An Implementation of Multi-Chip Architecture for Real-Time Ray Tracing Based on Parallel Frame Rendering

JINYOUNG LEE, JAE-HO NAH, WOONAM CHUNG, TAE-HYOUNG LEE, AND WOO-CHAN PARK

1Department of Computer Engineering, Sejong University, Seoul 05006, South Korea
2Department of Computer Science, Sangmyung University, Seoul 03016, South Korea
Corresponding author: Woo-Chan Park (pwchan@sejong.ac.kr)

ABSTRACT In this paper, we propose a multi-chip architecture based on parallel frame rendering suitable for real-time ray tracing in dynamic scenes. In multi-chip architecture with the commonly used screen partitioning method, the acceleration structure data, such as a tree, updated in a dynamic scene must be transmitted to each chip. In the proposed frame division method, a tree build and ray tracing are performed on the same chip, and each frame is allocated to a predesignated multi-chip. Thus, the proposed method can achieve scalable performance improvement not only of ray tracing but also of a tree build. We implemented a multi-chip architecture on three field-programmable gate array (FPGA) boards and built 12 ray-tracing cores in the FPGA chip of each board. This configuration means that the inter-chip operates using the frame division method, while the inner chip operates using the screen partitioning method. The results of experiments showed that the proposed multi-chip architecture improved frames per second (FPS) performance by an average of 2.83 times compared to a single-chip architecture.

INDEX TERMS Ray tracing, hardware, multicore processing.

I. INTRODUCTION

With the recent release of ray-tracing GPUs, the real-time ray tracing era has begun. NVIDIA and AMD launched the Ampere [1] and RDNA 2 architectures [2], respectively. The Khronos Group officially added Ray Tracing Extensions to the Vulkan API [3]. Microsoft has released DirectX 12 Ultimate, the latest version of the API, which supports ray tracing [4]. Intel has announced the oneAPI Toolkit, including the Intel Embree, Intel OSPRay, and Intel Open Image Denoise libraries, which are specialized in ray tracing rendering [5].

These ray-tracing GPUs integrate many cores dedicated to ray tracing into a single chip. NVIDIA GeForce RTX series include additional RT cores; there are 46 and 68 RT cores in RTX 2080 and 3080, respectively [6], [7]. AMD’s RDNA 2 GPUs also add a separate ray accelerator to each stream processor [8].

With recent advances in digital devices (for example, over 4K), the GPU performance requirements have accordingly increased significantly. One direct approach to meet these requirements is to configure multiple GPUs in a multi-chip form. For example, NVIDIA GPUs’ Scalable Link Interface (SLI) supports mounting two GPUs, and AMD’s CrossFire can combine up to four GPUs.

Parallel rendering on multi-chip GPUs has been studied for a long time and is well described in [9]. Most of them perform parallel rendering by partitioning the screen [10]–[12], and the result is finally merged in the frame buffer. This screen partitioning method is commonly used in modern multi-GPUs. An alternative is a parallel frame rendering method whereby frames are divided by the number of chips and a frame is allocated to and processed by each chip. Parallel

...
rendering of contiguous frames can increase the cache hit rate, which offers advantages in terms of memory traffic and energy consumption [13].

Most parallel rendering methods are based on rasterization. In a parallel rendering method based on ray tracing, we need to additionally consider the processing of the acceleration structure (AS) for the following reasons. First, the AS, such as a kd-tree or bounding volume hierarchy (BVH), is essential for ray-tracing acceleration [14], [15]. Second, for dynamic scenes, the AS should be re-built or be asynchronously built at every frame [16], [17]. Third, because a tree build algorithm based on the SAH (surface area heuristic) has an $N \log N$ complexity and is recursive, it is normally difficult to achieve scalable performance improvement via parallelization [18], [19].

When ray tracing is parallelized using screen partitioning, rendering performance may achieve scalable improvement. However, tree-build performance does not increase in proportion to rendering performance, which may lead to increased bottlenecks in a dynamic scene. This increasing bottleneck problem may also occur in a way that parallelization is performed in units of rays [20] rather than in units of screens on a multi-chip GPU. Moreover, since all chips must receive the re-built AS data at every frame, additional data transmission is required.

To overcome the above limitations of screen partitioning, we present a multi-chip-based ray tracing system with a frame division method capable of parallel frame rendering. The main benefit of our system is to facilitate tree-build acceleration based on a frame division method. By coupling a tree build and rendering for the corresponding frame in each chip, multiple tree-build units (TBUs) on multiple chips can independently construct trees for different frames. As a result, we can achieve scalable performance even if either tree-build or rendering, which is the core processes of ray tracing, causes a bottleneck. Also, parallel AS processing on multiple boards using SLI or Crossfire has not been attempted yet, so it is expected that it will arouse interest in the graphics, parallel processing, and hardware communities.

As in the proposed frame division method, studies on increasing performance scalability via multiple boards are showing a lot of interest in Nvidia and others to achieve high performance. [22] demonstrates that continuous performance scalability is possible by integrating GPU modules at the package level to overcome the limited performance of a single monolithic GPU. [20] presented many-chip-based architecture for ray tracing acceleration. [23] posted on the Nvidia developer community also suggested the possibility of frame pipelining via multi-GPU.

The above approach based on the frame division method requires an additional frame ordering mechanism. This is because the timing of frame generation by each chip may be different. If draw calls are processed without taking timing differences into account, frames that are out of order may be displayed on the screen. As a solution, we propose a method that uses a frame queue. This simple method only checks whether there is a frame in the frame queue to be displayed, thereby preventing out-of-order frames in a display.

We implemented the proposed multi-chip architecture in three field-programmable gate array (FPGA) boards which is a novel approach not attempted before. The FPGA chip of each board corresponds to the ray-tracing GPU consisted of 12 ray-tracing units (RTUs) and a tree-build unit (TBU). The entire multi-FPGA system operates using the frame division method, while a single FPGA processes sub-blocks of the screen using the traditional ray tracing method—that is, the screen partitioning method.

Our experiments in various test scenes prove scalable performance of our approach through parallel AS builds on multiple chips. According to the experimental results on three FPGA boards, the screen partitioning method showed mixed results: 0.99-2.44× performance with triple chips. In contrast, our approach showed consistent scalable results: 2.66-2.99× performance with the same setup. As a result, we can achieve interactive frame rates (15-42 FPS) at full-HD resolution with our FPGA implementation.

We performed an ASIC evaluation of a hardware prototype using the latest 8-nm process technology. The results showed that the operating frequency was 900 MHz, 12 RTUs used an area of 8.07 mm$^2$, and power consumption was about 2.37 W. According to the results, we estimate that the power consumption and silicon area of our architecture are one to two orders of magnitude lower than those of current ray-tracing GPUs [7], [8], [21].

The rest of this paper is organized as follows. Section 2 describes the background and prior studies. Section 3 describes the proposed multi-chip architecture, the ray-tracing system in which it was implemented, the overall flowchart, and the frame ordering scheme for arranging the display order. Section 4 describes the hardware implementation. Section 5 describes the experimental environment and results. Section 6 concludes the paper and discusses the limitations of the proposed method and directions for future work.

II. RELATED WORK AND BACKGROUND

Figure 1 shows a typical ray-tracing pipeline. This can be divided into a tree-build part and a rendering part. In the

![FIGURE 1. Example of a ray-tracing pipeline for dynamic scenes.](image-url)
tree-build part, the AS, which is the tree information of geometry data for the current frame, is built. In the rendering part, ray tracing and shading are performed based on the built AS. Finally, the rendered image is displayed, and this series of processes is repeated.

There have been several hardware projects to accelerate the above two main parts. The RPU architecture [24] is an early prototype of a combination of programmable shaders and ray-tracing accelerators, similar to the current ray-tracing GPUs. The T&I engine [25] focuses on the acceleration of the traversal and intersection test parts, and this architecture was extended to SGRT [26] with mobile reconfigurable processors. RayCore [27] is another mobile ray-tracing hardware architecture comprising multiple ray-tracing units (RTUs) and a tree-build unit (TBU) for supporting dynamic scenes. Lee et al. [28] later applied a load-balancing algorithm to increase the performance of RayCore in dynamic scenes. Lin et al. [29] presented a hardware architecture for accelerating traversal of dual-split trees [30], which are a new AS type including splitting and carving nodes.

The above architectures commonly include dedicated hardware logic for traversal and/or intersection tests. In contrast, TRaX [31] consists of fully programmable thread processors designed for ray tracing. The TRaX architecture has been improved in terms of memory accesses later [32], [33], and Mach-RT [20] introduces how to design multiple streaming ray-tracing chips on a board and effectively connect them.

Some projects have tried to accelerate tree construction for dynamic scenes, and they can be categorized into BVH construction, BVH refitting, and kd-tree construction. First, Doyle et al. [34] presented the first hardware architecture for SAH BVH construction. Later, Viitanen et al. presented two faster BVH construction architectures named MergeTree [35] and PLOCTree [36], which are based on the newer BVH construction algorithms (HLBVH [37] and PLOC [38]). Second, HART [16] is a heterogeneous system for hardware-accelerated BVH refitting and CPU-based BVH construction. This is possible by extending the asynchronous BVH construction [17]. Viitanen et al. [39] presented a more sophisticated refitting architecture to support quantized BVHs. Third, the TBU in RayCore [27] accelerates SAH kd-tree construction with binning- and sorting-based pipelines. FastTree [40] is another kd-tree construction engine based on the Morton code rather than the SAH.

From now, we will describe the background directly related to our paper in detail. Figure 2 shows an example of a block diagram of a ray-tracing system with a single FPGA. This is similar to the ray-tracing hardware system presented in [28]. One tree-build unit (TBU) performs tree-build, and sub-blocks of the screen are assigned to 12 RTUs, which perform rendering. The TBU and RTUs are improved versions of RayCore, a dedicated hardware architecture proposed by [27].

The host computer (CPU) runs and manages the ray-tracing application and scene manager. It also transfers geometry data and AS data of the system memory. The FPGA board consists of local memory for rendering, RTUs, a TBU, and a bus interface unit.

Figure 3 shows how the 12 RTUs are configured. Each RTU is equipped with a high-efficiency L1 cache, and four RTUs share one L2 cache. Each L2 cache can refer to the local memory through the bus interface. This is similar to the traditional hardware memory hierarchy composed of many cores.

Figure 4 shows the multi-FPGA ray-tracing system configured by extending the a single FPGA board to $n$-FPGA boards in the ray-tracing system of shown in Figure 2. In this multi-FPGA system, rendering is typically performed using the screen partitioning method, whereby multiple chips generate one frame in parallel.
The frame to be generated is partitioned in the screen space by the total number of FPGAs. Then, the partitioned frames are allocated to each FPGA board. The RTUs of each FPGA perform ray tracing of the allocated regions, and the calculated color value is stored in the frame buffer of the FPGA board #1 through the memory controller. Thus, each FPGA performs ray tracing in an area corresponding to about \( \frac{1}{n} \) of the frame.

To support dynamic scenes in Figure 4, an additional TBU is implemented in FPGA #1. The TBU receives geometry data for the current frame and constructs an AS by performing tree-build. The AS data are then transferred to other FPGA chips, and the RTUs in each FPGA perform ray tracing. The AS data transmission increases in proportion to the number of FPGAs used in the system. If the number of FPGAs is \( n \), AS data transmission occurs \( n \) times. This data transmission is performed at every frame.

Although multi-FPGA systems with screen partitioning can reduce the rendering time compared to a single FPGA system, the tree-build time remains the same. Moreover, additional data transmission is required. Therefore, this method may not improve performance via multiple FPGAs when TBU bottlenecks occur. Also, it is not homogeneous because tree-build is performed only on one chip. For this reason, applying the screen partitioning method to multi-chip ray tracing has limitations in terms of performance improvement.

III. THE PROPOSED MULTI-CHIP ARCHITECTURE
A. MULTI-CHIP RAY-TRACING SYSTEM FOR PARALLEL FRAME RENDERING
Figure 5 shows a multi-FPGA ray tracing system capable of parallel frame rendering. Each FPGA is implemented in the form of coupled RTUs and a TBU, and different consecutive frames are simultaneously processed. This frame division approach can proportionally improve a tree build and ray-tracing performance as the number of FPGAs increases. Moreover, because each FPGA processes different frames, additional data transmission of AS data between FPGAs does not occur.

Different predetermined frames are assigned to FPGAs #1, #2, #3, and #\( n \). Each FPGA receives geometry data for the assigned frame from the host and performs a tree build and ray tracing. The 12 RTUs on each FPGA perform ray tracing by dividing the corresponding frames into sub-blocks of \( k \times k \) pixels, like typical ray-tracing hardware such as [27]. Finally, the frames generated by each FPGA board are stored in the frame buffers of the host PC allocated to each FPGA board and are outputted to the display through the frame ordering scheme described in Section 3.3.

B. OVERALL FLOWCHART
Figure 6 shows the overall flowchart of the proposed system. In Part A, each FPGA checks whether rendering is completed and prepares to process the next frame. If rendering is completed, the system proceeds to Parts B and C. Part B performs tasks to display the generated frame on the screen. Part C performs setting tasks for the next frame and determines whether the scene ends.

Part A includes the MaxCardNum check, the Frame assignment, and the Geometry data transmission stages. In the MaxCardNum check stage, the total number of FPGA boards in the multi-FPGA system is checked and set to \( n \), and the draw number is set to 1. In the Frame assignment stage, frames to be generated are assigned to each FPGA board. The number of allocated frames is determined according to the number of the FPGA boards as shown in Figure 5 (FPGA #1, #2, #3 and #\( n \)). In the Geometry data transmission stage, each FPGA board receives geometry data for the corresponding frame.
Part B which is described in detail in section 3, includes the Store to frame buffer, Store to frame queue, Frame number check, and Draw frame stages. In the Store to frame buffer and Store to frame queue stages, the generated frames are sequentially stored in the frame buffer and queue, respectively. In the Frame number check stage, the system checks whether the number in which the frames stored in the frame queue matches the draw number. If the frame and draw numbers do not match, the frame waits, if they match, it is outputted to the display in the Draw frame stage. The draw number increases by one each time the Draw frame stage is performed.

Part C includes the Frame number setting and Total frame number check stages. In the Frame number setting stage, the frame number for the next frame is set. The frame number is calculated by adding the number of FPGAs used in the system to the initially set frame number. For example, the current frame number is six and the number of FPGAs (n) is three, the next frame number will be nine. In the Total frame number check stage, the system checks whether the set frame number is higher than the total frame number of the scene to determine whether the scene has ended.

C. FRAME ORDERING SCHEME
The frame division method can process many frames simultaneously, depending on the number of FPGAs. In this case, the order in which the frame generation is completed may not coincide with the display order. Therefore, the generated frames must be arranged according to the display order.

To achieve this, an ordering scheme with the frame number, frame buffer, and frame queue is proposed so that the generated frames are outputted in the display order.

A frame number is assigned to each FPGA as described in Section 3.2. The frame number refers to the order in which the corresponding frame is displayed. When frame rendering is completed, each FPGA board stores it in the allocated frame buffer in the memory of the host computer. If the frame buffer is insufficient, a frame may be overwritten by the next frame if the latter is generated before the former is displayed. To prevent overwriting, each FPGA has one or more frame buffers. In this paper, each FPGA has three frame buffers.

Frames stored in the frame buffers are sequentially sent to the frame queue in the memory of the host computer. The frame queue is where to check if the frame number and draw number of the generated frames match and to wait for the draw call. The size of the frame queue is the same as the total number of frame buffers.

The frames are stored in the frame queue in an order corresponding to their frame numbers. Then, frames are displayed on the screen through use of the draw call function. The calling order is determined by the draw number. The initial draw number value of 1 increases by 1 each time the draw call function is called. If the draw and frame numbers match, the frame is displayed; if not, the frame is not displayed. Therefore, draw calls are performed in order from frame #1, so that the display order can be arranged regardless of the order in which the frame is generated.

Figure 7 shows an example of a frame ordering scheme in a system using three FPGA boards. FPGA boards #1, #2, and #3 generate frames #1, #2, and #3, respectively. Then, the next frame to be generated is assigned to the FPGAs by adding n (the total number of FPGAs) to the frame number of the previous frame. Each FPGA board has three frame buffers that store the generated frames sequentially. Frames #1, #4, and #7 are stored in frame buffers 1, 2, and 3 of FPGA board

Figure 6. Overall flowchart of the proposed multi-chip architecture.

Figure 7. Block diagram of the proposed frame ordering scheme.
#1; frame #2, #5, and #8 are stored in frame buffers 1, 2, and 3 of FPGA board #2; frames #3, #6, and #9 are stored in frame buffers 1, 2, and 3 of FPGA board #3.

The frames stored in the frame buffers are stored in the frame queue in the order of their frame numbers. Frames #1, #4, and #7, which are stored in frame queues 1, 4, and 7, respectively. The frames stored in the frame buffers of FPGA board #2 and #3 are stored in the frame queues in the same way. The frames stored in the frame queues are displayed on the screen in order by drawing. The draw call function is called sequentially by comparing the draw and frame numbers from frame queue 1.

### IV. HARDWARE IMPLEMENTATION

#### A. MULTI-FPGA PROTOTYPE

Figure 8 shows the multi-FPGA prototype with the proposed multi-chip architecture. The system consists of a host computer with an Intel Core i9-9920X (3.5 GHz) CPU and three FPGA boards with VCU118 FPGA boards [41]. Each FPGA board is connected to the host computer through the PCI Express interface. Because the Intel Core i9-9920X CPU does not support built-in graphics, NVIDIA GeForce GT 1030 GPU was mounted for monitor output.

![Figure 8. The multi-FPGA ray-tracing system.](image.png)

The multi-FPGA system operates at a speed of 200 MHz. Table 1 shows the resource utilization of the proposed multi-chip architecture implemented in an FPGA.

#### TABLE 1. Resource utilization of the proposed hardware.

|       | LUT | REGISTER | BLOCK MEMORY | URAM  | DSP  |
|-------|-----|----------|---------------|-------|------|
| RTUs  | 60.15% | 39.64% | 59.7% | 14.69% | 12.68% |
| TBU   | 9.75% | 4.74% | 3.5% | 0% | 8.2% |

#### B. ASIC EVALUATION

For the ASIC evaluation, we used recent 8-nm low-power process technology. The TBU and RTUs were synthesized up to 1 GHz and 900 MHz, respectively, with a clock period margin of 30%. Table 2 shows the Power, Performance and Area (PPA) results. The total area of the 12 RTUs is around 8.07 mm² and the silicon area of the TBU is around 0.26 mm². The total power consumption of the 12 RTUs and the TBU is 2.37 W and 131 mW, respectively. Considering that the NVIDIA GeForce RTX 3090 has 350W of thermal design power (TDP) and an area of 628 mm² in 8-nm technology, we believe that the proposed multi-chip architecture can be highly efficient in terms of power consumption and area.

#### V. EXPERIMENT

Figure 9 shows the six dynamic benchmark scenes used in the experiments; Sword (37k primitives), Wood (30k primitives), Church (261k primitives), Butterfly (49k primitives), Movie (338k primitives), and Jewel (17k primitives). Table 3 shows detailed information on each dynamic benchmark scene, including the total numbers of primitives (static and dynamic), the numbers of textures and lights, and the total frames of each scene. To ensure accuracy, we replicated the procedures three times and obtained the averages. The experiments were conducted at a resolution of 1920 × 1080.

![Figure 9. Dynamic scenes for the benchmark test; (top left) Sword, (top right) Wood, (middle left) Church, (middle right) Butterfly, (bottom left) Movie, (bottom right) Jewel.](image.png)

#### A. EXPERIMENTAL RESULTS

Table 4 shows the FPS changes according to the number of FPGA boards when ray tracing was performed using the
### Table 3. Features of the test scenes.

| Scene | Total Primitives | # of Textures | # of Lights | Total Frames |
|-------|------------------|---------------|-------------|--------------|
| Sword | 37666            | 59            | 2           | 321          |
| Wood  | 30688            | 6             | 2           | 1001         |
| Church| 261681           | 27            | 5           | 2301         |
| Butterfly | 19135         | 10            | 3           | 301          |
| Movie | 338708           | 29            | 3           | 1051         |
| Jewel | 8770            | 5             | 1           | 851          |

### Table 4. Experimental results.

| Scene   | Method      | # of FPGA | FPS | Performance Ratio |
|---------|-------------|-----------|-----|-------------------|
| Sword   | Screen      | 1         | 5.6 | 100%              |
|         | Partitioning| 2         | 10  | 87%               |
|         | Frame       | 2         | 10.8| 92%               |
|         | Division    | 3         | 15.8| 282%              |
| Wood    | Screen      | 1         | 5   | 100%              |
|         | Partitioning| 2         | 8.7 | 100%              |
|         | Frame       | 2         | 17.2| 199%              |
|         | Division    | 3         | 25.8| 299%              |
| Church  | Screen      | 1         | 3   | 100%              |
|         | Partitioning| 2         | 5.5 | 241%              |
|         | Frame       | 2         | 20.1| 197%              |
|         | Division    | 3         | 42.3| 287%              |
| Butterfly| Screen    | 1         | 3   | 100%              |
|         | Partitioning| 2         | 12.1| 121%              |
|         | Frame       | 2         | 21.5| 177%              |
|         | Division    | 3         | 24.0| 197%              |
| Movie   | Screen      | 1         | 3   | 100%              |
|         | Partitioning| 2         | 18.4| 122%              |
|         | Frame       | 2         | 29.6| 198%              |
|         | Division    | 3         | 40.8| 272%              |
| Jewel   | Screen      | 1         | 3   | 100%              |
|         | Partitioning| 2         | 15.7| 115%              |
|         | Frame       | 2         | 26.2| 192%              |

### Table 5. Average FPS performance ratios according to the number of FPGAs.

| Scene   | # of FPGA | 1 → 2 | 2 → 3 |
|---------|-----------|-------|-------|
| Sword   | 5.6       | 196.4%| 100%  |
| Wood    | 6.7       | 283.2%| 100%  |
| Church  | 10.8      | 132.2%| 100%  |

### Table 6. Processing times of RTUs and TBU according to the number of FPGAs with the screen partitioning method (in milliseconds).

| # of FPGA | RTUs | TBU | RTUs | TBU | RTUs | TBU |
|-----------|------|-----|------|-----|------|-----|
| 1         | 101.5| 178.4| 58.8 | 178.0| 42.6 | 177.8|
| 2         | 42.6 | 115.9| 23.7 | 115.1| 21.5 | 115.9|
| 3         | 67.9 | 13.9 | 37.8 | 15.6 | 28.2 | 15.8 |
| Jewel     | 82.4 | 19.3 | 46.4 | 18.7 | 33.7 | 18.1 |

Table 6 shows the processing times of RTUs and TBU according to the number of FPGAs with the screen partitioning method. Although this method reduced the processing time of RTUs in proportion to the number of FPGAs, there was no significant difference in TBU processing time. This means that even if the number of FPGAs increases, the FPS converges to the performance of the TBU. Convergence to TBU performance can be mitigated by improving TBU performance, but it is may not a fundamental solution because it can decrease area- and/or power-efficiency. Therefore, if the TBU performance is too low for the scene complexity, the improvement in performance due to multi-chip expansion is insignificant. Our approach solves the problem of parallel acceleration structure builds.

### B. PERFORMANCE EVALUATION OF ASIC VERSION OF MULTI-CIP ARCHITECTURE WITH FRAME DIVISION METHOD

We used the same metric as RayCore [27] to evaluate the performance of our architecture. In the proposed parallel frame rendering method, because frame latency increases in

...
The expected performance of four chips, considering we evaluated ASIC performance at a max frequency of 900 MHz for RTUs, is six times faster than the FPGA implementation due to the difference in the clock frequency (200 MHz → 900 MHz) and the number of chips (3 → 4). Therefore, the performance of the ASIC version is expected to be about 254 FPS for the Church scene in FHD. This is achieved with an area of 33.3 mm² (48 RTU + 4 TBU) and power consumption of about 10.0 W.

The latest RTX 30 series from NVIDIA (RTX 3090) uses 8-nm process technology, has an area of 628 mm², clock frequency of 1395 MHz, and TDP of 350 W. Even considering that the RT cores occupy a small portion of the total area of RTX 3090 if the proposed multi-chip architecture is manufactured with the same process technology and area, it is expected that real-time performance with ultra-high resolution (e.g., over 4k) will be obtained even if it is manufactured with a single chip rather than a multi-chip.

Table 7 shows the kd-tree construction performance of the FPGA prototype’s TBU for the scenes in Figure 9. According to the results in Table 7, the kd-tree build time on the FPGA grows approximately linearly with the primitive counts. The longest time is 178.4 ms for the Sword scene with 26k dynamic primitives. The total memory traffic for the scene is only up to 9.71 MB/frame (including frame read memory traffic, 8.3 MB/frame). In this case, 0.582 GB/s of memory bandwidth is required for 60 FPS performance, which is significantly lower considering that modern mobile application processors (e.g., Qualcomm Snapdragon 888) support up to 51.2 GB/s of memory bandwidth. This result means that at 1 GHz, the ASIC version of the TBU can update all the benchmarks shown in Figure 9 at frame rates of over 60 FPS without high memory bandwidth requirements.

### VI. CONCLUSION, LIMITATIONS AND FUTURE WORK

#### A. CONCLUSION

In this paper, we propose a multi-chip architecture based on a frame division method suitable for real-time ray tracing. Each chip is implemented with a TBU and RTUs in the form of coupling so that each chip generates different frames at the same time. This method improves performance almost proportionally to the number of chips and has excellent scalability. It can achieve linear performance improvement even in dynamic scenes, whereas the screen partition method, which is commonly used in parallel rendering, may cause performance bottlenecks in such scenes.

We implemented the proposed architecture in three FPGAs. The experimental results showed that FPS performance improved by up to 2.99 times in various dynamic scenes, confirming that the proposed method can achieve linearly scalable performance improvement. We also performed ASIC evaluations of our architecture and compared the results with the latest ray-tracing GPUs. The proposed architecture showed dramatic results in terms of occupied area and power consumption. In the future, it is expected to be sufficiently embedded in application processors such as smartphones.

#### B. LIMITATIONS AND FUTURE WORK

The proposed method achieves scalable performance proportional to the number of chips. However, frame latency also increases proportionally. This may not be an issue if the frame rate is very high but may not be appropriate if a very fast response speed is required. For this reason, the proposed architecture is limited in terms of increasing the number of chips and is thus more suitable for multi-chip than many-chip systems. To address this issue, we intend to study the extent to which frame latency is generally acceptable to humans and apply it to the proposed system.

Another expected problem is that overall performance is limited by the maximum tree-build throughput on each chip. This may be an issue in scenes with very complex geometry. This problem could be alleviated by increasing the number of TBUs in each chip. However, this would lead to an increase in area, and a careful work distribution over multiple TBUs is needed to get sufficient scalable performance and/or tree quality. Therefore, a more detailed investigation is required to determine the optimal performance ratio of the TBUs and RTUs. In the future, we would like to propose an enhanced hybrid system that combines screen partitioning and frame division methods to solve these problems.

### ACKNOWLEDGMENT

The EDA tool was supported by the IC Design Education Center.

### REFERENCES

1. D. James. (2020). *Nvidia Ampere—RTX 3090, RTX 3080, and RTX 3070 Release Date, Specs, Performance, and Pricing*. [Online]. Available: https://www.pcgamer.com/nvidia-ampere-release-date-specs-performance/

2. J. Walton. (2020). *AMD Big Navi RDNA 2 GPUs: Release Date, Specs, Everything we Know*. [Online]. Available: https://www.tomshardware.com/news/amd-big-navi-rdna2-all-we-know

3. Khronos Group. *Vulkan Ray Tracing Final Specification Release*. Accessed: Jan. 4, 2021. [Online]. Available: https://www.khronos.org/blog/vulkan-ray-tracing-final-specification-release

4. Jianye. *DirectX 12 Ultimate for Holiday 2020*. Accessed: Jan. 4, 2021. [Online]. Available: https://devblogs.microsoft.com/directx/directx-12-ultimate-for-holiday-2020/
S. Eilemann, A. Bilgili, M. Abdellah, J. Hernando, M. Makhinya, and R. Pajarola, “Equalizer: A scalable parallel rendering framework,” IEEE Comput. Graph. Appl., vol. 14, no. 4, pp. 23–32, Jul. 1994.

S. Eilemann, M. Makhinya, and R. Pajarola, “Equalizer: A scalable parallel rendering framework,” IEEE Trans. Vis. Comput. Graphics, vol. 15, no. 3, pp. 436–452, May 2009.

S. Eilemann, A. Bilgili, M. Abdellah, J. Hernando, M. Makhinya, R. Pajarola, and F. Schürmann, “Parallel rendering on hybrid multi-GPU clusters,” in Proc. Eurographics Symp. Parallel Graphics. Visualizat., 2012, pp. 109–117.

P. Yin, X. Jiang, J. Shi, and R. Zhou, “Multi-screen tiled displayed, parallel rendering system for a large terrain dataset,” Int. J. Virtual Real., vol. 5, no. 4, pp. 47–54, 2006.

H. Jordan, S. Pellegrini, P. Thoman, K. Kofler, and T. Fahringer, “Parallel frame rendering: Trading responsiveness for energy on a mobile GPU,” in Proc. 22nd Int. Conf. Parallel Architectures Compilation Techn., Oct. 2013, pp. 83–92.

M. Pharr, W. Jakob, and G. Humphreys, Physically Based Rendering: From Theory to Implementation. San Mateo, CA, USA: Morgan Kaufmann, 2016.

J.-H. Nah, J.-W. Kim, J. Park, W.-J. Lee, J.-S. Park, S.-Y. Jung, and T.-D. Han, “HART: A hybrid architecture for ray tracing animated scenes,” IEEE Trans. Vis. Comput. Graphics vol. 21, no. 3, pp. 389–401, Nov. 2014.

I. Wald, T. Ize, and S. G. Parker, “Fast, parallel, and asynchronous construction of BVHs for ray tracing animated scenes,” Comput. Graph., vol. 32, no. 1, pp. 3–13, Feb. 2008.

I. Wald and V. Havran, “On building fast KD-trees for ray tracing, and on doing that in \(O(N \log N)\),” in Proc. IEEE Symp. Interact. Ray Tracing, Sep. 2006, pp. 61–69.

M. Shevtsov, A. Soupick, and A. Kapustin, “Highly parallel fast KD-tree construction for interactive ray tracing of dynamic scenes,” Comput. Graph. Forum, vol. 26, no. 3, pp. 395–404, Sep. 2007.

E. Vasiou, K. Shikurko, E. Brunvand, and C. Yuksel, “Mach-RT: A many chip architecture for HighPerformance ray tracing,” IEEE Trans. Vis. Comput. Graphics, early access, Sep. 1, 2020, doi: 10.1109/TVCG.2020.3021048.

TechPowerUp. AMD Radeon RX 6900 XT. Accessed: Jan. 4, 2021. [Online]. Available: https://www.techpowerup.com/gpu-specs/raiden-rx-6900-xt.c3481

A. Arunkumar, E. Bolotin, B. Cho, U. Milic, E. Ebrahimi, O. Villa, and D. Nellans, “MCM-GPU: Multi-chip-module GPUs for continued performance scalability,” ACM SIGARCH Comput. Archit. News, vol. 45, no. 2, pp. 320–332, 2017.