gVulkan: Scalable GPU Pooling for Pixel-Grained Rendering in Ray Tracing

Yicheng Gu, Yun Wang, Yunfan Sun, Yuxin Xiang, Yufan Jiang, Xuyan Hu, Zhengwei Qi, and Haibing Guan, *Shanghai Jiao Tong University*

https://www.usenix.org/conference/atc24/presentation/gu-yicheng

This paper is included in the Proceedings of the 2024 USENIX Annual Technical Conference.

July 10–12, 2024 • Santa Clara, CA, USA

978-1-939133-41-0

gVulkan: Scalable GPU Pooling for Pixel-Grained Rendering in Ray Tracing

Yicheng Gu, Yun Wang, Yunfan Sun, Yuxin Xiang,
Yufan Jiang, Xuyan Hu, Zhengwei Qi, Haibing Guan

Shanghai Jiao Tong University

Abstract

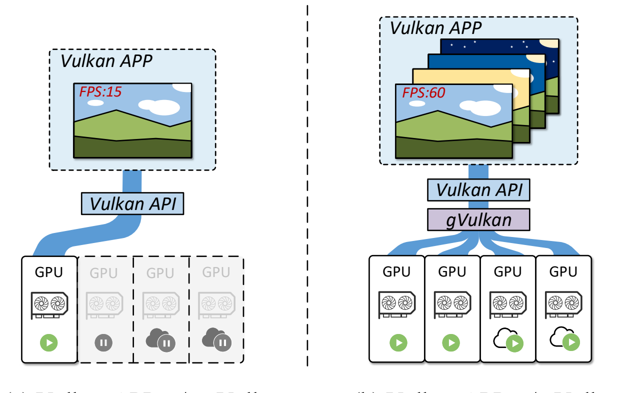
Ray tracing rendering technology enhances scene realism and offers immersive experiences. However, it demands significant computational resources to trace and compute light-object interactions. As a result, traditional local GPU rendering might not meet the demands for high image quality and low latency. Moreover, many applications are tailored to utilize the resources of a single GPU, limiting their capacity to increase computational power through additional GPUs.

This paper presents gVulkan, the first transparent multi-GPU acceleration rendering solution for Vulkan-based ray tracing. To address the bottleneck caused by limited local GPU resources, gVulkan can offload ray tracing rendering to the cloud via API-forwarding. In the cloud, gVulkan employs Split Frame Rendering (SFR) to enable an arbitrary number of GPUs to accelerate rendering in parallel, while dynamically self-rebalancing the workload at a pixel-grained level across GPUs. Experiments demonstrate that gVulkan can accelerate Vulkan-based ray tracing programs in an application-unaware manner. By dynamically rebalancing each GPU's workload, gVulkan achieves good linearity with 3.81× speedup across 4 GPUs on average.

1 Introduction

Real-time rendering is applied in many fields, such as art design, gaming, and visualization [23, 40, 42], and has continuously attracted people's attention. This technique pivots on two crucial metrics for enhancing user experience: high quality and low latency. High-quality rendering deepens user immersion in visuals, while low latency reduces user wait times. Traditionally, real-time rendering has relied on rasterization, which compromises some image quality to maintain low latency, a standard practice for several decades. However, with the integration of hardware ray tracing pipelines in GPUs [11, 44], the ray tracing approach has also entered the realm of real-time rendering.

Ray tracing is a common technique in mainstream graphics rendering. In contrast to rasterization rendering [1], ray



(a) Vulkan APP w/o gVulkan (b) Vulkan APP w/ gVulkan

Figure 1: Traditional Vulkan APP w/ and w/o gVulkan. Traditional Vulkan APP can only use one GPU lead to low FPS, while use gVulkan can utilize GPU resources in cloud to accelerate rendering.

tracing is particularly effective at generating more realistic lighting and shadow effects. Ray tracing uses the principle of reversibility of light paths, emitting rays from the camera towards the screen. After colliding with the scene, these rays produce reflections, refractions, and diffuse reflections, ultimately capturing their true colors. Therefore, it can accurately reproduce various reflection effects, which is unachievable by rasterization.

However, ray tracing is a computationally intensive task that requires significant resources to accurately simulate the interaction between light rays and objects in a virtual environment [5, 21, 37]. As the demand for more complex and larger-scale scenes continues to grow, the computational requirements for ray tracing within these scenes have correspondingly increased. GPUs primarily designed for rasterization lack the necessary computational power to meet these demands. Therefore, traditional local GPU rendering is insufficient to satisfy the requirements for high computation and low latency.

Several solutions have been proposed to accelerate ray tracing, which mainly focuses on hardware-level support [4, 26]. However, these optimizations either require significant modifications to modern GPU workflow or require specific hardware support that introduces a high manufacturing overhead. While such optimizations have shown promise in accelerating the ray tracing process, they may not be practical or cost-effective.

Furthermore, in computer graphics, leveraging multiple GPUs for concurrent rendering is a classic approach for acceleration [6]. Alternate frame rendering (AFR) and split-frame rendering (SFR) are crucial techniques in this context. However, AFR does not reduce the latency introduced by each frame, making SFR a more strategic choice. SFR encompasses three primary strategies: sort-first, sort-middle, and sort-last [24, 28, 41]. Existing research on sort-middle mainly focuses on rasterization [35, 38], while sort-last requires considerable effort to merge the split results [10]. Image parallelization methods like sort-first [20, 25], while not achieving even workload distribution in rasterization, tend to be more favorable for the workload of ray tracing.

However, as shown in Fig. 1a, most ray tracing applications are developed for a single GPU. In general, an application can only use one GPU. Therefore, these solutions to accelerate ray tracing require intrusive changes to the source applications. Consequently, the newly designed solutions cannot be quickly integrated into the existing application ecosystem. They can only be applied in the future when new applications incorporate these designs during development, but it will take a long time before they truly become effective.

To accelerate rendering in an application-unaware manner, choosing the right graphics API is also crucial. Vulkan, as a high-performance, cross-platform, open-source graphics API [34], plays an important role in high-quality real-time rendering. As a low-overhead, low-latency, and low-level graphics API, Vulkan requires developers to control resources finely. Compared to other open-source graphics APIs like OpenGL, Vulkan can fully utilize hardware capabilities for performance optimization [7, 19]. It also provides support for hardware-accelerated ray tracing and offers better support for multi-threading [27, 29].

This paper introduces gVulkan, an application-unaware scalable multi-GPU acceleration rendering solution designed for Vulkan-based ray tracing applications. gVulkan offloads ray tracing rendering to the cloud, effectively alleviating local GPU pressure without necessitating a specific hardware architecture, while maintaining transparency for use-cases with respect to cloud-based acceleration. Within the cloud, gVulkan customizes the API streaming as well as shader required by each GPU, employs split-frame rendering (SFR), enabling multiple GPUs to process pixel-grained tasks concurrently and balance workload dynamically. This approach enhances GPU utilization and reduces computational latency. Finally, the generated video stream is compressed using FFmpeg and transmitted back to the user end for output.

To date, we have developed a prototype of gVulkan for both Windows and Linux platforms. Performance results demonstrate that gVulkan imposes only negligible memory overhead, and achieves high scalability and dynamically adjustable design goals with reduced intra-frame latency. To the best of our knowledge, gVulkan is the first transparent multi-GPU acceleration rendering solution for Vulkan-based ray tracing that supports high scalability and dynamic self-rebalancing.

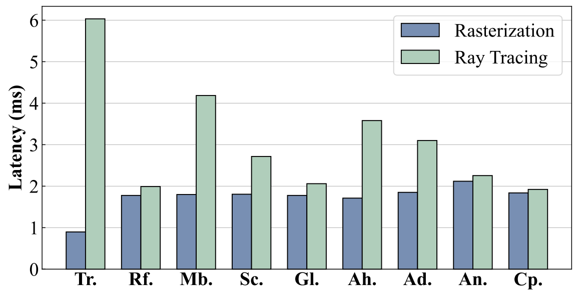


Figure 2: The latency of rasterization and ray tracing.

Overall, this work makes the following contributions:

1. We implemented a prototype of gVulkan, a transparent multi-GPU acceleration solution, offering high scalability and pixel-grained dynamic self-rebalancing for Vulkan-based ray tracing.
2. gVulkan introduces a latency-determined adaptive load balancing mechanism, dynamically adjusting the GPU load at pixel-grained to achieve single-frame rendering with minimum latency.
3. gVulkan proposes a dependency-decoupled parallel rendering approach that customizes the API streaming for each GPU and accelerates the rendering process.
4. gVulkan proposes a resource-classified transparent forwarding scheme for the Vulkan API, which fully utilizes the rendering power of the GPU pool in use-cases transparent manner, achieving high ecological compatibility.
5. To address existing Vulkan APIs' limitations that lack support for partial rendering in ray tracing, gVulkan introduces *Shader Customizer*, which enables transparent and non-uniform GPU splitting in use-cases without modifying the driver.
6. Compared to locally executed Vulkan use-cases, gVulkan achieved a speedup of 3.81× using 4 GPUs. It provides QoS assurance for use-cases that were previously unable to be assured without altering the source code.

2 Motivation

In this section, we briefly introduce the current status of ray tracing technology. We discuss the challenges of high-quality real-time rendering in online interactive scenarios.

2.1 The Latency of Ray Tracing

The two most common rendering techniques in computer graphics are rasterization and ray tracing. Rasterization converts objects in a 3D scene into 2D pixels and renders them

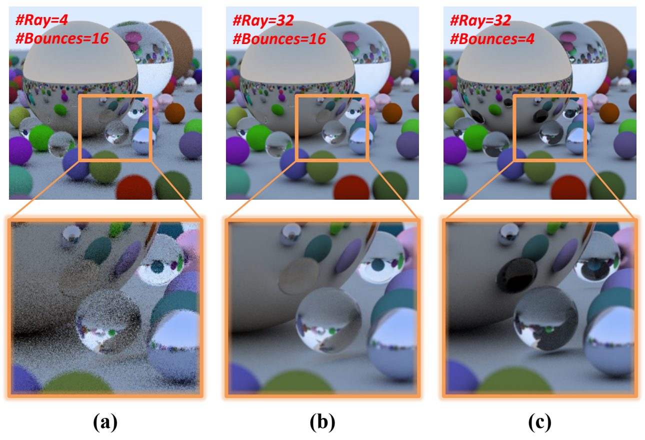


Figure 3: The image quality between Vulkan and gVulkan in same latency. (b) is able to emit more rays within a single frame compared to (a), and the rays in (b) have more bounce times than those in (c).

based on the color and depth information of the pixels. Rasterization is fast and suitable for real-time rendering, but its image quality is relatively poor and does not accurately simulate the propagation and reflection of light.

In contrast, ray tracing is a rendering method based on physical optics that generates images by simulating the propagation and interaction of light in a scene. Ray tracing can simulate the propagation and reflection of light more realistically, resulting in high-quality images. However, because ray tracing requires multiple iterations of computation for each pixel, it exhibits relatively longer rendering times, which limits its use in real-time applications and interactive scenes.

Ray tracing takes more computing power compared to rasterization. Fig. 2 compares the latency of rasterization and ray tracing when rendering the same scene. The ray tracing use-case of Tr. represents the latency obtained by using ray tracing on the whole triangle. The other use-cases are from NVIDIA's open-source ray tracing examples[1], using ray tracing only for selected parts of the scene. In the same scene, pure ray tracing results in up to 6.7× higher latency compared to rasterization. Partial ray tracing also incurs varying degrees of increased latency compared to rasterization. The impact of ray tracing latency can be even greater in complex scenes.

2.2 Challenges for high-quality Real-Time Rendering

Creating immersive online interactive scenarios has always been a pursuit for people. Rasterization, as a traditional real-time rendering solution, can meet the demands of real-time conditions well, rapidly generating the required images. However, it cannot reflect the actual effects of light and shadow, as rasterization does not bounce light rays.

[1]https://github.com/nvpro-samples/vk_raytracing_tutorial_KHR

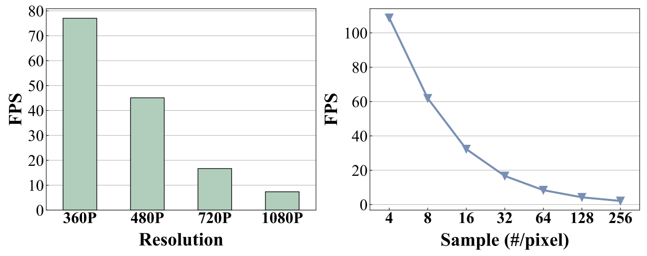


Figure 4: Impact of resolution and the number of ray samples per pixel on FPS under a T4 GPU.

With the advancement of GPU computing power and the emergence of hardware ray tracing pipelines, the latency required for ray tracing is continuously decreasing. As a solution that provides high-quality images, ray tracing is also being used in real-time rendering. However, this high-quality real-time rendering solution also faces many challenges.

Challenge 1: Delivering low-latency, high-quality visual effects in real-time rendering. Achieving both low latency and high image quality is the most direct demand in scenarios such as gaming. To ensure high quality in real-time rendering, we need to use ray tracing technology. However, ray tracing not only increases latency several times compared to rasterization but also affects latency at higher resolutions and with more ray samples per pixel (SPP). As shown in Fig. 4, with the continuous increase in resolution, the latency required for rendering the same scene also increases. Also, the number of SPP and ray bounces are important indicators determining image quality. From Fig. 3a and 3b, it can be seen that with a fixed number of bounces, a smaller SPP introduces more noise into the image, thereby affecting the image quality. Fig. 3b and 3c show that with a fixed SPP, too few ray bounces may lead to incorrect rendering results, as some bounce effects are not displayed. However, as shown in Fig. 4, with the continuous increase of SPP, the rendering latency also keeps increasing. Therefore, high quality and low latency are opposing factors in real-time rendering, and finding a balance between them becomes a challenge.

Challenge 2: The limited local rendering power available for users. The contradiction between low latency and high image quality is mainly due to computational power. Sufficient computational power can obtain higher-quality images in a given time. However, ordinary users will only purchase a small number of GPUs. They may lack high-performance hardware or reliable Internet connections, which could hinder their ability to achieve high-quality real-time rendering.

Additionally, secure containers [2, 15] are widely used in serverless computing. However, current secure containers cannot use GPUs, which has an impact on their scope of use.

Currently, cloud rendering is an excellent solution to this problem. GPU resources can be utilized more efficiently by centralizing resources to serve more users. The rental model can also lower the threshold for rendering, allowing access

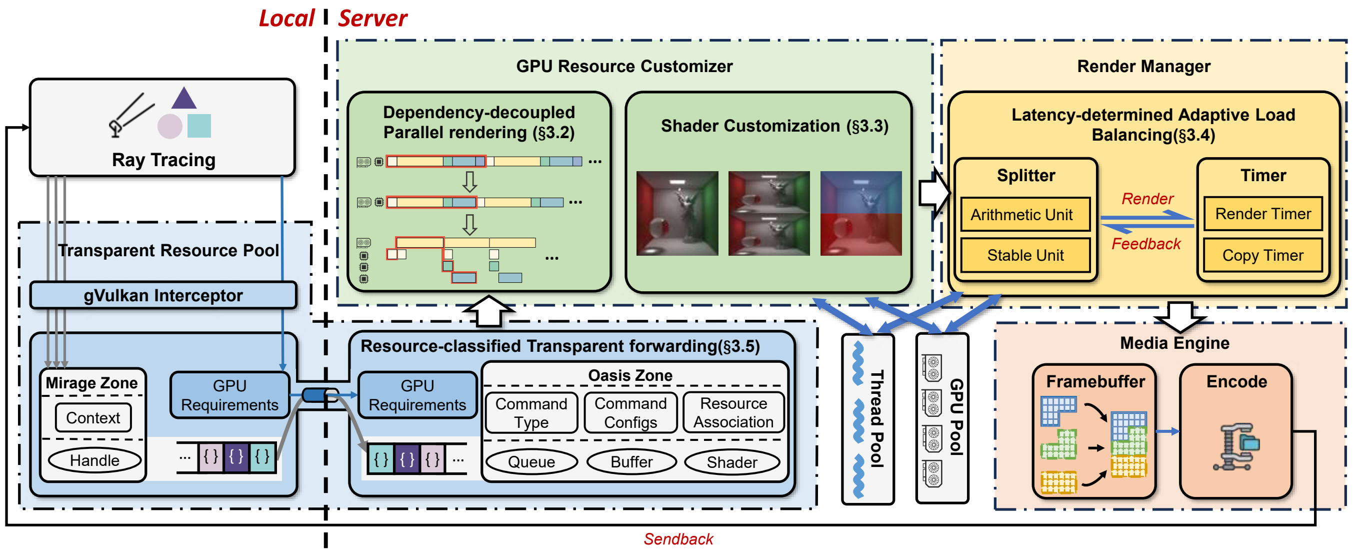


Figure 5: Architecture of gVulkan.

to more computational power with limited expenditure.

Challenge 3: Ecosystem compatibility of solutions. Contemporary use-cases utilize rendering APIs such as OpenGL, Vulkan, and DirectX. Among these, Vulkan is an emerging open-source API capable of fully leveraging hardware performance. However, Fig. 1a illustrates that only a few applications are designed with consideration for multiple GPUs and can be manually configured for multi-GPU mode. Most Vulkan applications are designed primarily for rendering on a single GPU. Therefore, even if users have multiple GPUs available, whether locally or in the cloud, they can only use one of the GPUs for rendering, making it difficult to fully utilize their available computation power.

To address this issue, this paper proposes the gVulkan architecture, as illustrated in Fig. 1b. When local computing power is insufficient, gVulkan transparently intercepts the Vulkan API in an application-unaware manner and transmits it to the server side. It creates custom API streams and shaders for each GPU, rendering in a dynamic and self-balancing manner. This method allows applications initially designed for a single GPU to utilize existing GPU resources locally or in the cloud entirely. With unchanged image quality, gVulkan can utilize more GPUs to improve FPS. Similarly, as shown in Fig. 3, if users emit more rays through multiple GPUs with the same latency, gVulkan can reduce image noise and achieve superior image quality.

3 System Design

3.1 Overview

We show the gVulkan architecture in Fig. 5. gVulkan encompasses two primary perspectives: **gVulkan-local** and **gVulkan-server**, as well as four critical components: (1) a **Transparent Resource Pool** (TRP) that intercepts and sends API resources to the server while the ray tracing application is running (§3.5 and §4.1); (2) a **GPU Resource Customizer** that customizes the API streaming and shader used by each GPU (§3.2 and §3.3); (3) a **Rendering Manager** that adaptively schedules the workloads rendered on each GPU (§3.4); (4) a **Media Engine** that merges and compresses images to send back to the application (§4.2).

Given a ray tracing application, gVulkan's Transparent Resource Pool runs with the application, intercepts the Vulkan APIs, and sends the resources used by the APIs to the server for processing. With these resources, the GPU Resource Customizer customizes the required API and shader to achieve the target rendering effect based on the region to be rendered by the corresponding GPU. When multiple GPUs render together, the slower rendering speed of some GPUs can affect the overall rendering latency. Therefore, the rendering manager decides how to distribute the workload among each GPU based on the latency of each GPU during rendering to achieve the lowest latency rendering solution. Ultimately, the Media Engine combines the images rendered by each GPU and compresses them back to the ray tracing application for rendering. We will describe each component of gVulkan as follows.

3.2 Dependency-decoupled Parallel Rendering

Reproducing the API flow on the server side can get the application running. However, if the original program uses only one GPU, the server side cannot utilize more GPUs to accelerate rendering. Therefore, the GPU Resource customizer creates multiple *vkDevices* on the server side. It implements the API streaming from the local side on multiple logical devices, thus enabling a basic multi-GPU parallel rendering scheme. However, there is still considerable space for

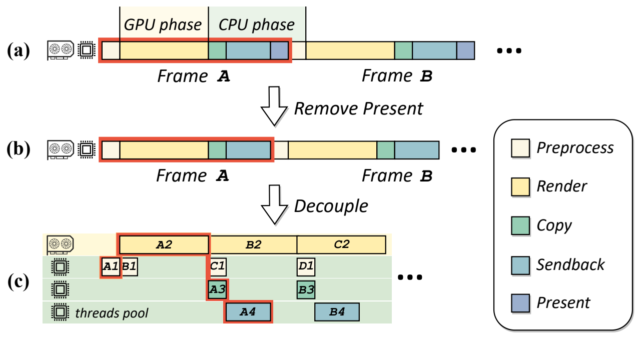


Figure 6: Present elimination and dependency-decoupled parallel rendering accelerate FPS.

optimizing this approach.

Present elimination. As shown in Fig. 6a, multiple GPUs rendering together introduces time overhead for copying, sendback, and some pre-processing and also renders additional frames on the server side for presentation. However, the server does not need to display this frame of the image; it only needs to send back the rendered result. Therefore, this part of the work causes waste on the server side. The APIs related to the presentation are mainly associated with the *swapchain*, and removing the present API alone could cause errors in the program. gVulkan addresses this issue by using a custom *swapchain* and *imageIndex*, eliminating the error and preventing the server from having to present the image, as shown in Fig. 6b.

Dependency-decoupled parallel rendering. However, this approach does not fully utilize the GPU's total computing power. When CPU computations are being performed, the GPU remains idle. Therefore, gVulkan decouples the phases required for a frame based on dependency relationships and utilizes multi-threading to maximize GPU computation, as shown in Fig. 6c. The pre-processing phase needs to be executed before rendering. gVulkan continues to maintain the swapchain method for rendering multiple frames simultaneously, allowing all pre-processing phases to be executed immediately. Nevertheless, running the pre-processing phase too early can increase the latency of a frame. Therefore, in search of a balance between latency and efficiency, gVulkan only processes the pre-processing phase for two frames ahead. The pre-processing phase for frame n+2 only begins after the rendering of frame n is complete. The copying phase depends on the completion of rendering, and the send backing phase relies on the completion of the copying phase. Pre-processing, copying, and send backing phases are continuously executed by different threads. With a fully parallel implementation, when the rendering time exceeds the pre-processing time, the FPS is only related to the rendering time. In this way, the GPU is constantly tasked with rendering, fully utilizing its computing power. Meanwhile, the CPU handles pre-processing, copying, send backing, and other CPU-intensive tasks during GPU rendering. Since high-quality real-time rendering is often GPU-intensive, this approach allows the FPS to be solely related to rendering, achieving the fastest rendering speed.

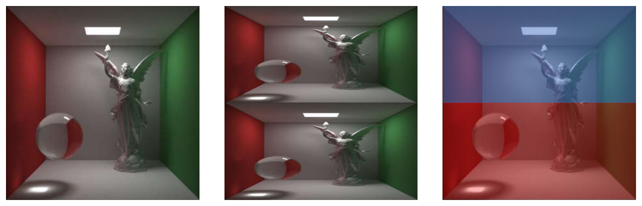


Figure 7: The reason for customizing the shader in Vulkan. (The different colored masks highlight the area rendered by different GPUs.)

3.3 Shader Customization for Compromising Interfaces Provided

The ray generation shader in Vulkan delineates the requisite processes to engender rays for individual pixels, while the driver encapsulates a nested loop iterating across all pixels. The driver furnishes the shader with *gl_LaunchSizeEXT* and *gl_LaunchIDEXT*, representing the overall image size and the current pixel's position being rendered.

Ideally, a modification or an interface in the driver would enable ray tracing use-cases to select a portion of the window to render — similar to the render pass offered for rasterization techniques. However, *vkCmdTraceRaysKHR* (Vulkan API for ray tracing rendering) can only specify the total size of the window so that the same shader can only generate two congruent scaled images instead of a half-window image, as shown in Fig. 7.

Although modifying the driver constitutes an elegant solution, obtaining permission from non-open-source drivers to implement this architecture on all GPUs may pose challenges. Consequently, as a provisional solution, gVulkan provides a shader customizer that dynamically modifies shaders to achieve rendering splits.

As shown in Fig. 8, the shader received by Vulkan is in the compiled SPIR-V format. To transform them into the target file, the shader customizer comprises three components: a compiling module, Cshader, and a decompiling module.

The decompiling module translates binary files into more human-readable and writable GLSL files. Cshader is a shader code generator that produces a corresponding shader based on each GPU's workload. The compiling module is responsible for recompiling the GLSL files into SPIR-V files, which are subsequently submitted to Vulkan for processing.

3.4 Latency-determined Load Balancing

Although gVulkan proposes a dependency-decoupled parallel rendering solution, this method only addresses the efficiency issues within a single GPU. When multiple GPUs

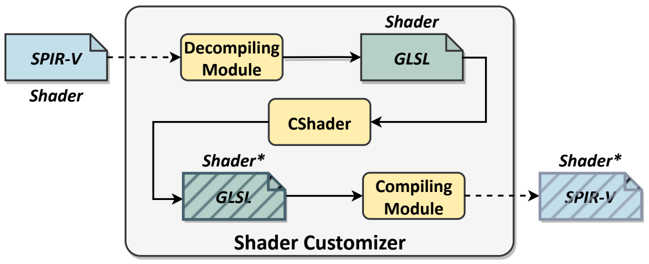


Figure 8: Workflow for the shader customizer.

work simultaneously, there can still be situations where some GPUs slow down, causing the other GPUs to be waiting. As shown in Fig. 9, evenly distributing tasks initially allows multiple GPUs to start rendering with minimal delay. However, suppose a GPU suddenly receives additional tasks from another program, causing an increase in delay. In that case, it forces the other GPUs to wait until the rendering task is completed before starting the next frame. Even though the *swapchain* can facilitate the early rendering of the next frame, ultimately, it still results in waiting for the slowest GPU. The "bucket effect" situation emphasizes the importance of dynamic rebalancing among the GPUs. Furthermore, during the copying phase, the primary GPU, due to waiting, does not minimize the rendering latency to the greatest extent, indicating that there is further room for optimization in terms of latency reduction.

We propose improvement suggestions to alleviate the impact on latency during both the rendering and copying phases. Firstly, we implement a heuristic algorithm to minimize latency generated during the rendering phase. Secondly, we consider the impact of the copying phase. We allocate more rendering responsibility to the primary GPU, equalizing its duration with the copy phase duration of other GPUs. Finally, we employ multi-threading techniques to parallelize the present phase, enhancing overall system performance.

Table 1: Definitions of Key Variables and Measures

Variable	Definition
g_i	The GPU_i which server provide
T_i	The current workload for GPU_i
NT_i	The future workload for GPU_i
RP_i	The rendering power of the GPU_i
CP_i	The unit performance for $GPU_i(i \neq 0)$ to copy the rendered image to the primary GPU
RL_i	The latency of rendering of the GPU_i
CL_i	The latency of copying of the GPU_i
α_i	The coefficient of NT_i

To optimize GPU resource utilization, we introduce a heuristic algorithm to allocate rendering tasks among the GPUs. The gVulkan-server can provide n GPUs, denoted as

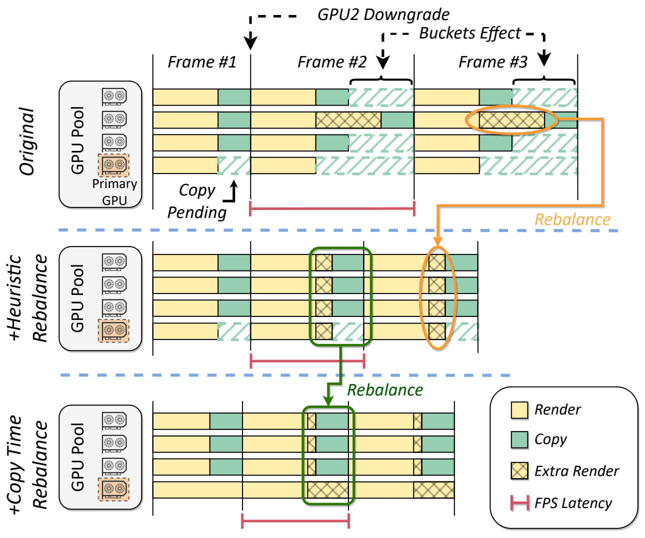


Figure 9: Based on the utilization of the rendering and copying phases, heuristic algorithms are proposed to maximize the use of GPU resources and minimize latency.

$G = \{g_1, g_2, ..., g_n\}$, for a given use-case. We represent the current workload of g_i as T_i and the future workload assigned to g_i as NT_i. Additionally, RP_i and RL_i indicate the rendering power and rendering latency of g_i, while CP_i and CL_i represent the unit power required to copy the rendered image to the primary GPU and the copying latency for $g_i (i \neq 0)$, respectively.

Table 1 provides the definitions of the key variables in the algorithm. Our objective is to maximize GPU resource utilization, which entails minimizing latency for each frame. This latency is determined by the maximum latency among all GPUs. Thus, the problem with the goal of latency minimization per frame can be formulated as follows:

$$\begin{aligned} \text{minimize} \quad & max\{\frac{NT_i}{RP_i} + \frac{NT_i}{CP_i}, \frac{NT_0}{RP_0}\}(i \neq 0) \\ \text{subject to} \quad & \forall i, NT_i > 0 \\ & \sum_{i=0}^{n} T_i = \sum_{i=0}^{n} NT_i = T_{total} \\ & \frac{T_i}{RP_i} = RL_i, \frac{T_i}{CP_i} = CL_i (i \neq 0) \end{aligned} \tag{1}$$

Equation 1 represents the objective function, which calculates the maximum delay among all GPUs. The constraint ensures that the sum of workload for all GPUs remains equal at all times and that every GPU resource is utilized. And RP_i and CP_i can be calculated by known values.

According to the properties of this convex function, the minimum value is attained when and only when:

$$\frac{NT_i}{RP_i} + \frac{NT_i}{CP_i} = \frac{NT_0}{RP_0}(i \neq 0) \tag{2}$$

Based on Equation 2, we derive the coefficient of NT_i as:

$$\begin{cases} \alpha_i = \dfrac{RL_i + CL_i}{T_i} \quad (i \neq 0) \\ \alpha_0 = \dfrac{RL_0}{T_0} \end{cases} \tag{3}$$

Since the variables constituting α_i are all known quantities, α_i is a constant factor. We observe that the workload g_i needs to allocate are proportional to each other:

$$NT_i : NT_j = \frac{1}{\alpha_i} : \frac{1}{\alpha_j} (i \neq j) \tag{4}$$

Consequently, we can determine the specific value of NT_i from this proportion:

$$NT_i = \frac{T_{total}}{\alpha_i * \sum_{j=0}^{n} (\frac{1}{\alpha_j})} \tag{5}$$

The workload allocation approach detailed above exhibits high scalability and can be readily extended to accommodate any number of GPUs. By efficiently computing the workloads of all GPUs with O (1) time complexity and dynamically allocating the workload distribution based on workload disparities, our system fully exploits the resources of the GPU pool, thereby achieving optimal performance in rendering images.

Latency-determined adaptive load balancing mechanism. With the above algorithm, gVulkan introduces a latency-determined adaptive load balancing mechanism. After obtaining the rendering and copying latency of GPUs through the Timer, gVulkan calculates the optimal allocation scheme based on the algorithm mentioned above. It then divides the workload among each GPU according to pixel-grained. The number of rays in a ray tracing use case determines the amount of workload, and the number of rays is related to the number of pixels. Therefore, it is feasible to split the rendering workload in terms of pixel-grained.

Nonetheless, GPU latency may experience brief fluctuations due to unforeseen conditions. If such changes are detected by the splitter and cause alterations in task allocation, system jitter may occur.

To mitigate this issue, the splitter incorporates a stable unit, which enhances the system in two ways. First, it employs a threshold: workload changes are implemented when the difference between the currently assigned workload and the workload to be assigned surpasses a predetermined threshold. Second, it utilizes a multi-check approach: even if the workload exceeds a certain threshold, the switch is not executed immediately. Instead, the workload is altered when the behavior of surpassing the threshold occurs sequentially several times. Through these methods, the splitter reduces system jitter and bolsters the stability and reliability of the system.

3.5 Resource-classified Transparent Forwarding

To address local limitations and achieve ecological compatibility discussed in Section 2.2, gVulkan introduces Resource-classified transparent forwarding. This technology takes into account the differences between various APIs [8], accelerating the server-side offloading of API-related resources on the local side. It allows some resources to use the server-side real values directly while others are presented to the application as virtual resources through the mirage zone. It also informs the server of their corresponding resources in the oasis zone to achieve transparent utilization of remote resources.

To speed up the transformation rate and avoid intolerable latency, gVulkan categorizes Vulkan APIs into three types based on their characteristics: Context APIs, handle APIs, and requirement APIs. Context APIs set context information and only need to return whether the setting was successful. Most APIs fall into this category. Handle APIs require obtaining a resource handle and returning this handle for subsequent usage. Resources such as queues, buffers, images, fences, and pipelines all have their own specific invocation handles. Requirement APIs obtain specific requirements of the current GPU for use in later computations. These computations occur between API calls, and it is challenging to predict how the use-case will utilize the requirements provided by the GPU.

For instance, when a Vulkan application calls *vkCreateBuffer* (handle type), the API returns a buffer handle and a signal indicating success. When *vkGetBufferMemoryRequirements* (requirement type) is called, the API returns the buffer's memory requirements for the current GPU, and the application allocates memory based on this information. The *vkBindBufferMemory* (context type) binds the buffer and memory together and returns only a success or failure signal.

Although handle APIs must return handles to the use-case, these handles are merely binary values, and these values do not involve any computation. To minimize data transfer between the gVulkan-local and gVulkan-server, we introduce the *mirage zone* to store virtual handles that are returned to the use-case locally [8]. The use-case uses the virtual handle as the resource handle for subsequent operations. When the use-case utilizes this handle to manipulate resources, the virtual handle is converted into the real handle, which is stored in the *oasis zone* on the gVulkan-server. Similarly, resources for context API are stored in *mirage zone* like virtual handles. Only the requirement API needs to return GPU requirements in real time.

The use-case is unaware of the processing that gVulkan performs on its APIs and merely obtains the handle and requirements provided by the GPU for subsequent calculations. However, the virtualized resources perceived by the use-case have been divided and processed in the physical environment on the remote side, accelerating rendering, increasing FPS,

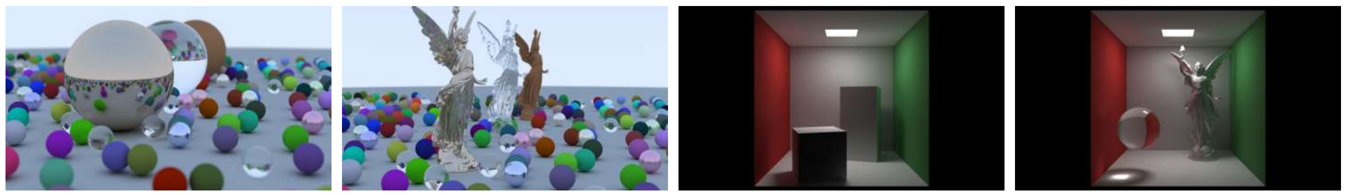

(a) Outdoor-Simple (b) Outdoor-Lucy (c) Cornell-Simple (d) Cornell-Lucy

Figure 10: Scenes that experimental use.

and enhancing image fluidity. Therefore, gVulkan is seamlessly integrated, easily blending into the existing application ecosystem.

4 Implementation

4.1 gVulkan Interceptor

gVulkan use Vulkan layer, a mechanism provided by the Vulkan API, to set up the gVulkan interceptor. Compared to the approach of intercepting dynamic libraries through *LD_PRELOAD*, the gVulkan Interceptor is more closely aligned with Vulkan applications and can intercept the Vulkan API for statically compiled applications. Since the Layer mechanism is provided by Vulkan itself, it does not require additional processing for applications, operating systems, or drivers.

4.2 Media Engine

All GPU-generated images are merged in the framebuffer and sent back to the client via the network for display. Using FFmpeg, the framebuffer can be compressed to reduce the amount of data transmitted and alleviate the pressure on network bandwidth. However, encoding and decoding also consume time and computational resources, so the specific codec needs to be considered based on the scenario.

Additionally, many applications use high-level APIs like Graphics Library Framework (GLFW) to create windows, but Vulkan cannot intercept the window handles from GLFW and directly modify them. This leads to the need to open a new window for rendering, which does not achieve true application-unawareness. gVulkan has discovered that such high-level APIs like Graphics Library Framework GLFW eventually call low-level APIs like X C Binding (XCB), and Vulkan can obtain the window handles from XCB APIs. Therefore, gVulkan writes the returned framebuffer into the XCB window and refreshes it to display the rendered images in the original window, achieving true application-unawareness.

4.3 Prototype Implementation

Due to the complex nature of nested structures and pointer-intensive parameters in the Vulkan API, engineering implementation is prone to bugs. To address this issue, gVulkan employs protobuf for serialization and deserialization of information to be transmitted.

Utilizing protobuf enhances understandability, reduces the occurrence of errors, and minimizes network bandwidth consumption. Currently, gVulkan supports 94 critical APIs through more than 30k lines of C code. These 94 APIs encompass all 28 ray tracing extension APIs, 18 resource allocation APIs, 18 resource release APIs, 14 rendering drawing APIs, 6 GPU resource acquisition APIs, 6 synchronization APIs, 3 resource binding APIs, and 1 resource update API. gVulkan is capable of supporting the classic Vulkan ray tracing benchmark, RayTracingInVulkan[2], which allows customization of the various scenes used to display.

Besides that, according to the nature of ray tracing, server-side tasks in gVulkan are independent of one another, making them easily parallelizable through multi-threading. Upon receiving commands, the gVulkan-server promptly partitions the workload into multiple segments and assigns them to threads corresponding to the number of GPUs in use.

5 Evaluation

Currently, no publicly available support with a multi-GPU architecture for Vulkan ray tracing exists. So in this section, we analyze gVulkan to answer the following questions:

- How is the performance of gVulkan? What amount of resources is required to achieve such a performance? Is gVulkan scalable? (5.2)
- Can gVulkan balance the workload in a timely manner when the GPU performance changes? How effective is the dynamic self-rebalancing? (5.3)
- How much improvement in FPS can be achieved by employing a transparent resource pool? (5.4)
- Does using gVulkan affect image quality? (5.5)

Based on the questions, the comparative study is mainly focused on four aspects of gVulkan: 1) the performance and scalability of gVulkan, 2) the effect of dynamic rebalancing of gVulkan, and 3) the effect of TRP on FPS. 4) the image quality of gVulkan.

[2]https://github.com/GPSnoopy/RayTracingInVulkan

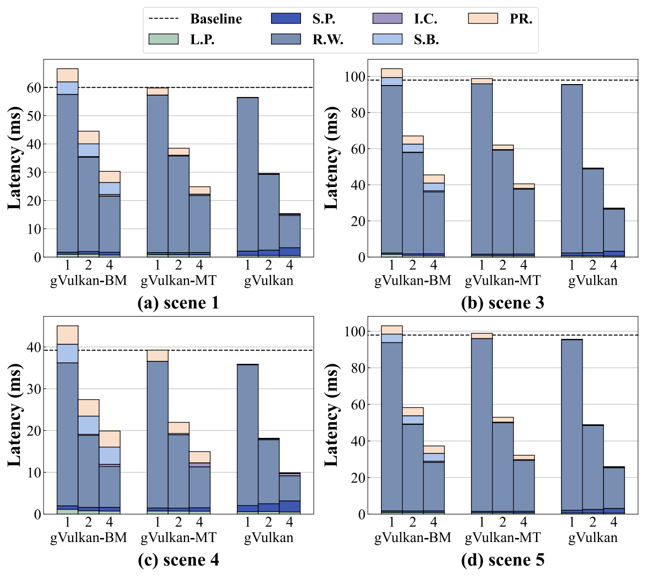


Figure 11: The latency of different phases for scenes.

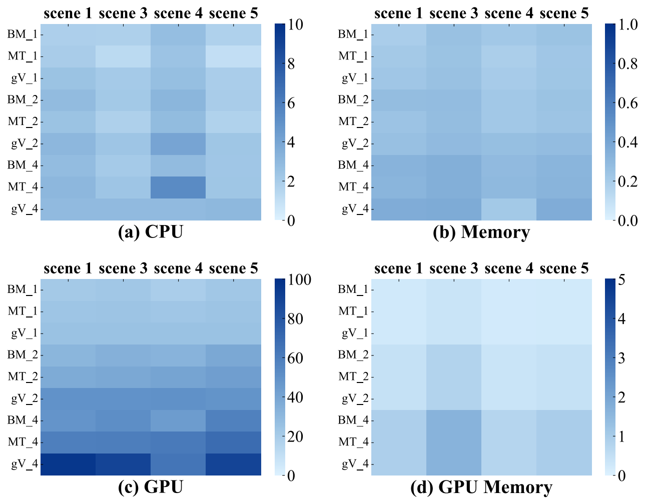


Figure 12: Use-cases resource cost for gVulkan-BM (BM), gVulkan-MT (MT) and gVulkan (gV).

5.1 Experimental Setup

To measure the effect of gVulkan, we design four sets of experiments as evaluation.

Configurations: We configured the experiments with Intel® Xeon® Platinum 8163 CPU @2.50GHz, 4 GPUs with T4 and 372 GiB DRAM. We conduct all experiments in Alibaba Cloud (ecs.gn6i-c24g1.24xlarge), which runs on the Ubuntu 22.04. We locked the frequency of GPUs to guarantee performance consistency.

Benchmarks: We used a popular Vulkan Ray-Tracing Benchmark called RayTracingInVulkan as gVulkan's benchmark. RayTracingInVulkan can be tested using a variety of customized scenes. gVulkan used four of the scenes it provides as the benchmark for this experiment. The scenes are shown in Fig. 10. Fig. 10a depicts the basic scene from "Ray Tracing in One Weekend," an outdoor setting. Fig. 10b builds upon this by adding a statue called Lucy, increasing the scene's complexity. Fig. 10c illustrates a Cornell box scene, an indoor setting. Fig. 10d builds on 10c by also incorporating a statue named Lucy, further increasing the scene's complexity.

Baseline: As gVulkan is the first architecture that employs multiple GPUs to partition the Vulkan ray tracing use-cases and leverages a remote resource pool to accelerate local rendering tasks, we evaluate the performance of the use-cases executed on the local GPU as a baseline for comparison. Meanwhile, to present the breakdown optimizations within the gVulkan, we propose two methods: gVulkan-BM and gVulkan-MT. gVulkan-BM only implements the basic rendering splitting function, and gVulkan-MT implements multi-threading on the basis of gVulkan-BM.

5.2 Performance and Scalability of gVulkan

Evaluation 1 is a comprehensive comparison of gVulkan operating on varying numbers of GPUs and under different scenes, aiming to assess resource cost, Quality of Service (QoS), and scalability.

Fig. 11 illustrates the latency of two use-cases at two resolutions utilizing 1, 2, and 4 GPUs, divided into six components: layer pre-processing (L.P.), server pre-processing (S.P.), render waiting (R.W), image copying (I.C.), send backing (S.B.) and presenting (Pr.). The dashed line of the baseline represents the latency experienced when operating a single local GPU, simulating the performance before implementing the gVulkan architecture.

The R.W., S.B., and Pr. phases have a significant impact on latency. In different scenes, the proportions of these stages vary slightly. The latency in the S.B. phase can be addressed through multi-threading, the latency in the Pr. phase can be completely eliminated through API streaming rewriting, and the latency in the R.W. phase can be proportionally reduced as the number of GPUs increases. The latency of other phases remains relatively constant, fluctuating within a few milliseconds. While using a single GPU, the local application fails to reach the 30 FPS threshold as shown in Fig. 11a, but with 2 GPUs, gVulkan successfully met the Basic QoS guarantee. Moreover, with 4 GPUs, gVulkan successfully achieved the QoS guarantee of 60 FPS. The average speedup for the four scenes under the gVulkan with 4 GPUs can reach 3.81.

Fig. 12 shows the CPU, GPU, memory, and GPU memory utilization rates (normalized to the percentage of 100% hardware capability) for these four scenes at 720p resolution. We conclude that gVulkan exhibits low resource costs in terms of CPU, memory, and GPU memory while effectively employing available GPU resources for rendering acceleration.

In addition to this, the resolution of the image and the

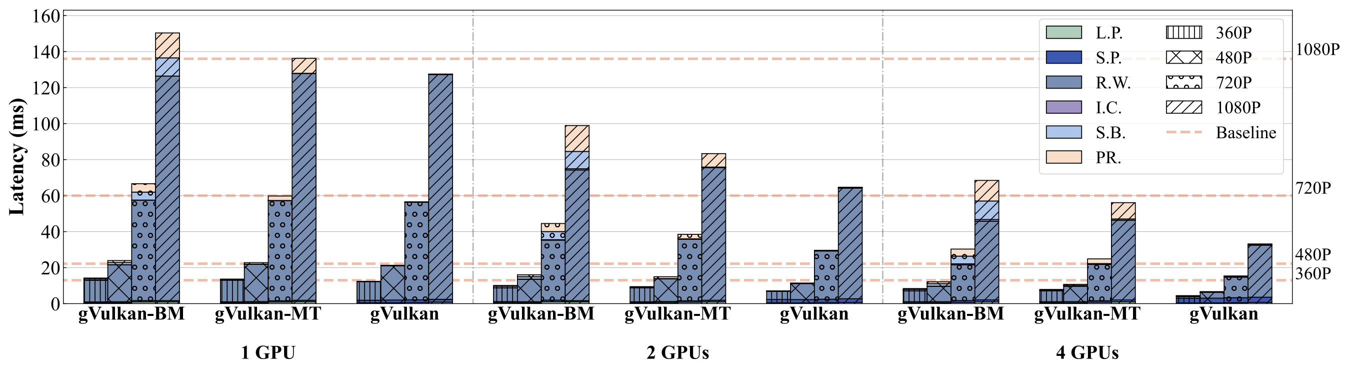


Figure 13: Compare the latency of each phase at different resolutions.

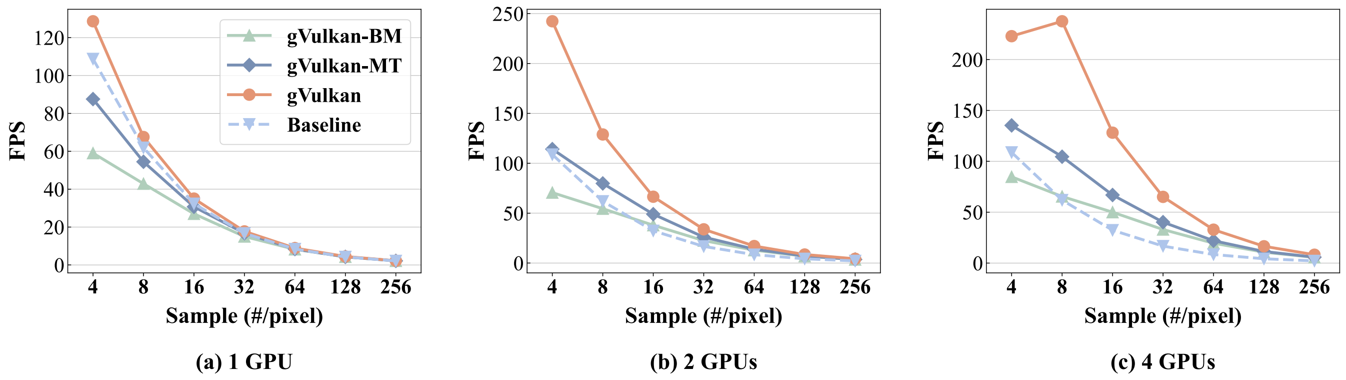


Figure 14: Contrast FPS with SPP changes.

number of rays sampled per pixel point also have an impact on the rendering latency.

Fig. 13 compares the FPS of the outdoor-simple scene at different resolutions. As the resolution increases, both the latency required for rendering and the latency needed for presentation are continuously increasing, resulting in a decrease in FPS. gVulkan-BM, gVulkan-MT, and gVulkan all reduce the latency in multi-GPU scenarios, but it is clear that gVulkan achieves an almost linear reduction effect, especially in higher-resolution scenes.

Fig. 14 compares the FPS degradation of the different methods with an increasing number of light samples per pixel for different numbers of GPUs and the comparison with local rendering. The gVulkan scheme outperforms the other two methods as well as the local runtime results, regardless of the number of GPUs. When the FPS exceeds 250, the total FPS is determined by the pre-process because the computation time of phases such as L.P., S.P., and I.C. is higher than the rendering time, resulting in the scheme not achieving a linear increase at a higher FPS.

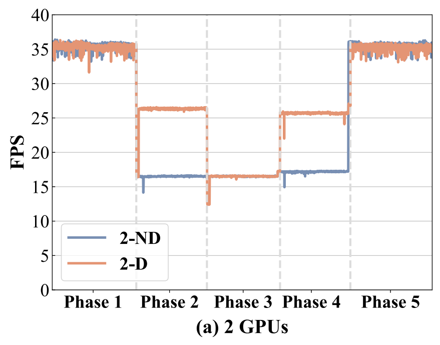
In terms of gVulkan's scalability, the Pr. and S.B. phases can be negligible through present elimination and dependency-decoupled parallel rendering, and the R.W. phase workload can be linearly diminished. The cumulative latencies of the L.P., S.P., and I.C. phases persist below 5 ms as a fixed duration. Since our goal is to accelerate high-quality real-time rendering to ensure QoS ($\sim$60 FPS), the required rendering latency is generally high. However, before the rendering time falls below the computation time for pre-processing, computational phases such as L.P., S.P., and I.C. do not affect the total FPS. Therefore, we believe that gVulkan can linearly reduce latency.

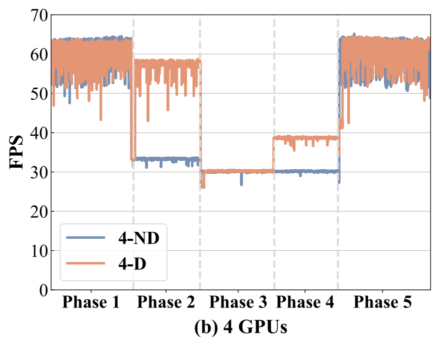
5.3 Dynamic Rebalance of gVulkan

Evaluation 2 is a comparison of gVulkan running multiple GPUs to assess the impact of dynamic rebalancing and system jitter. We compare the FPS of gVulkan with and without dynamic rebalancing using 2 and 4 GPUs, and examine the effects of integrating different stable units.

Fig. 15 shows the FPS and latency for each GPU across five phases: ①uniform rendering power, ②decrease in single GPU rendering power, ③decrease in all GPU rendering power, ④increase in single GPU rendering power, and ⑤increase in all GPU rendering power.

As shown in Fig. 15a and 15b, a sudden drop in the rendering power of a single GPU can significantly impact FPS. However, dynamic rebalancing can swiftly reallocate the workload to utilize the resources of each GPU fully. When the rendering power of a single GPU suddenly increases, dynamic rebalancing assigns an additional portion of the work to that GPU, improving FPS compared to an unoptimized state. Dynamic rebalancing functions effectively for





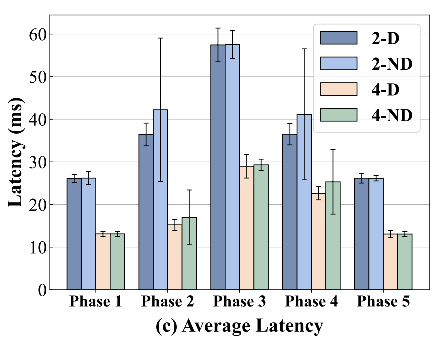


Figure 15: Fps and latency from dynamically adjusting of gVulkan in 2 GPUs w/ dynamic rebalancing (2-D), 2 GPUs w/o dynamic rebalancing (2-ND), 4 GPUs w/ dynamic rebalancing (4-D) and 4 GPUs w/o dynamic rebalancing (4-ND).

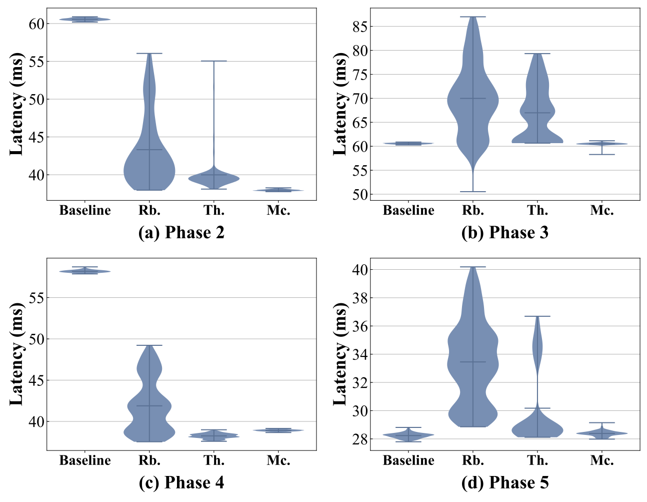


Figure 16: The fluctuation of different dynamic solutions in the latter four phases.

any number of GPUs. Additionally, latency jitter in Fig. 15a and 15b mainly stems from the test environment running at over 30 FPS.

Fig. 15c demonstrates the maximum latency of all GPUs for 2 GPUs with dynamic rebalancing (2-D), 2 GPUs without dynamic rebalancing (2-ND), 4 GPUs with dynamic rebalancing (4-D), 4 GPUs without dynamic rebalancing (4-ND), and the maximum value of GPU latency differences upon stabilization. In phases ①, ③, and ⑤, the rendering power of all GPUs is consistent, making dynamic rebalancing ineffective in reducing latency and instead causing minor latency fluctuations. In phases ② and ④, GPUs exhibit non-uniform rendering power, and dynamic rebalancing significantly reduces maximum latency while substantially decreasing the maximum difference between all latencies.

We evaluate the influence of stable units on dynamic rebalancing in reducing fluctuations during the latter four phases. Fig. 16 compares the fluctuations of gVulkan without dynamic rebalance (Baseline), dynamic rebalancing without a stable unit (Rb.), dynamic rebalancing with a stable unit featuring only a threshold function (Th.), and dynamic rebalancing with a completely stable unit (Mc.) across these phases. It is evident that stable units can effectively reduce fluctuations and latency.

5.4 FPS Optimization of Transparent Resource Pool

Evaluation 3 is a comparison of gVulkan, aimed to evaluate the FPS optimizations brought by TRP. We use Linux's Traffic Control, *tc*, to artificially simulate the network latency.

Instead of sending messages to the server individually, TRP facilitates message batching for concurrent dispatch. Fig. 17 depicts the FPS difference between individually sending command requests and dispatching command requests in batches under diverse network latency conditions. With increasing network latency, the FPS optimization advantages offered by batching become increasingly evident.

Table 2: PSNR for each scene.

	360P	**480P**	**720P**	**1080P**
Outdoor-Simple	54.60	54.63	54.68	54.68
Outdoor-Lucy	41.64	41.70	42.06	42.07
Cornell-Simple	30.25	30.25	31.52	38.78
Cornell-Lucy	37.11	37.14	38.41	38.26

5.5 Image Quality of gVulkan

While it is possible to speed up the acquisition of images through gVulkan, is it possible to guarantee the quality of the images at the same time?

In this experiment, we compare the difference between images rendered locally and by gVulkan through a classic image quality assessment index, Peak Signal-to-Noise Ratio (PSNR). To be fair, the rendered images are after 10000 rays are emitted, regardless of the rendering rate. Table 2 shows the PSNR values of the gVulkan rendered images compared to the locally rendered images in four different scenes with four different resolutions. All PSNR values are greater than 30, which shows that gVulkan can improve the rendering rate while maintaining image quality.

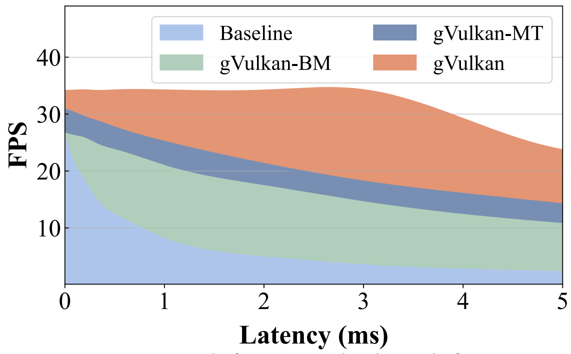


Figure 17: FPS among different methods in different network latency.

6 Related Work

6.1 Cloud Offloads

GPUs have become a popular choice for accelerating compute-intensive tasks in public clouds due to their high computing capacities and accuracy [33]. Various works have offloaded workloads to the cloud for scalability, resilience, and cost-efficiency, including thin-client architecture and cloud-edge cooperation.

Thin-client architecture is widely used in many cloud-gaming systems [3]. This architecture migrates all the applications to the cloud [16], and the client only serves as a displayer, leaving the client-side performance wasted.

Compared with the thin-client architecture, cloud-edge cooperation aims to utilize both server-side and client-side resources fully. One of the representative technologies in cloud-edge cooperation is API-forwarding. For example, gRemote [32] supports OpenGL command forwarding and enables the full utilization of client-side resources and achieving server-side CPU-GPU workload balance. ShareRender [43] intercepts graphics APIs, including Direct3D and OpenGL, and offloads graphics workloads directly to GPUs to optimize GPU usage. DroidCloud and CARE address the system-resource redundancy issues, leveraging resources in another dimension [14, 31]. Recently, more detailed cloud-edge workload partitioning based on API-forwarding has been explored in VR applications to achieve effective collaborative rendering, including both foreground-background partitioning [13, 18, 22] and foveal partitioning [39].

Furthermore, Secure containers [2, 15] are the current direction of development in cloud computing and are widely used. They offer higher isolation compared to containers and are lighter in weight compared to virtual machines. However, current secure containers cannot utilize GPUs, which imposes some limitations on their use cases. gVulkan can enhance secure containers with API forwarding capabilities so that GPU cloud offload solutions can be implemented in these containers.

6.2 Ray Tracing Acceleration

In recent years, there have been several efforts to improve the performance of ray tracing. To solve the problem of irregular memory access for ray tracing, STRaTA [12] proposes a hardware architecture that uses a streaming data model and a reconfigured ray stream memory to reduce energy consumption in massively parallel graphics processors. Dual Streaming [30], separates memory access of ray tracing into two streams (a scene stream and a ray stream) to minimize memory access conflicts between the two streams. Mach-RT [36] combines a new hardware architecture with a new ray ordering scheme to reduce the memory access bottleneck of ray tracing. Garanzha addresses the issue of incoherent reflection rays in GPU ray tracing. This work proposes an approach that organizes reflection rays into coherent packets to improve memory coherence and GPU SIMT efficiency [9]. Faced with the same problem, Liu approaches the problem differently [17]. It introduces a new ray intersection predictor in the GPU to eliminate redundant operations and directly evaluate primitives that may intersect rays.

However, all the current work mainly focuses on hardware-level support and requires specific hardware support or significant modifications to modern GPU workflow. To the best of our knowledge, gVulkan is the first cloud-rendering solution for Vulkan-based ray tracing. The above optimization works are not in conflict with gVulkan. Combined with the aforementioned hardware optimizations, gVulkan can deploy advanced hardware on the cloud and fully utilize scalable cloud hardware resources to achieve efficient ray tracing rendering.

7 Conclusion

In this paper, we introduce gVulkan, the first multi-GPU acceleration rendering solution with high scalability and dynamic self-rebalancing for Vulkan-based ray tracing rendering. We introduced resource-classification transparent forwarding to accelerate resource offloading, utilized GPU resources efficiently through dependency-decoupled parallel rendering, and adopted a latency-determined adaptive load balancing mechanism to allocate workloads at pixel granularity. Furthermore, we circumvented the limitations of existing interface designs by customizing shaders. Our experimental results show that gVulkan achieves high linear speedup, scalability, and dynamic rebalancing of workload across GPUs.

In our future work, we will make more efficient use of local resources while utilizing server-side resources, and also address scenarios when failures occur.

Acknowledgement

We sincerely thank our anonymous reviewers for their helpful comments and suggestions. This work was supported in part by National NSF of China (No. 62141218), the HighTech Support Program from STCSM (No.22511106200), the cooperation project from Ant Group ("Wasm-enabled Managed language in security restricted scenarios"), and Shanghai Key Laboratory of Scalable Computing and Systems.

References

[1] Zainul Abi Din, Panagiotis Tigas, Samuel T King, and Benjamin Livshits. {PERCIVAL}: Making {In-Browser} perceptual ad blocking practical with deep learning. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 387–400, 2020.

[2] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight virtualization for serverless applications. In *17th USENIX symposium on networked systems design and implementation (NSDI 20)*, pages 419–434, 2020.

[3] Ricardo A. Baratto, Leonard N. Kim, and Jason Nieh. THINC: a virtual display architecture for thin-client computing. In Andrew Herbert and Kenneth P. Birman, editors, *Proceedings of the 20th ACM Symposium on Operating Systems Principles 2005, SOSP 2005, Brighton, UK, October 23-26, 2005*, pages 277–290. ACM, 2005.

[4] R. Barringer, M. Andersson, and T. Akenine-Möller. Ray accelerator: Efficient and flexible ray tracing on a heterogeneous architecture. *Computer Graphics Forum*, 36(8):166–177, 2017.

[5] Benedikt Bitterli, Chris Wyman, Matt Pharr, Peter Shirley, Aaron Lefohn, and Wojciech Jarosz. Spatiotemporal reservoir resampling for real-time ray tracing with dynamic direct lighting. *ACM Transactions on Graphics (TOG)*, 39(4):148–1, 2020.

[6] Sangjin Choi, Taeksoo Kim, Jinwoo Jeong, Rachata Ausavarungnirun, Myeongjae Jeon, Youngjin Kwon, and Jeongseob Ahn. Memory harvesting in {Multi-GPU} systems with hierarchical unified virtual memory. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 625–638, 2022.

[7] Oscar Ferraz, Paulo Menezes, Vitor Silva, and Gabriel Falcao. Benchmarking vulkan vs opengl rendering on low-power edge gpus. In *2021 International Conference on Graphics and Interaction (ICGI)*, pages 1–8. IEEE, 2021.

[8] Di Gao, Hao Lin, Zhenhua Li, Chengen Huang, Yunhao Liu, Feng Qian, Liangyi Gong, and Tianyin Xu. Trinity:{High-Performance} mobile emulation through graphics projection. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 285–301, 2022.

[9] Kirill Garanzha and Charles Loop. Fast Ray Sorting and Breadth-First Packet Traversal for GPU Ray Tracing. *Computer Graphics Forum*, 2010.

[10] A. V. Pascal Grosset, Manasa Prasad, Cameron Christensen, Aaron Knoll, and Charles Hansen. Tod-tree: Task-overlapped direct send tree image compositing for hybrid mpi parallelism and gpus. *IEEE Transactions on Visualization and Computer Graphics*, 23(6):1677–1690, 2017.

[11] Jacob Haydel, Cem Yuksel, and Larry Seiler. Locally-adaptive level-of-detail for hardware-accelerated ray tracing. *ACM Transactions on Graphics (TOG)*, 42(6):1–15, 2023.

[12] Daniel Kopta, Konstantin Shkurko, Josef Spjut, Erik Brunvand, and Al Davis. An energy and bandwidth efficient ray tracing architecture. In *Proceedings of the 5th High-Performance Graphics Conference*, HPG '13, page 121–128, New York, NY, USA, 2013. Association for Computing Machinery.

[13] Zeqi Lai, Y. Charlie Hu, Yong Cui, Linhui Sun, and Ningwei Dai. Furion: Engineering high-quality immersive virtual reality on today's mobile devices. In *Proceedings of the 23rd Annual International Conference on Mobile Computing and Networking*, MobiCom '17, page 409–421, New York, NY, USA, 2017. Association for Computing Machinery.

[14] Linsheng Li, Bin Yang, Cathy Bao, Shuo Liu, Randy Xu, Yong Yao, Mohammad R Haghighat, Jerry W Hu, Shoumeng Yan, and Zhengwei Qi. Droidcloud: Scalable high density androidtm cloud rendering. In *Proceedings of the 28th ACM International Conference on Multimedia*, pages 3348–3356, 2020.

[15] Zijun Li, Jiagan Cheng, Quan Chen, Eryu Guan, Zizheng Bian, Yi Tao, Bin Zha, Qiang Wang, Weidong Han, and Minyi Guo. {RunD}: A lightweight secure container runtime for high-density deployment and high-concurrency startup in serverless computing. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 53–68, 2022.

[16] Li Lin, Xiaofei Liao, Guang Tan, Hai Jin, Xiaobin Yang, Wei Zhang, and Bo Li. Liverender: A cloud gaming system based on compressed graphics streaming. In *Proceedings of the 22nd ACM international conference on Multimedia*, pages 347–356, 2014.

[17] Lufei Liu, Wesley Chang, Francois Demoullin, Yuan Hsi Chou, Mohammadreza Saed, David Pankratz, Tyler Nowicki, and Tor M. Aamodt. Intersection prediction for accelerated gpu ray tracing. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '21, page 709–723, New York, NY, USA, 2021. Association for Computing Machinery.

[18] Xing Liu, Christina Vlachou, Feng Qian, Chendong Wang, and Kyu-Han Kim. Firefly: Untethered multi-user VR for commodity mobile devices. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 943–957. USENIX Association, July 2020.

[19] Michael Lujan, Michael McCrary, Blake W Ford, and Ziliang Zong. Vulkan vs opengl es: Performance and energy efficiency comparison on the big. little architecture. In *2021 IEEE International Conference on Networking, Architecture and Storage (NAS)*, pages 1–8. IEEE, 2021.

[20] Bingzheng Ma, Ziqiang Zhang, Yusen Li, Wentong Cai, Gang Wang, and Xiaoguang Liu. Spider: An effective, efficient and robust load scheduler for real-time split frame rendering. In *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 672–682. IEEE, 2022.

[21] Daniel Meister, Shinji Ogaki, Carsten Benthin, Michael J Doyle, Michael Guthe, and Jiří Bittner. A survey on bounding volume hierarchies for ray tracing. In *Computer Graphics Forum*, volume 40, pages 683–712. Wiley Online Library, 2021.

[22] Jiayi Meng, Sibendu Paul, and Y. Charlie Hu. Coterie: Exploiting frame similarity to enable high-quality multiplayer vr on commodity mobile devices. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 923–937, New York, NY, USA, 2020. Association for Computing Machinery.

[23] Zili Meng, Tingfeng Wang, Yixin Shen, Bo Wang, Mingwei Xu, Rui Han, Honghao Liu, Venkat Arun, Hongxin Hu, and Xue Wei. Enabling high quality {Real-Time} communications with adaptive {Frame-Rate}. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 1429–1450, 2023.

[24] S. Molnar, M. Cox, D. Ellsworth, and H. Fuchs. A sorting classification of parallel rendering. *IEEE Computer Graphics and Applications*, 14(4):23–32, 1994.

[25] Brendan Moloney, Marco Ament, Daniel Weiskopf, and Torsten Moller. Sort-first parallel volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 17(8):1164–1177, 2011.

[26] Jae-Ho Nah, Hyuck-Joo Kwon, Dong-Seok Kim, Cheol-Ho Jeong, Jinhong Park, Tack-Don Han, Dinesh Manocha, and Woo-Chan Park. Raycore: A ray-tracing hardware architecture for mobile devices. *ACM Trans. Graph.*, 33(5), sep 2014.

[27] David Pankratz, Tyler Nowicki, Ahmed Eltantawy, and José Nelson Amaral. Vulkan vision: Ray tracing workload characterization using automatic graphics instrumentation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 137–149. IEEE, 2021.

[28] Xiaowei Ren and Mieszko Lis. Chopin: scalable graphics rendering in multi-gpu systems via parallel image composition. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 709–722. IEEE, 2021.

[29] Mohammadreza Saed, Yuan Hsi Chou, Lufei Liu, Tyler Nowicki, and Tor M Aamodt. Vulkan-sim: A gpu architecture simulator for ray tracing. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 263–281. IEEE, 2022.

[30] Konstantin Shkurko, Tim Grant, Daniel Kopta, Ian Mallett, Cem Yuksel, and Erik Brunvand. Dual streaming for hardware-accelerated ray tracing. In *Proceedings of High Performance Graphics*, HPG '17, New York, NY, USA, 2017. Association for Computing Machinery.

[31] Dongjie Tang, Cathy Bao, Yong Yao, Chao Xie, Qiming Shi, Marc Mao, Randy Xu, Linsheng Li, Mohammad R Haghighat, Zhengwei Qi, et al. Care: Cloudified android oses on the cloud rendering. In *Proceedings of the 29th ACM International Conference on Multimedia*, pages 4582–4590, 2021.

[32] Dongjie Tang, Yun Wang, Linsheng Li, Jiacheng Ma, Xue Liu, Zhengwei Qi, and Haibing Guan. gremote: Api-forwarding powered cloud rendering. In Manish Parashar, Vladimir Vlassov, David E. Irwin, and Kathryn Mohror, editors, *HPDC '20: The 29th International Symposium on High-Performance Parallel and Distributed Computing, Stockholm, Sweden, June 23-26, 2020*, pages 197–201. ACM, 2020.

[33] Kun Tian, Yaozu Dong, and David Cowperthwaite. A full {GPU} virtualization solution with mediated {Pass-Through}. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 121–132, 2014.

[34] Lars Olav Tolo, Ivan Viola, Atle Geitung, Harald Soleim, and Daniel Patel. Multi-gpu rendering with the open vulkan api. In *Norsk IKT-konferanse for forskning og utdanning*, 2018.

[35] Will Usher, Ingo Wald, Jefferson Amstutz, Johannes Günther, Carson Brownlee, and Valerio Pascucci. Scalable ray tracing using the distributed framebuffer. *Computer Graphics Forum*, 38(3):455–466, 2019.

[36] E. Vasiou, K. Shkurko, E. Brunvand, and C. Yuksel. Mach-rt: A many chip architecture for ray tracing. In *Proceedings of the Conference on High-Performance Graphics*, HPG '19, page 1–6, Goslar, DEU, 2022. Eurographics Association.

[37] Luiz Velho, Vinicius da Silva, and Tiago Novello. Immersive visualization of the classical non-euclidean spaces using real-time ray tracing in vr. In *Graphics Interface 2020*, 2020.

[38] Jorge Luis Williams and Robert E Hiromoto. Sort-middle multi-projector immediate-mode rendering in chromium. In *VIS 05. IEEE Visualization, 2005.*, pages 103–110. IEEE, 2005.

[39] Chenhao Xie, Xie Li, Yang Hu, Huwan Peng, Michael Taylor, and Shuaiwen Leon Song. Q-vr: System-level design for future mobile collaborative virtual reality. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, page 587–599, New York, NY, USA, 2021. Association for Computing Machinery.

[40] Alex Yu, Ruilong Li, Matthew Tancik, Hao Li, Ren Ng, and Angjoo Kanazawa. Plenoctrees for real-time rendering of neural radiance fields. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 5752–5761, 2021.

[41] Haitang Zhang, Junchao Ma, Zixia Qiu, Junmei Yao, Mustafa A Al Sibahee, Zaid Ameen Abduljabbar, and Vincent Omollo Nyangaresi. Multi-gpu parallel pipeline rendering with splitting frame. In *Computer Graphics International Conference*, pages 223–235. Springer, 2023.

[42] Wei Zhang, Binghao Chen, Zhenhua Han, Quan Chen, Peng Cheng, Fan Yang, Ran Shu, Yuqing Yang, and Minyi Guo. {PilotFish}: Harvesting free cycles of cloud gaming with deep learning training. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 217–232, 2022.

[43] Wei Zhang, Xiaofei Liao, Peng Li, Hai Jin, and Li Lin. Sharerender: Bypassing GPU virtualization to enable fine-grained resource sharing for cloud gaming. In Qiong Liu, Rainer Lienhart, Haohong Wang, Sheng-Wei "Kuan-Ta" Chen, Susanne Boll, Yi-Ping Phoebe Chen, Gerald Friedland, Jia Li, and Shuicheng Yan, editors, *Proceedings of the 2017 ACM on Multimedia Conference, MM 2017, Mountain View, CA, USA, October 23-27, 2017*, pages 324–332. ACM, 2017.

[44] Yuhao Zhu. Rtnn: accelerating neighbor search using hardware ray tracing. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 76–89, 2022.