. Zhu, "Lumina: Real-time mobile neural rendering by exploiting computational redundancy," *arXiv preprint arXiv:2506.05682*, 2025.

[11] Y. Feng, W. Lin, Z. Liu, J. Leng, M. Guo, H. Zhao, X. Hou, J. Zhao, and Y. Zhu, "Potamoi: Accelerating neural rendering via a unified streaming architecture," *ACM Transactions on Architecture and Code Optimization*, vol. 21, no. 4, pp. 1–25, 2024.

[12] Y. Feng, Z. Liu, J. Leng, M. Guo, and Y. Zhu, "Cicero: Addressing algorithmic and architectural bottlenecks in neural rendering by radiance warping and memory optimizations," in *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2024, pp. 1293–1308.

[13] D. Han, J. Ryu, S. Kim, S. Kim, and H.-J. Yoo, "2.7 metavrain: A 133mw real-time hyper-realistic 3d-nerf processor with 1d-2d hybrid-neural engines for metaverse on mobile devices," in *2023 IEEE International Solid- State Circuits Conference (ISSCC)*, 2023, pp. 50–52.

[14] H. He, G. Li, F. Liu, L. Jiang, X. Liang, and Z. Song, "Gsarch: Breaking memory barriers in 3d gaussian splatting training via architectural support," in *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2025, pp. 366–379.

[15] X. Huang, H. Zhu, T. Ma, Y. Xiong, F. Liu, Z. He, Y. Gan, Z. Liu, J. Leng, Y. Feng *et al.*, "Splatonic: Architecture support for 3d gaussian splatting slam via sparse processing," *arXiv preprint arXiv:2511.18755*, 2025.

[16] Y. Jiang, J. Tu, Y. Liu, X. Gao, X. Long, W. Wang, and Y. Ma, "Gaussianshader: 3d gaussian splatting with shading functions for reflective surfaces," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2024, pp. 5322–5332.

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

[18] Y. Kim, W. Yang, and O. Mutlu, "Ramulator: A fast and extensible dram simulator," *IEEE Computer architecture letters*, vol. 15, no. 1, pp. 45–49, 2015.

[19] S. Kirkpatrick, C. D. Gelatt Jr, and M. P. Vecchi, "Optimization by simulated annealing," *science*, vol. 220, no. 4598, pp. 671–680, 1983.

[20] A. Knapitsch, J. Park, Q.-Y. Zhou, and V. Koltun, "Tanks and temples: Benchmarking large-scale scene reconstruction," *ACM Transactions on Graphics*, vol. 36, no. 4, 2017.

[21] S. Laine and T. Karras, "High-performance software rasterization on gpus," in *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*, 2011, pp. 79–88.

[22] J. C. Lee, D. Rho, X. Sun, J. H. Ko, and E. Park, "Compact 3d gaussian representation for radiance field," *arXiv preprint arXiv:2311.13681*, 2023.

[23] 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.

[24] J. Lee, S. Jeon, J. Lee, J. Park, and J. Sim, "Grtx: Efficient ray tracing for 3d gaussian-based rendering," *arXiv preprint arXiv:2601.20429*, 2026.

[25] J. Lee, S. Lee, J. Lee, J. Park, and J. Sim, "Gscore: Efficient radiance field rendering via architectural support for 3d gaussian splatting," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, 2024, pp. 497–511.

[26] C. Li, S. Li, L. Jiang, J. Zhang, and Y. C. Lin, "Uni-render: A unified accelerator for real-time rendering across diverse neural renderers," *arXiv preprint arXiv:2503.23644*, 2025.

[27] 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.

[28] C. Li, B. Wu, P. Vajda, and Y. Lin, "Mixrt: Mixed neural representations for real-time nerf rendering," in *International Conference on 3D Vision (3DV)*, 2024.

[29] L. Li, J. Qin, J. Peng, Z. Wan, H. Qu, Y. Han, P. Zheng, H. Zhang, Y. Cao, T. Chen *et al.*, "Rtgs: Real-time 3d gaussian splatting slam via multi-level redundancy reduction," in *Proceedings of the 58th IEEE/ACM International Symposium on Microarchitecture*, 2025, pp. 1838–1851.

[30] R. Li, M. Tancik, and A. Kanazawa, "Nerfacc: A general nerf acceleration toolbox," *arXiv preprint arXiv:2210.04847*, 2022.

[31] 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.

[32] T. Li, M. Slavcheva, M. Zollhoefer, S. Green, C. Lassner, C. Kim, T. Schmidt, S. Lovegrove, M. Goesele, R. Newcombe *et al.*, "Neural 3d video synthesis from multi-view video," in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2022, pp. 5521–5531.

[33] W. Lin, Y. Feng, and Y. Zhu, "Metasapiens: Real-time neural rendering with efficiency-aware pruning and accelerated foveated rendering," in *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, 2025, pp. 669–682.

[34] L. Ling, Y. Sheng, Z. Tu, W. Zhao, C. Xin, K. Wan, L. Yu, Q. Guo, Z. Yu, Y. Lu *et al.*, "Dl3dv-10k: A large-scale scene dataset for deep learning-based 3d vision," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2024, pp. 22 160–22 169.

[35] 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.

[36] Y. Liu, K. Zhu, G. Wu, Y. Ren, B. Liu, Y. Liu, and J. Shan, "Mv-deepsdf: Implicit modeling with multi-sweep point clouds for 3d vehicle reconstruction in autonomous driving," in *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 2023, pp. 8306–8316.

[37] J. Luiten, G. Kopanas, B. Leibe, and D. Ramanan, "Dynamic 3d gaussians: Tracking by persistent dynamic view synthesis," in *2024 International Conference on 3D Vision (3DV)*. IEEE, 2024, pp. 800–809.

[38] N. Max, "Optical models for direct volume rendering," *IEEE Transactions on visualization and computer graphics*, vol. 1, no. 2, pp. 99–108, 2002.

[39] 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," in *European Conference on Computer Vision*, 2020.

[40] T. Müller, A. Evans, C. Schied, and A. Keller, "Instant neural graphics primitives with a multiresolution hash encoding," *ACM Trans. Graph.*, vol. 41, no. 4, pp. 102:1–102:15, Jul. 2022.

[41] A. Parashar, P. Raina, Y. S. Shao, Y.-H. Chen, V. A. Ying, A. Mukkara, R. Venkatesan, B. Khailany, S. W. Keckler, and J. Emer, "Timeloop: A systematic approach to dnn accelerator evaluation," in *2019 IEEE international symposium on performance analysis of systems and software (ISPASS)*. IEEE, 2019, pp. 304–315.

[42] M. Pei, G. Li, J. Si, Z. Zhu, Z. Mo, P. Wang, Z. Song, X. Liang, and J. Cheng, "Gcc: A 3dgs inference architecture with gaussian-wise and cross-stage conditional processing," in *Proceedings of the 58th*

IEEE/ACM International Symposium on Microarchitecture, 2025, pp. 1824–1837.

[43] J. Philip, M. Gharbi, T. Zhou, A. A. Efros, and G. Drettakis, "Multi-view relighting using a geometry-aware network." *ACM Trans. Graph.*, vol. 38, no. 4, pp. 78–1, 2019.

[44] C. Rao, H. Yu, H. Wan, J. Zhou, Y. Zheng, M. Wu, Y. Ma, A. Chen, B. Yuan, P. Zhou *et al.*, "Icarus: A specialized architecture for neural radiance fields rendering," *ACM Transactions on Graphics (TOG)*, vol. 41, no. 6, pp. 1–14, 2022.

[45] C. Reiser, R. Szeliski, D. Verbin, P. Srinivasan, B. Mildenhall, A. Geiger, J. Barron, and P. Hedman, "Merf: Memory-efficient radiance fields for real-time view synthesis in unbounded scenes," *ACM Transactions on Graphics (TOG)*, vol. 42, no. 4, pp. 1–12, 2023.

[46] A. Samajdar, Y. Zhu, P. N. Whatmough, M. Mattina, and T. Krishna, "Scale-sim: Systolic cnn accelerator," *CoRR*, 2018.

[47] S. Samudrala, S. Kondguli, and P. Gratz, " Benchmarking 3D Gaussian Splatting Rendering ," in *2025 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2025, pp. 227–238. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/ISPASS64960.2025.00029

[48] S. Sarangi and B. Baas, "DeepScaleTool: A tool for the accurate estimation of technology scaling in the deep-submicron era," in *2021 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, 2021, pp. 1–5.

[49] P. Shirley, M. Ashikhmin, and S. Marschner, *Fundamentals of computer graphics*. AK Peters/CRC Press, 2009.

[50] Siemens, "Catapult High-Level Synthesis and Verification," 2024.

[51] Z. Song, H. He, F. Liu, Y. Hao, X. Song, L. Jiang, and X. Liang, "Srender: Boosting neural radiance field efficiency via sensitivity-aware dynamic precision rendering," in *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2024, pp. 525–537.

[52] Synopsys, "PrimePower," 2022.

[53] Synopsys, Inc., "Fusion Compiler: Achieve Your Simply Better PPA," 2024, https://www.synopsys.com/implementation-and-signoff/physical-implementation/fusion-compiler.html.

[54] T. Takikawa, O. Perel, C. F. Tsang, C. Loop, J. Litalien, J. Tremblay, S. Fidler, and M. Shugrina, "Kaolin wisp: A pytorch library and engine for neural fields research," 2022.

[55] M. Tancik, E. Weber, E. Ng, R. Li, B. Yi, J. Kerr, T. Wang, A. Kristoffersen, J. Austin, K. Salahi, A. Ahuja, D. McAllister, and A. Kanazawa, "Nerfstudio: A modular framework for neural radiance field development," in *ACM SIGGRAPH 2023 Conference Proceedings*, ser. SIGGRAPH '23, 2023.

[56] A. Tewari, J. Thies, B. Mildenhall, P. P. Srinivasan, E. Tretschk, Y. Wang, C. Lassner, V. Sitzmann, R. Martin-Brualla, S. Lombardi, T. Simon, C. Theobalt, M. Nießner, J. T. Barron, G. Wetzstein, M. Zollhöfer, and V. Golyanik, "Advances in neural rendering," in *Computer Graphics Forum*, vol. 41, no. 2. Wiley Online Library, 2022, pp. 703–735.

[57] D. Verbin, P. Hedman, B. Mildenhall, T. Zickler, J. T. Barron, and P. P. Srinivasan, "Ref-nerf: Structured view-dependent appearance for neural radiance fields," in *2022 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE, 2022, pp. 5481–5490.

[58] D. Verbin, P. P. Srinivasan, P. Hedman, B. Mildenhall, B. Attal, R. Szeliski, and J. T. Barron, "Nerf-casting: Improved view-dependent appearance with consistent reflections," in *SIGGRAPH Asia 2024 Conference Papers*, 2024, pp. 1–10.

[59] H. Wang, Z. Zhu, T. Zhao, Y. Xiang, Z. Wang, J. Yu, H. Yang, Y. Xie, and Y. Wang, "React3d: Real-time edge accelerator for incremental training in 3d gaussian splatting based slam systems," in *Proceedings of the 58th IEEE/ACM International Symposium on Microarchitecture*, 2025, pp. 1852–1866.

[60] G. Wu, T. Yi, J. Fang, L. Xie, X. Zhang, W. Wei, W. Liu, Q. Tian, and X. Wang, "4d gaussian splatting for real-time dynamic scene rendering," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2024, pp. 20 310–20 320.

[61] B. Xiong, Z. Li, and Z. Li, "Gauu-scene: A scene reconstruction benchmark on large scale 3d reconstruction dataset using gaussian splatting," *arXiv preprint arXiv:2401.14032*, 2024.

[62] Q. Yang, K. Yang, Y. Xing, Y. Xu, and Z. Li, "A benchmark for gaussian splatting compression and quality assessment study," in *Proceedings of the 6th ACM International Conference on Multimedia in Asia*, 2024, pp. 1–8.

[63] Z. Yang, H. Yang, Z. Pan, and L. Zhang, "Real-time photorealistic dynamic scene representation and rendering with 4d gaussian splatting," *arXiv preprint arXiv:2310.10642*, 2023.

[64] C. Ye, Y. Nie, J. Chang, Y. Chen, Y. Zhi, and X. Han, "Gaustudio: A modular framework for 3d gaussian splatting and beyond," *arXiv preprint arXiv:2403.19632*, 2024.

[65] Z. Ye, Y. Fu, J. Zhang, L. Li, Y. Zhang, S. Li, C. Wan, C. Wan, C. Li, S. Prathipati *et al.*, "Gaussian blending unit: An edge gpu plug-in for real-time gaussian-based rendering in ar/vr," in *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2025, pp. 353–365.

[66] H. Yu, J. Julin, Z. Á. Milacski, K. Niinuma, and L. A. Jeni, "Cogs: Controllable gaussian splatting," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2024, pp. 21 624–21 633.

[67] W. Zielonka, T. Bolkart, and J. Thies, "Instant volumetric head avatars," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2023, pp. 4574–4584.