Unleashing the Power of GPU for Physically-Based Rendering via Dynamic Ray Shuffling

Yashuai Lü[†] Libo Huang*[‡] Li Shen[‡] Zhiying Wang[‡]
[†]Key Laboratory, Astronautical Engineering University, Beijing, China
[‡]State Key Laboratory of High Performance Computing and School of Computer,
National University of Defense Technology, Changsha, China
{yashuailv,libohuang,lishen,zywang}@nudt.edu.cn

ABSTRACT

Computer graphics is generally divided into two branches: real-time rendering and physically-based rendering. Conventional graphics processing units (GPUs) were designed to accelerate the former which is based on the standard Z-buffer algorithm. However, many applications in entertainment, science, and industry require high quality visual effects such as soft-shadows, reflections, and diffuse lighting interactions which are difficult to achieve with the Z-buffer algorithm, but are straightforward to implement using physically-based rendering methods. Physically-based rendering can already be implemented on present programmable GPUs. However, for physically-based rendering on GPUs, a large portion of the processing power is wasted due to low utilization of SIMD units. This is because the core algorithm of physically-based rendering, *ray tracing*, suffers from Single Instruction, Multiple Thread (SIMT) control flow divergences. In this paper, we propose the Dynamic Ray Shuffling (DRS) architecture for GPUs to address this problem. Our key insight is that the primary control flow divergences are caused by inconsistent ray traversal states of a warp, and can be eliminated by dynamically shuffling rays. Experimental results show that, for an estimated 0.11% area cost, DRS significantly improves the SIMD efficiency for the tested benchmarks from 41.06% to 81.04% on average. With this, the performance of a physically-based rendering method such as path tracing can be improved by 1.67×−1.92×, and 1.79× on average.

CCS CONCEPTS

• **Computer systems organization → Single instruction, multiple data**;

KEYWORDS

GPU, physically-based rendering, warp divergence

* Corresponding author.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
MICRO-50, October 14–18, 2017, Cambridge, MA, USA
© 2017 Copyright held by the owner/author(s). Publication rights licensed to Association for Computing Machinery.
ACM ISBN 978-1-4503-4952-9/17/10...$15.00
https://doi.org/10.1145/3123939.3124532

ACM Reference format:
Yashuai Lü[†] Libo Huang*[‡] Li Shen[‡] Zhiying Wang[‡]. 2017. Unleashing the Power of GPU for Physically-Based Rendering via Dynamic Ray Shuffling. In *Proceedings of MICRO-50, Cambridge, MA, USA, October 14–18, 2017,* 14 pages.
https://doi.org/10.1145/3123939.3124532

1 INTRODUCTION

Real-time rendering poses severe computational requirements to computers. The required computing power can be delivered by special purpose, highly parallel hardware, which is dedicated to the particular problem of rendering, that is, the Graphics Processing Unit (GPU). Conventional GPUs were built to support a single graphics rendering algorithm: the Z-buffer in which the underlying physical model is simplified to cope with performance requirements. This simplification is called the local illumination model, because it assumes that a point can be shaded using only its own properties, and independent of other surfaces in the scene. With the simplified rendering model, conventional GPUs can provide a highly interactive experience at a relatively low cost. However, the local illumination model is extremely different from the physically accurate model of light transfer. Achieving complex illumination effects such as soft-shadows, reflections, and refractions is difficult. On the contrary, these effects can be easily implemented using physically-based rendering methods [30] that can produce photorealistic images which are crucial for modern movie special-effects industry, future real-time 3D graphics applications, and many other application fields.

GPUs have now evolved to support general-purpose programming beyond traditional fixed-function pipelines. Current programmable GPUs get their computing power by running in the order of thousands of threads in parallel, and the Single Instruction, Multiple Thread (SIMT) programming paradigm used by CUDA [27] and OpenCL [15] allows the programmer to treat these threads just as if they were scalar threads. Physically-based rendering algorithms are often viewed as being "embarrassingly parallel." Extracting parallelism seems as trivial as constructing many independent samples in parallel. However, the SIMT hardware has strong single-instruction, multiple data (SIMD) traits. Scalar threads are grouped together into SIMD batches, which are referred to as warps by NVIDIA. If the threads of a warp diverge in control flow, the warp serially executes both branches. During each such divergence, part of the warp's

RIGHTSLINK

threads are inactive; hence, the underlying SIMD processing unit is partially utilized for the duration of each divergent branch. In practice, for physically-based rendering on GPUs, the loss in SIMD efficiency could be extremely high due to severe control flow divergences caused by the *ray tracing* algorithm. As a consequence, many researchers believed that the Multiple Instruction, Multiple Data (MIMD) threaded architecture is more efficient than the SIMT architecture for ray tracing [18, 19, 21, 22, 24], as threads can run independently without affecting each other.

This paper introduces a novel architecture named Dynamic Ray Shuffling (DRS), showing a contrary conclusion to this perspective. Rays are dynamically resorted using DRS to prevent the occurrence of warp divergence caused by inconsistent ray traversal states; thus, the SIMD efficiency is greatly improved. Experimental evaluations show that with ray shuffling, the potential computing power of a modern GPU can be unleashed for physically-based rendering with marginal hardware overhead. In summary, the contributions of this paper are the following:

- It shows an approach that the warp divergence problem can be addressed by shuffling register data without touching the warp control logic.
- It proposes the Dynamic Ray Shuffling architecture that eliminates inconsistent ray traversal states to improve SIMD efficiency for ray tracing on GPUs.
- Experimental evaluations show that with marginal hardware overhead, both the SIMD utilization and the performance of physically-based rendering on GPUs can be greatly improved. This result demonstrates that modern GPU hardware is capable of rendering photorealistic images with high hardware efficiency.

The remainder of this paper is organized as follows. Section 2 discusses the background and motivation. The DRS architecture details are presented in Section 3. The experimental evaluation is presented in Section 4. Related works are discussed in Section 5. Finally, Section 6 contains the conclusion.

2 BACKGROUND AND MOTIVATION

This section provides a brief background on physically-based rendering, discusses the problem of ray tracing on GPUs, and describes the motivations for dynamic ray shuffling. We use NVIDIA's terminology throughout the paper for convenience.

2.1 Physically Based Rendering

Rendering is the process of producing an image from the description of a 3D scene, and many approaches can be used. Physically-based rendering methods [30] attempt to produce highly realistic images by simulating reality; that is, they use principles of physics to model the interaction of light and matter. Almost all physically based rendering methods are based on *ray tracing* such as the widely used rendering method, *path tracing*. Path tracing incrementally generates path samples by simulating a random walk through a scene. A path starts with a *primary ray* at the camera. It is traced into the scene, and on each surface hit the *Bidirectional Scattering Distribution Function* (BSDF) is sampled to generate a new direction; a *secondary ray* is then traced from the hit point on the surface along the sampled direction to find the next hit point. The progress of path extension is repeated until a ray is traced out of the scene, or a ray hits a light source, or a hard-coded maximum path depth is reached. In general, a pixel in the final resulting image is rendered by evaluating hundreds or thousands of ray paths. When combined with physically accurate models of surfaces, light sources, and cameras, path tracing can produce images that are indistinguishable from photographs.

A physically-based approach may seem to be the most evident method to rendering. However, it is not adopted by conventional GPUs due to its high computational cost, which is mainly caused by the *ray tracing* algorithm. Most algorithms such as ray generation, BSDF evaluation, and ray shading in a physically-based rendering method have the time complexity of $O(n)$ except for ray tracing which has the highest time complexity. The task of a ray tracing algorithm is finding ray-object intersections. Determining the closest object that intersects a ray is a searching problem among all scene objects. Hence, the time complexity of ray tracing is $O(n \times m)$, where n is the number of rays and m is the number of objects in the scene.

The most effective method to improve the performance of physically-based rendering is *improving the performance of ray tracing*. Hence, ray tracing has received continuous research interests from computer graphics researchers. Ray tracing is viewed as being "embarrassingly parallel," because each ray is independent of each other. Hence, the multi-core system with multithreading becomes an attractive platform to implement ray tracing. Conceptually, the ray tracing algorithm maps nicely to SIMT processors such as modern programmable GPUs, since individual rays can be mapped to a single thread, resulting in thousands of individual threads with no inter-thread data dependencies. However, this ray tracing implementation does not yield high hardware efficiency on GPUs.

2.2 Inefficiency of Ray Tracing on GPUs

Ray tracing is a searching problem that finds the closest object that intersects a ray. To accelerate this search, tree data structures are widely used, such as a kd-tree [6] or Bounding Volume Hierarchies (BVH). Leaf nodes of the tree contain scene objects, and the inner nodes of the tree are used to subdivide a larger spatial representation into multiple smaller spatial areas (kd-tree) or break the objects in the scene into smaller sets of constituent objects (BVH). Rays are the input into the tree and are recursively traversed down from the root node to leaf nodes. Algorithm 1 shows a typical ray tracing algorithm which has four loops. The outermost loop (line 1) is used to fetch new rays from memory and initialize ray variables. The inner loop of line 3 is used to ensure that

RIGHTSLINK

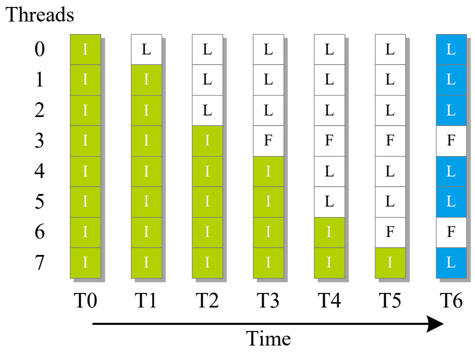


Figure 1: An illustrative example for the inefficiency of ray tracing on GPUs

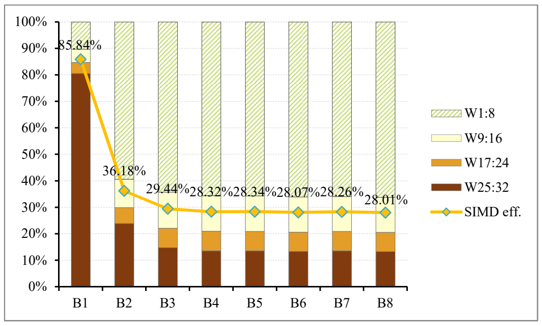


Figure 2: SIMD efficiency and utilization breakdown of Aila's ray tracing kernel [4] on the conference room benchmark

all relevant tree leaf nodes are tested before a ray is completely processed. The first innermost loop statement (line 4) is used to traverse inner nodes of the tree until it finds a leaf node. The second innermost loop statement (line 7) is used to test all objects of a leaf node.

Algorithm 1: Typical Ray Tracing Algorithm

```
1  while there are rays that have not been traced do
2      fetch a new ray from memory and initialize;
3      while ray not terminated do
4          while node is not a leaf do
5              traverse inner nodes of the tree;
6          end
7          while node contains untested objects do
8              perform ray-object intersection tests;
9          end
10     end
11 end
```

Suppose that a ray tracing kernel is implemented based on Algorithm 1 with the tree traversal of each ray being mapped to a single thread. The example in Figure 1 depicts the occurrence of low SIMD utilization in a warp. To simplify the explanation, we suppose that the warp size is eight. Letter **I** in Figure 1 denotes a ray that must traverse inner nodes of the tree (for brevity, this state is noted as the *inner* state); letter **L** denotes a ray that must traverse leaf nodes (noted as the *leaf* state); letter **F** denotes that a ray is terminated, and that the thread must fetch a new ray from memory (noted as the *fetching* state). The presence of colored lattices indicate corresponding threads are active, whereas uncolored lattices indicate corresponding threads are inactive.

Initially, all rays of the warp are in the *inner* state. Thus, all threads are traversing inner nodes of the tree with full SIMD utilization. At time T1, thread #0 gets into the leaf state, and should traverse the leaf nodes given that traversing paths of different rays may be different from each other. However, as other threads are all in the *inner* state, thread #0 must be inactive and must wait for other threads. As time goes by, more and more threads have changed from the *inner* state into other states. At time T5, only thread #7 is in the *inner* state, and all other threads must wait for it. The SIMD utilization at time T5 is only 1/8. At time T6, no thread is in the *inner* state, and the warp is traversing leaf nodes. However, thread #3 and thread #6 are in the *fetching* state; thus, the SIMD unit at time T6 is not fully utilized. This example shows that for ray tracing on GPUs, the number of clock cycles for any ray in a warp to complete is equal to the longest ray in the warp, resulting in poor SIMD unit utilization.

Aila et al. proposed several software optimizations [3, 4] to ray tracing kernels that are based on Algorithm 1 to improve SIMD efficiency, including persistent threads, speculative traversal, and replacing terminated rays, resulting in the fastest ray tracing kernel on NVIDIA's GPUs to date [1, 8]. However, without hardware modifications, the proposed techniques are only able to alleviate the low SIMD utilization problem to some extent, but cannot fundamentally address it. Figure 2 illustrates the SIMD efficiency and utilization breakdown of Aila's ray tracing kernel [4] on the conference room benchmark. The data are collected from the GPGPU-Sim simulator [5]. Bn in Figure 2 denotes the result data of the nth bounce of path tracing. Category Wm:n is the percentage of warp instructions issued to the pipeline with m to n active threads in the warp. The experimental details will be described in Section 4. Figure 2 shows that although the SIMD efficiency of primary rays (bounce #1) is high, it degrades greatly for secondary rays (bounce #2 - bounce #8). The main reasons for this phenomenon are the following: 1) the primary rays which are generated from the camera are highly coherent resulting in high SIMD efficiency; 2) secondary rays, which are randomized by ray bouncing, are much less coherent than primary rays, leading to frequent warp divergences across loop iterations and poor SIMD efficiency. The experimental data of Figure 2 exemplify that software techniques have limitations on improving the SIMD efficiency for ray tracing on GPUs. Hence, we are motivated to search for a hardware solution.

2.3 Motivation

First, we review prior micro-architectural proposals that address the control flow divergence problem on GPUs.

One approach addresses the SIMD efficiency problem by exploring intra-warp parallelism. Dual-Path Execution (DPE) [33] and Multi-Path Immediate Post-Dominator (MP-IPDOM) [10] enables interleaved execution of two or more divergent paths within a warp while maintaining IPDOM. Warps in Dynamic Warp Subdivision (DWS) [23] are selectively subdivided into warp-splits which are treated as independently schedulable units. Intra-Warp Compaction [38] improves SIMD efficiency by using cycle compression techniques. Thread-frontier re-convergence [9] improves SIMD efficiency by making threads of a warp re-converge at the same PC which is usually earlier than the IPDOM re-convergence. These techniques that focus on intra-warp parallelism *cannot improve the number of active threads within a warp* for ray tracing. Our goal is for every warp to execute ray traversals with all its threads being active.

Another approach addresses the control flow divergence problem by exploring inter-warp parallelism, i.e., recombining threads from different warps [7, 12, 13, 25, 32, 40]. While these approaches are more aggressive than those exploring intra-warp parallelism and provide good performance improvement for some divergent applications, they introduce complexity in the micro-architecture of SIMT hardware. First, instruction scheduler and other warp instruction control logic need to be modified to implement implicit warp barriers and manage additional warp execution states. Second, these approaches require highly-banked, per-SIMD-lane addressable register files, which introduces additional area and energy costs.

One common point of the above mentioned approaches is that the SIMD efficiency is improved by modifying instruction control hardware. In this paper, to improve the SIMD efficiency of ray tracing on GPUs, we take a different approach other than making modifications to the instruction control logic. As shown in Figure 1, for ray tracing on GPUs, warp divergence occurs when rays of a warp have different traversal states. Our key insight is that *if the rays within a warp can keep the same traversal state, then no control flow divergence on the four loop statements of Algorithm 1 will occur*. To achieve this condition, we propose dynamic ray shuffling (DRS) to enable the rays of a warp to always maintain the same traversal state. This method avoids introducing complexity into the warp control hardware. It can achieve optimal performance improvement with marginal hardware cost. The DRS approach can coexist with most of the prior approaches mentioned above, as it does not require modifications on the instruction control hardware.

3 DYNAMIC RAY SHUFFLING

In this section, we present the DRS architecture, its workflow and a walkthrough example.

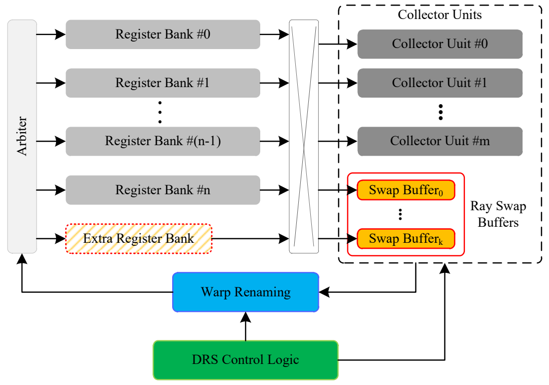


Figure 3: Dynamic ray shuffling architecture

3.1 Architecture

3.1.1 Overview. Figure 3 illustrates the DRS architecture and how it is integrated with a GPU. For ray tracing kernels implemented on current GPUs, all live variables of a ray such as origin, direction, and hit length, can be stored in architectural registers during ray traversal, since current GPU register files are sufficiently large. As a consequence, the DRS hardware mainly interacts with GPU register files to manage live rays.

To provide large bandwidth without unduly increasing complexity, current GPU register file is constructed with multiple single ported SRAM banks, as shown in Figure 3. To provide the appearance of a multiported register file, NVIDIA GPUs adopt a hardware structure named "operand collector" which is composed of a set of buffers and arbitration logic. The buffers, which are called collector units, are used to buffer the source operands of a warp instruction.

The DRS hardware is composed of four parts, which are indicated by the colored parts of Figure 3, as follows: (1) optional extra register banks, which are used to accommodate extra rays; (2) swap buffers, which are used as temporary data buffers for ray data swapping between registers; (3) the warp renaming logic, which is used to allow a warp to freely access an arbitrary row of rays in the register file; and (4) the DRS control logic, which controls ray swapping and warp renaming.

3.1.2 ISA and Register Extension. The DRS hardware interacts with a ray tracing kernel by using an instruction set extension (*rdctrl reg_d*) and a special register (*reg_ray_state*). The *rdctrl* instruction is used to read a control value from the DRS hardware. This control value is used by a ray tracing kernel to determine the control flow that the current thread will take. A difference between the *rdctrl* and normal register move instructions is that the *rdctrl* instruction can stall the issue of a warp instruction. The special register *reg_ray_state* is used by the ray tracing kernel to inform the DRS hardware of the ray traversal state of the current thread.

3.2 Workflow

In this subsection, we describe the workflow of the DRS by first introducing the ray tracing kernel for the DRS, then describing the organization of the live rays in a GPU core. Finally, we describe the hardware implementation details of the DRS.

3.2.1 Ray Tracing Kernel for the DRS. The pseudo-code of the ray tracing kernel for the DRS is shown in Kernel 1. The layered "while-while-while" loop structure of Algorithm 1 is changed into a layered "while-if" structure with three "if" statements nested in a "while" loop. The first "if" statement (line 3) is used to get a new ray from memory, which is equivalent to line 2 of Algorithm 1. The second "if" statement (line 7) is used to traverse inner nodes, and the third "if" statement (line 11) is used to perform ray-object intersection tests. At the end of each "if" statement, the next ray traversal state (*next_trav_state*) is stored into the special register *reg_ray_state* to inform the DRS hardware if the current ray requires traversing inner nodes or leaf nodes, or gets terminated. In line 5, after a ray has been initialized, its next traversal state is always *inner*, because it needs to first traverse the root node.

Kernel 1: Ray Tracing Kernel for the DRS

```
1  read $trav\_ctrl\_val$;
2  while $trav\_ctrl\_val \neq EXIT$ do
3      if $trav\_ctrl\_val = FETCH$ then
4          fetch a new ray from memory and initialize;
5          reg_ray_state← $INNER$;
6      end
7      if $trav\_ctrl\_val = TRAV\_INNER$ then
8          traverse inner nodes of the tree;
9          reg_ray_state← $next\_trav\_state$;
10     end
11     if $trav\_ctrl\_val = TRAV\_LEAF$ then
12         perform ray-object intersection tests;
13         reg_ray_state← $next\_trav\_state$;
14     end
15     read $trav\_ctrl\_val$;
16 end
```

The main control flow of Kernel 1 is determined by the variable *trav_ctrl_val*. The variable is assigned by the clause "read *trav_ctrl_val*" in line 1 and line 15 of Kernel 1. This procedure is performed by reading the value from the DRS hardware using the *rdctrl* instruction. Clearly, if the threads of a warp get the same *trav_ctrl_val* value, then the threads will execute the same "if" branch with no warp divergence.

For an arbitrary warp, the workflow of Kernel 1 is as follows. (1) When the warp executes the clause "read *trav_ctrl_val*," the DRS hardware maps the warp to a group of rays that are in the same state to avoid control flow divergence. The returned *trav_ctrl_val* value is dependent on the ray traversal state. For example, if the corresponding rays are in the inner state, the returned *trav_ctrl_val* value will be TRAV_INNER, and the warp will take the control flow of the second "if" statement. (2) When one of the "if" statements has been completed by the warp, the ray traversal states of the warp may be changed, and some of these states may be different from each other. The next ray traversal states are all stored in the *reg_ray_state* register at the end of each "if" statement to inform the DRS hardware about the changes of the ray states. (3) When the warp executes the clause "read *trav_ctrl_val*" again, the DRS hardware may map it to another group of rays that are in the same state.

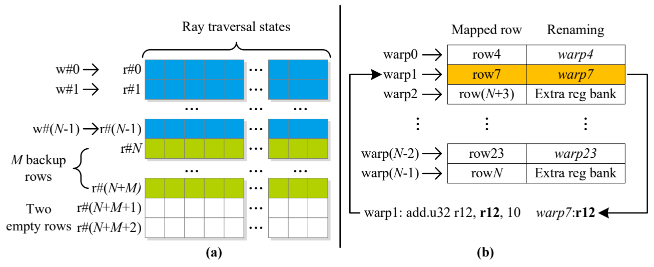
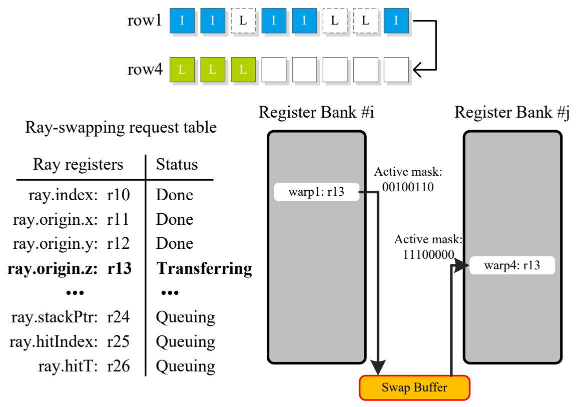


Figure 4: (a) Ray state table; (b) warp mapping and renaming.

3.2.2 Organization of Live Rays. As stated above, all live variables of a ray are mapped to registers. We compiled Kernel 1 into the assembly code, and manually identified the corresponding registers of live ray variables. These registers are regarded as *ray registers*.

The DRS logically organizes the live rays in a register file into $(N+M+2)$ rows with row size of 32 (the same with the warp size), where N is the number of warps can be spawned on a GPU core or a Streaming Multiprocessor (SMX) in NVIDIA's terminology, M is the number of backup ray rows, and the number "2" denote two rows of empty slots, as shown in Figure 4(a). Initially, N warps are mapped to the first N rows. The purpose of maintaining M rows of backup rays is that when the rays of a warp change into different traversal states, the DRS can switch the warp to work on another row of rays that is not bound to any warp; thus, the warp can keep on working without stall. With the empty slots, rays can be reorganized into different rows, which will be shown in the walkthrough example in the following subsection. The DRS control uses a ray state table with the size of $(N+M+2)\times32$ to monitor ray states, as shown in Figure 4(a).

The ray state transitions in this table are performed by assigning values to the *reg_ray_state* register (line 5, line 9 and line 13 of Kernel 1). For example, suppose that traversing one inner node has been completed (line 8 of Kernel 1) by a warp which is mapped to row #1 of the ray state table, and the rays of the second and the third thread must traverse leaf nodes while other rays of the warp must traverse inner node. In line 9 of Kernel 1, the second and the third thread will assign the *leaf* state to register *reg_ray_state*, while other threads in the warp will assign the *inner* state to *reg_ray_state*. The consequence is that the second and the



Ray-swapping request table

Ray registers	Status
ray.index: r10	Done
ray.origin.x: r11	Done
ray.origin.y: r12	Done
ray.origin.z: r13	**Transferring**
...	**...**
ray.stackPtr: r24	Queuing
ray.hitIndex: r25	Queuing
ray.hitT: r26	Queuing

Figure 5: Workflow of ray swapping

third column of row #1 will be changed into the *leaf* state, while other columns of row #1 will keep the *inner* state.

3.2.3 Warp Mapping and Renaming. In general, the tasks of ray shuffling are two-fold: (1) mapping a warp to a row of rays and (2) ray swapping. We first describe the implementation of warp mapping.

When a *rdctrl* instruction is scheduled for issue (line 1 and line 15 of Kernel 1), the DRS control checks the ray states of this warp. If the ray states differ, then the DRS control will try to map the warp to a new row of rays with the same ray states. If such a row cannot be found due to ongoing ray shuffling, then the issue of the warp is temporarily suspended until ray shuffling is complete.

Mapping a warp to an arbitrary row of rays is achieved by warp renaming, as shown in Figure 4(b). The DRS control maintains a renaming table which has N entries, each entry corresponding to a warp, and two columns with the first column indicating the row where the warp is mapped and the second column indicating how to rename warps.

Suppose without warp renaming, the ith warp should operate on the ith row of rays. If a warp is mapped to row $\#j$ $(0 \leq j \leq N-1, i \neq j)$ by the DRS control, then the warp renaming scheme will allow the warp to access the *ray registers* that originally belongs to warp $\#j$. For example, as shown in Figure 4(b), warp #1 is mapped to row #7, and must access register *r12*. Warp #1 accesses the actual register, which is register *r12* of warp #7. One row is not allowed to be bound to more than one warp. If a warp is mapped to row $\#k$ $(k \geq N)$, then the warp will access *ray registers* in the extra register bank.

3.2.4 Ray Swapping. Ray swapping moves ray data between different logical ray rows to eliminate the rows that have different ray states. We use a greedy method to achieve this task.

At each clock cycle, the DRS control will initiate ray swapping if a row which has different ray states exists. Ray swapping is performed by designating three special rows and moving ray data between the special and other rows. The three

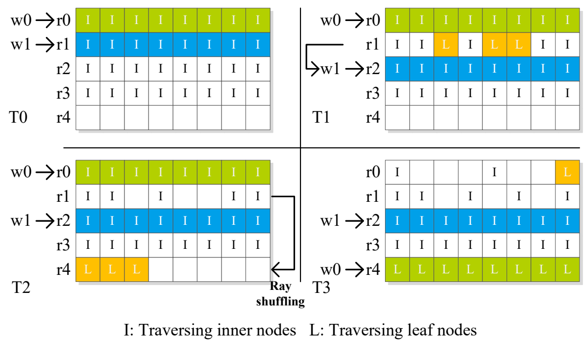


Figure 6: Walkthrough example for the DRS

special rows are a *fetching*-state ray collecting, a *leaf*-state ray collecting, and an *inner*-state ray ejecting row. The *fetching*-state and the *leaf*-state ray collecting rows are used to collect rays that are in the *fetching* state and the *leaf* state from other rows, respectively. The ray collection process will result in rows that are full of *fetching* or *leaf*-state rays. The *inner*-state ray ejecting row is selected to be a row that has one or more *inner*-state rays. The DRS control moves *inner*-state rays of this row to empty slots on other rows, or exchanges *inner*-state rays for *fetching* or *leaf*-state rays on other rows. This process will fill other rows with the *inner*-state rays.

It should be noted that ray swapping between two rows cannot be completed in one cycle, because the live variables of a ray are spread over ray registers. The DRS control translates a ray swap between two rows into several register read/write requests. A swapping request table is used to track the status of these requests, and the swap buffers are used to temporarily store the variables. In Figure 5, the process of ray swapping between two rows is composed of two steps: the first step is reading data from the registers corresponding to the source row into swap buffers; the second step is writing data from the swap buffers into the registers corresponding to the destination row.

3.3 Walkthrough Example

We use a walkthrough example to further explain the workflow of the DRS, as illustrated in Figure 6.

As mentioned above, the live rays in a SMX are logically organized into several rows, with the row size equals the warp size as depicted in Figure 6. A blank lattice in Figure 6 indicates that this lattice is currently unmapped to any ray. To clarify, we use two warps with the warp size of eight for the example in Figure 6.

At time T0, warp #0 and #1 are working on row #0 and #1, respectively. The rays of row #0 and #1 are in the *inner* state. Thus, no warp divergence occurs. At time T1, three rays of row #1 have changed into the *leaf* state, while other rays of row #1 are still in the *inner* state. If warp #1 remains working on row #1, then a warp divergence will occur because of different ray traversal states. To avoid this

Parameter	Value
SMX Clock Frequency	980 MHz
SIMD lanes	32
SMXs/GPU	15
Warp Scheduler	Greedy-Then-Oldest [34]
Warp Schedulers/SMX	4
Inst. Dispatch Units/SMX	8
Registers/SMX	65536
L1 Data Cache	48 KB
L1 Texture Cache	48 KB
L2 Cache	1536 KB

Table 1: GPU microarchitectural parameters

condition, warp #1 is moved to work on row #2. The warp divergence is avoided because the rays of row #2 are in the inner state. The DRS control initiates ray shuffling with row #4 being used to collect the *leaf*-state rays, because row #1 has different ray traversal states at this time. At time T2, the three rays of row #1 that are in the *leaf* state have been moved to row #4. Time T3 shows that row #4 has been filled with the *leaf*-state rays by ray shuffling, with warp #0 working on it. No warp divergence occurred when ray shuffling is used, because the two warps always worked on the rows, which ray traversal states are the same.

4 EXPERIMENTS AND EVALUATION

4.1 Methodology

We used the GPGPU-Sim [5] simulator (version 3.2.1) for evaluating the DRS. The simulator was modified and configured to model an NVIDIA GeForce GTX780 GPU (Kepler architecture [26]) as our baseline GPU architecture. The configuration parameters are listed in Table 1.

For evaluating ray tracing, we used four benchmark scenes with different complexities, as illustrated in Figure 7. The *conference room* benchmark is an indoor scene and has a medium number of objects that are not evenly distributed throughout the scene. The *fairy forest* benchmark is an outdoor scene and an example of the "teapot in a stadium" problem with a small detailed model in a simple large open environment. The *crytek sponza* benchmark has a similar number of objects with the *conference room* benchmark, but the architectural structure of the *crytek sponza* benchmark is particularly complex for testing the physically-based rendering methods. The *plants* benchmark is another outdoor scene but with a large number of objects that are densely distributed.

These benchmark scenes were rendered by PBRT [30] using the path tracing light transport algorithm with a maximum ray bounce depth of eight. The path tracing algorithm was selected because this algorithm is the most widely used physically-based rendering method and can generate incoherent and widely scattered secondary rays that provide a worst-case stress test for a ray tracing system. The scenes were rendered at a resolution of 640×480 with the low-discrepancy

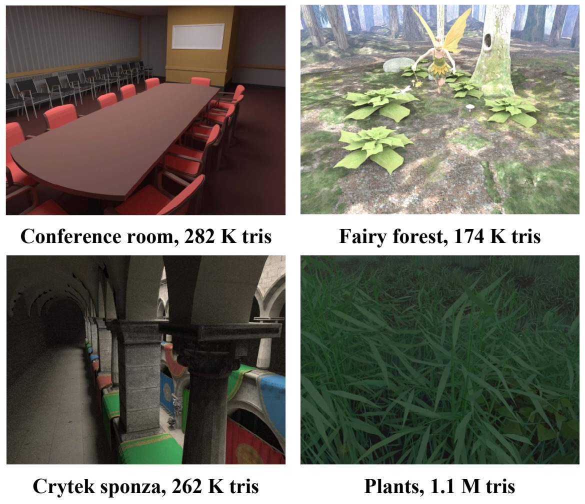


Figure 7: Benchmark scenes used for evaluation

sampling method and 64 samples per pixel to achieve reasonable simulation time. We treat shading and ray generation as a "black box," because this work focuses on the ray tracing algorithm. We streamed traces of rays captured from PBRT and fed these traces to ray tracing kernels as input.

For evaluating the DRS, we compared our method with the software ray tracing method [4] proposed by Aila et al. To the best of our knowledge, Aila's ray tracing kernel is currently the fastest ray tracing kernel on the GPUs of NVIDIA. Kernel 1 is used for evaluating the DRS, and is built on Aila's kernel by removing the code for speculative traversal and by changing the "while-while" structure into the "while-if" structure. Other parts of Aila's kernel were left unchanged. For software-based ray tracing, the "while-while" structure is better than the "while-if" structure [3]. Consequently, Aila's kernel and Kernel 1 used by the DRS are slightly different from each other. For both kernels, the BVH acceleration structure is used and accessed through the L1 texture cache, which is consistent with Aila's kernel. Aila's kernel can spawn 48 warps, and Kernel 1 used for the DRS can spawn 60 warps depending on the number of registers used per thread in each kernel.

4.2 Extra Register Bank

Two parameters, namely, the size of the extra register bank and the number of swap buffers, are known to influence the hardware cost of the DRS. In the DRS, the extra register bank is used to accommodate extra backup rays, because when a warp has different ray states, this warp can be scheduled to work on another row of rays, which have the same traversal state without a stall. The swap buffers are used to temporarily store ray variables for ray swapping between the two rows. For Kernel 1 used by the DRS, the variables of a ray are composed of 17 integers and floats. A swap buffer can only store one integer or one float. Hence, using additional swap buffers results in the short time of ray swapping costs.

RIGHTSLINK

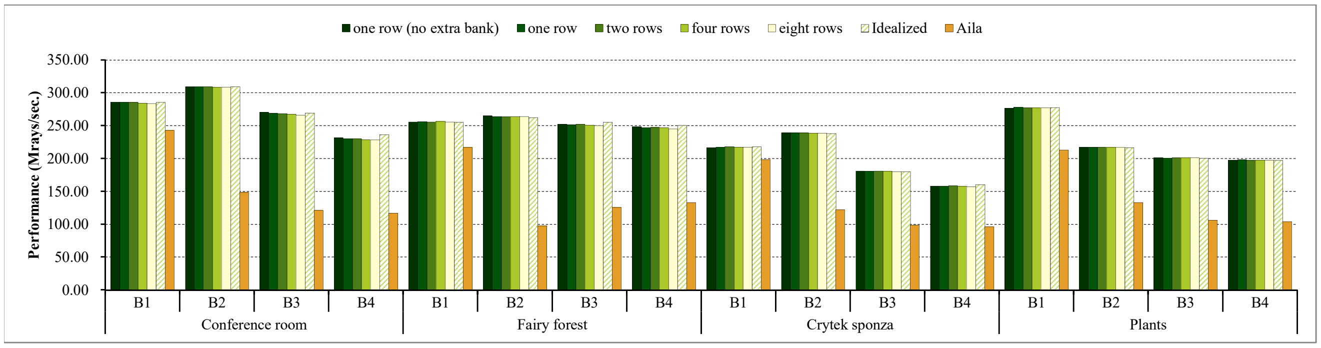


Figure 8: Simulated ray tracing performance (Mrays/sec.) for 2 million rays with different configurations of backup ray rows

This subsection explores the influence of the extra register bank size on the ray tracing performance. The sensitivity study of the swap buffers will be examined in the next subsection.

We evaluated the first four bounces for each scene and two million rays for each bounce to explore the parameters, because the overhead of using a GPU simulator to evaluate ray tracing is extremely high. For the sensitivity study of the extra register bank size, we used a configuration of nine swap buffers. The swap buffers are evenly divided for the *leaf*-state ray collecting, *fetching*-state ray collecting, and *inner*-state ray ejecting, indicating that each of them has three swap buffers for ray swapping. The size requirement of the extra register bank depends on the number of rows used for backup rays. We examined using one row, two rows, four rows and eight rows of backup rays for the DRS. We also examined a configuration of using one row of backup rays but without using the extra register bank. For this configuration, the original register file has to make room for one row of backup rays. Therefore, the number of spawned warps is reduced from 60 to 58. The simulated ray tracing performance is depicted in Figure 8, where Bn is short for bounce #n. For comparison purposes, the results of an idealized DRS (the ray shuffling can be completed in one cycle), and Aila's method are also presented in Figure 8.

One of the conclusions that can be drawn from the experimental results of Figure 8 is that the DRS greatly improves the ray tracing performance when compared with Aila's software method. The detailed analysis will be presented later.

Another important and interesting conclusion that can be drawn from Figure 8 is that the ray tracing performance is insensitive to the extra register bank size. This conclusion may be contrary to one's first instinct, because further backup rays may lead to a few warp issue stalls incurred by ray shuffling, and the performance should be improved consequently. The experimental data show that the number of warp issue stalls on the *rdctrl* instruction is reduced with increasing rows of backup rays, as depicted in Figure 9 (given the limited paper space, only the results of the *conference room* and the *fairy forest* benchmarks are presented, and the results of other benchmarks show the similar trend).

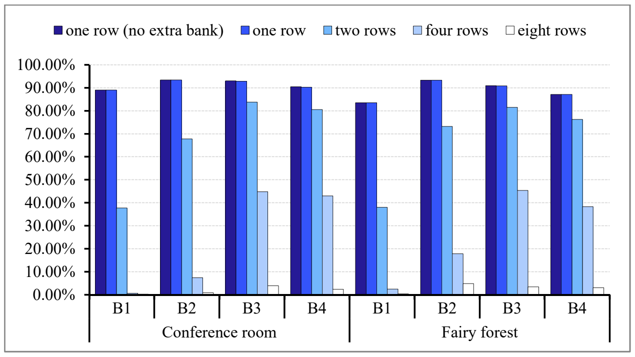


Figure 9: Warp issue stall rate of the *rdctrl* instruction in the *conference room* and the *fairy forest* benchmarks

Figure 9 indicates that using one-row backup rays can incur the warp issue stall of the *rdctrl* instruction in the rates of 83.50%-93.45%. In comparison, using eight-row backup rays only incur a stall rate of 4.81% at the maximum. One may raise the question that why the high warp issue stall rates (83.50%-93.45%) does not degrade the performance. The investigation into this issue reveals two primary reasons. First, the warp issue stalls incurred by ray shuffling only occur to the *rdctrl* instruction, because the *rdctrl* instruction is used to enable the DRS control to bind the current warp to an idle row of rays that have the same ray state. After a warp is bound to a row of rays by executing the *rdctrl* instruction, the ray shuffling scheme will not affect the execution of the warp until the *rdctrl* instruction is scheduled for issue again. The issue of the *rdctrl* instruction does not occur frequently, given that when compiled into the assemble code, the main while loop of Kernel 1 is composed of over 300 lines of instructions, where the *rdctrl* instruction only takes up one line. For the configuration of one-row backup rays, the experimental data show that the *rdctrl* instruction scheduled for issue only occupied a maximum of 6.63% of the total warp scheduled for all the ray bounce tests examined. The second reason depends on the abundant warps spawned (60 warps). Each SMX of the GTX780 GPU has four warp schedulers with eight instruction dispatch units. Because the issue of the *rdctrl* instruction does not occur frequently, a warp scheduler can usually switch to schedule the instructions of another warp if the warp scheduler fails to issue the *rdctrl* instruction. The infrequent execution of the *rdctrl* instruction does not mean the warp divergence has a low occurrence frequency. The *rdctrl* instruction is a key point for handling warp divergence. If the divergence is not handled at the *rdctrl* instruction, then the warps executed afterwards will all have low SIMD utilization due to divergence.

In Figure 8, the differentials between the different configurations of backup ray rows are minimal, and the ray tracing performances are close to that of the idealized ray shuffling. An interesting phenomenon is that the ray tracing performance is usually slightly improved with fewer backup ray rows. In particular, the configuration of one-row backup rays achieves the best ray tracing performance for most cases. Further investigation into the experimental data reveals that additional backup rays usually lead to added memory requests (added nodes of the BVH structure are accessed by different rays) resulting in L1 cache thrashing and the slight degradation of the cache hit rate. Another interesting phenomenon is that for some scenes (*conference room*, *fairy forest*, and *crytek sponza*), the ray tracing performance of the second bounce (B2) is higher than that of the primary rays. The rays of B2 are generated from the surfaces that the primary rays hit. Given that lots of the primary rays were casted towards the ground in the scenes, most of these primary rays were reflected upwards, which resulted in quite a number of reflected rays hitting the upper bound or traveling out of a scene. One characteristic of the BVH or kd-tree acceleration structure is that the rays that hit the bounds of the scene can quickly get terminated. With improved SIMD utilization by the DRS, the average ray tracing time of B2 rays was shorter than that of primary rays for some scenes, because many of the B2 rays quickly got terminated. For the *plants* scene, most of the reflected rays were occluded by densely distributed triangles. Thus, this phenomenon does not occur.

In summary, configuring one-row backup rays without using extra register bank is sufficient for the DRS (in fact, this configuration achieves the best ray tracing performance for most cases). We will use this configuration for the following experiments.

4.3 Swap Buffers

As above mentioned, the swap buffers are used to temporarily store ray variables for ray swapping. The more swap buffers used, the more ray variables can be swapped simultaneously, which results in reduced ray-swapping time. We examined four swap buffer configurations: 6, 9, 12, and 18 swap buffers. For each configuration, the swap buffers are evenly divided for the *leaf*-state ray collecting, *fetching*-state ray collecting, and *inner*-state ray ejecting. The experimental results are listed in Table 2. The best configuration for each test is indicated by the bold type.

#buffer / Tests		#6	#9	#12	#18
Conference room	B1	285.27	**285.79**	285.63	285.32
	B2	309.10	**309.22**	309.06	309.07
	B3	**270.88**	270.61	270.42	270.70
	B4	**231.93**	231.47	230.92	230.70
Fairy forest	B1	255.35	255.70	**255.96**	255.86
	B2	264.52	**264.91**	263.80	264.21
	B3	**253.50**	252.16	252.84	251.96
	B4	**249.04**	248.63	247.57	246.78
Crytek sponza	B1	**216.73**	216.44	217.00	216.45
	B2	238.82	238.94	238.98	**239.22**
	B3	**186.45**	186.11	185.53	185.79
	B4	**168.99**	168.26	168.40	168.12
Plants	B1	276.84	276.73	**277.30**	277.24
	B2	**217.66**	216.79	217.30	217.32
	B3	201.02	201.10	**201.14**	200.90
	B4	**198.03**	197.75	197.83	197.58

Table 2: Ray tracing performance with different configurations of swap buffers

Similar to the case of the extra register bank, the data of Table 2 indicate that the influence of different swap buffer configurations on the performance is unclear, and no one configuration that can always achieve the best performance (in summary, the six-buffer configuration achieves the best performance for the majority of cases). The performance differentials between the different swap buffer configurations are minimal and usually within 1 Mrays per second. Further investigation of the experimental data indicates that the average clock cycles cost by ray swapping for each configuration (6, 9, 12, and 18) are 31.6, 25.0, 24.3, and 22.0, respectively. This result indicates that increasing the number of swap buffers does not make ray swapping time be markedly reduced. This result is due to the register data transfer for ray swapping is also affected by the bank conflicts of a register file. The effect of reducing ray swapping time on the overall performance is limited, because the influence of warp issue stalls caused by ray shuffling on the performance is limited.

In summary, the configuration of one backup ray row with six swap buffers and without using extra register bank can be a good default configuration for the DRS. We used this configuration in the overall performance and hardware overhead evaluation. The primary reason that the configuration parameters of the DRS have small influence on the performance lies in the abundant warps spawned. Compared with long latency operations such as memory access, the latency of ray shuffling is relatively short which can be easily hidden by simultaneous execution of abundant warps. Hiding latencies by massively parallel execution of threads/warps is one important design philosophy of current GPUs.

RIGHTSLINK

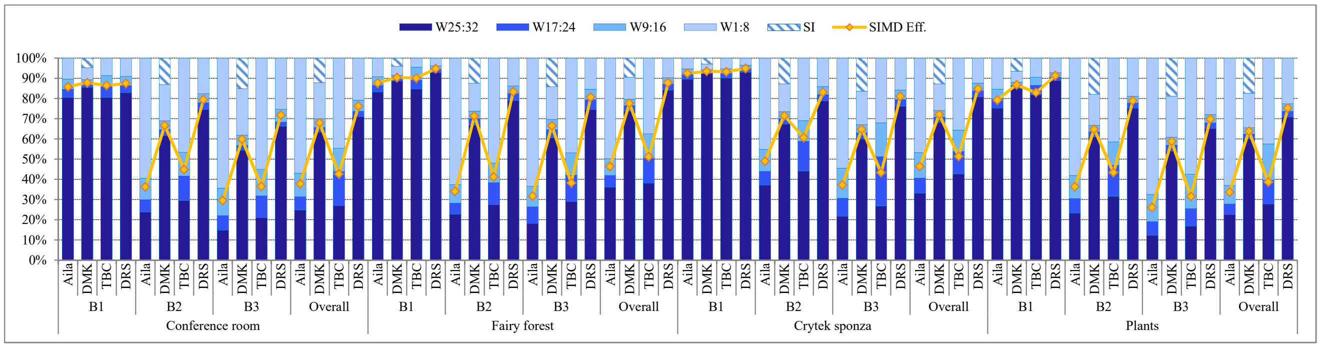


Figure 10: SIMD efficiency and utilization breakdown

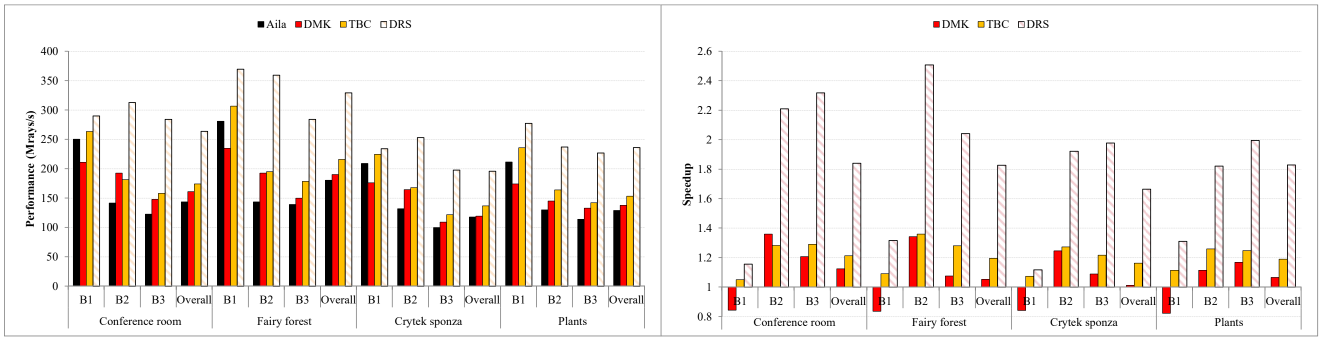


Figure 11: (a) Simulated ray tracing performance (Mrays/sec.) and (b) the speedup comparisons of DMK and DRS (speedups are normalized to the performance of Aila's software-based method)

4.4 Overall Performance Evaluation

We compared the DRS with two previously proposed thread-recombining techniques: Dynamic Micro-Kernel (DMK) [40] and Thread Block Compaction (TBC) [12]. The DMK was designed to improve the SIMD efficiency of ray tracing on GPUs, which shares the same goal with the DRS. TBC targets for improving performance for various divergent applications on GPUs. For evaluating the DMK, the on-chip spawn memory per SMX was configured to have 32 banks. For evaluating TBC, each thread block was configured to have 6 warps, which is the same with the configuration specified in [12]. One thing should be noted that TBC assumes a per-SIMD-lane addressable register file model. With this model, when threads are regrouped into different warps, simultaneous accesses to the register file from different threads of a new formed warp are not affected. However, to save the energy and area of a register file, current GPU register files are per-warp addressable, i.e. accessed with warp IDs, which is also used by GPGPU-Sim 3.x. We implemented TBC on GPGPU-Sim 3.2.1, but assumed that simultaneous accesses to the register file from the threads of a new formed warp are not affected in TBC.

Figure 10 presents the SIMD efficiency and utilization breakdown of Aila's software-based ray tracing method, DMK, TBC, and our DRS approach. The same with Figure 2, category $Wm : n$ is the percentage of warp instructions issued to the pipeline that have m to n active threads in the warp. To make clear comparisons, the statistics of the micro-kernel spawn-related instructions (instructions used for micro-kernel spawn, data dumping and loading, which is indicated as **SI** in Figure 10) in the DMK is separated from the statistics of other instructions in the SIMD utilization breakdown, because these extra instructions do not contribute to the overall computation. Figure 11 shows the simulated ray tracing performance and the speedup comparisons of DMK, TBC, and DRS. Given the limited paper space, for each scene benchmark, only the results of the first three bounces and the overall results are illustrated in Figures 10 and 11. This limitation does not destroy the overall illustration of the experimental results, because the SIMD utilization and the ray tracing performance of the last five bounces are very close to that of the third bounce, which is similar to the trend expressed in Figure 2. The similarity is due to the rays after the third bounce are highly randomized over the space of the scene, thus making a slight difference on the results of different bounces. The overall SIMD efficiency is computed as $\frac{\sum_{i=1}^{n}(Active\ threads\ of\ warp\ inst_i)}{n\times 32}$, where n is the total number of warp instructions issued in all of the eight bounces, and the overall ray tracing performance is computed as $\frac{Total\ number\ of\ rays\ traced\ in\ all\ 8\ bounces}{Total\ clock\ cycles\ of\ all\ 8\ bounces}$.

Figure 10 illustrates that the DRS significantly improves SIMD efficiency for the case of secondary rays (B2-B8) when compared with Aila's software method. For example, for the *conference room* benchmark, the SIMD efficiency of Aila's method on the secondary rays is 28.01%-36.18%, whereas the SIMD efficiency of the DRS is 72.26%-79.46%. For primary rays, the SIMD efficiency improvement is slightly insignificant, because the SIMD efficiency of Aila's method on primary rays is already high (79.24%-92.49%). The reason for this is that the primary rays which are generated from the camera are usually coherent and not prone to cause control flow divergences, whereas the secondary rays are usually incoherent because of randomization and prone to cause divergences. In summary, compared with Aila's method, the DRS improves the overall SIMD efficiency from 33.70%-46.42% to 75.18%-87.79%, and from 41.06% to 81.04% on average. One thing should be noted that the DRS focuses on resolving the primary warp divergences in the ray tracing kernel, which are caused by inconsistent ray states and overlooks other potential minor warp divergences. For example, within the three "if" statements of Kernel 1, several branch statements may cause control flow divergences. This is the reason for the DRS to fail from a SIMD efficiency of 100%. To further improve the SIMD efficiency and the performance, the approach that explores intra-warp parallelism [7, 10, 23, 33, 38] can be used with the DRS. The DRS does not require modifications on the warp instruction control logic, the DRS approach is orthogonal to these techniques. One may concern about register file accesses increased by ray shuffling. Experimental data show that for primary rays, the register file accesses caused by ray shuffling occupy 7.36% of the total register file accesses for the four scenes on average. For secondary rays, this percent increases to 18.79%. Although the register file access pressure is increased to some extent by ray shuffling, the total number of register file accesses is decreased due to improved SIMD utilization when compared with the software method.

The SIMD efficiency of the DMK is higher than that of Aila's software method but lower than that of the DRS. Specifically, the overall SIMD efficiency improvement of DMK on Aila's method is 25.84%-31.23%, and 29.35% on average, but the SIMD efficiency reduction of DMK on the DRS is 8.34%-12.73%, and 10.63% on average. The SIMD efficiency reduction is mainly caused by the micro-kernel spawn-related instructions. For coherent primary rays, the spawn related instructions represent 2.76%-6.32% of the total instructions executed. For incoherent secondary rays, the spawn-related instructions constitute 11.21%-18.92% of the total instructions executed. Interestingly, if the spawn-related instructions are excluded, then the SIMD efficiencies of the DMK and the DRS are close to each other.

Compared with DMK and DRS, the SIMD efficiency of TBC is relatively lower. Specifically, the overall SIMD efficiency of TBC is 38.72%-51.36%, and 46.01% on average. Its SIMD efficiency improvement over Aila's method is 12.08% on average. The reasons behind this are as follows. First, although the target of TBC is improving SIMD utilization, a compacted warp with all its threads being active is not guaranteed in TBC. In comparison, both DMK and DRS aim for 100% SIMD utilization for ray traversal. Second, TBC considers the entire thread block as a single unit when a branch diverges, and uses block-wide reconvergence stacks instead of warp reconvergence stacks. Consequently, the warps of a thread block must synchronize at divergence points to enable compaction opportunities. For TBC, the number of warps in a thread block is usually not set to very large, because large number of warps could cause long synchronization latencies. With limited number of warps, the SIMD utilization improvement by compaction is restricted.

With the significantly improved SIMD efficiency, compared with Aila's software method, the DRS can remarkably improve the overall ray tracing performance, which is mainly contributed by the performance improvement on secondary rays. Specifically, the DRS improve the performance on primary rays by 11.93%-31.68%, and 22.61% on average. For the secondary rays, the performance improvement is 58.39%-150.92%, and 90.25% on average. To render an image based on ray-tracing, tracing secondary rays represents the majority of the rendering time. In summary, in Figure 11, the DRS achieves 1.84×, 1.92×, 1.67× and 1.83× overall ray tracing performance speedup (compared with Aila's method) for the *conference room*, *fairy forest*, *crytek sponza*, and *plants* benchmarks, respectively, and 1.79× on average. Of the four benchmarks, the performance speedup on the *crytek sponza* benchmark is relatively less than that of the other benchmarks. The development of further explanation on this benchmark is significant. The *crytek sponza* benchmark incurs the worst ray tracing performance among the four benchmarks regardless of the ray tracing methods used. What is interesting is that the SIMD efficiencies of this benchmark are not the lowest. For example, for Aila's method, the overall SIMD efficiency of the *crytek sponza* benchmark is 46.32%, which is higher than that of the *conference room* and the *plants* benchmarks. The same phenomenon occurs to DMK, TBC, and the DRS. The investigation into this result reveals two related reasons. First, given its complex architectural structure, the rays are harder to be traced out of the scene or hitting the light source, i.e., hard to terminate. The *conference room* scene, although it is an indoor scene, has lights on its ceiling. The rays are easier to terminate than those in the *crytek sponza* benchmark on average. The consequence is that the rays in the *crytek sponza* scene require visiting more BVH nodes on average than those in other benchmarks. This result slows down the total ray tracing time. Second, given the increased memory divergences caused by requiring to visit additional BVH nodes, the L1 texture cache miss rates (the BVH nodes are accessed through the L1 texture cache) in the *crytek sponza* benchmark tests are relatively higher than those in the tests of other benchmarks, which is another reason for the performance slowdowns.

One may notice that although the DMK greatly improves SIMD efficiency, its performance improvement is rather limited, which is 1.06× speedup on average for overall performance. It even incurs slowdown on the case of tracing primary rays (bounce #1). In comparison, TBC shows better performance improvement than DMK, which is 1.18× speedup on average, even its SIMD efficiency improvement is much lower than that of DMK. This is because DMK suffers serious spawn memory bank conflicts that are caused by the extra data loading/dumping instructions executed for the micro-kernel spawn. To qualify the conflicts, $\frac{\#bank\ conflicts\ per\ SMX}{total\ cycles}$ can be used. Its value is 7.95% for primary rays, and is between 17.25% and 19.97% for secondary rays in the *conference room* benchmark. The situations in other benchmarks are similar. In essence, the DMK approach regroups a set of rays by splitting up old warps and forming new warps. To swap corresponding ray data, it must first copying the data from the register file into the spawn memory (splitting up old warps), then copying the data from the spawn memory back into the register file (forming new warps), indicating the process of accessing spawn memory occurs twice for regrouping a set of rays, resulting in additional bank conflicts. Furthermore, the extra cycles incurred by bank conflicts cannot be hidden as the data movement is explicitly performed by executing extra data dumping/loading instructions. As a consequence, traversing a set of rays costs more cycles in the DMK approach than that in TBC and DRS because of explicit data movement. In comparison, the regrouping of threads (i.e. compacting warps) in TBC does not need any register data movement, and the improvement of SIMD utilization directly leads to performance improvement. The ray data movement in the DRS approach is implicitly conducted by the DRS control without using any instruction. Furthermore, Aila's ray tracing kernel is the state-of-the-art ray-tracing kernel on the GPUs of NVIDIA, and is superior to the Radius-CUDA ray tracing kernel, which was used as the comparison point in [40]. This is the reason why the DMK does not show remarkable performance improvement as claimed in [40]. In summary, for the four scenes, DRS achieves an average 68.47% and 51.58% performance improvement over DMK and TBC respectively.

4.5 Hardware Overhead

One great advantage of the DRS is its marginal hardware overhead. The hardware area overhead is mainly composed of the six swap buffers and the DRS control logic. For the six swap buffers, the storage overhead is $6\times(warp_size - 1)\times 32$bits = 744 bytes. For the DRS control logic, the primary storage overhead depends on the ray state table that is used to manage live rays. The storage requirement of the ray state table is $61\times32\times2$bits = 488 bytes to manage 61×32 live rays (58 warps, one backup ray row and two empty ray rows). With some additional control state, the total storage requirement is approximately 1.4 KB per SMX. Given this, the storage overhead is 0.55% of the register file per SMX (the size of the GTX 780 GPU register file is 256 KB per SMX).

For the DMK, the minimum capacity of on-chip spawn memory required for storing the data passed between threads per SMX is $54 \times 32 \times 17 \times 32$ bits = 114.75KB, not including the memory storage required for storing the metadata of dynamic threads, spawn lookup table and additional control states. As for TBC, the main storage overhead comes from the thread IDs in the warp buffer which is $10 \times 32 \times 64$ bits = 2.5KB (1024 max threads per block and 64 max warps per SMX in the Kepler architecture). Additionally, TBC requires per-SIMD-lane addressable register file which is substantially different from the register file of Kepler GPUs.

To further prove the implementation practicality, the DRS design with register file for a GPU core was described in HDL and synthesized in TSMC 28 nm standard cell technology under typical operating conditions. The areas of the target hardware were calculated by Synopsys Design Compiler and measured at their maximum supported frequencies. Experimental results show that the DRS occupies only 0.042 mm^2 for each GPU core without using any custom logic. The cycle delay for the DRS design is 0.47ns, which can run over 2GHz. In total, for a Kepler-sized 15 SMX GPU (550mm^2), DRS adds approximately 0.11% more area. This result demonstrates that the hardware impact of the DRS is well within the reach of modern GPGPU architectures.

4.6 Further Discussion

In general, most prior micro-architectural proposals approach the warp divergence problem from the *control* side. Approaching from the *control* side requires modifications to the warp instruction control logic, and may require further modifications on the register file. In contrast, the DRS addresses the warp divergence problem from the *data* side, which leads to minor modifications to the original design of a GPU. First, the instruction control logic is left untouched. Second, the DRS is designed to be compatible with the register files of current GPUs, keeping the main structure of the register file unmodified, and the DRS control logic itself is independent of other parts of a GPU. These advantages make the DRS applicable to modern GPUs.

Physically-based rendering is the trend of future graphics processing technology [11]. For example, the PowerVR Wizard GPU is specially designed to improve the ray tracing performance [16, 29]. DRS follows this trend and focuses on improving the computation efficiency. Though its focus is ray tracing, the main idea of DRS has the potential that could benefit other divergent applications on GPUs. We can envision some characteristics of a more general method that is similar to DRS and overcomes some of the limitations of current thread-recombining approaches (for example TBC): (1) the data not the threads of different warps are shuffled; (2) as divergence is mitigated on the data side, block-wide reconvergence stack is not needed; (3) on a divergent point that synchronizes warps, once the SIMD utilization of a warp is improved to some extent by shuffling thread data with other warps, then the warp is released for issue, avoiding long synchronization latencies. In this work, we focus on the problem of ray tracing, and leave the study of applying the idea to other divergent applications in our future work.

5 RELATED WORKS

We have discussed the related works that target the control flow divergence problem on GPUs earlier. In this section, we review some other related works.

The hardware ray tracing architectures can be separated into two types: they are either dedicated architectures designed from the ground up for ray tracing or redesigned versions of programmable multicore/many-core architectures optimized for ray tracing.

Many dedicated ray-tracing architectures have been proposed over the last decade. Early hardware based architectures like SaarCOR [35] and RPU [39] use SIMD processing of ray packets, and demonstrate the benefits of using a dedicated pipeline for traversing coherent rays. The StreamRay [31] architecture includes a filter engine for incoherent rays and a ray engine. The T&I Engine [24] introduces dedicated hardware units for traversal and intersection.

Other recent research focuses on redesigning programmable multicore/many-core architectures. Govindaraju et al. proposed Copernicus [14] which is a tile-based parallel ray-racing system running on 128 programmable cores. TRaX [36] and MIMD threaded multiprocessors (TMs) [19] introduce MIMD processing which was considered to be better suited for tracing incoherent rays. However, the DRS shows that the SIMT architecture can also be highly efficient for tracing incoherent rays. Aila and Karras [2] proposed a hardware architecture based on NVIDIA Fermi GPUs to reduce memory traffic via a treelet-based approach. Keely [17] proposed a technique that improves the efficiency of ray-tracing hardware through low-cost reduced-precision arithmetic.

Software worklist (or work-queue) is an approach used by many prior works for improving the SIMD efficiency of irregular computations on GPUs [20, 28, 37]. Using worklists within the ray tracing algorithm is not very practical on GPUs. If a ray is regarded as a worklist item, to move rays between threads, the ray states have to be moved between registers and memory, which degrades the performance due to tremendously increased memory accesses. However, worklists can be used as pipelines between different stages (kernels) of a GPU path tracer to filter out terminated rays and to direct rays to their corresponding processing kernels [20].

6 CONCLUSION

This paper proposed a novel hardware architecture named DRS for improving the performance of physically-based rendering on GPUs by addressing the control flow divergence problem of the ray tracing algorithm. The control flow divergence problem is approached from the data side by the DRS, which is different from prior work. Compared with state-of-the-art software-based and hardware-based ray traversal methods, the DRS greatly improved the ray tracing performance, and required marginal hardware cost. For our future work, we will explore how to apply the idea to other divergent applications.

7 ACKNOWLEDGEMENTS

The authors want to thank the anonymous reviewers for their insightful comments. This work was supported by the NSF of China (under Grants 61433019, 61472435, 61202129, 61572058, 61472431, 61402501, U1435217), Young Elite Scientists Sponsorship Program By CAST (under Grant YESS-20150090).

REFERENCES

[1] Attila T. Áfra and László Szirmay-Kalos. 2014. Stackless Multi-BVH Traversal for CPU, MIC and GPU Ray Tracing. *Comput. Graph. Forum* 33, 1 (Feb. 2014), 129–140. https://doi.org/10.1111/cgf.12259

[2] Timo Aila and Tero Karras. 2010. Architecture Considerations for Tracing Incoherent Rays. In *Proceedings of the Conference on High Performance Graphics (HPG '10)*. Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, 113–122.

[3] Timo Aila and Samuli Laine. 2009. Understanding the Efficiency of Ray Traversal on GPUs. In *Proceedings of the Conference on High Performance Graphics 2009 (HPG '09)*. ACM, New York, NY, USA, 145–149.

[4] Timo Aila, Samuli Laine, and Tero Karras. 2012. *Understanding the Efficiency of Ray Traversal on GPUs – Kepler and Fermi Addendum*. NVIDIA Technical Report NVR-2012-02. NVIDIA Corporation.

[5] A Bakhoda, G. L Yuan, W. W. L Fung, H Wong, and T. M Aamodt. 2009. Analyzing CUDA workloads using a detailed GPU simulator. In *IEEE International Symposium on PERFORMANCE Analysis of Systems and Software*. 163–174.

[6] Jon Louis Bentley. 1975. Multidimensional Binary Search Trees Used for Associative Searching. *Commun. ACM* 18, 9 (Sept. 1975), 509–517.

[7] Nicolas Brunie, Sylvain Collange, and Gregory Diamos. 2012. Simultaneous Branch and Warp Interweaving for Sustained GPU Performance. In *Proceedings of the 39th Annual International Symposium on Computer Architecture (ISCA '12)*. IEEE Computer Society, Washington, DC, USA, 49–60.

[8] Tomáš Davidovič, Jaroslav Křivánek, Miloš Hašan, and Philipp Slusallek. 2014. Progressive Light Transport Simulation on the GPU: Survey and Improvements. *ACM Trans. Graph.* 33, 3, Article 29 (June 2014), 19 pages. https://doi.org/10.1145/2602144

[9] Gregory Diamos, Benjamin Ashbaugh, Subramaniam Maiyuran, Andrew Kerr, Haicheng Wu, and Sudhakar Yalamanchili. 2011. SIMD Re-convergence at Thread Frontiers. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-44)*. ACM, New York, NY, USA, 477–488. https://doi.org/10.1145/2155620.2155676

[10] Ahmed ElTantawy, Jessica Wenjie Ma, Mike O'Connor, and Tor M. Aamodt. 2014. A scalable multi-path microarchitecture for efficient GPU control flow. *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)* 00 (2014), 248–259.

[11] Marcos Fajardo. 2014. How Ray Tracing Conquered Cinematic Rendering, Keynote. In *Proceedings of High Performance Graphics (HPG '14)*.

[12] Wilson W. L. Fung and Tor M. Aamodt. 2011. Thread Block Compaction for Efficient SIMT Control Flow. In *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture (HPCA '11)*. IEEE Computer Society, Washington, DC, USA, 25–36.

[13] Wilson W. L. Fung, Ivan Sham, George Yuan, and Tor M. Aamodt. 2007. Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 40)*. IEEE Computer Society, Washington, DC, USA, 407–420.

[14] Venkatraman Govindaraju, Peter Djeu, Karthikeyan Sankaralingam, Mary Vernon, and William R. Mark. 2008. Toward a Multicore Architecture for Real-time Ray-tracing. In *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 41)*. IEEE Computer Society, Washington, DC, USA, 176–187.
[15] Khronos Group. 2016. *The OpenCL C Specification V2.0*. Technical Report.
[16] Imagination. 2017. *PowerVR Graphics Processors*. Technical Report.
[17] S. Keely. 2014. Reduced Precision for Hardware Ray Tracing in GPUs. In *Proceedings of High Performance Graphics (HPG '14)*. Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, 29–40.
[18] Daniel Kopta, Konstantin Shkurko, Josef Spjut, Erik Brunvand, and Al Davis. 2013. An Energy and Bandwidth Efficient Ray Tracing Architecture. In *Proceedings of the 5th High-Performance Graphics Conference (HPG '13)*. ACM, New York, NY, USA, 121–128.
[19] D. Kopta, J. Spjut, E. Brunvand, and A. Davis. 2010. Efficient MIMD architectures for high-performance ray tracing. In *IEEE International Conference on Computer Design*. 9–16.
[20] Samuli Laine, Tero Karras, and Timo Aila. 2013. Megakernels Considered Harmful: Wavefront Path Tracing on GPUs. In *Proceedings of the 5th High-Performance Graphics Conference (HPG '13)*. ACM, New York, NY, USA, 137–143. https://doi.org/10.1145/2492045.2492060
[21] Won-Jong Lee, Youngsam Shin, Jaedon Lee, Jin-Woo Kim, Jae-Ho Nah, Seokyoon Jung, Shihwa Lee, Hyun-Sang Park, and Tack-Don Han. 2013. SGRT: A Mobile GPU Architecture for Real-time Ray Tracing. In *Proceedings of the 5th High-Performance Graphics Conference (HPG '13)*. ACM, New York, NY, USA, 109–119.
[22] G. Liktor and K. Vaidyanathan. 2016. Bandwidth-efficient BVH Layout for Incremental Hardware Traversal. In *Proceedings of High Performance Graphics (HPG '16)*. Eurographics Association, Aire-la-Ville, Switzerland, 51–61.
[23] Jiayuan Meng, David Tarjan, and Kevin Skadron. 2010. Dynamic Warp Subdivision for Integrated Branch and Memory Divergence Tolerance. In *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA '10)*. ACM, New York, NY, USA, 235–246.
[24] Jae-Ho Nah, Jeong-Soo Park, Chanmin Park, Jin-Woo Kim, Yun-Hye Jung, Woo-Chan Park, and Tack-Don Han. 2011. T&I Engine: Traversal and Intersection Engine for Hardware Accelerated Ray Tracing. In *Proceedings of the 2011 SIGGRAPH Asia Conference (SA '11)*. ACM, New York, NY, USA, Article 160, 10 pages.
[25] Veynu Narasiman, Michael Shebanow, Chang Joo Lee, Rustam Miftakhutdinov, Onur Mutlu, and Yale N. Patt. 2011. Improving GPU Performance via Large Warps and Two-level Warp Scheduling. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-44)*. ACM, New York, NY, USA, 308–317.
[26] NVIDIA Corporation. 2012. *NVIDIA Kepler GK110 Architecture Whitepaper V1.0*. Technical Report.
[27] NVIDIA Corporation. 2017. *CUDA C Programming Guide V8.0*. Technical Report.
[28] Sreepathi Pai and Keshav Pingali. 2016. A Compiler for Throughput Optimization of Graph Algorithms on GPUs. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2016)*. ACM, New York, NY, USA, 1–19. https://doi.org/10.1145/2983990.2984015
[29] Luke Peterson and Tobias Hector. 2016. Ray Tracing on the Wizard GPU. In *Imagination Developers Connection (IDC '16)*. https://community.imgtec.com/downloads/ray-tracing-api-overview/
[30] Matt Pharr and Greg Humphreys. 2004. *Physically Based Rendering: From Theory to Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.
[31] Karthik Ramani, Christiaan P. Gribble, and Al Davis. 2009. StreamRay: A Stream Filtering Architecture for Coherent Ray Tracing. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIV)*. ACM, New York, NY, USA, 325–336.
[32] Minsoo Rhu and Mattan Erez. 2012. CAPRI: Prediction of Compaction-adequacy for Handling Control-divergence in GPGPU Architectures. In *Proceedings of the 39th Annual International Symposium on Computer Architecture (ISCA '12)*. IEEE Computer Society, Washington, DC, USA, 61–71.
[33] Minsoo Rhu and M. Erez. 2013. The dual-path execution model for efficient GPU control flow. In *Proceedings of IEEE Symp. on High Performance Computer Architecture (HPCA '13)*. 235–246.
[34] Timothy G. Rogers, Mike O'Connor, and Tor M. Aamodt. 2012. Cache-Conscious Wavefront Scheduling. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-45)*. IEEE Computer Society, Washington, DC, USA, 72–83.
[35] Jörg Schmittler, Sven Woop, Daniel Wagner, Wolfgang J. Paul, and Philipp Slusallek. 2004. Realtime Ray Tracing of Dynamic Scenes on an FPGA Chip. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware (HWWS '04)*. ACM, New York, NY, USA, 95–106.
[36] Josef Spjut, Andrew Kensler, Daniel Kopta, and Erik Brunvand. 2009. TRaX: A Multicore Hardware Architecture for Real-time Ray Tracing. *Trans. Comp.-Aided Des. Integ. Cir. Sys.* 28, 12 (Dec. 2009), 1802–1815.
[37] Markus Steinberger, Michael Kenzel, Pedro Boechat, Bernhard Kerbl, Mark Dokter, and Dieter Schmalstieg. 2014. Whippletree: Task-based Scheduling of Dynamic Workloads on the GPU. *ACM Trans. Graph.* 33, 6, Article 228 (Nov. 2014), 11 pages. https://doi.org/10.1145/2661229.2661250
[38] Aniruddha S. Vaidya, Anahita Shayesteh, Dong Hyuk Woo, Roy Saharoy, and Mani Azimi. 2013. SIMD Divergence Optimization Through Intra-warp Compaction. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA '13)*. ACM, New York, NY, USA, 368–379.
[39] Sven Woop, Jörg Schmittler, and Philipp Slusallek. 2005. RPU: A Programmable Ray Processing Unit for Realtime Ray Tracing. In *ACM SIGGRAPH 2005 Papers (SIGGRAPH '05)*. ACM, New York, NY, USA, 434–444.
[40] Joseph Zambreno and Michael Steffen. 2010. Improving SIMT Efficiency of Global Rendering Algorithms with Architectural Support for Dynamic Micro-Kernels. *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2010)* (2010), 237–248.