Zatel: Sample Complexity–Aware Scale–Model Simulation for Ray Tracing

Davit Grigoryan
University of British Columbia
Canada
davitggc@student.ubc.ca

Yuan Hsi Chou
University of British Columbia
Canada
yuanhsi@ece.ubc.ca

Tor M. Aamodt
University of British Columbia
Canada
aamodt@ece.ubc.ca

***Abstract*—Ray tracing is a computationally intensive rendering technique that simulates the behavior of light rays as they interact with objects in a scene. It is becoming increasingly popular in video games and is already the de facto standard for animated movies. However, current hardware still struggles to efficiently ray trace complex scenes and requires further research. To evaluate early-stage hardware proposals that accelerate ray tracing for GPUs, one either uses cycle-accurate simulators, which are highly accurate and flexible but slow, or other models that are an order of magnitude faster but provide limited output with high error margins. In this paper, we propose Zatel, a prediction methodology for evaluating GPU performance on ray tracing workloads. We observe that the desired metrics can be estimated with reasonable accuracy by only tracing a representative subset of pixels. Furthermore, the parallel nature of GPUs allows us to split the scene into chunks, which lets Zatel execute faster using downscaled GPU configurations. We incorporate these two optimization steps into Zatel and evaluate it on a benchmark suite for ray tracing using Vulkan-Sim, a cycle-accurate simulator. By relying on Vulkan-Sim, architectural changes are captured through the simulator, and Zatel does not need to be updated to support each change. Zatel records less than 1% error with 10× simulation time speedup for measuring simulation cycles on a mobile GPU.**

***Index Terms*—Ray Tracing, Simulation, Graphics processors, Modeling methodologies**

I. Introduction

Ray tracing is a rendering technique that simulates the path of light, creating photorealistic effects that are either difficult or near-impossible with rasterization. However, ray tracing is computationally expensive to execute and, thus, is primarily used in offline rendering applications such as films [1] and computer-aided design (CAD) [2]. Many advancements are currently being made in the hardware space such as the implementation of specialized accelerators so that real-time ray-tracing applications like video games can run at higher frame rates [3], [4].

When generating early-stage hardware proposals for ray tracing workloads, one way to evaluate them is through cycle-accurate simulators as they provide the most accurate and detailed profiling information while allowing high flexibility for GPU configurations. However, they are also very slow due to simulating a large number of GPU cores serially on a central processing unit (CPU) [5]–[7]. For instance, simulating a single 1080p frame from a real-world workload can take up to weeks to finish. Therefore, workload simulations can become a bottleneck during the early design phase, when fast decision-making is crucial [8]. The long simulation times also push researchers to evaluate their proposed architectural changes on miniature workloads [9], which do not adequately represent a real-world workload.

Such a demand resulted in the creation of performance models that are an order of magnitude faster but less accurate, which can be used to quickly evaluate different hardware ideas and choose the most optimal subset to investigate further. Current ones use either sampling or analytical modeling techniques to estimate the performance of the proposed architectures. The sampling method simulates a representative portion of the workload and extrapolates the output to approximate the final results [10]–[12]. Analytical models use traced information in combination with mathematical equations to predict the metrics [8], [13]–[15]. These performance models, however, also have their shortcomings. Most notably, they are not fitted for ray tracing as they cannot fully capture the complexity of such workloads, resulting in high mean absolute error (MAE) [16].

There have been some attempts to use analytical models for ray tracing workloads; however, they were designed for older generations of GPUs and can not capture the complexity of modern GPU architectures [17], [18]. Additionally, most of the models provide limited profiling output. For instance, the current state-of-the-art GPU analytical model, GCoM, proposed by Lee et al. [15] can only construct the cycles-per-instruction (CPI) stack and does not provide information on other metrics like cache miss rate and ray tracing unit utilization statistics, which are highly desirable for the design stage. Furthermore, when the microarchitecture changes, the performance models may also need to be drastically modified. MDM [14] was proposed as one of the first analytical models, GPUMech [13], did not adequately model memory-divergent applications. GCoM was released after GPUs underwent significant changes like dividing cores into sub-cores, which MDM did not capture.

In this paper, we present Zatel[1], a prediction methodology for evaluating GPU performance on ray tracing workloads that reduces the workload simulation bottleneck while avoiding many of the key problems observed in other performance

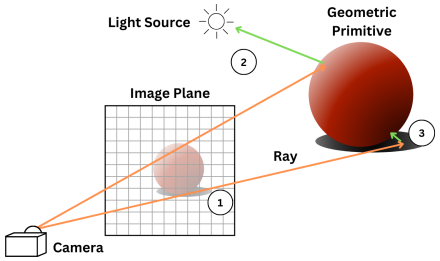

Fig. 1. Overview of ray tracing rendering method

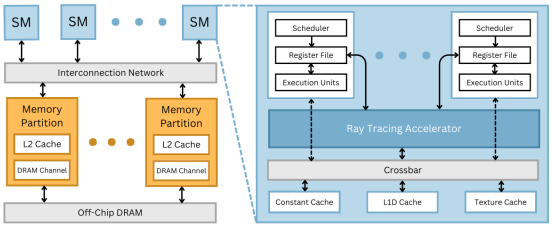

Fig. 2. Vulkan-Sim's modeled GPU architecture [7]

models. Zatel exploits the inherent parallelism of GPUs by partitioning the workload into groups and simulating each group concurrently using a downscaled GPU. It further reduces the simulation time by selecting representative pixels from each group to trace. Liu et al. [19] propose a similar technique for scaling down the CPU architecture. They reduce the number of cores and shared resources, then run the scaled-down model on the target workload without splitting it. Since they are not downsizing the workload, they also need to extrapolate the performance using machine learning. We offer a simpler methodology that runs the downscaled GPU on workload partitions and combines the results by either averaging or summing them. Zatel is also designed to use an already available cycle-accurate simulator, Vulkan-Sim [7], in its evaluations. Since the methodology uses the simulator at its core, it can estimate any metric that Vulkan-Sim provides, as desired by the user. Moreover, Zatel only requires minimal changes to represent new hardware proposals as most microarchitecture details are captured by Vulkan-Sim, which can be easily updated to follow trends in GPU design. Thus, our paper makes the following contributions:

1) We provide a prediction methodology for evaluating ray tracing workloads on GPUs.
2) We propose a performance estimation technique that is independent of the underlying GPU architecture.
3) We present a novel approach of GPU downscaling with the selection of representative pixels to predict the performance metrics.

II. Background & Motivation

A. Ray Tracing

Ray tracing is a rendering method that generates photorealistic images by simulating lighting. It models the path the light takes from the camera to the final intersecting object for each pixel of the image plane. We illustrate a simple example of ray tracing in Fig. 1 that shoots primary and secondary rays. The algorithm begins by first tracing the primary ray as shown in (1). If the ray hits an object in the scene, the red sphere, in this case, we cast a secondary ray towards the light source (shown in green). If the path of the secondary ray is unoccluded, like in (2), that means the pixel is not a shadow. As for the ray in (3), the casted secondary ray gets obstructed, which leads to a shadow. To reduce the search space for the ray intersecting the scene, the base geometric primitives are structured into an acceleration structure, typically as a bounding volume hierarchy (BVH) tree [20]. The BVH tree is a spatial tree-like structure where each subtree is a collection of geometric primitives tightly enclosed in an axis-aligned bounding box (AABB).

[1]Zatel in Armenian means both divide and separate, describing our two core optimization strategies.

B. Graphics Processing Units (GPUs)

GPUs are highly parallel devices capable of concurrently executing numerous threads. To downscale the target GPU, we need to understand its underlying architecture. The architecture, as depicted in Fig. 2, features Streaming Multiprocessors (SMs) consisting of execution units, L1D, texture, and constant cache, connected via a crossbar [21]. Modern GPUs incorporate ray tracing accelerators (RT units) in each SM to offload application-specific computations [22]. SMs connect to memory partitions via an interconnection network, and memory controllers manage an L2 cache, DRAM scheduler, and off-chip DRAM channel. Programs running on GPUs execute multiple threads in parallel, organized into warps of 32 scalar threads. In applications like ray tracing, each thread in a warp processes the same program with different data, such as unique ray directions for pixel tracing on the image plane.

C. Motivation

Current ray tracing programs face performance challenges due to the computational intensity of the rendering technique, causing bottlenecks in real-time applications. Thus, accelerating these programs requires further research and development on ray tracing accelerator units [23]. While building a GPU with a ray tracing accelerator for testing is an option, the process is extremely time-consuming and comes with exorbitant costs. To address this, researchers resort to cycle-level simulators known for their flexibility, capable of simulating various architectural changes in GPUs. However, these simulators are exceptionally slow, requiring weeks for a single simulation run on a realistic workload. If errors occur, reruns necessitate additional waiting time. Moreover, there are currently no models tailored for evaluating ray tracing workloads specifically, given their substantial differences from other GPU workloads [16]. To bridge this gap, we introduce Zatel, designed explicitly for evaluating GPU performance in ray tracing workloads.

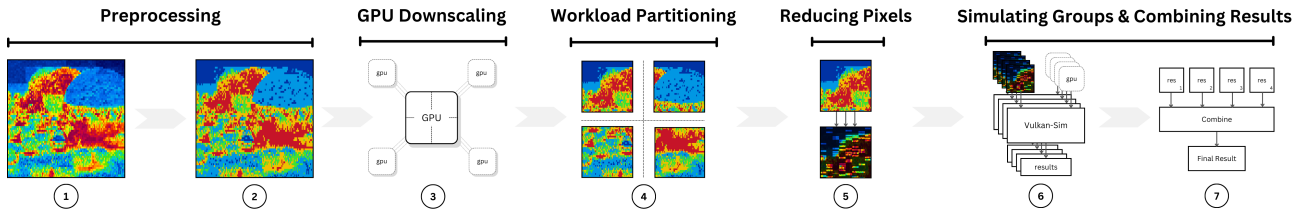

Fig. 3. High-level steps that Zatel takes to produce performance statistics.

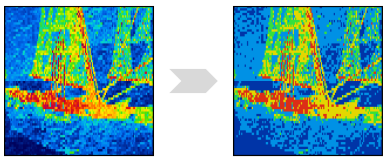

Fig. 4. Quantizing the generated heatmap to remove noise

III. Zatel's Design

Zatel is an effective prediction methodology for evaluating GPU performance metrics on ray tracing workloads at large resolutions. It exploits the parallel nature of GPUs and ray tracing workloads to simulate parts of a scene concurrently and further reduce the number of simulated pixels for each part to reduce the simulation time. Parallel simulation is achieved by partitioning the workload into groups and assigning each a downscaled GPU, which allows simulating each group simultaneously on different CPU cores. Zatel further reduces the simulation time of each simulation instance by selecting the representative pixels to trace.

A. Overview

Zatel goes through seven high-level steps to estimate the desired metrics of a workload. All the steps are visualized in Fig. 3. Zatel first performs preprocessing in steps (1) and (2) by obtaining the execution time heatmap of the desired workload and doing color quantization to remove the noise. In steps (3) and (4), Zatel chooses a scaling factor of K (in this example, $K = 4$) to downscale the GPU configuration and divide the scene into K groups. Afterwards, it reduces the simulation time by selecting representative pixels for each group in step (5). Finally, Zatel runs a simulation instance for each group and combines the results in steps (6) and (7).

B. Preprocessing

To identify the critical sections of the workload, we first generate an execution time heatmap, which highlights the time-intensive sections of the scene. We generate it in (1) by profiling the runtime of each pixel and normalizing it by the longest runtime, which is then mapped to a temperature color using NVIDIA's heat gradient [24], where warmer colors indicate lengthier ray trace times. This is followed by color quantization (2) using K-Means clustering to merge similar colors and create distinct groups, eliminating noise. Fig. 4 shows an example of a quantized heatmap, where, for instance, the darker blueish colors are merged into a distinct dark blue.

Profiling can be done on real GPU hardware or using Vulkan-Sim's functional mode. As the heatmap highlights time-consuming regions of the ray tracing algorithm, both options yield comparable results. The specific GPU hardware used is mostly irrelevant for the same reason. We choose to generate the heatmap on a hardware GPU as it can be done in seconds, unlike Vulkan-Sim's functional mode.

C. Downscaling GPU Configuration

The primary concept behind Zatel's optimization strategy is to partition the workload's image plane into groups and assign each group to a downsized GPU for ray tracing (3). This benefits from the inherent parallelism of GPUs, where independent processing units simultaneously trace different portions of the image plane.

To downscale the GPU, we focus on reducing independent components, with SMs identified as a key element. From shared components, we choose to only reduce memory partitions since they can be modeled as being divided between SMs, and thus, their count can be proportionally reduced.

We choose the downscaling factor K to be the greatest common divisor (gcd) of the number of selected components. After picking K, we divide the number of selected components by it. As an example, assume we want to downscale a GPU with 80 SMs and 10 memory controllers. We get the gcd of these two numbers as $K = 10$. After dividing the number of such parts by K, we get a new, downsized GPU configuration with 8 SMs and 1 memory partition.

Regarding other shared components, such as the off-chip memory, interconnection network, and LLC, there is no need to downscale them by the same factor K. Since the peak off-chip DRAM bandwidth is proportional to the number of memory controllers, we automatically shrink the DRAM by dividing the number of memory partitions. Moreover, each memory controller also contains the L2 cache; thus, downscaling memory partitions also proportionally reduces the number of LLC available. The mesh topology of the interconnect changes automatically with the number of SMs and memory controllers. Hence, there is no need to explicitly change any configuration of shared GPU resources.

D. Dividing the Image Plane

Once the downsizing factor K is determined and the GPU configuration is adjusted accordingly, the next step (4) is to

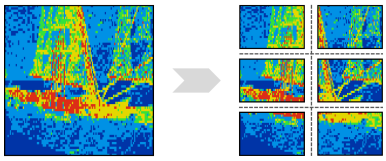

Fig. 5. Splitting the plane into 3 rows and 2 columns using coarse-grained method for $K = 6$

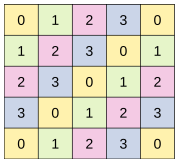

Fig. 6. Visualization of fine-grained division method

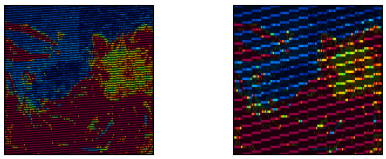

Fig. 7. Pixels of group $i = 0$ using fine-grained method; we picked chunks' height to be 2 pixels for the left image and 8 for the right one

partition the image plane into K groups. The goal is to provide each downscaled Vulkan-Sim instance with a group of pixels to trace. This paper introduces and contrasts two methods of scene division – coarse-grained and fine-grained divisions. In both approaches, the scene is segmented into distinct groups, each containing an equal number of pixels.

For the coarse-grained division method, we directly split the scene into K groups, where K is the chosen downscaling factor. Fig. 5 shows an example of dividing the scene into $K = 6$ groups using coarse-grained partitioning.

The fine-grained method divides the image plane in a more interleaved fashion to better sample the characteristics of the scene. Fig. 6 shows an image plane split into small, same-sized chunks and is evenly assigned in a round-robin fashion to four downscaled GPUs (0-3). Fig. 7 shows the result of fine-grained division on a sample heatmap. The left heatmap is divided into chunks of 32×2 pixels, and, for visualization purposes, we choose the height of a chunk to be 8 pixels when dividing the right heatmap. By making the chunks' area small, we can recognize the fox in these heatmaps, allowing each group to homogeneously capture the complexity of the workload. Zatel chooses the chunk's width to be 32 to match the warp size of 32 threads. Zatel also chooses to keep the chunk height small (2 pixels) to maintain a small chunk area while retaining thread divergence characteristics.

The fine-grained method captures the overall scene's complexity but also promotes thread divergence as rays in different chunks may diverge during traversal. Meanwhile, the coarse-grained option emphasizes ray locality since rays are close to each other. We compare these methods in Section IV-E.

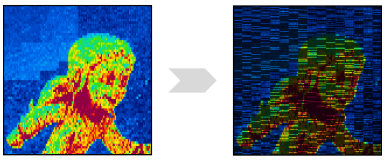

Fig. 8. Representative pixels subset of a group

E. Selecting Representative Pixels

After splitting the image plane into groups, we can further reduce the number of traced rays by selecting a subset of pixels per group to simulate (5). Similar to the scaling factor K, by simulating fewer pixels, Zatel will execute faster while only losing minimal accuracy. When selecting a subset of pixels from the group, we need to specify the number of pixels to trace and which pixels to select before launching each individual simulation instance on each group.

To choose the number of pixels to simulate, the first naive method chooses a constant percentage of pixels. This simple method allows us to better predict the reduction in simulation time and evaluate the error margins with higher confidence. However, we found that tracing different groups with the same percentage leads to each taking drastically different simulation times and resulting in varied accuracies. To solve this, we propose varying the number of pixels per group based on their heatmap's temperature distribution. We discovered that the accuracy strongly correlates with how effectively the GPU gets saturated when ray tracing our selected pixels. Thus, the warmer the average temperature of our selected pixels is, the more the GPU gets utilized, resulting in an overall better accuracy for the group. In addition, having similar average temperatures for each group's selected pixels will result in each taking similar simulation times, leading to better load balancing. We further empirically found that tracing less than 30% of pixels gives intolerable error and more than 60% doesn't provide dramatic improvements in accuracy. We thoroughly explore how the selected percentage influences both accuracy and speedup in Section IV-D.

By combining the above two observations, we can derive an equation for the number of pixels to be traced (1). We denote the percentage of pixels to be traced as P, the number of pixels in the group M, and assign each quantized color with a value $c_i \in [0, 1]$ based on their shifted hue parameter, which shows how cool the color is (0 meaning the color is hot and 1 meaning the color is cold). Zatel uses the following expression to compute P:

$$P = \frac{1}{M} \times \left\lfloor \sum_{i=0}^{M-1} c_i \right\rfloor \text{ for } 0.3 \leq P \leq 0.6 \qquad (1)$$

Equation (1) selects the subset's size proportional to how cold the heatmap's average temperature is and then bounds P between 0.3 and 0.6, keeping the groups' accuracy and speedup relatively the same.

Now that we know how many pixels to trace, we need to select which exact pixels represent the group. This process is

done in two steps: the first step divides the group into section blocks and the second step selects these blocks until it reaches the desired color distribution and number of pixels. For the first step of this process, since the fine-grained method already divides the scene into chunks, we only need to further divide each group into a grid of section blocks for the coarse-grained method. Fig. 8 shows an example of selecting section blocks with the coarse-grained group. In this example, the group gets split into section blocks of 32×2 pixels. Afterwards, Zatel chooses enough section blocks to meet the percentage requirement set by equation (1).

We choose the block's width as the number of threads in a warp, so it maps nicely to a warp and better occupies the GPU. This leads to metrics like average warp occupancy being captured more accurately. For block height, since the number of pixels to be selected is fixed, a larger block height results in fewer blocks being chosen, leading to some parts of the group being left uncaptured. However, larger block heights emphasize simulating rays closer to each other, better capturing the traced region's spatial locality. Smaller block sizes lead to higher ray divergence and, at the same time, allow us to trace pixels from different areas of the group. Thus, as a balancing measure, we choose the height to be two.

The second step is to decide which section blocks to include. We assign each quantized color a distribution percentage p_j based on the occurrence of the color in the group, which determines how many pixels of that color to include. Next, to distribute the quantized colors, this paper proposes two methods: uniformly and temperature-based. Uniform distribution assigns colors such that they match the color distribution of the group itself. For example, if 50% of the colors in the group are dark blue, approximately half of the subset's pixels should be colored dark blue as well. Such an approach captures the overall complexity of the scene better; however, it might not saturate the GPU as well as the temperature-based method. The temperature-based distribution emphasizes the pixels that take longer to trace, stressing the hardware components better. It assigns each color a value $c'_j = 1 - c_j$, representing its warmth, then maps each value to a percentage using the functions below:

$$p_j = \frac{c'_j}{C} \text{ where } C = \sum_{j=0}^{M-1} c'_j \tag{2}$$

$$p_j = \frac{c'^5_j}{C} \text{ where } C = \sum_{j=0}^{M-1} c'^5_j \tag{3}$$

We can intuitively associate each quantized color's warmth value c'_j to the weight that the color carries on saturating the GPU. The more weight the color has, the more important it is to include that color in the representative pixels. In equation (2), by summing the color values c'_j, we determine that colored region's overall impact on stressing the GPU. To normalize this value, we divide it by the sum of the weights of all pixels C. We can further modify (2) by heuristically amplifying each color's weight by raising it to the power of five, as shown in (3). Amplifying the warm colors' values stresses the hardware even more, resulting in better accuracy while maintaining similar simulation times. If there are not enough pixels with the desired color, we randomly choose other section blocks until we reach the required number of pixels.

```
.reg .u32 %shader_passes;
filter_shader %shader_passes;
.reg .pred %does_pass;
setp.eq.u32 %does_pass, %shader_passes, 0;
@%does_pass bra shader_exit;
```

Listing 1. Injected PTX code snippet in the ray generation shader

F. Modifying the Simulator

After selecting the subset of pixels for all groups, we generate K files, each containing the group's pixels' coordinates that need to be traced. We then feed these coordinates to Vulkan-Sim so that it will only trace the specified pixels. Since Vulkan-Sim doesn't have a built-in way of filtering out pixels, we create a custom PTX instruction `filter_shader` that writes 0 to the destination register if the pixel shouldn't be traced and a 1 otherwise. We inject that instruction at the beginning of the ray tracing shader, as shown in Listing 1, which exits it if the pixel should be skipped. Since the filtered-out shaders do not do anything else after exiting, their impact on the final performance statistics is negligible.

G. Extrapolating the Predictions for Each Group

When evaluating Zatel, we detected systematic bias toward metrics like IPC, L2 cache miss rate, DRAM efficiency, and bandwidth utilization. The IPC metric, for instance, is usually underestimated since the number of instructions increases faster than simulation cycles with the larger percentage of pixels we trace. When a group accesses an uncached data, it causes a cache miss, after which the value gets fetched into the cache. Since Zatel runs simulation instances for each group independently, they do not share the L2 cache entries. Thus, each simulated group causes a cache miss, leading to a higher predicted L2 cache miss rate. DRAM efficiency and bandwidth utilization are frequently underestimated for a similar reason. Therefore, the predicted values should be extrapolated based on the percentage of pixels traced in the group. The direct way is to linearly extrapolate absolute metrics, such as the simulation cycles. For example, assume that after tracing 10% of pixels, we get the simulation cycles as $100,000$. To extrapolate, Zatel divides the simulation cycles by the fraction of pixels traced, yielding $100,000/0.1 = 1,000,000$ simulation cycles as our final prediction.

Another option is to use a regression model for extrapolating our results. Since the absolute error decreases exponentially the more pixels get traced (more in Section IV-F), we propose using an exponential regression model to extrapolate the prediction results. We then simulate the group at three different percentages to feed into our regression model and extrapolate the value for 100% of pixels traced.

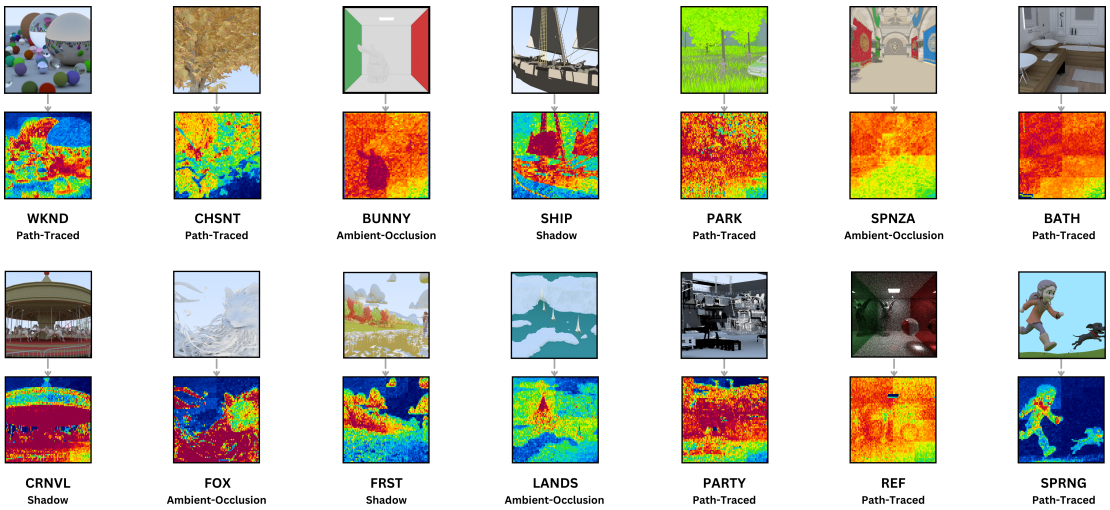

Fig. 9. The used LumiBench scenes with their respective heatmaps [16]

TABLE I
METRICS EVALUATED

Metric	Description
GPU Instructions Per Cycle	# of instructions executed per cycle
GPU Simulation Cycles	# of cycles required to ray trace the scene
L1D Total Cache Miss Rate	Total cache miss rate over all L1D instances
L2 Total Cache Miss Rate	Total cache miss rate over all L2 instances
RT Unit Avg Efficiency	Average # of active rays per warp over all ray tracing accelerator units
DRAM Efficiency	DRAM bandwidth utilization with pending requests waiting to be processed
Bandwidth Utilization	DRAM bandwidth utilization without pending requests waiting to be processed

H. Combining the Results

The last step of Zatel is combining Vulkan-Sim's output from each simulation group into a final performance prediction (7). Some metrics are encapsulated when simulated on just a single group, such as the cache miss rates, while other metrics, such as IPC, require combining the results across all the groups. As an example, assume Zatel splits the image plane into two groups and gets 20 IPC with 0.70 L1D miss rate for the first group and 50 IPC with 0.60 L1D miss rate for the second one. In Zatel, we split the image plane into groups using the fine-grained division method. This ensures that each group homogeneously samples the scene's characteristics. Thus, the instruction count executed by each group should be very close to each other. Since GPU's cores are modeled as processing units that run in parallel, then in the same cycle, the first half of the GPU executes 20 instructions while the second half executes 50 instructions, totaling to 70 IPC during the original GPU's overall execution. Meanwhile, if the two groups capture the complexity of the workload, then the overall L1D cache miss rate during the whole GPU's execution should be averaged to 0.65.

IV. EVALUATION

A. Methdology

Zatel consists of two optimization steps: dividing the scene into groups on which we run the downscaled GPU and further selecting representative pixels for each group. We test each optimization step separately by determining their effect on execution time speedup and prediction accuracy. Afterwards, we choose optimal values for the predictor's parameters to achieve maximal speedup without losing much accuracy. Each optimization step is evaluated on LumiBench's selected subset of scenes shown in Fig. 9. We simulate these workloads at 512×512 resolution with 2 samples per pixel to stress the GPU like a real-world workload while maintaining a reasonable simulation time. These scenes were selected to specifically stress individual aspects of GPU hardware. We run the simulations using Vulkan-Sim, and the metrics used to evaluate Zatel are described in Table I. Note that we refer to the speedup as the decrease in simulation time rather than improved workload performance unless stated otherwise. We also extrapolate the results of each group linearly as our baseline. To illustrate the impact of GPU saturation on the model's accuracy, we run all experiments on two different GPU configurations in Table II – a Mobile System-on-Chip (SoC) and an NVIDIA Turing RTX 2060. We first evaluate the fully optimized version of Zatel, then explain how each optimization strategy contributes to the gained speedup and the predicted metrics' accuracies. The code is available at https://github.com/ubc-aamodt-group/zatel-scale-model-sim-rt.

B. Fully Optimized Results

We choose to evaluate the final version of Zatel on the PARK scene, which is the most difficult path tracing workload from LumiBench. Thus, such a scene saturates a GPU close to a real-world 1080p workload. Fig. 10 plots the absolute error of different metrics on the PARK scene for Mobile SoC and RTX 2060 GPU configurations. Since Mobile SoC contains 8

TABLE II
GPU CONFIGURATIONS FOR EVALUATION [7]

	Mobile SoC	Turing RTX 2060
# Streaming Multiprocessors (SM)	8	30
# Memory Controllers	4	12
# Registers / SM	32768	65536
# RT Units / SM	1	
# Max Warps / SM	32	
# Warp Size	32	
Warp Scheduler	Greedy-then-Oldest	
RT Unit Max Warps	4	
RT Unit MSHR Size	64	
L1D Data Cache & Shared Mem	64KB, Fully assoc. LRU, 20 cycles	
L2 Unified Cache	3MB, 16-way assoc. LRU, 160 cycles	
Instruction Cache	128KB, 16-way assoc., 20 cycles	
Compute Core Clock	1365 MHz	
Interconnect Clock	1365 MHz	
L2 Clock	1365 MHz	
Memory Clock	3500 MHz	

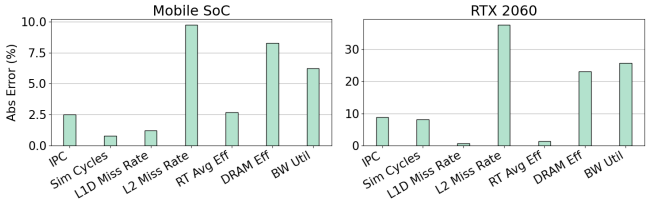

Fig. 10. Errors of metrics using Mobile SoC and RTX 2060 on PARK

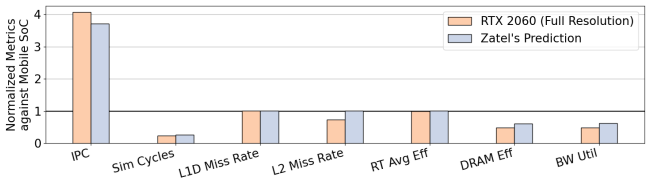

Fig. 11. RTX 2060 architecture's performance improvement over Mobile SoC

SMs and 4 memory partitions, we use a downscaling factor of $K = 4$. The RTX 2060 configuration has 30 SMs with 12 memory partitions, giving us a scaling factor of $K = 6$. When running Zatel on the PARK scene using Mobile SoC, we register a speedup of 9.2× while only getting 0.7% error for the simulation cycles with other metrics falling within 10% error. For RTX 2060, we get a speedup of 11.6× with three metrics within the 10% error and four metrics below 40% error. Overall, Zatel achieves an MAE of 4.5% for Mobile SoC and 15.1% for RTX 2060 with both configurations gaining about 10× speedup. When evaluating an arbitrary scene, Zatel uses equation (1) to ensure enough pixels are traced to achieve reasonable accuracy. However, since PARK quickly saturates most GPUs, we also assign Zatel to trace only up to 10% of pixels for each group. By drastically reducing the percentage of pixels traced, we get 50× speedup with only a 5.2% MAE on Mobile SoC, cutting down the simulation wait time from almost a week to less than three hours. These low MAE values confirm that Zatel is an excellent choice for evaluating new GPU architectures, especially mobile GPUs, on workloads that

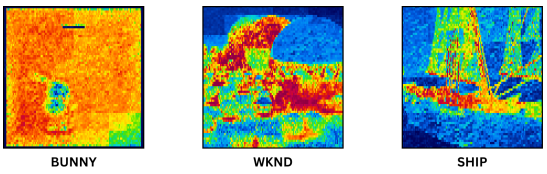

Fig. 12. Three heatmaps with different temperature distribution

fully stress the hardware, which most real-world workloads do. Furthermore, Zatel would not require being constantly updated since it runs a cycle-level simulator at its core.

Fig. 11 plots the normalized metrics of the RTX 2060 config relative to the Mobile SoC baseline. The orange bar shows the results of Vulkan-Sim with the RTX 2060 configuration when simulated at a full resolution of 512×512 and the blue bars show the predicted metrics using Zatel. Zatel is able to capture the relative performance trends when changing the simulated GPU architecture as seen from the similar blue and orange bars for each metric, where the maximum difference in normalized metrics is 37.6% for the L2 cache miss rate and minimum difference is 0.6% for L1D cache miss rate. This also highlights Zatel's ability to accurately predict performance results when designing a new architecture, making it a reliable tool to quickly estimate performance with large scene resolutions.

The state-of-the-art analytical model for GPGPU applications, GCoM, achieves a much higher MAE of 26.7% against Accel-Sim with a speedup of only 7.6x for a single design point, significantly worse than Zatel for ray tracing workloads. Moreover, unlike Zatel, if a significant architectural change is introduced, GCoM's error could increase substantially. Principal Kernel Analysis (PKA) [11] is more similar to Zatel since it selects representative kernels and thread blocks to capture the performance of a workload. However, these GPU models can not be executed on graphics applications. Even if they could, most would not capture the unique characteristics of ray tracing. For instance, PKA consists of two methods: Principal Kernel Selection and Principal Kernel Projection. Principal Kernel Selection is irrelevant since ray tracing only launches one kernel. Principal Kernel Projection terminates the simulation when the desired metric stabilizes. Since most of our evaluated workloads, especially ones with many reflective objects, involve tracing highly divergent rays, Principal Kernel Projection might stop the simulation too early, outputting a value with high error.

C. Selecting the Distribution Method and Section Block Size

We tune several optimization parameters in Zatel to produce the results in Section IV-B. We test four section block sizes: 32×1, 32×2, 32×16, and 32×32, and three distribution methods (Section III-E): uniform, linearly dependent on temperature (lintmp), and exponentially dependent on temperature (exptmp). We test each possible combination on the three scenes in Fig. 12 with different temperature distributions. These scenes were generated relative to each other by using the

TABLE III
TUNING ZATEL BY CHOOSING THE MOST OPTIMAL DISTRIBUTION AND SECTION SIZE FOR EACH METRIC OF SCENES

Metrics	SHIP			WKND			BUNNY		
	Best Dist	Best Section	MAE	Best Dist	Best Section	MAE	Best Dist	Best Section	MAE
GPU IPC	uniform	any	36.0%	any	32×16	29.5%	any	32×32	16.3%
GPU Sim Cycles	uniform	any	73.1%	uniform	any	88.3%	any	any	13.6%
L1D Miss Rate	uniform	any	13.9%	uniform	any	2.7%	any	any	0.7%
L2 Miss Rate	uniform	any	8.8%	any	any	6.6%	any	any	3.8%
BW Utilization	any	any	37.3%	any	any	34.4%	any	any	16.3%
DRAM Efficiency	uniform	any	12.5%	any	any	6.7%	any	32×32	5.7%
RT Avg Efficiency	any	any	19.9%	exptmp	any	3.9%	any	any	8.1%

same scaling value; thus, identically colored pixels represent the same tracing time. We choose to trace 2-4% of the overall pixels to understand whether the accuracy trend is impacted by the percentage. Since selecting blocks out of viable options is random, we run Zatel on each scene five times and average the results. The sampled results for each scene are in Table III; the "any" cell means no clear distinction between options.

The prediction MAEs over the listed metrics for SHIP (coldest), WKND (mix of warm and cold), and BUNNY (warmest) are 21.0%, 13.9%, and 8.5%. Such an observation suggests that the uniformly warmer the heatmap is (i.e. the better the scene saturates the modeled GPU), the more accurate Zatel will be. We further notice that the errors of each metric decrease from SHIP to BUNNY, except for the RT unit's average efficiency. For that metric, we get an MAE of 3.9% for WKND and 8.1% for BUNNY. By including up to 4% of the pixels with the exptmp distribution, the traced pixels would come from the warmest regions of WKND and BUNNY, which are redder for WKND. This leads to the RT unit metrics other than RT average efficiency also being more accurately predicted for WKND. Such an observation suggests that for evaluating RT unit-related metrics, the exptmp distribution is a more optimal choice. We also notice a high MAE of greater than 70% for simulation cycles in the SHIP and WKND scenes while for the BUNNY scene, it is only 13.6%. Such a variance in errors occurs due to tracing only 2% to 4% of the overall pixels, leading to an overcalibrated scaling since GPU simulation cycles usually follows a logarithmic trendline.

Based on results in Table III, we choose uniform distribution for the used metrics. Since the block's size has negligible impact on the overall prediction results, we choose a dimension of 32×2. Such a height value promotes ray divergence in warps while maintaining some locality. If users want to estimate more RT unit-related metrics, they should use exptmp distribution in conjunction with the uniform distribution.

D. Effect of Selecting Representative Subset of Pixels

Next, we check how the number of pixels traced affects accuracy and execution time speedup. We run the model on $\{10\%, 20\%, \ldots, 90\%\}$ of pixels without GPU downscaling and compare the estimated results to the simulator's output when ray tracing the full workload. Although we evaluate the correlation between accuracy and the percentage of pixels simulated for both the Mobile SoC and RTX 2060 configurations,

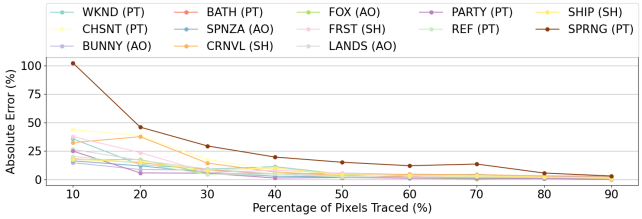

Fig. 13. Simulation cycles error per scene (RTX 2060)

we find both results in the same trends. We show figures for RTX 2060 since it is more popularly used.

Fig. 13 shows the correlation between the number of pixels traced and the Zatel's error in estimating the number of simulation cycles for each scene. We pick simulation cycles as our primary metric of examination since it is used to compare the performance of different hardware designs and ultimately select the best option. At 10% of pixels traced, we notice that the absolute error for simulation cycles drastically varies between scenes. We get more than 100% of absolute error for the SPRNG scene and 14.7% for the BUNNY scene. For the Mobile SoC configuration, we also get a high absolute error of 9.1% for CHSNT and the lowest error of 0.4% for SPNZA, around 23 times difference. As the percentage of pixels traced increases, the errors exponentially converge to 0. When tracing 50% of pixels, the difference between most scenes' absolute errors decreases to 4.1% for RTX 2060 configuration and 3.6% for Mobile SoC. One reason for the error variance, especially at 10%, is we choose section blocks randomly. Another reason is error bounds also depend on the workload.

The SPRNG scene is a special case in Fig. 13. Since there are only two objects in the scene, most rays end up terminating early. Thus, the GPU gets underutilized, and it takes a similar number of simulation cycles to trace from 10% to 100% of pixels of SPRNG. Zatel estimates the number of simulation cycles by linearly extrapolating the results to 100%. So, for 10%, it estimates a much higher number by assuming that the GPU is properly stressed.

Fig. 14 shows the simulation time it took to trace the given percentage of pixels. We notice that the longer it takes to simulate the pixels (i.e. the better the GPU is saturated), the less the workload's error bound becomes. For both configurations, one of the longest-running scene by a high margin is BATH with a rising slope of 0.21 hours per percentage for

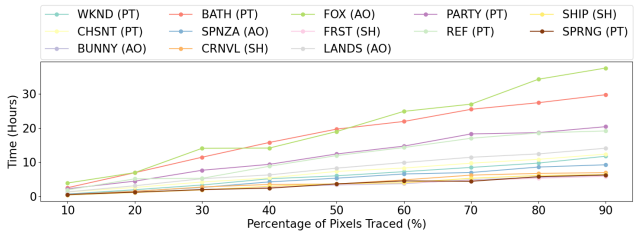

Fig. 14. Running time of Zatel per scene (RTX 2060)

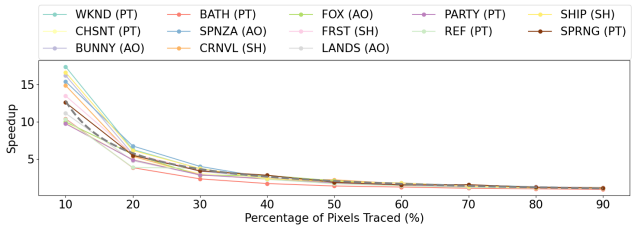

Fig. 15. Running time Speedups per scene (RTX 2060)

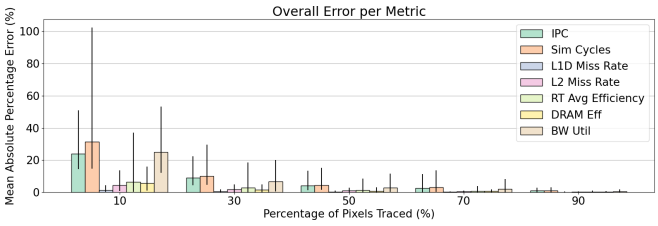

Fig. 16. Mean absolute error per listed metric over all scenes (RTX 2060)

Mobile SoC configuration and 0.34 hours per percentage for RTX 2060 configuration. Similarly, BATH also occupies the lowest absolute errors for simulation cycles, being bounded by 2.5% for Mobile SoC configuration and 20% for RTX 2060 configuration. Such a trend confirms our hypothesis that the better the scene saturates the GPU, the more accurate Zatel estimates performance metrics.

When considering the speedup gained by each scene in Fig. 15, we notice that all the scenes share similar speedups for each percentage of pixels traced for both configurations. Moreover, they all exponentially converge to $1\times$ as we trace more pixels. From the collected data points, we derive (4), plotted as a gray curve in Fig. 15, that predicts the gained speedup based on the percentage of pixels traced, helping users choose the best configuration of Zatel for their study.

$$\text{speedup}(\text{perc}) = 181 \times \text{perc}^{-1.15} \text{ for perc} \geq 10\% \quad (4)$$

We further present the absolute error of each metric over all the selected scenes against the selected percentage of pixels in Fig. 16. The error lines describe each metric's maximum and minimum error percentage. Similar to Fig. 13 and 15, we notice that the MAE for all metrics decreases exponentially the more pixels we trace. As a comparison, the recorded highest absolute error at 10% of pixels traced above 100% for the

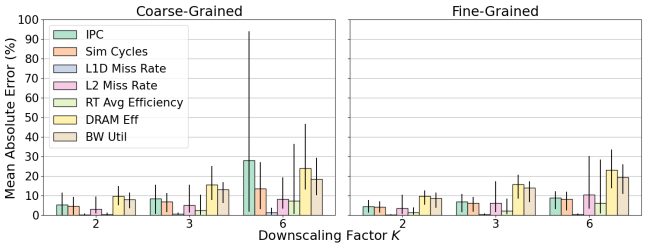

Fig. 17. Metrics' overall error per downscaling factor on the representative subset of LumiBench scenes

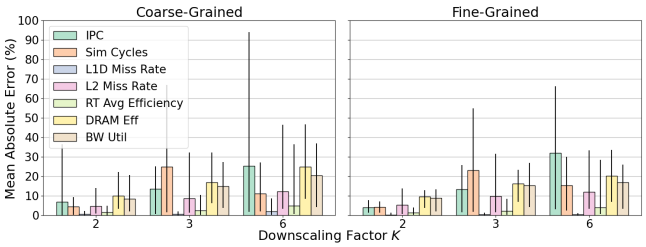

Fig. 18. Metrics' overall error per downscaling factor on the used LumiBench scenes

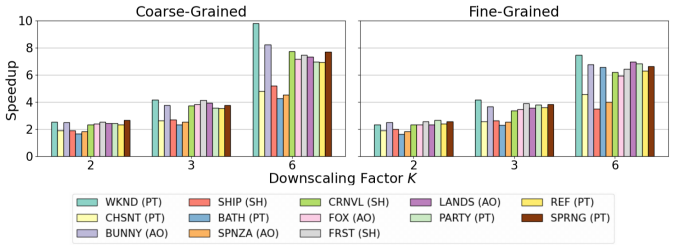

Fig. 19. Gained speedup from GPU downscaling

simulation cycles on RTX 2060 configuration. By tracing only 20% more pixels, we get the highest error down by more than 2 times for RTX 2060 and around 3 times for Mobile SoC. Additionally, we find that the metrics that are getting saturated the quickest, such as L1D and L2 cache metrics, exhibit the smallest error margins. For RTX 2060, at 10% pixels traced, the recorded MAEs for both cache metrics differ around two times since the L2 cache is 3MB while the L1D cache is only 64KB. On the Mobile SoC configuration, however, the errors of the L1D and L2 cache metrics are around the same (below 2%) independent of the percentage value since such smaller GPUs get fully stressed more easily.

E. Effect of Downscaling GPU

Zatel splits the image plane into groups and simulates each using a downscaled GPU in parallel. In this section, we sweep the downscaling factors from 2 to 6 and then evaluate them on LumiBench's scenes. We also assess how our proposed group division techniques – fine-grained and coarse-grained – affect the simulation time speedup and accuracy of Zatel.

We examine how downscaling affects the accuracy of predicted metrics in Fig. 17 for the representative subset of

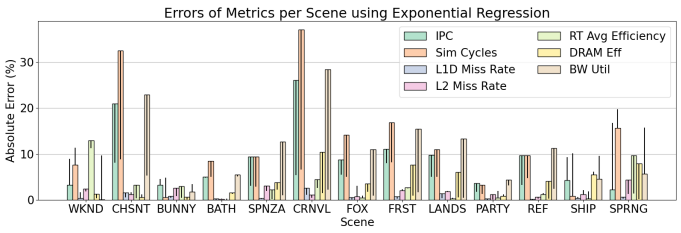

Fig. 20. Error per scene using exponential regression (RTX 2060)

scenes outlined by LumiBench. Similar to Fig. 16, we see an exponential rise in MAE as we trace fewer pixels. However, certain metrics show consistent absolute errors regardless of the scaling factor when compared to directly reducing the traced pixel count. For fine-grained division, we observe maximum absolute error for simulation cycles and IPC under 12% even for the downscaling factor of 6, which allows us to trace only 16.7% of pixels. In Fig. 16, we see that for 10% of traced pixels, even minimum absolute errors for simulation cycles and IPC are greater than 15%. For other metrics like DRAM efficiency, we see that reducing the number of memory partitions actually resulted in a much higher MAE of 20%, compared to less than 8% by simply tracing 10% of pixels. We suspect that read and write requests to DRAM and the number of active cycles do not scale linearly as we hoped. Extending the subset of scenes and running on the ones in Fig. 9, results in higher absolute errors for the IPC and simulation cycles metrics, as shown in Fig. 18. When splitting some of the scenes outside the representative subset into groups, like SPRNG, they do not adequately stress the downscaled GPU, leading to higher errors. However, we also suspect that such larger errors can result from modifying part of Zatel or the container it runs on during its development. Since fine-grained division captures the overall complexity of the scene, we get lower and more stable MAE values compared to coarse-grained division.

We included the running time speedups in Fig. 19 to examine if downscaling the GPU gives a meaningful speedup in comparison to only reducing number of pixels. When comparing the simulation time speedups with the ones in Fig. 15, we get similar results. Thus, downscaling the GPU configuration does not significantly reduce the execution time of Zatel. On the upside, this allows us to predict the speedup using equation (4). We choose to go with the fine-grained division method for better and more consistent accuracy and simulation time speedup per scene.

F. Choosing the Extrapolation Method

Beyond the previous optimizations, we also explore another model of extrapolation, the exponential regression model, by comparing it to linear extrapolation as our baseline (Section III-G). We simulate three times at 20%, 30%, and 40% to obtain three data points to feed into our regression model and extrapolate the value for 100% of pixels. Fig. 20 illustrates the estimated metrics' error per scene using regression. We compare these errors with the ones directly from tracing only 40% of pixels. The error lines in the plot represent the difference between the two. If the error line is inside the bar, the predicted value using regression is less accurate by the line's size than if we traced only 40%. For RTX 2060, we get that 62% of metrics have higher errors when we use regression compared to directly tracing 40% of pixels. Moreover, we only gained a maximum of less than 10% in accuracy while losing around 25% in metrics like IPC and bandwidth utilization for the CHSNT scene. Similarly, for Mobile SoC, around the same percentage of metrics have higher errors than the baseline, and we gain approximately as much accuracy as we lose (4%). Since our data points are already partially inaccurate, the model overfits them, resulting in a high error. Thus, regression does not provide a clear advantage over using one data point while requiring running the simulator three times.

V. Related Work

A. Evaluation using Simulation

To thoroughly evaluate the proposed architectural changes, the standard go-to method is using detailed simulators [5]–[7]. Villa et al. [25] and Sun et al. [26] achieved a speedup of around 2-3 times by making them multi-threaded. Villa et al. also sped up their simulation time by running a single iteration of the target workload instead of the whole application. They then extrapolate the results. Similarly, PKA and TBPoint [27] only simulate the representative part of the workload based on the recorded microarchitecture-independent features. Wang et al. [28] present a more statistical approach that estimates the performance based on the trace information gotten from profiling the source code. Yu et al. [29] generate downscaled but representative workloads to speed up the simulation. Zatel downscales both the workload and the GPU configuration to estimate the performance statistics with a fast simulation time.

B. Evaluation using GPU Analytical Models

One of the first analytical models for GPUs was GPUMech [13] which used interval analysis to estimate the CPI stacks of the workload. However, it gave high errors for the emerging memory-divergent workloads. MDM [14] was able to more accurately predict such workloads with a 6.1× speedup. GCoM [15] further modeled GPU subcore effects and is the current state-of-the-art analytical model. In industry, companies like NVIDIA also try creating analytical models like Need for Speed [8] to speed up the estimation of their GPUs' performance. Liu et al. [16] show in the LumiBench paper that current analytical models were not able to capture the complexity of ray tracing workloads.

C. Evaluation using Machine Learning Models

Another way to estimate GPUs' performance is to use ML models. They work like a black box and are not dependent on the modeled GPU microarchitecture. Wu et al. [30] use machine learning estimation techniques to predict both the performance and power consumption of tested GPU hardware. They train their model on numerous GPU hardware configurations. Guerreiro et al. [31] propose a model that takes in the

PTX instructions with the information about the hardware to estimate the performance. Poise [32] uses a regression model to select a warp scheduling decision for a new application. We tried using a regression model to extrapolate our results but it did not provide a clear advantage.

VI. Conclusion

We propose Zatel, a prediction methodology for estimating GPU performance on ray tracing workloads. Zatel downscales the given GPU configuration by a factor of K, divides the scene into K groups, for each group selects a representative subset of pixels, and assigns a cycle-level simulator to trace the chosen pixels for each group. Using these optimization steps, Zatel records up to less than 1% error with 10 times simulation time speedup for measuring a metric like simulation cycles on a mobile GPU. We promote Zatel as a fast and reasonably accurate methodology for evaluating GPUs' performance on complex ray tracing workloads.

Acknowledgment

The authors thank the reviewers for their feedback. We would also like to thank Jonathan Lew, Lufei Liu, and Mohammadreza Saed for their feedback on earlier drafts of this paper. The paper is funded in part by grants from Huawei Technologies. This research was also supported in part through computational resources and services provided by Advanced Research Computing at the University of British Columbia.

References

[1] P. Christensen, J. Fong, J. Shade, W. Wooten, B. Schubert, A. Kensler, S. Friedman, C. Kilpatrick, C. Ramshaw, M. Bannister, B. Rayner, J. Brouillat, and M. Liani, "Renderman: An advanced path-tracing architecture for movie rendering," *ACM Trans. Graph.*, vol. 37, no. 3, 2018.

[2] Ray Tracing Rendering Software. [Online]. Available: https://www.autodesk.com/solutions/ray-tracing

[3] Nvidia Corporation. (2023) Nvidia Ada GPU architecture. [Online]. Available: https://images.nvidia.com/aem-dam/Solutions/geforce/ada/nvidia-ada-gpu-architecture.pdf

[4] Advanced Micro Devices Inc. (2023) AMD RDNA architecture. [Online]. Available: https://www.amd.com/en/technologies/rdna

[5] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt, "Analyzing CUDA workloads using a detailed GPU simulator," in *Proc. IEEE Symp. on Perf. Analysis of Systems and Software (ISPASS)*, 2009, pp. 163–174.

[6] M. Khairy, Z. Shen, T. M. Aamodt, and T. G. Rogers, "Accel-Sim: An extensible simulation framework for validated GPU modeling," in *Proc. IEEE/ACM Int'l Symp. on Computer Architecture (ISCA)*, 2020, pp. 473–486.

[7] M. Saed, Y. H. Chou, L. Liu, T. Nowicki, and T. M. Aamodt, "Vulkan-Sim: A GPU architecture simulator for ray tracing," in *Proc. IEEE/ACM Symp. on Microarch. (MICRO)*, 2022, pp. 263–281.

[8] O. Villa, D. Lustig, Z. Yan, E. Bolotin, Y. Fu, N. Chatterjee, N. Jiang, and D. Nellans, "Need for speed: Experiences building a trustworthy system-level gpu simulator," in *Proc. IEEE Symp. on High-Perf. Computer Architecture (HPCA)*, 2021, pp. 868–880.

[9] Y. H. Chou, T. Nowicki, and T. M. Aamodt, "Treelet prefetching for ray tracing," in *Proc. IEEE/ACM Symp. on Microarch. (MICRO)*, 2023, p. 742–755.

[10] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," *SIGPLAN Not.*, vol. 37, no. 10, p. 45–57, oct 2002.

[11] C. Avalos Baddouh, M. Khairy, R. N. Green, M. Payer, and T. G. Rogers, "Principal kernel analysis: A tractable methodology to simulate scaled gpu workloads," in *Proc. IEEE/ACM Symp. on Microarch. (MICRO)*. Association for Computing Machinery, 2021, p. 724–737.

[12] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe, "Smarts: Accelerating microarchitecture simulation via rigorous statistical sampling," in *Proc. IEEE/ACM Int'l Symp. on Computer Architecture (ISCA)*. Association for Computing Machinery, 2003, p. 84–97.

[13] J.-C. Huang, J. H. Lee, H. Kim, and H.-H. S. Lee, "Gpumech: Gpu performance modeling technique based on interval analysis," in *Proc. IEEE/ACM Symp. on Microarch. (MICRO)*, 2014, pp. 268–279.

[14] L. Wang, M. Jahre, A. Adileho, and L. Eeckhout, "Mdm: The gpu memory divergence model," in *Proc. IEEE/ACM Symp. on Microarch. (MICRO)*, 2020, pp. 1009–1021.

[15] J. Lee, Y. Ha, S. Lee, J. Woo, J. Lee, H. Jang, and Y. Kim, "Gcom: A detailed gpu core model for accurate analytical modeling of modern gpus," in *Proc. IEEE/ACM Int'l Symp. on Computer Architecture (ISCA)*, 2022, p. 424–436.

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

[17] V. Govindaraju, P. Djeu, K. Sankaralingam, M. Vernon, and W. R. Mark, "Toward a multicore architecture for real-time ray-tracing," in *Proc. IEEE/ACM Symp. on Microarch. (MICRO)*, 2008, pp. 176–187.

[18] P. Ganestam and M. Doggett, "Auto-tuning interactive ray tracing using an analytical gpu architecture model," in *Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units*, ser. GPGPU-5. Association for Computing Machinery, 2012, p. 94–100.

[19] W. Liu, W. Heirman, S. Eyerman, S. Akram, and L. Eeckhout, "Scale-model architectural simulation," in *Proc. IEEE Symp. on Perf. Analysis of Systems and Software (ISPASS)*, 2022, pp. 58–68.

[20] A. Marrs, P. Shirley, and I. Wald, Eds., *Ray Tracing Gems II*. Apress, 2021.

[21] T. Aamodt, W. Fung, M. Martonosi, and T. Rogers, *General-Purpose Graphics Processor Architectures*. Morgan & Claypool Publishers, 2018.

[22] L. Liu, W. Chang, F. Demoullin, Y. H. Chou, M. Saed, D. Pankratz, T. Nowicki, and T. M. Aamodt, "Intersection prediction for accelerated GPU ray tracing," in *Proc. IEEE/ACM Symp. on Microarch. (MICRO)*, 2021, pp. 709–723.

[23] J. Peddie, "Applications of ray tracing," *Ray Tracing: A Tool for All*, 2019.

[24] Profiling DXR Shaders with Timer Instrumentation. [Online]. Available: https://developer.nvidia.com/blog/profiling-dxr-shaders-with-timer-instrumentation/

[25] O. Villa, A. Tumeo, S. Secchi, and J. B. Manzano, "Fast and accurate simulation of the cray xmt multithreaded supercomputer," *IEEE Transactions on Parallel and Distributed Systems*, vol. 23, no. 12, pp. 2266–2279, 2012.

[26] Y. Sun, T. Baruah, S. A. Mojumder, S. Dong, X. Gong, S. Treadway, Y. Bao, S. Hance, C. McCardwell, V. Zhao, H. Barclay, A. K. Ziabari, Z. Chen, R. Ubal, J. L. Abellán, J. Kim, A. Joshi, and D. Kaeli, "Mgpusim: Enabling multi-gpu performance modeling and optimization," in *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*, 2019, pp. 197–209.

[27] J.-C. Huang, L. Nai, H. Kim, and H.-H. S. Lee, "Tbpoint: Reducing simulation time for large-scale gpgpu kernels," in *Proc. IEEE Int'l Parallel and Distributed Processing Symp. (IPDPS)*, 2014, pp. 437–446.

[28] X. Wang, K. Huang, A. Knoll, and X. Qian, "A hybrid framework for fast and accurate gpu performance estimation through source-level analysis and trace-based simulation," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2019, pp. 506–518.

[29] Z. Yu, L. Eeckhout, N. Goswami, T. Li, L. K. John, H. Jin, C. Xu, and J. Wu, "Gpgpu-minibench: Accelerating gpgpu micro-architecture simulation," *IEEE Transactions on Computers*, pp. 3153–3166, 2015.

[30] G. Wu, J. L. Greathouse, A. Lyashevsky, N. Jayasena, and D. Chiou, "Gpgpu performance and power estimation using machine learning," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, 2015, pp. 564–576.

[31] J. Guerreiro, A. Ilic, N. Roma, and P. Tomás, "Gpu static modeling using ptx and deep structured learning," *IEEE Access*, 2019.

[32] S. Dublish, V. Nagarajan, and N. Topham, "Poise: Balancing thread-level parallelism and memory system performance in gpus using machine learning," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2019, pp. 492–505.