2025 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS) | 979-8-3315-0294-2/25/$31.00 ©2025 IEEE | DOI: 10.1109/ISPASS64960.2025.00016

Hierarchical Traversal Stack Design Using Shared Memory for GPU Ray Tracing

Eunsoo Jung[†], Eunbi Jeong[†], Gunjae Koo[‡], Yunho Oh[§*], and Myung Kuk Yoon[†*]
[†]*Department of Computer Science and Engineering, Ewha Womans University, Seoul, Korea*
[‡]*Department of Computer Science and Engineering, Korea University, Seoul, Korea*
[§]*School of Electrical Engineering, Korea University, Seoul, Korea*
{*jsoo615, eb0313, myungkuk.yoon*}*@ewha.ac.kr, gunjaekoo@korea.ac.kr, yunho_oh@korea.ac.kr*

***Abstract*—Ray tracing is widely used to generate photorealistic images by tracing the paths of light rays through a scene and their interactions with scene objects. To accelerate ray tracing, an acceleration structure—typically a bounding volume hierarchy—organizes scene primitives into an efficient spatial data structure, commonly traversed using a traversal stack. Modern GPUs are equipped with specialized ray tracing acceleration units to accelerate traversal and intersection tasks. With limited on-chip storage, the traversal stack is kept short, leading to frequent spilling and reloading operations between on-chip buffers and off-chip memory during stack overflows. This paper reveals that such overflows increase off-chip memory traffic, degrading overall performance. To address this, we propose SMS, a novel GPU architecture that leverages shared memory as a secondary traversal stack. The proposed design uses the shared memory stack to complement the primary on-chip stack, thus reducing off-chip traffic caused by stack overflows. Additionally, two optimizations for managing shared memory stacks are proposed: skewed bank access and dynamic intra-warp reallocation. Through effective management of traversal stacks, the proposed SMS architecture achieves a 23.2% performance improvement over a baseline GPU that uses only a primary on-chip stack.**

I. Introduction

Ray tracing is a rendering method known for producing photorealistic images by simulating the paths of light rays within a scene and their interactions with scene objects. In a naïve approach, each ray is tested for intersections against all scene primitives, typically triangles. However, this method is computationally expensive, particularly for complex modern scenes containing millions of triangles. To address this issue, an acceleration structure is used to organize scene primitives into a hierarchical spatial data structure, enabling efficient pruning of the search space [11], [19], [32], [37]. The most widely adopted structure is the bounding volume hierarchy (BVH) [32], illustrated on the right side of Fig. 1. BVH arranges primitives into a tree of bounding volumes, typically axis-aligned bounding boxes (AABBs). As each ray traverses the tree, intersections are first tested against the bounding volumes. If a node's bound is not intersected, its entire subtree is skipped, reducing the number of primitive intersection tests. BVH is commonly traversed with a depth-first search approach, using a traversal stack for backtracking. While BVH traversal reduces computational complexity, performance remains limited by long-latency memory operations due to rays following diverse paths, causing divergent memory accesses and frequent cache misses [7], [23].

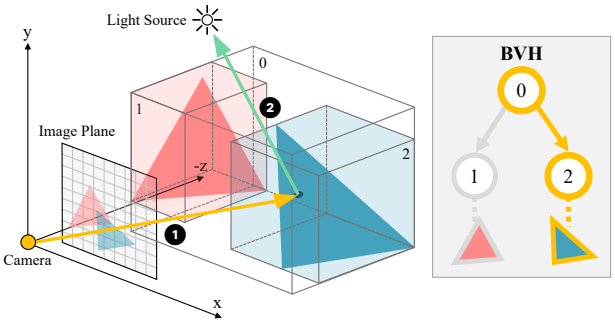


Fig. 1: Overview of the ray tracing process utilizing a BVH

In modern GPUs, real-time ray tracing is enabled by integrating a specialized hardware acceleration unit for ray tracing within each streaming multiprocessor (SM). These ray tracing accelerators handle traversal and intersection tasks, with each thread (representing a ray) maintaining its own traversal stack [8], [33]. Due to limited on-chip storage, these stacks are implemented as short stacks, with a small number of entries stored in on-chip buffers [33]. When the stack overflows during push operations, older entry values are spilled into thread-local off-chip memory. Conversely, during pop operations, the most recently spilled values are reloaded into the on-chip stack to maintain consistency. This stack management process introduces substantial memory traffic, reducing overall ray tracing performance [6], [7], [40].

One straightforward approach to mitigate these traffic issues is to increase the on-chip stack size; however, it is too costly and may result in underutilization of the stack during traversal. Alternatively, increasing the L1 data cache (L1D) size can help mitigate the performance loss from stack overflows; however, based on our initial evaluation, its effectiveness is limited by the incoherent ray traversal paths, which may incur frequent cache evictions. In this paper, we explore a cost-effective yet powerful solution: reconfiguring a portion of L1D as shared memory to manage stack overflows. Leveraging this approach, we propose SMS architecture for ray tracing acceleration that introduces a two-level hierarchical on-chip memory traversal stack. We employ a shared memory stack (SH stack) as a

[*]Yunho Oh and Myung Kuk Yoon are co-corresponding authors.

2766-0486/25/$31.00 ©2025 IEEE
DOI 10.1109/ISPASS64960.2025.00016

secondary stack, complementing the primary on-chip stack.

To further enhance performance, two optimization strategies are introduced. First, we propose a skewed access pattern in shared memory banks. This strategy minimizes bank conflicts by interleaving initial bank accesses across threads, thereby effectively utilizing shared memory. Second, we propose intra-warp reallocation of SH stacks, based on our insight that threads require varying stack depths during ray traversal and complete their traversals at different times. By reallocating unused SH stacks from early finished threads to threads within the same warp that need additional stack entries, this approach effectively prevents stack overflows into off-chip memory.

We evaluate SMS architecture using Vulkan-Sim [33], a cycle-level GPU simulator with ray tracing acceleration units. The evaluation is conducted on ray tracing benchmark scenes from Lumibench [27]. Our evaluation demonstrates that SMS architecture delivers a significant performance improvement of 23.2% compared to the baseline GPU, which employs an 8-entry on-chip stack per thread. Furthermore, the proposed architecture exhibits a strong alignment with the performance of a full per-ray stack stored in on-chip memory, which is impractical for actual hardware implementation. By effectively balancing the use of L1D, shared memory, and off-chip memory, SMS efficiently manages stacks during ray traversal, achieving significant improvements in overall performance.

Overall, this paper makes the following contributions:

- We identify off-chip memory traffic from traversal stack management as a critical performance bottleneck in GPU-based ray tracing.
- We propose a two-level on-chip stack architecture that leverages shared memory to reduce off-chip stack traffic.
- We introduce two optimization techniques—skewed bank access and intra-warp reallocation—to enhance the efficiency of shared memory usage.
- We provide a quantitative evaluation demonstrating that the proposed design achieves performance close to that of a full on-chip stack implementation.

II. Background

A. Ray Tracing

Ray tracing is a popular rendering technique that generates photorealistic images. Fig. 1 illustrates a simplified ray tracing process. A primary ray is cast from the camera through each pixel location on the image plane (❶), determining the pixel color by tracing its interactions with objects (object primitives). When the ray intersects a primitive, a secondary or shadow ray is cast from the intersection point toward a light source to check for obstructions (❷). If the shadow ray is blocked by another object, the pixel is rendered in shadow; otherwise, it is illuminated with the intersected primitive's color. To achieve realistic visual effects, path tracing [21], a fully ray-traced rendering technique, is employed. Path tracing generates secondary rays in random directions at each intersection point, simulating light bounces throughout the scene. These rays are traced recursively until they either hit

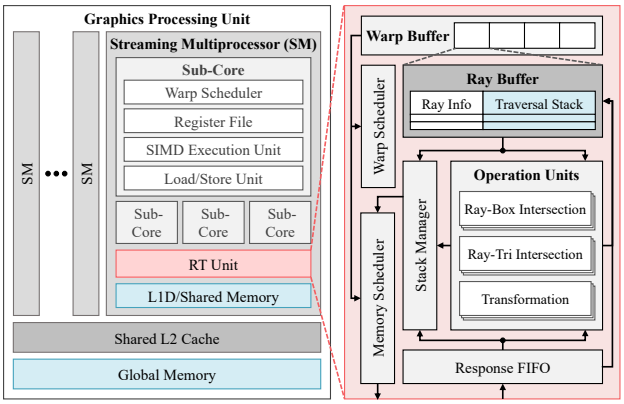


Fig. 2: Baseline GPU architecture integrating the RT unit

a light source or reach the maximum recursion depth. To improve accuracy and reduce noise, multiple rays are cast per pixel, and their results are averaged to compute the final color. While increasing the number of samples per pixel (SPP) enhances image quality, it also raises computational costs, creating a trade-off between quality and rendering time.

To optimize performance, ray tracing uses acceleration structures, most commonly BVHs [32]. A BVH organizes scene primitives into a hierarchical tree of bounding volumes, typically AABBs (Fig. 1), and is usually traversed via depth-first search using a stack. Each stack entry stores the memory address of a BVH node, which is used to load the corresponding node data from global memory. The algorithm begins at the root (an internal node), where ray-AABB intersection tests are performed for all child nodes. The closest intersected child is visited next, while the address of the other hit child is pushed onto the stack for later traversal. When the ray reaches a leaf node, ray-triangle intersections are tested. Traversal continues by popping the next node address from the stack and repeating the process until the stack is empty. This approach reduces computational costs by replacing costly ray-triangle intersection tests with faster ray-AABB intersection tests [29].

B. GPU Architecture and Ray Tracing Acceleration Units

Fig. 2 shows the baseline GPU architecture integrating ray tracing acceleration units referred to as RT units. GPUs are highly parallel processors that execute tasks as threads, with each thread serving as an independent unit of computation. A GPU consists of multiple SMs, each containing several sub-cores [2]. Each sub-core comprises a warp scheduler, a register file, a SIMD execution unit, and a load/store unit. These sub-cores execute threads organized into warps, which are groups of 32 concurrent threads processed in a single instruction multiple thread (SIMT) manner. Each SM contains on-chip memory, such as a unified L1 cache and shared memory, shared by its sub-cores. Beyond the SM level, all SMs share access to an off-chip L2 cache and global memory.

Modern GPUs incorporate a dedicated RT unit within each SM, termed the RT Core in NVIDIA GPUs [1], [2], [4]. RT units accelerate two key tasks in ray tracing: (1) pointer chasing during BVH traversal and (2) intersection tests between

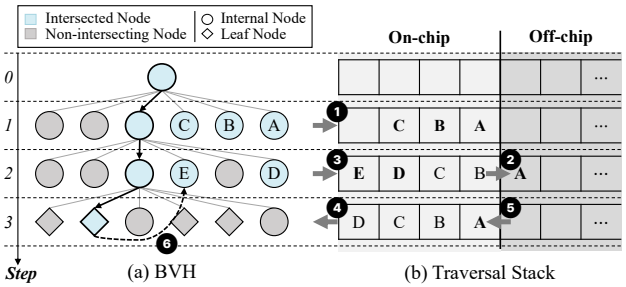


Fig. 3: Example of BVH6 traversal using 4-entry short stack

rays and scene geometry. The RT unit retrieves BVH node data from memory, decodes geometry information, and performs intersection tests. The results are then used by general-purpose computation units (*i.e.,* SIMD execution units), to calculate pixel colors during the shading process. Offloading these traversal and intersection tasks to the RT unit allows other operations, such as shading, to execute concurrently [4].

A warp issuing a trace ray instruction is forwarded to the RT unit during the execute stage of the pipeline. It is held in the warp buffer until all 32 threads are processed. Each warp maintains a ray buffer, which holds a traversal stack and ray information—such as ray ID, ray properties, and current ray status—for every thread, with each thread mapped to a ray. The traversal stack is maintained as a short stack with 8 entries per ray [33]. To manage stack overflows during traversal, the stack manager generates global memory requests to spill or reload stack entry values between the ray buffer and off-chip memory, which are then issued by the memory scheduler.

The warp scheduler in the RT unit follows a greedy-then-oldest (GTO) policy, prioritizing the same warp until a stall occurs. For the scheduled warp, the top entry of each thread's stack is read to retrieve the next memory address. Memory requests to load the corresponding node data are then collected across all 32 threads. After the data is returned to the response FIFO, it is forwarded to the corresponding operation unit along with the ray properties. The operation units include: (1) ray-box intersection units, (2) ray-triangle intersection units, and (3) object-to-world and world-to-object transformation units.

Once the BVH operation is complete, the results update the ray status, and the traversal stacks for the requested rays are popped. During the stack pops, the stack manager checks for spilled addresses and generates memory requests to reload them. New addresses for the intersected child nodes are then pushed onto the stack. Before pushing, the stack manager checks for stack overflows. If no overflow occurs, the addresses are pushed onto the stack. Otherwise, the stack manager generates memory requests to free up space.

C. Memory Traffic in Traversal Stack Management

The traversal stack for BVH traversal is often implemented as a short stack with a limited number of entries [25], [30], [33]. When the stack overflows, older values are spilled to off-chip local memory to free up space [33]. When the spilled stack is popped, the most recently spilled entry value is reloaded into on-chip memory to preserve stack consistency.

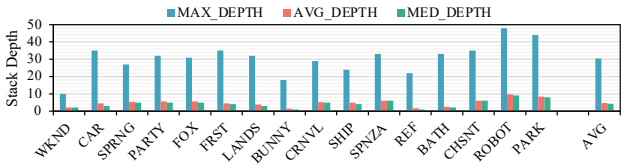


Fig. 4: Summary of stack depths for each workload

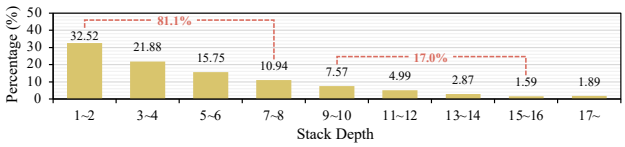


Fig. 5: Average stack depth distribution across all workloads

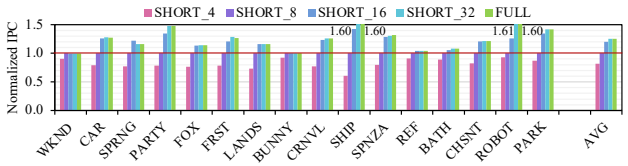
A wide BVH (BVHk) is a variant where each internal node can have up to k children, resulting in a higher branching factor [15], [36], [38]. Fig. 3 shows a BVH6 traversal using a 4-entry stack. Starting from the root, ray-AABB intersections are tested against all its six child nodes. The closest hit child is then visited, and the addresses of the remaining intersected children (A, B, C) are pushed onto the stack (❶). At step 2, after pushing one node address (D), the stack becomes full. Before pushing the remaining child address E, the oldest entry value (A) must be spilled to off-chip memory (❷) via a global memory store request, freeing space in the stack for E (❸). After visiting a leaf node, the traversal proceeds by popping the next node address (E) from the stack (❹), triggering a global memory load request to reload the most recently spilled address (A) from off-chip memory (❺). Traversal continues with the popped node, backtracking through the tree (❻).

Although this strategy enables continuous ray traversal with smaller on-chip stack entries, the frequent off-chip memory accesses for stack maintenance introduce significant memory traffic [6], [7], [40]. In off-chip memory, spilled entry values are thread-specific, which prevents efficient coalescing of global memory accesses during stack overflows, especially as rays follow divergent paths. As a result, traversal stack traffic significantly increases memory bandwidth usage, in addition to the traffic for scene geometry loads. This issue becomes more pronounced, particularly with smaller on-chip stack sizes [7].

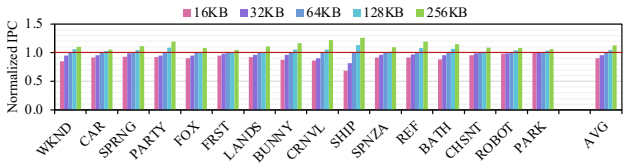
III. Motivation

A. Impact of On-chip Traversal Stack Size

The traversal stack traffic caused by stack overflows significantly impacts performance. Increasing the stack size can help mitigate this issue. To determine the optimal stack size, we measure stack usage during ray traversal across 16 ray tracing workloads (see §VII-A for a detailed evaluation methodology). Fig. 4 shows the maximum, average, and median stack depths observed during traversal. While the overall average and median depths range between 4 and 5, the maximum depth reaches around 30. This gap highlights the challenge of selecting an optimal stack size. While *over-allocation* of stack entries causes underutilized on-chip storage, *under-allocation* leads to frequent off-chip memory accesses for rays with long traversal lengths, thereby degrading overall performance.



(a) IPC with different traversal stack sizes



(b) IPC with different L1 data cache sizes

Fig. 6: IPC with varying stack and L1 cache configurations

We further examine the impact of traversal stack sizes on performance. Fig.6a shows the instructions per cycle (IPC) under varying stack configurations, normalized to a baseline configuration using an 8-entry stack. Reducing the stack size to 4 entries leads to an 18.4% IPC drop, while increasing it to 16 and 32 entries results in IPC improvements of 19.9% and 25.2%, respectively. However, performance gains beyond 32 entries are marginal. Based on the results, we conclude that *although the average required stack depth is only 5 entries, overflows at higher depths significantly degrade performance due to increased off-chip memory traffic*.

To better understand stack usage throughout traversal, we refer to Fig. 5, which presents the average distribution of stack depths across all workloads. For each workload, the stack depth is recorded at every push and pop operation across all rays. Although the maximum required depth reaches 30, increasing the stack size beyond 16 entries is not cost-effective, as only 1.9% of traversal steps exceed this size. On the other hand, frequent stack spills—occurring in 17.0% of traversal steps that require 9 to 16 entries—explain the performance gains observed when increasing the stack size to 16.

Therefore, increasing the stack size beyond the 8-entry baseline is essential for improving overall performance. However, expanding the on-chip stack size comes with significant hardware overhead. For example, allocating an 8-entry stack per thread requires 8KB (8B$\times$8-entry$\times$128-thread) of storage per SM. Doubling the stack to 16 entries adds another 8KB, which accounts for nearly 25% of the GPU's register file size—already known as one of the most power-hungry components in modern GPUs [22], [26].

Likely due to such hardware constraints, prior works have explored stackless traversal methods [6], [9], [12], [18], [24] or stack-based approaches with minimal on-chip stacks [24], [35], often at the cost of additional computational overhead. Other works aim to reduce traffic from accessing fully off-chip traversal stacks by utilizing on-chip memory resources. For example, one study [7] introduces a stack-top cache in the register file to improve access efficiency. Another work [40] uses shared memory to store the first few entries of an off-chip stack, designed specifically for compressed BVHs and compact stack formats. These constraints highlight that *effective traversal stack management is crucial for reducing latency caused by insufficient on-chip stack capacity.*

B. Impact of L1 Data Cache Size

Given the challenges of limited on-chip stack capacity, increasing the size of the L1D—the nearest storage in the memory hierarchy—can support stack management. Although not a direct substitute for a larger on-chip stack, a larger L1D can help mitigate performance degradation by enabling faster access to stack entries, as long as they remain in the cache and are not replaced by other memory operations. Beyond aiding stack management, a larger L1D also reduces access latency for scene geometry data, further contributing to traversal performance. Fig. 6b presents IPC results across varying L1D configurations, normalized to a 64KB baseline. Expanding the L1D to 128KB improves performance by 4.5%, while a 256KB configuration results in a 12.6% gain. Conversely, reducing the L1D to 32KB and 16KB leads to performance drops of 4.5% and 9.6%, respectively.

While increasing the L1D size improves performance, its effectiveness is relatively limited compared to expanding the on-chip stack size. For example, doubling the stack size from 8 to 16 entries (*i.e.,* an 8KB increase) yields a 7.3 higher percentage points (PP) than increasing the L1D size from 64KB to 256KB (*i.e.,* a 192KB increase). This is due to the *incoherence of rays* in global illumination models [7], where rays tend to follow different traversal paths through the BVH and scene, leading to divergent memory access patterns. Such divergence causes frequent cache evictions, increasing latency when accessing both spilled stack entries and scene geometry stored in global memory. As scene complexity continues to grow, memory traffic often exceeds the L1D capacity [28], further limiting its effectiveness. Thus, *although increasing the L1D size improves performance, its benefits are less significant compared to expanding the on-chip stack size, primarily due to the divergent traversal behavior of incoherent rays.*

C. Exploring Cost-effective yet Powerful Solution for Traversal Stack Management

To summarize, managing traversal stacks efficiently is a significant challenge for achieving high ray tracing performance. While increasing the on-chip stack size may appear to be a straightforward solution to maximize performance, it comes with drawbacks, such as higher energy consumption and the risk of underutilizing stack resources [14], [16]. A more practical alternative is to leverage the flexibility of modern GPUs equipped with RT units, which allow the L1D to be reconfigured as programmable or shared memory. Our observation (Fig. 6b) further suggests that reducing the L1D size to make room for shared memory results in only a minor performance loss. In the following section, we demonstrate that dedicating shared memory exclusively to stack management can offer significant performance gains, outweighing the small loss from reducing the L1D size.

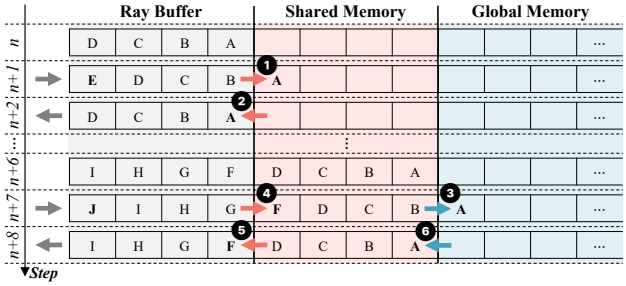


Fig. 7: Memory transactions for traversal stack management in the proposed SMS architecture

IV. SMS Architecture

In this section, we propose a novel GPU architecture for ray tracing acceleration—named SMS—designed around ***S**hared **M**emory **S**tack management*. SMS architecture introduces a hierarchical traversal stack design, comprising two levels of on-chip storage: (1) the stack within the ray buffer serves as the *primary stack*, holding the most recently pushed node addresses; (2) when the primary stack reaches its capacity, shared memory is used as a *secondary stack*, storing node addresses spilled from the primary stack. In the following subsections, we first provide a conceptual overview of the SMS architecture and a detailed explanation of its traversal stack management. Then, we present the effectiveness of using SH stacks during ray traversal.

A. Hierarchical Traversal Stack Management

Fig. 7 illustrates how traversal stacks are managed in the proposed SMS architecture, highlighting the memory transactions involved in handling stack overflows. In the baseline architecture, stack entry values are directly spilled and reloaded between the primary ray buffer stack (RB stack) and global memory. However, to mitigate off-chip memory traffic, SMS architecture introduces a two-level on-chip stack design using shared memory. When the primary RB stack overflows, the oldest entry value is spilled to the secondary SH stack (❶). During a stack pop, the most recently spilled address is retrieved from the SH stack and placed back into the RB stack (❷). If the SH stack also becomes full, global memory is accessed; the oldest node address in shared memory is moved to global memory (❸), freeing space for the spilled address from the RB stack (❹). When a pop operation involves entries that have been spilled into global memory, the newest address in the SH stack is transferred back to the RB stack (❺), followed by reloading the spilled address from global memory into shared memory (❻). In this manner, SMS architecture holds recent node addresses in faster, on-chip storage, while older addresses migrate to slower, off-chip global memory. By significantly reducing off-chip traffic, this approach offers a robust solution for managing ray traversal in modern GPUs.

As the RB stack is dedicated to each thread (or ray), SMS also assigns a distinct set of stack entries in shared memory for each thread. This SH stack is implemented as a circular queue, with the detailed implementation presented in §VI.

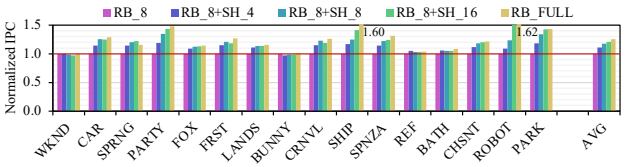


Fig. 8: IPC improvements with different L1D/Shared memory configurations

While traversal stacks are maintained at the individual thread level, memory accesses for spilled entry values are processed at the warp level within the RT unit. Similar to how global memory accesses are handled in the baseline architecture, the memory scheduler collects and issues shared memory accesses across all 32 threads within a scheduled warp. To ensure that the requested data is returned to the corresponding thread, the stack manager also tracks the associated threads for shared memory load accesses.

B. Effectiveness of Shared Memory Traversal Stack

To demonstrate the effectiveness of using the SH stack, we measure the performance improvements of the SMS architecture. Fig. 8 presents IPC results normalized to a baseline configuration with an 8-entry RB stack (`RB_8`). Here, `RB_`*N* and `SH_`*M* denote an *N*-entry RB stack and an *M*-entry SH stack, respectively. Note that the shared memory size is configured to match the capacity of the SH stack; thus, increasing the SH stack size proportionally reduces the available L1D capacity within the unified memory. Using a full RB stack (`RB_FULL`) results in a 25.3% performance improvement over the baseline. In comparison, adding a 4-entry SH stack to the baseline (`RB_8+SH_4`) yields an 11.0% gain, while expanding the SH stack to 8 (`RB_8+SH_8`) and 16 (`RB_8+SH_16`) entries achieves 17.4% and 21.2% improvements, respectively.

However, these improvements come at the cost of increased shared memory usage. As illustrated in Fig. 5, most stack depth requirements fall within the range of 1 to 16 entries. Therefore, a configuration combining an 8-entry RB stack with an 8-entry SH stack effectively covers the majority of cases. Based on these observations, we propose using an 8-entry SH stack per ray, along with a 56KB L1D and 8KB of shared memory in our SMS architecture. This configuration provides a good balance between performance and resource utilization, delivering significant performance gains while keeping the on-chip ray buffer relatively small.

V. Optimization of SMS

To further enhance the efficiency of the SMS architecture, we propose two optimization strategies, each driven by specific observations. First, significant shared memory bank conflicts arise from regular access patterns across threads. To mitigate this, we introduce a skewed access pattern to better distribute memory usage across banks. Second, ray incoherence can lead to underutilization of SH stacks, particularly when some rays finish traversal earlier than others. To address this, we suggest dynamically reallocating SH stacks to active threads within

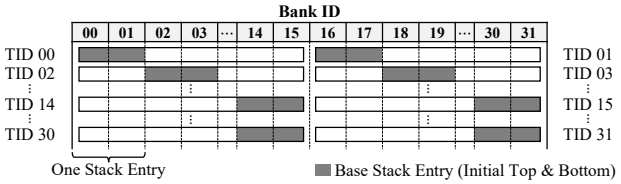


Fig. 9: Shared memory bank assignment across threads for `SH_8` and base entry mapping with the skewed access strategy

the same warp. Together, these strategies leverage the unique characteristics of ray tracing workloads to optimize shared memory usage and improve overall performance.

A. Skewed Bank Access

Observation. Unlike the primary traversal stack in the ray buffer, accessing entries in the SH stack follows the constraints of shared memory architecture. Shared memory is divided into equally sized memory banks, allowing for parallel processing of memory load or store operations [2], [3], [5]. When threads access addresses that map to different banks, these operations can proceed in parallel. However, if multiple threads access different words within the same bank, bank conflicts occur, leading to serialized accesses. This serialization reduces the efficiency of shared memory bandwidth and delays subsequent memory accesses during ray traversal, further exacerbating performance penalties. In our initial design, each thread is assigned a fixed number of SH stack entries and begins accessing them from entry index 0. However, severe bank conflicts arise as threads often access the same stack entry index that maps to the same shared memory bank—particularly under the 8-entry SH stack configuration.

Fig. 9 illustrates an example where each thread is assigned an 8-entry SH stack. Each 8-byte stack entry spans two adjacent banks, meaning a thread with an 8-entry stack accesses 16 adjacent banks. With 32 shared memory banks, threads with even indices (0, 2, ..., 30) access the first 16 banks, while threads with odd indices (1, 3, ..., 31) access the remaining 16 banks. When all threads access their SH stacks starting from entry 0, the likelihood of bank conflicts increases, as threads often access their stacks in a similar circular order during traversal. This regularity leads to unbalanced bank accesses and severe bank conflicts, ultimately degrading performance.

Solution. To address this issue, we propose a skewed access pattern for shared memory banks across threads within a warp. By introducing irregularity in how threads access their stack entries, we can balance shared memory bank utilization and reduce conflicts. Specifically, we offset the base stack entry—the initial entry accessed when the stack is empty—for each thread based on its thread ID and the SH stack size. For example, with an 8-entry SH stack per thread (Fig. 9), threads 0 and 16 start with entry 0 (bank 0–1), while threads 2 and 18 start with entry 1 (bank 2–3). Similarly, threads 1 and 17 begin with entry 0 (bank 16–17), and threads 3 and 19 begin with entry 1 (bank 18–19). Starting from their respective base stack entries, subsequent entries are accessed in a circular manner.

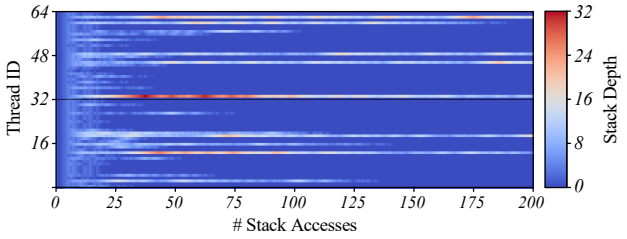


Fig. 10: Traversal stack depths across threads for PARTY

This approach reduces bank conflicts by more evenly distributing bank accesses, minimizing latency in shared memory transactions. Furthermore, it is scalable, as it ensures consistent performance gains across different stack sizes. By implementing this skewed access pattern, SH stacks are utilized more efficiently, resulting in further performance improvements.

B. Dynamic Intra-Warp Reallocation of Shared Memory Stack

Observation. Assigning an equal number of stack entries to each thread can be less effective due to the varying stack depth requirements of incoherent rays during traversal. This is illustrated in Fig. 10, where the x-axis represents the number of stack accesses (push and pop operations) during traversal, the y-axis corresponds to individual threads, and the color at each point indicates the stack depth of the thread at that access. Although the figure presents traversal stack access patterns for two specific warps in the `PARTY` scene, similar trends are observed across other warps and scenes.

One key observation is that threads complete their traversals at different times. Threads that finish early leave their SH stacks unused until the entire warp completes the traversal. Another observation is that certain threads require larger stack depths, reflecting a higher number of intersected nodes. These threads often need global memory accesses to handle overflows, introducing delays in subsequent stack operations. As a result, stack accesses for these threads are delayed to later cycles in the timing model, while threads with smaller stack depths at the same x-axis position experience fewer delays. This imbalance increases the likelihood that threads with longer traversals will borrow unused SH stacks.

Solution. To address this issue, we propose dynamically reallocating unused SH stacks from threads that terminate early. Initially, each thread is assigned a fixed number of entries in shared memory. However, once a thread's dedicated stack is fully utilized, it can access additional SH stacks released by early-finishing threads, reducing the need to spill entry values to global memory. This reallocation of multiple SH stacks gives the thread the appearance of having an extended SH stack. The strategy is effective because, while most threads initially require a similar number of entries, their stack requirements diverge as the traversal progresses, making static allocation less effective. By leveraging the observed patterns in ray tracing workloads, this approach helps reduce off-chip memory traffic and improves overall performance.

To simplify implementation, we limit SH stack reallocation to threads within the same warp (*intra-warp*), rather than

across different warps (*inter-warp*). Once all 32 threads in a warp complete their traversal, the warp is removed from the warp buffer, allowing the next warp to enter the RT unit. Inter-warp reallocation would involve complex tracking and management of stack ownerships, as threads would need to return borrowed stacks to the newly entered warp. By limiting reallocation to intra-warp, we minimize this overhead and achieve performance improvements.

VI. Implementation

This section outlines the architectural support for SMS. Fig. 11 illustrates the modified RT unit design in the SMS architecture, focusing on the management of SH stacks. The hardware components that have been modified or newly added in our proposed architecture are highlighted in pink. To support SH stack management, we introduce additional fields in the ray buffer and modify the stack manager hardware unit. Furthermore, we incorporate a response FIFO to handle shared memory access requests. The following subsections provide a detailed explanation of these modifications and discuss the overhead associated with the added logic.

A. Shared Memory Stack Management

Extended Ray Buffer. As discussed in §IV-A, we extend the ray buffer to track the state of the SH stack. For each thread, we introduce three fields: `Top`, `Bottom`, and `Overflow`. The `Top` field represents the index of the most recently filled entry in the SH stack, while the `Bottom` field indicates the index of the oldest entry. The `Overflow` field flags whether the SH stack has overflowed, indicating that its entry values have been spilled to global memory. Using these fields, the stack manager unit tracks the status of traversal stacks and generates the appropriate memory requests for spilling or reloading entry values. These memory requests are then passed to the memory scheduler for execution.

SMS Stack Manager. During traversal, the SMS stack manager hardware unit ensures that both the traversal stacks and the associated ray buffer fields are correctly updated. When performing a stack pop operation, the stack manager first checks for overflow conditions in both the RB stack and SH stack using the `Overflow` field. If the RB stack has overflowed, the most recently spilled value is reloaded from the SH stack into the RB stack via a shared memory load operation. The corresponding shared memory address is computed using the `Top` field, which tracks the most recently filled entry. If the SH stack has also overflowed, additional steps are required to reload an entry value from global memory to the SH stack. This involves a global memory load followed by a shared memory store, where the destination address for the store is computed using the `Bottom` field. Note that these memory requests are issued sequentially, with a subsequent request begin issued only after the prior load request has returned to the response FIFO.

A stack push operation occurs when the operation unit returns the address of an intersected child node (*Entry Value* in Fig. 11) for each node visit. Similar to the pop operation, the

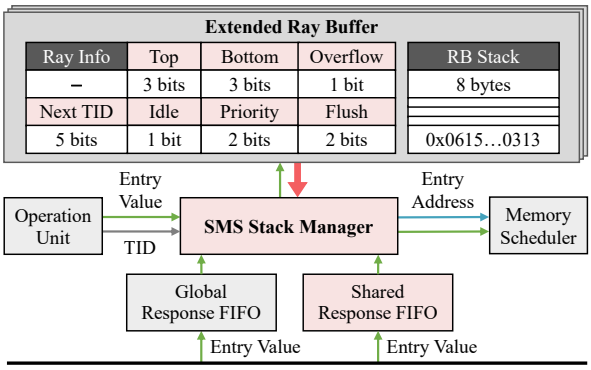


Fig. 11: Extended RT unit design in SMS architecture

stack manager first checks for overflow conditions in both the RB and SH stacks. If the RB stack is full, the oldest pushed address must be spilled into the SH stack before the new address can be added. If the SH stack is also full, the oldest address in the SH stack must be moved to global memory. As a result, when only the RB stack is full, the stack manager generates a shared memory store request. When both stacks are full, the sequence of requests is as follows: a shared memory load, a global memory store, and a shared memory store.

B. Shared Memory Address Generation

Shared memory addresses for load and store operations are computed by the stack manager using the thread's ID (`TID`) along with the corresponding `Top` and `Bottom` fields. Since each thread is initially allocated a specific, static region of shared memory as its SH stack (Fig. 9), `TID` determines the base address of this stack region. The `Top` or `Bottom` field serves as the entry index within the stack: `Top` is used for data transfers between the SH and RB stacks, while `Bottom` is used for transfers between the SH stack and global memory. To enable skewed bank access and intra-warp reallocation, additional steps are required during address generation.

Skewed Bank Access. To balance bank utilization and reduce conflicts in shared memory, threads within each warp initially align their *top* entries based on their thread IDs and the SH stack size (N). The stack manager calculates this starting entry (*Base Stack Entry* in Fig. 9) for each thread's stack using the following formula:

$$\text{Base Entry Index} = \left(\frac{\text{TID}}{k}\right) \bmod N, \; where \; k = \frac{32}{N \times 2}$$

This base index is assigned to both the `Top` and `Bottom` fields, enabling subsequent stack entries to be accessed in a circular manner with more balanced shared memory bank usage. By skewing the initial *top* entries across threads—without requiring modifications to the memory scheduler—shared memory banks are automatically distributed as threads begin filling their stacks from their respective starting points.

Dynamic Intra-Warp Reallocation. When a thread exhausts all entries in its dedicated SH stack, the stack manager assigns an available SH stack from an early-finished thread within the

Fig. 12: Access order of linked SH stacks after reallocations

same warp. To track the availability of SH stacks, the stack manager maintains an `Idle` field for each thread. This field is set to 1 when the the thread's SH stack is available, indicating that the thread has completed its traversal and the stack is not currently being borrowed by another thread. Conversely, if the thread has not completed its traversal or its stack is being borrowed, the field is set to 0.

Our proposed reallocation scheme allows each thread to borrow SH stacks from multiple threads within the same warp. Fig. 12 illustrates how each thread can access its multiple SH stacks, using a simple example with three SH stacks: one dedicated and two borrowed. The most recently allocated stack becomes the *top stack*, which is accessed during push or pop operations until it becomes empty. To track the order of multiple reallocations, the SMS stack manager maintains `Next TID` for each SH stack. This field stores the ID of the next borrowed stack, creating a link to the subsequent stack. By sequentially following these `Next TID` links, the stack manager can navigate through all borrowed stacks and locate the current top stack for a given thread. When the current top stack is emptied, it is immediately released, and the next-most-recently allocated stack in the chain becomes the new top stack and is accessed in the same manner.

Excessive reallocations can degrade performance due to the latency involved in searching for the top stack. To mitigate this, we limit the number of concurrently borrowed stacks per thread to four, based on heuristics derived from our observations (§III-A). This setup provides a total on-chip stack capacity of 48 entries: 40 SH entries (8-entry×5-stack) and 8 RB entries, which covers the maximum stack depth required across all evaluated workloads.

During dynamic reallocation, a situation may arise where a thread attempts to push a new node address, but all borrowed SH stacks are full and no additional stacks are available. To maintain stack consistency—where stacks are filled from the bottom up—the oldest address in shared memory must be spilled to global memory to accommodate a new spill from the RB stack. Since borrowed SH stacks are linked via a field and may not reside contiguously in shared memory, shifting entries across stacks (from top to bottom) is impractical. Instead, an alternative mechanism is introduced: the entire set of entries in the bottom SH stack is flushed to global memory, and the emptied stack is then prompted to the top position, allowing it to hold the most recently spilled values.

While this flushing mechanism effectively simulates access to an additional stack, it is limited to three consecutive flushes per allocated SH stack. In the the worst-case scenario, a thread operates with one dedicated SH stack and one borrowed stack, with no further stacks available. With support for three flushes per stack, the thread can effectively operate with 8 SH stacks—one dedicated, one borrowed, and six simulated via flushing. Together with the RB stack, this setup provides a total stack depth of 72 entries, which we have found sufficient to meet the required depth (*i.e.,* 48) across all evaluated workloads.

To track the number of stack flushes, an extra `Flush` field is introduced for each SH stack or thread. Additionally, a `Priority` field is maintained for each stack to track the allocation order, ensuring correct stack access. Stack flushes are infrequent, as dynamic reallocation is only triggered when threads begin to finish their traversal. At this point, many threads are likely to have completed their work, leaving their stacks unused and reducing contention for borrowing stacks.

C. Hardware Overhead

The proposed design achieves efficient stack management with minimal on-chip memory overhead. As mentioned earlier, each per-thread SH stack requires several additional fields in the ray buffer: `Top`, `Bottom`, and `Overflow` for independent stack management, and `Next TID`, `Idle`, `Priority`, and `Flush` for dynamic intra-warp reallocation. For an SH stack with 2^N entries, the `Top` and `Bottom` fields each require N bits. For example, with a 2^3-entry SH stack, each thread requires 3 bits per field, resulting in 96 bytes (2-field×3-bit×32-thread×4-warp) across 128 threads in the RT unit. Additionally, the `Overflow` and `Idle` fields require 1 bit each, while the `Next TID` field requires 5 bits to identify one of 32 threads within the warp. The `Priority` and `Flush` fields each require 2 bits, as the number of concurrent reallocations is limited to four, while continuous flushes are limited to three. These additional fields contribute a total overhead of 176 bytes (11-bit×32-thread×4-warp) to the on-chip storage per RT unit.

In total, the traversal stack management in the SMS architecture adds 272 bytes of on-chip memory per SM. Notably, this overhead is significantly smaller than increasing the RB stack size by 8 entries, which would require 8KB (8-byte×8-entry×32-thread×4-warp). Thus, we conclude that the SMS architecture offers a cost-effective and efficient solution for traversal stack management.

VII. Evaluation

A. Methodology

We use Vulkan-Sim [33], a cycle-level GPU simulator for ray tracing. The baseline GPU configuration is from the original Vulkan-Sim work, which represents a mobile System-on-Chip (SoC) GPU [33], as detailed in TABLE I. To optimize resource allocation, *shared memory* is set to the minimum required for SH stacks, maximizing the remaining capacity for the L1D. As shown in TABLE II, we evaluate benchmark scenes from Lumibench [27], rendered with the path tracing (PT) shader. Most scenes are simulated at a resolution of 128×128 with 2 SPP, except for `CHSNT`, `ROBOT`, and `PARK`. Due to their long evaluation times, these three more complex scenes are evaluated at a 32×32 resolution with 1 SPP. However, similar results are expected at this reduced scale, as performance trends have been observed to remain consistent across varying workload sizes [13], [27].

TABLE I: Baseline GPU parameters

Component	Parameter	Description
General	# SMs	8
	warp size	32
	warp scheduler	GTO
	# registers per SM	32,768
RT Unit	# RT units per SM	1
	max # warps per RT unit	4
Memory	L1D/shared memory	64KB, fully associative, LRU, 20 cycles
	L2 unified cache	3MB, 16-way associative, LRU, 160 cycles

TABLE II: Benchmark scenes

Scene	# Triangles	BVH (MB)	Scene	# Triangles	BVH (MB)
WKND	0	0.2	CAR	12.7M	1,328.2
SPRNG	1.9M	178.0	PARTY	1.7M	156.1
FOX	1.6M	648.5	FRST	4.2M	380.5
LANDS	3.3M	303.5	BUNNY	144.1K	13.2
CRNVL	449.6K	60.7	SHIP	6.3K	0.5
SPNZA	262.3K	22.8	REF	448.9K	40.4
BATH	423.6K	112.8	CHSNT	313.2K	28.3
ROBOT	20.6M	1,869.0	PARK	6.0M	542.5

Note that our proposed architecture is evaluated under a mobile GPU configuration, reflecting the growing adoption of hardware-accelerated ray tracing in mobile devices for real-time applications. Nonetheless, the architecture is scalable to larger GPUs, as ray tracing workloads typically assign a single ray traversal to each thread. Thus, the per-ray on-chip stack pressure remains a persistent issue across GPU configurations, leading to performance degradation—even on high-end GPUs like the RTX 2060 used in the original Vulkan-Sim work [33]. Consistent with prior work [27], applying our architecture to larger GPUs exhibits similar performance trends.

B. Performance

Fig. 13 shows the IPC improvements achieved by the SMS architecture, normalized to the baseline configuration (`RB_8`). By default, SMS employs an 8-entry SH stack (`SH_8`) and integrates two optimization strategies: skewed bank access (`SK`) and dynamic intra-warp reallocation (`RA`). Introducing the SH stack (`+SH_8`) delivers an average IPC improvement of 15.1% over the baseline. Adding skewed bank access (`+SK`) further enhances performance by 4.3 PP, effectively reducing bank conflicts and distributing accesses more evenly across threads. Finally, incorporating intra-warp reallocation (`+RA`) increases the IPC improvement to 23.2% over the baseline. This performance is comparable to that of the full stack, which achieves an IPC gain of 25.3% over the baseline.

The benefits of SMS vary with scene complexity. Scene complexity—which affects ray traversal length and simulation time—is not determined by a single factor. While scenes with more primitives and larger, deeper BVHs tend to require longer traversals, well-structured BVHs can efficiently prune large portions of the tree [27]. More complex scenes, such as `ROBOT` and `PARK`, typically involve longer traversals and exhibit significant IPC improvements. These gains arise from increased

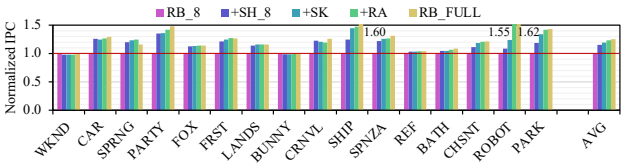


Fig. 13: IPC improvements of SMS architecture

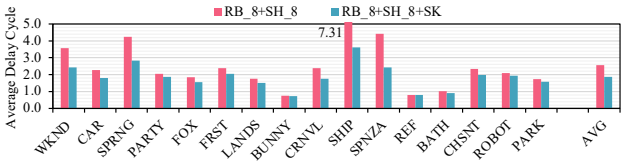
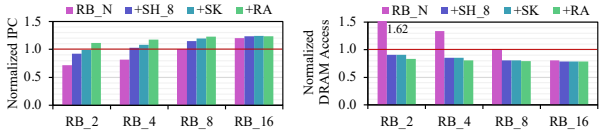


Fig. 14: Effect of skewed bank access



(a) IPC improvements (b) Normalized DRAM accesses

Fig. 15: Impact of primary RB stack sizes

stack depth demands, which allow SMS to effectively utilize the secondary SH stack. In contrast, simpler scenes like `REF` and `BATH` show smaller improvements, as the 8-entry primary stack is generally sufficient to handle most traversal needs. Notably, scenes with unique characteristics—such as `SHIP`, which features long, thin primitives that result in a high ratio of leaf node accesses to total node accesses—achieve substantial performance gains, despite having relatively few primitives and a compact BVH.

C. Effect of Skewed Bank Access

To evaluate the impact of the skewed bank access strategy in SMS, we analyze the average delay cycles caused by bank conflicts during shared memory access scheduling. As shown in Fig. 14, we compare the delay cycles before (`RB_8+SH_8`) and after applying the strategy (`RB_8+SH_8+SK`) across all workloads. On average, the strategy reduces delay cycles by 27.3% compared to using the SH stack alone. This improvement highlights that the strategy more evenly distributes bank accesses across threads, effectively utilizing shared memory bandwidth and thereby enhancing overall performance.

D. Impact of Primary Traversal Stack Sizes

We further evaluate the impact of primary traversal stack (RB stack) sizes on SMS performance, as shown in Fig. 15a. All results are normalized to the baseline configuration with an 8-entry RB stack (`RB_8`). Additionally, off-chip memory access counts are presented in Fig. 15b. As discussed in §III, reducing the RB stack size leads to performance degradation due to increased off-chip traffic. For example, with a 2-entry RB stack (`RB_2`), performance drops by 28.3%, while off-chip memory accesses rise by 62.3% compared to the

baseline (RB_8). By introducing SMS—which incorporates a secondary stack in shared memory along with two optimization strategies—this performance loss is effectively mitigated. Compared to using only the primary stack (RB_2), SMS improves performance by 39.7 PP and reduces off-chip memory accesses by 79.2 PP. A similar improvement is observed with the 4-entry RB stack (RB_4) when combined with SMS. Remarkably, even with reduced RB stack sizes (2 or 4 entries), SMS outperforms the baseline configuration featuring an 8-entry RB stack. This highlights that integrating the proposed SMS architecture enables the use of smaller RB stacks, offering an alternative for hardware implementation that reduces on-chip storage—one of the most power-hungry components in GPUs, as widely recognized in prior studies [22], [26]. In contrast, with a larger 16-entry RB stack (RB_16), the performance gain from SMS is more modest, at 3.5 PP. This limited improvement is primarily due to the already low off-chip memory traffic achieved with a larger primary stack. Note that increasing the RB stack is not a practical solution, as it incurs substantial hardware cost and energy consumption associated with scaling on-chip storage [14], [16], [22], [26].

VIII. Related Work

A. Short Stack or Stackless BVH Traversal

Prior works have explored short stack and stackless approaches to BVH traversal. Laine [24] introduced the restart trail for binary BVHs, enabling both stackless and short stack traversal via restarts by storing a single bit of data per hierarchy level. Vaidyanathan *et al.* [35] extended this method to wide BVHs by using a short stack along with an array of counters—one for each tree level—to track the number of previously traversed children. Hapala *et al.* [18] proposed a stackless traversal method that uses parent pointers for backtracking with simple state logic to determine the next node to traverse. Barringer and Akenine-Möller [9] built upon this idea by introducing two stackless traversal algorithms for binary BVHs that eliminate the need to reevaluate the child traversal order. Áfra and SzirmayKalos [6] further extended this stackless approach to wide BVHs. Binder and Keller [12] introduced efficient stackless traversal method with constant-time backtracking, designed specifically for GPUs. These approaches handle traversal while removing off-chip memory traffic for stack management, replacing them with additional traversal steps as some nodes may be revisited. Such methods can be applied orthogonally to our work. By providing additional stack entries in shared memory, restarts or backtracking can be minimized, occurring only when the secondary stack overflows. This reduction in computational overhead can lead to further performance improvements.

B. Ray Grouping with Group-Local Traversal Stack

Ray grouping involves clustering rays with similar paths, allowing them to follow a common traversal path through the BVH using a single traversal stack per group. Wald *et al.* [39] proposed a packet-frustum BVH traversal scheme that supports both static and deformable scenes. Günther *et al.* [17] introduced a parallel packet traversal algorithm for GPU ray tracing. Benthin and Wald [10] proposed the simultaneous tracing of multiple frusta using SIMD units, enabling real-time soft shadow rendering. These approaches effectively reduce the total number of traversal stack entries and minimize stack traffic by sharing a common stack among rays within each group. Although effective for coherent primary rays, these methods often struggle with incoherent ray types, which take divergent paths and may result in unnecessary node visits. Such methods can be applied orthogonally to our work. Since shared memory is accessible by threads within a warp, a per-group traversal stack can be allocated in shared memory and efficiently utilized with our optimization strategies.

C. Shared Memory Utilization in GPGPU

Prior works have explored various methods to improve GPU performance and occupancy by utilizing shared memory. Hayes and Zhang [20] proposed a technique that leverages shared memory for register spilling, significantly reducing the reliance on off-chip memory. Sakdhnagool *et al.* [34] improved upon this by addressing inefficiencies caused by overly conservative register allocation. Yoon *et al.* [41] introduced a technique that exploits underutilized portions of the register file and shared memory to enable fast context switching, thereby increasing thread-level parallelism. Building on this idea, Oh *et al.* [31] proposed a method employing fine-grained memory management to further enhance the efficiency of resource utilization. While these methods aim to optimize the performance of general-purpose GPU applications, our work focuses specifically on shared memory management for ray tracing workloads. This involves addressing the unique memory access patterns associated with traversal stack management in modern GPU-based ray tracing.

IX. Conclusion

This paper proposes SMS, a powerful and cost-effective GPU architecture that efficiently manages traversal stack for ray tracing acceleration. SMS incorporates a secondary SH stack as an alternative solution to enlarging the primary stack in the ray buffer, effectively mitigating off-chip memory traffic caused by stack overflows. To further optimize SH stack utilization, we present skewed bank access and dynamic intra-warp reallocation strategies. Based on our evaluation, SMS improves performance by 23.2% over the baseline that relies solely on the primary stack. This performance is comparable to a design that maintains a full per-ray stack in on-chip memory, demonstrating that the proposed architecture achieves substantial performance gains with minimal hardware overhead.

Acknowledgements

This work was supported in part by the National Research Foundation of Korea (NRF) funded by the Ministry of Science and ICT (MSIT) of the Government of South Korea (NRF-2021R1C1C1012172), by the Ministry of Education of the

Republic of Korea and the National Research Foundation of Korea under Grant NRF-2022R1C1C1011021, by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (RS-2024-00404972, Development of 5G-A vRAN Research Platform), and by the NRF grant funded by the Korea government (MSIT) (RS-2025-00553645).

REFERENCES

[1] "Nvidia ada gpu architecture," https://images.nvidia.com/aem-dam/Solutions/Data-Center/l4/nvidia-ada-gpu-architecture-whitepaper-v2.1.pdf, (Accessed on 11/18/2024).

[2] "Nvidia ampere ga102 gpu architecture," https://www.nvidia.com/content/PDF/nvidia-ampere-ga-102-gpu-architecture-whitepaper-v2.pdf, (Accessed on 11/18/2024).

[3] "Nvidia hopper tuning guide," https://docs.nvidia.com/cuda/hopper-tuning-guide/index.html#unified-shared-memory-l1-texture-cache, (Accessed on 11/30/2024).

[4] "Nvidia turing gpu architecture," https://images.nvidia.com/aem-dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf, (Accessed on 11/18/2024).

[5] "Nvidia volta tuninig guide," https://docs.nvidia.com/cuda/volta-tuning-guide/index.html#unified-shared-memory-l1-texture-cache, (Accessed on 11/30/2024).

[6] A. T. Áfra and L. Szirmay-Kalos, "Stackless multi-bvh traversal for cpu, mic and gpu ray tracing," in *Computer Graphics Forum*, vol. 33, no. 1. Wiley Online Library, 2014, pp. 129–140.

[7] T. Aila and T. Karras, "Architecture considerations for tracing incoherent rays," in *Proceedings of the Conference on High Performance Graphics*, 2010, pp. 113–122.

[8] T. Aila and S. Laine, "Understanding the efficiency of ray traversal on gpus," in *Proceedings of the conference on high performance graphics 2009*, 2009, pp. 145–149.

[9] R. Barringer and T. Akenine-Möller, "Dynamic stackless binary tree traversal," *Journal of Computer Graphics Techniques*, vol. 2, no. 1, pp. 38–49, 2013.

[10] C. Benthin and I. Wald, "Efficient ray traced soft shadows using multi-frusta tracing," in *Proceedings of the Conference on High Performance Graphics 2009*, 2009, pp. 135–144.

[11] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Communications of the ACM*, vol. 18, no. 9, pp. 509–517, 1975.

[12] N. Binder and A. Keller, "Efficient stackless hierarchy traversal on gpus with backtracking in constant time." in *High Performance Graphics*, 2016, pp. 41–50.

[13] Y. H. Chou, T. Nowicki, and T. M. Aamodt, "Treelet prefetching for ray tracing," in *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, 2023, pp. 742–755.

[14] C. Duan, A. J. Gotterba, M. E. Sinangil, and A. P. Chandrakasan, "Energy-efficient reconfigurable sram: Reducing read power through data statistics," *IEEE Journal of Solid-State Circuits*, vol. 52, no. 10, pp. 2703–2711, 2017.

[15] M. Ernst and G. Greiner, "Multi bounding volume hierarchies," in *2008 IEEE Symposium on Interactive Ray Tracing*. IEEE, 2008, pp. 35–40.

[16] A. Garg and T. T.-H. Kim, "Sram array structures for energy efficiency enhancement," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 60, no. 6, pp. 351–355, 2013.

[17] J. Günther, S. Popov, H.-P. Seidel, and P. Slusallek, "Realtime ray tracing on gpu with bvh-based packet traversal," in *2007 IEEE Symposium on Interactive Ray Tracing*. IEEE, 2007, pp. 113–118.

[18] M. Hapala, T. Davidovič, I. Wald, V. Havran, and P. Slusallek, "Efficient stack-less bvh traversal for ray tracing," in *Proceedings of the 27th Spring Conference on Computer Graphics*, 2011, pp. 7–12.

[19] V. Havran, R. Herzog, and H.-P. Seidel, "On the fast construction of spatial hierarchies for ray tracing," in *2006 IEEE Symposium on Interactive Ray Tracing*. IEEE, 2006, pp. 71–80.

[20] A. B. Hayes and E. Z. Zhang, "Unified on-chip memory allocation for simt architecture," in *Proceedings of the 28th ACM international conference on Supercomputing*, 2014, pp. 293–302.

[21] J. T. Kajiya, "The rendering equation," in *Proceedings of the 13th annual conference on Computer graphics and interactive techniques*, 1986, pp. 143–150.

[22] V. Kandiah, S. Peverelle, M. Khairy, J. Pan, A. Manjunath, T. G. Rogers, T. M. Aamodt, and N. Hardavellas, "Accelwattch: A power modeling framework for modern gpus," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021.

[23] D. Kopta, K. Shkurko, J. Spjut, E. Brunvand, and A. Davis, "Memory considerations for low energy ray tracing," in *Computer Graphics Forum*, vol. 34, no. 1. Wiley Online Library, 2015, pp. 47–59.

[24] S. Laine, "Restart trail for stackless bvh traversal," in *Proceedings of the Conference on High Performance Graphics*. Citeseer, 2010, pp. 107–111.

[25] W.-J. Lee, Y. Shin, J. Lee, J.-W. Kim, J.-H. Nah, S. Jung, S. Lee, H.-S. Park, and T.-D. Han, "Sgrt: A mobile gpu architecture for real-time ray tracing," in *Proceedings of the 5th high-performance graphics conference*, 2013, pp. 109–119.

[26] J. Lim, N. B. Lakshminarayana, H. Kim, W. Song, S. Yalamanchili, and W. Sung, "Power modeling for gpu architectures using mcpat," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 2014.

[27] L. Liu, M. Saed, Y. H. Chou, D. Grigoryan, T. Nowicki, and T. M. Aamodt, "Lumibench: A benchmark suite for hardware ray tracing," in *2023 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2023, pp. 1–14.

[28] D. Meister, J. Boksansky, M. Guthe, and J. Bittner, "On ray reordering techniques for faster gpu ray tracing," in *Symposium on Interactive 3D Graphics and Games*, 2020, pp. 1–9.

[29] D. Meister, S. Ogaki, C. Benthin, M. J. Doyle, M. Guthe, and J. Bittner, "A survey on bounding volume hierarchies for ray tracing," in *Computer Graphics Forum*, vol. 40, no. 2. Wiley Online Library, 2021, pp. 683–712.

[30] J.-H. Nah, J.-S. Park, C. Park, J.-W. Kim, Y.-H. Jung, W.-C. Park, and T.-D. Han, "T&i engine: Traversal and intersection engine for hardware accelerated ray tracing," in *Proceedings of the 2011 SIGGRAPH Asia Conference*, 2011, pp. 1–10.

[31] Y. Oh, M. K. Yoon, W. J. Song, and W. W. Ro, "Finereg: Fine-grained register file management for augmenting gpu throughput," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2018, pp. 364–376.

[32] S. M. Rubin and T. Whitted, "A 3-dimensional representation for fast rendering of complex scenes," in *Proceedings of the 7th annual conference on Computer graphics and interactive techniques*, 1980, pp. 110–116.

[33] M. Saed, Y. H. Chou, L. Liu, T. Nowicki, and T. M. Aamodt, "Vulkan-sim: A gpu architecture simulator for ray tracing," in *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2022, pp. 263–281.

[34] P. Sakdhnagool, A. Sabne, and R. Eigenmann, "Regdem: Increasing gpu performance via shared memory register spilling," *arXiv preprint arXiv:1907.02894*, 2019.

[35] K. Vaidyanathan, S. Woop, and C. Benthin, "Wide bvh traversal with a short stack," in *Proceedings of the Conference on High-Performance Graphics*, 2019, pp. 15–19.

[36] T. Viitanen, M. Koskela, P. Jääskeläinen, and J. Takala, "Multi bounding volume hierarchies for ray tracing pipelines," in *SIGGRAPH ASIA 2016 Technical Briefs*, 2016, pp. 1–4.

[37] C. Wächter and A. Keller, "Instant ray tracing: The bounding interval hierarchy." *Rendering Techniques*, vol. 2006, no. 139-149, p. 130, 2006.

[38] I. Wald, C. Benthin, and S. Boulos, "Getting rid of packets-efficient simd single-ray traversal using multi-branching bvhs," in *2008 IEEE Symposium on Interactive Ray Tracing*. IEEE, 2008, pp. 49–57.

[39] I. Wald, S. Boulos, and P. Shirley, "Ray tracing deformable scenes using dynamic bounding volume hierarchies," *ACM Transactions on Graphics (TOG)*, vol. 26, no. 1, pp. 6–es, 2007.

[40] H. Ylitie, T. Karras, and S. Laine, "Efficient incoherent ray traversal on gpus through compressed wide bvhs," in *Proceedings of High Performance Graphics*. Association for Computing Machinery, 2017, pp. 1–13.

[41] M. K. Yoon, K. Kim, S. Lee, W. W. Ro, and M. Annavaram, "Virtual thread: Maximizing thread-level parallelism beyond gpu scheduling limit," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3, pp. 609–621, 2016.