DOI: 10.1111/cgf.14873
High-Performance Graphics 2023
J. Bikker and C. Gribble
(Guest Editors)

COMPUTER GRAPHICS *forum*
Volume 42 (2023), Number 8
e14873 *pp. 1 - 11*

Data Parallel Multi-GPU Path Tracing using Ray Queue Cycling

Ingo Wald[†] Milan Jaroš[‡] Stefan Zellmann[◇]

[†]NVIDIA [‡]IT4Innovations, VSB – Technical University of Ostrava, Ostrava, Czech Republic [◇]University of Cologne

Figure 1: *A high-resolution version of the Disney Moana* island *model, with nearly 600 million triangles before instancing, 31 million instances, and 33 GB of textures, for a total of 84 GBs of model data excluding acceleration structures. At* 2560×1080 *pixels and* 8 *paths per pixel, our method runs this at* 2.9 *frames per second (FPS) on a DGX-2 (with 16 Volta class GPUs and NVLink and NVSwitch), at* 2.9 *FPS on an HGX (similar architecture, but with 8 A100 GPUs), and at* 6.0 *FPS, respectively, on an RTX Server with 8 Ampere class GPUs with ray tracing cores on PCIe. An important feature of our method is that it is almost entirely oblivious to how geometry gets partitioned across GPUs, and does not require any spatially or object-space coherent assignment whatsoever. Right: A false-color image where an object's color encodes which GPU it is on; showing a near-random assignment that works just fine in our method.*

Abstract

We propose a novel approach to data-parallel path tracing on single-node/multi-GPU hardware that builds on ray forwarding, but which aims—above all else—at generality and practicability. We do this by avoiding *any attempts at reducing the number of traces or forward operations performed, and instead focus on always using all GPUs' aggregate compute and bandwidth to effectively trace each ray on every GPU. We show that—counter-intuitively—this is both feasible and desirable; and that when run on typical data-center/cloud hardware, the resulting framework not only achieves good performance and scalability, but also comes with significantly fewer limitations, assumptions, or preprocessing requirements than existing techniques.*

1. Introduction

Modern GPUs have become highly efficient at ray tracing, and what can or cannot efficiently be ray traced today is almost entirely governed by what can or cannot be fit into GPU memory. For games, this constraint is addressed by aggressively managing what is or is not in GPU memory at any point in time—using techniques like streaming, LOD, compression, etc. For more general rendering outside of gaming, however, such techniques seem to be less applicable, and limited GPU memory is frequently a serious issue.

One solution to rendering models larger than GPU memory is to adopt data-parallel rendering, where the model gets distributed across the memories of multiple different GPUs and/or nodes that then work together. This is not applicable to gaming where users only have a single GPU; but for most professional uses of rendering more than one GPU is either already the norm, or an easy-to-adopt option. However, despite a rich history of data parallel rendering research, in practice this technology seems to be entirely confined to scientific visualization, and hardly used at all outside of that field.

Why this may be so is an interesting topic for debate; however, we believe the three most important reasons are the following: first, most existing approaches to data parallel rendering have focused on multi-node cluster/MPI setups, but those are often constrained in terms of bandwidth, and are too complicated to set up and use for the average user. Second, existing approaches in sci-vis typically rely on *sort-last* image compositing, which does not work at all for path tracing. Data parallel path tracing requires very different communication patterns—either frequent forwarding of rays, or fetching data on the fly—that are significantly more challenging to realize. Lastly, for existing techniques that do use fetching or forwarding, it matters *a lot* how exactly the scene is partitioned

across the different GPUs (see, e.g., the discussion in [ZMWP20]), which in practice means that these techniques are inherently fragile regarding which content they can or cannot handle well.

In this paper, we address these three issues by following the mantra of *make it simple, make it work*. First, we exclusively focus on single-node, multi-GPU hardware; this cannot scale to the kind of "hero run" problems sometimes encountered in sci-vis, but for most applications reduces the problem of what the user can render to what machine he or she has access to. Machines such as NVIDIA DGX-2 and A100/HGX (or similar hardware from other vendors) today are widely available in many data centers, supercomputers, and through virtually any cloud provider; and often have aggregate GPU memory that is sufficient for even very large production models. Second, we consciously abandon the very idea of using some clever partitioning to minimize the amount of work or bandwidth generated—and instead focus on maximizing the *throughput* of the aggregate system by fully utilizing every GPU's compute and bandwidth in every stage of our method. In particular, our method will eventually trace every ray on every GPU. This sounds very inefficient, but as we will show it is not: doubling the number of GPUs does indeed double the number of times any ray needs forwarding and tracing—but it also doubles the overall system's aggregate compute and memory capabilities available for doing that, and simultaneously also reduces how many rays each GPU has to trace; so when properly utilizing all resources these effects cancel each other out. Tracing each ray on every GPU means that it no longer matters how the scene gets partitioned across different GPUs, allowing our method to be applied to virtually any input, with scene content assigned to GPUs on-the-fly, and with no constraints other than that the model must fit into the aggregate memory of all GPUs.

2. Background and Related Work

We assume familiarity with the basics of modern ray and path tracing; for reference, we point the reader to Alarcon's explanation of the RTX pipeline [Ala20], and Boksanksy and Marrs' *Reference Path Tracer* [BM21]. We also assume familiarity with the concept of *wavefront* path tracing (see, e.g., [LKA13]).

Path tracing depends on the ability to efficiently trace lots of rays. Our method's core ideas are general, but for this paper, we explicitly target modern GPUs. Leveraging such GPUs' ray tracing capabilities requires the use of APIs such as OptiX [PBD*10], DirectX, or Vulkan. In this paper, we build on top of NVIDIA OptiX [PBD*10], using the OWL library [WMH20], but the same techinques should also map to other vendors' APIs, or to vendor-independent APIs such as DXR or Vulkan.

2.1. Data Parallel Rendering

When dealing with models larger than what fits into a single node or GPU, one option is to use *out of core*, in which geometry and/or rays are temporarily paged out to disk or host memory. This was first proposed in Pharr's seminal *memory coherent ray tracing* [PKGH97]; a more recent example is Disney's *Hyperion* renderer [BAC*18]. An alternative to out of core rendering is *data parallel rendering*, where the model gets split across the memories of multiple different render nodes or GPUs. This was used as early as the nineties to realize ray tracing on early parallel computers [SG89]. Today, data parallel rendering is almost entirely constrained to scientific visualization (sci-vis), typically via sort-last image compositing [Mor11, Eil19].

For ray and path tracing, image compositing does not apply. In this context, methods can be classified into those that *fetch* scene data to the processors that need them; or those that *send/forward* rays to whichever processor has the data that those rays need.

Data fetching. usually relies on caching to reduce bandwidth [WSB01, DGP04, IBH11]; this works great *most of the time*, but tends to catastrophic stalls whenever sudden changes in visible content invalidate the caches. Jaros et al. [JRSS21] described a method that also uses fetch-and-cache, but leverages NVIDIA GPUs' *managed memory* to fetch data with driver- and hardware support. To further reduce memory transfers they also pre-compute which virtual memory pages will get the most accesses, and replicate those to each GPU. The downside to their approach is that this requires a-priori knowledge of what the user will render, and that this is incompatible with vendor-optimized and/or hardware-accelerated solutions where the acceleration structure cannot be changed by the user, or even with hardware texture units that cannot access memory on other GPUs.

Ray forwarding has been looked at by several different researchers (e.g., [SG89, NCFL14, Nav10, Rei95]. Fouladi et al. [FSP*22] proposed this for low-cost (offline-)rendering in the cloud. Wald and Parker [WP22] recently proposed the *BriX* framework that uses a combination of object-space partitioning, partial replication, and cleverly designed next-node kernels to reduce the number of times rays need to be sent across the network. *BriX* did show that scene partitioning does not have to be spatial—but itself also heavily relies on a suitable (object-space) partitioning. In *Kilauea*, Kato et al. [KS02] proposed an architecture where the scene could be arbitrarily distributed across multiple nodes. A single rendering node would broadcast every ray to every scene node, then select the closest intersection returned by any such node.

Partitioning. Irrespective of underlying algorithm, data parallel rendering requires some sort of *partitioning* of the model into smaller pieces. This is typically done via *spatial* partitioning; however, this can be problematic for modern production content with lots of instances, hard-to-split objects, or highly varying geometric density [ZMWP20]. In a sci-vis context, the model partitioning can also be pre-ordained by the application that produced the data.

2.2. (Multi-)GPU Technology

Throughout this paper it is important to understand how device memory works, and how data can move from one GPU to another. On the lowest level, each GPU talks to any other GPU—or main memory—via PCIe, or via NVLink where available. Conceptually both NVLink and PCIe behave the same way, except for NVLink having higher bandwidth and lower latency (e.g., PCI 4.0 has a theoretical peak of 32 GB/s in each direction [Sol14], while fourth-generation NVLink on a HGX-2 has up to 900 GB/s [Cor20]).

Both NVLink and PCIe allow for what is referred to as *peer access*, where GPUs can directly read from, or write to, other GPUs'

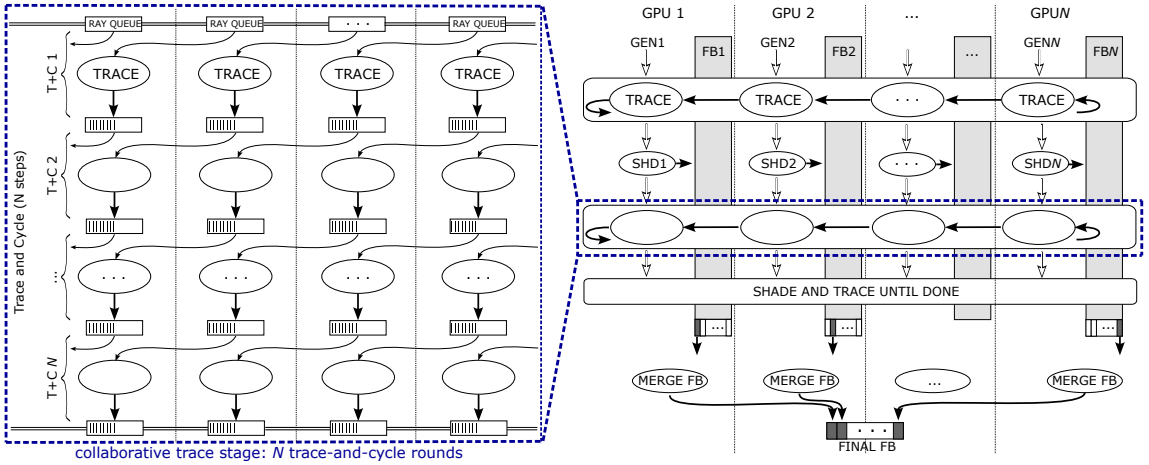

Figure 2: *Illustration of our method. Given N GPUs (that each have their own queue rays, for one-N'th of all pixels) rays are traced through N successive* trace-and-cycle *(T&C) stages. In each such T&Cstage (left) each GPU reads the rays from its right neighbor, traces these against the data stored on that GPU, and then stores these rays in its own outgoing ray queue. After N such stages each ray has been traced on each GPU, and is on the GPU that originally spawned. Right: the entire pipeline, with each rounded-edge* trace *box corresponding to the N steps on the left. Each GPU traces one-N'th of all pixels; and has a local accumulation buffer for those pixels that its* shade *stages write into. Once all paths have been traced these get tone mapped and merged into the final application frame buffer.*

physical memories. Peer access only works for GPUs that are *direct* peers, such as being on the same PCI root complex, or on the same NVLink bridge or switch. Older hardware like DGX-1 allowed peer access only between certain pairs of GPUs; newer hardware is *fully switched*, meaning each GPU can peer access any other. Both PCIe and NVLink are *bidirectional* (each GPU can both send *and* receive at the same time), and at least when fully switched will also allow for multiple concurrent communications, meaning *aggregate* bandwidth is much higher than for any individual GPU.

Modern GPUs have *unified virtual addressing*, where the GPU performs translation between virtual and physical addresses. When using CUDA *managed memory*, this can be used to share allocated memory across multiple GPUs, and/or host memory. Depending on how the driver has mapped the pages, a given memory access can either be resolved locally, or through peer access, or it can cause a page fault that the driver then uses to either migrate or copy the respective page. Using `cudaMemAdvise` the program can *suggest* certain access patterns, but the actual mapping is up to the driver.

We observe that NVLink is a vendor specific technology, but that PCIe, peer access, virtual addressing of GPU memory, etc., are not.

3. Method Overview

The fundamental insight that motivated our method is twofold: First, that in most data parallel techniques "clever" scene partitioning is the key to performance—but also the main source of overhead and limitations regarding what content that technique can or cannot handle. Second, that at least on single-node multi-GPU hardware it is both feasible and desirable to *not* use a clever partitioning, and instead, to trace each ray on every GPU. This is desirable because *if* we can do so we automatically solve the aforementioned source of limitations, getting an approach to data parallel rendering that will be *much* more general in what kind of content it can handle. Counter-intuitively, it is also feasible: tracing each ray on every GPU does indeed consume a lot of compute and bandwidth, and doubling the number of GPUs doubles that cost—but that at least when the GPUs are connected in the right manner (using fully switched PCIe or NVLink) doubling the number of GPUs will also double the aggregate compute and bandwidth available to do this, and will have only half as many rays per GPU per step.

The key aim of our method thus is to trace every ray on every GPU, and to leverage the aggregate bandwidth and compute to achieve this. This means we avoid techniques that channel work to specific GPUs, and instead use every GPU in every step. To do this, we adopt a design where we view all N GPUs as forming a ring in which each GPU has a clearly defined left and right neighbor it communicates with; each GPU contains some of the scene geometry, with few constraints as to how this partitioning is attained.

When starting a new frame, each GPU will initially generate its own wavefront of primary rays, for one N'th of the pixels (i.e., all GPUs *together* trace all pixels). All GPUs then trace their rays against their own geometry, in parallel; i.e., each GPU traces a different one-N'th of all rays against a different one-N'th of the geometry. Once all GPUs are done with this step we perform what we call a ray queue *cycling* step, where all GPUs concurrently pass their current set of partially traced rays to their respective right neighbor, taking their left neighbor's rays in turn. Each GPU then again traces the rays it now has—at which point all rays have been traced against two N'th of the geometry, etc. After N such trace-and-cycle (T&C) steps, each group of rays arrives back on the GPU that they were once spawned on—but every ray has now been traced on every GPU, against every piece of scene content. The GPUs can now perform shading of their rays, can each generate a new secondary wavefront for their portion of the pixels, etc.

When considering this algorithm, intuition wrongly suggests this to be a terrible idea, because doubling the number of GPUs obviously doubles the number of times each ray has to be forwarded and traced—which *seems* very inefficient. However, it is not: if we

double the number of GPUs we also have each GPU trace half as many rays, and we have twice the aggregate number of cores and bandwidth to do that. Since these effects do cancel each other out we cannot necessarily expect a major speed-up from adding more GPUs; but we absolutely *can* expect to be able to trace models that are N times as large without getting slower. And once we have enough GPUs we can also employ island parallelism (Section 4.11) to gain both larger models *and* higher performance, too.

4. Implementation

Whereas the preceding section sketched some general *concept*, in this section we describe a sample wavefront path tracer that is built on this idea. For the sake of reproducibility we sketch all the different stages of this path tracer; however, we observe that with very few exceptions—that we will explicitly point out—this path tracer looks like any other GPU wave-front path tracer. In fact, the entire renderer we are about to sketch was largely based on pieces that existed before this core idea was even conceived. This is good, as we believe this to be a strong indicator that these same ideas should be easy to integrate into any other wave-front renderer.

4.1. Input Structure and Partitioning

Our method is not limited to a specific type of scene content; but for this paper we focus on production style content that is organized similar to PBRT's [PJH16]: A scene consists of one or more instances of logical *objects*; each object consists of one or more triangle meshes; and each triangle mesh has one Disney "Principled BRDF"-style material [Bur12], and possibly color and alpha textures. As we trace each ray on each GPU, *how* we partition that scene will not matter much. Consequently, our first implementation simply assigned objects to GPUs in a round-robin manner.

Geometry Weight Estimate. The main issue we encountered with this totally random assignment is that different GPUs can end up with vastly different memory load—wasting resources. To address this we implemented what we call a content *weight estimate*, which can predict how much GPU memory a given object would require. This is the sum of all vertices, indices, texture memory, etc., but also needs to account for how much memory OptiX will spend on acceleration structures. The former we can compute exactly; the latter we can only estimate. We measured OptiX' BVH sizes for different triangle and instance counts, then hand-fitted two constants (for triangles and instances, respectively) to approximate the observed sizes. These constants depend on what type of GPU OptiX is running on: for GPUs with hardware ray tracing cores, we use 50 and 400 bytes per triangle and instance, respectively; for those without we use 100 and 400 bytes, respectively. Other geometry types such as hair, fur, etc., would require some equivalent weight estimates, but should then work in exactly the same way.

Weight-based Partitioning. Using this weight estimate we can easily achieve a more equal memory load across our GPUs (or, conversely, use less GPUs per model): during loading we make a list of all objects in the scene, and compute the weight of each such object (including all the instances pointing to it). We then sort these objects by their size, and, starting with the heaviest object, greedily assign each to the then respectively least loaded GPU. This is trivially cheap, and can be done on-the-fly.

Big-Object Splitting. Another problem we observed is that production scenes often have one "ground object" that is much larger than all the others (also see, e.g., [ZMWP20]). If we always assign entire objects then the biggest such object would still dictate what kind of models we could or could not scale to. To address this we take objects that have more than a certain number of triangles, and split those into multiple objects with fewer meshes. Since we only split *objects*—not *meshes*—this can be done by simple operations on the scene graph, at near zero cost.

4.2. Per-GPU Acceleration Structures

Once the scene has been partitioned into the desired number of parts, each GPU picks one part, uploads its data, and builds its acceleration structures. For ray tracing we build on OptiX [PBD*10]: OptiX can make use of ray tracing cores where available, but also has a fast software fallback for GPUs that do not. We use OptiX through the OWL library [WMH20]; OWL allows for low-level access to performance-relevant features such as CUDA inter-op or asynchronous launches, but significantly reduces the effort required to build OptiX acceleration structures, shader binding tables, etc.

Building the OptiX acceleration structures is not conceptually different from any other OptiX/OWL based path tracer, except that each GPU will have different meshes and instances, and thus needs its own acceleration structures and shader binding table. We create N different OWL contexts—one per GPU—and treat these as independent. Each context has its own CUDA stream for asynchronous launches, which we later use to run these GPUs in parallel.

4.3. Rays and Ray Queues

Like any other wave-front path tracer we maintain queues of rays, where each ray is not just a geometric ray, but actually describes the end-point of a *path* that is being traced: Each ray stores information about the pixel it belongs to, the ray type (shadow or not), the current path length, state, and throughput, as well as information about that ray's intersection with the scene, etc. As in any other wavefront path tracer we eventually have three kernels that operate on these queues: a *wave generation* kernel that generates a new wavefront of rays; a *trace* kernel that computes each ray's intersection with the scenes; and a *shade/bounce* kernel that processes a ray's found intersection, possibly generating one or more new outgoing rays.

Ray Queue Management. To avoid dynamic memory allocations during rendering we allow the path tracer to only have at most two rays active per pixel at any time (see Section 4.7 below), then preallocate ray queues with twice as many slots as there are pixels per GPU. We also use two such queues per GPU, to allow kernels to have different input and output queues, where required. Both of these measures are common practice for wave-front path tracers.

Embedded Full Hit Information. Our rays and ray queues as described so far are not significantly different from common practice. One important difference, however, is in how a ray stores its

intersection with the scene: Usually, each ray would store only high-level information such as primitive, mesh, and instance IDs, barycentrics, etc., and leaves the computation of the shading point, normals, material data, etc., to the shade kernel.

In our case, however, rays may find an intersection on a GPU other than the one that will eventually shade that ray, so any IDs would not be valid. We solve this by having the ray store the full geometry of the intersection (position, shading normal, and geometric normal) as well as the full set of coefficients for a *Disney Principled*-style BRDF [Bur12].

Storing all this in a ray sounds prohibitive, but it is not: partly this is because we no longer have to store the IDs and barycentrics; partly it is because for normals and BRDF coefficients we can use half precision without fear of artifacts (we could likely use even less). Intersection point and distance require full single-precision. Our ray struct currently uses 88 bytes per ray; this is about twice as much as what is used by *BriX*, but for the hardware we are targeting (with order 10s to several 100s of GB/s of available bandwidth per GPU) this is quite manageable—and a price worth paying for the ability to equally distribute all rays across all GPUs.

4.4. CUDA Wave-front Generation Kernel

Our wave-front generation kernel is realized in CUDA. Unlike, for example, the *BriX* system we chose to explicitly *not* try to generate rays only where some heuristic might indicate so, and instead aim for distributing the load of rays as equally as possible. We currently simply have each GPU trace every N'th row of pixels, meaning that all N GPUs together spawn exactly one path per each pixel, with each GPU having one N'th of those. The kernel generates a new path for each assigned pixel, and atomically appends this to the outgoing ray queue. All GPUs' generation kernels are launched in parallel, each into its own GPU's CUDA stream.

4.5. (Per-GPU) Trace Kernel

The tracing kernel is realized in OptiX, using a *ray-gen* (RG) program to launch the rays, and a combination of *any-hit* (AH) and *closest-hit* (CH) programs to process intersections.

Ray-Gen (RG) Program. Despite the name the RG program does not *generate* rays—it is merely OptiX' way of naming this program, and in our case is the program we use to trace a given ray queue's worth of rays into the OptiX acceleration structures. The RG program gets launched with one GPU thread per ray in the input ray queue; its launch parameters include a device pointer to the ray queue to be traced. Each thread first reads the corresponding ray from the ray queue, then checks if that is a shadow ray that was already terminated on a previous GPU. If so, the respective thread simply terminates; otherwise, it stores a pointer to that ray's data in the OptiX *per-ray-data* (PRD) and asks OptiX to trace that ray.

Any-Hit (AH) Program. The AH program gets called by OptiX on any potential intersection. It first looks up the current intersection's material, and checks if the hit is fully transparent. This includes both material information and, where required, alpha texturing. If this indicates a fully-transparent hit the intersection gets discarded; otherwise, the program checks if the ray is a fully occluded shadow ray, and if so, marks it as occluded and terminates traversal.

Closest-Hit (CH) Program. The CH program takes the closest found intersection—if it exists—and stores that in the ray (using the pointer from the PRD). It first computes shading and geometry normal, transforms those to world space, and stores that using half precision. It then computes the BRDF coefficients and stores these in half precision as well; any textures get evaluated and baked into the BRDF coefficients. For the 3D intersection coordinates we experimented with only storing the distance to the hit point, but for the kind of models we use this resulted in objectionable surface acne. Instead we now use Wächter's method [WB19] to compute a stable intersection position and store that in single precision.

4.6. Ray Queue Cycling

The core idea of our method is to not just trace each ray once, but to cycle each ray through every GPU, and trace it on each. We do this by simply calling the above trace kernel N times, and *cycle* all GPUs' ray queues once between each step (cf. Figure 2). In each such cycle, every GPU passes its current ray queue to its respective right neighbor, and receives its left neighbor's queue in turn.

One way of doing this—we call this an *implicit copy*—is for the RG program to have two ray queue pointers: one that points to its right neighbor's input queue, and one to its own output queue. The program then reads from one and writes the traced rays back to the other, which means that rays get copied *while* they are being traced. In what we call an *explicit copy* we instead have the trace kernel operate on only its own local input queue, and have it write the rays back to that queue where required. Following this we then launch, on each GPU, a dedicated `cudaMemcpyPeerAsync` that copies this queue to its neighbor GPU's output queue. These N copies can get launched into the same streams as the trace, in which case they will automatically wait for their respective trace to finish, yet still run in parallel to each other. We finally wait for those N parallel copies to finish, then simply swap each GPU's input and output queue pointers; then iterate. We currently first trace then copy; changing this order should not matter.

In theory, the implicit version should be faster, because copying and tracing are interleaved. We do see a slight benefit on NVLink; however, for PCIe-based hardware the explicit version was faster, likely due to PCI struggling with the less temporally ordered nature of the implicit variant. Since the communication is more bottlenecked on PCI we decided to make the latter the default.

We now take these two kernel—`trace` and `cycle`—and call them N times. At this time, each ray is back on the GPU that originally spawned it, having been traced on each GPU (Figure 2).

4.7. CUDA Shade/Bounce kernel

The *bounce* kernel reads rays from the input queue, shades these rays, and appends any generated shadow or secondary rays to the output queue. This kernel first checks if the ray to be shaded is a shadow ray, and if so, whether it had any intersection. If so it gets discarded; otherwise, we take its throughput value (which states how much light it carries), and add this to the pixel it belongs to.

	num GPUs	GPU memory	fabric	bidir P2P BW†	#CUDA cores	Arch	RT Cores	location
DGX-2	16	16×32 GB=512 GB	2nd-gen NVLink+NVSwitch	270 GB/s	16×5,120	Volta	no	IT4Innovations
HGX/A100	8	8×40 GB=320 GB	3rd-gen NVLink+NVSwitch	475 GB/s	8×6,912	Ampere	no	IT4Innovations
RTX-server	8×A40	8×48=384 GB	PCIe 4.0, fully switched	36.7–51.9 GB/s‡	8×10,752	Ampere	yes	NVIDIA GPU Cloud

Table 1: *Hardware specifications for the machines used for our performance evaluations. †measured using the CUDA Toolkit's* `p2pBandwidth` *tool. ‡Bandwidth on this platform varies across different GPU pairs, likely due to varying number of available PCIe lanes.*

For non-shadow rays we first check if the ray did hit any geometry; if not, it gets shaded with environment illumination, and the result gets atomically added to the pixel that this ray belongs to. Otherwise, we first handle any potential emission of the hit surface, then compute a new outgoing ray based on BRDF, hit point, and current state of the ray. We first check if that ray exceeds a predetermined maximum number of specular or diffuse bounces, respectively, and if so, terminate that path. Otherwise, we update the ray's throughput value, origin, direction, etc. To avoid tracing lots of low-throughput rays we perform Russian-Roulette termination based on the throughput value. If the ray was not discarded we atomically append it to that GPU's output queue. The same atomic counter used to specify the next free output queue position can later also be used to determine how many rays are in each queue.

If the ray undergoes a diffuse bounce we also generate a single shadow ray. We do this by performing repeated reservoir sampling, first choosing one from possibly multiple different area light samples, then one from possibly multiple point lights, etc., then choosing one of those to trace a shadow ray to. Each of these different sampling stages use importance sampling based on how much light the shadow ray would ultimately carries. Once a shadow ray has been generated, we store this carried light (divided by the light sample's PDF) in that ray's throughput field, mark it as a shadow ray, and append it to the queue. Once all GPUs have finished shading we are back to having an input queue of rays to be traced on each GPU, and simply iterate back to the next set of T&C cycles.

4.8. Image Contributions and Local Frame Buffer Merging

The *bounce* kernel can generate image contributions that need to get added to the frame buffer. In a single GPU this would be done using a single device buffer, but in our method this can happen on N different GPUs in parallel. Unlike in *BriX* we do not have to deal with adding different GPUs' contributions to the same pixels, because image contributions happen only during shading, on the GPU that actually spawned the original path. However, we still have pixels interleaved between different GPUs, and these need to be merged into a single contiguous buffer.

One way of doing this is to have a single accumulation buffer on GPU 0, then all GPUs atomically add into this using peer access. On NVLink this actually works quite well; however, on PCIe the resulting atomics are really slow. Instead, we have each GPU maintain its own local accumulation buffer for only its one-N'th of the pixels. After all paths have been traced each GPU first performs simple Firefly-clamping, Gamma-correction, and `RGBA8` conversion on its own pixels (in parallel), then the resulting N small frame buffers are copied to GPU 0, which then re-arranges those into the proper order in a CUDA kernel. More complex tone mapping or denoising would require to merge pixels in float precision; but this is orthogonal to our method.

4.9. Taking it all Together

Using these four kernels the main components of our method are now complete, and rendering an entire frame is simply a matter of launching these kernels in the right order.

Wave Generation. First each GPU launches its generation kernel, in parallel. At that time, each GPU has one-N'th of all rays.

Distributed Trace. We then trace all rays through all GPUs by doing N trace-and-cycle iterations. In each iteration we first launch one *trace* kernel on each GPU (using OptiX' asynchronous launches to make those run in parallel), then launch the corresponding copy into that same stream. We then wait for these traces and copies to finish by synchronizing those per-GPU streams, then simply swap all GPUs' input and output queue pointers, and set each GPU's ray count to that of its right neighbor. This marks the end of the first trace-and-cycle iteration, and each GPU now has one N'th of the rays, each of which has been traced on one N'th of all GPUs, against its respective GPU's part of the data. We repeat this N times, at which point each ray is back to the GPU that spawned it, having been traced on every GPU, against all data.

Wave-front Path Tracing. After the distributed trace is complete we launch each GPU's *bounce* kernel, again in parallel using our N streams. We then wait for these to complete, and read each GPU's atomic ray queue size counter. If this is zero for every GPU we are done; otherwise we have another generation of rays to trace, and go back to the distributed trace as described above. Once a wave is traced through all its generations we can either stop tracing and proceed to merging the local frame buffers, or can call another wave generation kernel with an additional path per pixel, if so desired.

Final Frame Buffer Merge. After all paths of all waves have been traced we execute the final accumulation buffer to `RGBA8` conversion, and merge the resulting pixels on GPU 0 for saving or display.

4.10. Wave-front Merging

One issue we observed is that very often some pixels require many more bounces than the average pixel. This "long tail" problem is a known issue for any path tracer, in particular for GPUs: lots of small waves mean a lot of call overhead, and low GPU utilization.

To address this we added what we call *wave front merging*: Instead of sizing all ray queues to have only two ray slots per pixel we instead make these ray queues somewhat larger, for example, with 3 instead of 2 ray slots per pixel on that GPU. After each shade step we look at how many rays are still in the ray queue at this time, and check if there are now enough free ray slots to hold both these existing rays *and* a new primary wave-front. If not, we simply proceed as before; otherwise we call the wave generation kernel and have it put the new rays right behind the still active rays—and then proceed with tracing. This effectively merges all but the last wave's short queues into larger ones, leading to speedups of order 10%.

landscape (1.6 GB on disk)
30.0K insts, 370 meshes, 27.8M tris
1.4 GB textures

island (18.8 GB on disk)
39.3M insts, 7.1M meshes, 285.5M tris
4.1 GB textures

island-XL (84.1 GB on disk)
31.2M insts, 18.9K meshes, 592.4M tris
33.2 GB textures

museum (113.7 GB on disk)
3.1K insts, 397 meshes, 647.7M tris
44.9 GB textures

Figure 3: *The models we use for our evaluation. For the island model we include both the triangles-only PBRT version used in [WP22], as well as a much larger one created by having Blender perform subdivision surface tesselation.*

4.11. Combination with Island Parallelism

Another issue we encountered is that for any data-parallel ray tracer there is a quickly diminishing return for adding more GPUs than are strictly required for a model: adding more GPUs reduces how much geometry each GPU will end up having—but ray tracing is logarithmic in geometric complexity, so reducing the geometry per GPU will not result in much speedup. This is still OK from the perspective that without data-parallelism the model would not have rendered at all; however, it is still a waste of potential if the underlying machine has significantly more GPUs than required.

To recoup this potential we integrated *island parallelism* as described by Zellmann et al. [ZWB*22]. The idea of island parallelism is to take a configuration of N GPUs, and split these into M so-called *islands* of K GPUs each—with data-parallel rendering performed *within* each K-sized island, and data-*replicated* rendering performed *across* the M different islands. For our method, applying this is trivially simple (see Figure 4): During startup we partition the scene into K (not N) parts, and have each GPU $i < N$ pick part $i \pmod K$. For each GPU i we then set its right neighbor as GPU $(i+1) \pmod K$, at which point our N GPUs form M cycles of K GPUs each. Once this is done all we need to do is reduce the number of trace-and-cycle iterations from N to K, which is what actually gives us the desired speedup. Nothing else needs to be changed at all: Each GPU still spawns the same one-N'th of all pixels' rays; and the *trace*, *bounce*, *cycle*, or frame buffer merge kernels do not even have to know that this technique is being used.

Island parallelism is not only trivial to add to our method, we can even make it work fully automatically: Using the weight estimate from Section 4.1 we simply compute how many GPUs are needed to hold the model once. Then any multiple of this minimum number of GPUs is used to create more islands.

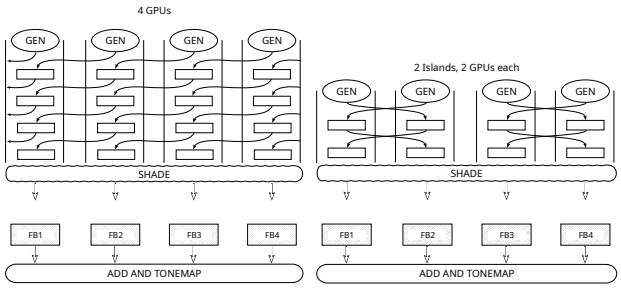

Figure 4: *Illustration of* island parallelism *[ZWB*22] within our method, here for 4 GPUs. Left, without islands each GPU holds one fourth of the model, but also requires four trace-and-cycle stages. Right: the same four GPUs, using a configuration of two islands of two GPUs each—requiring only two cycle stages.*

5. Results and Evaluation

We evaluate our framework on different scenes and hardware platforms. In practice we also use our method on consumer hardware with only one or two GPUs (when run on a single GPU, our method automatically behaves just like any other path tracer), and with or without NVLink bridges—but for this evaluation we focus on professional multi-GPU hardware. For the type of hardware most likely to be found in a cloud or data center we include both DGX-2's and HGX/A100 machines; these machines have fully switched NVLink interconnect, but no hardware ray tracing cores. For more rendering oriented hardware we also include an 8-GPU RTX Server, which has hardware ray tracing cores, but no NVLink. This machine has fully switched PCIe with still 30 to 50 GB/s of bidirectional bandwidth per GPU, but this is about an order of magnitude less than NVLink, while simultaneously having about an order of magnitude higher trace potential due to its hardware cores. Detailed specifications for these three machine types are given in Table 1. We observe that these machines (intentionally) cover opposite ends of the compute-to-bandwidth spectrum.

The models we use for evaluating are depicted in Figure 3. To enable a close comparison to *BriX* we include the same *landscape* and *island* models used in that paper. We also include two much larger models—*island-XL* and *museum*—as well as some up-sampled versions of *landscape* for reference. We observe that *island* and *island-XL* are conceptually the same model, but at very different scales: the former is from an export to PBRT (without subdivision surfaces), the latter is exported from Blender [Ble], with Blender doing subdivision surface tessellation. Both *museum* and *island-XL* also use significantly higher-resolution textures.

5.1. Fixed Model Size, Scaling Number of GPUs

The most obvious question for our method is how, for a given model, it will perform for different numbers of GPUs. We ran our framework on our different hardware platforms, and made it use only a user-specified number of GPUs. We also ran this experiment once with and once without islands; if islands are enabled the island is computed automatically as described in Section 4.1.

The result of these experiments is shown in Figure 5: Without island parallelism (upper half) we see a mostly binary outcome: until we get enough GPUs the model will not render at all; once we do get enough it will (which is the point), but adding more will not help much. Once we enable island parallelism (lower half), adding more GPUs also translates to higher performance. This can still result in some unused GPUs because we can only add GPUs in multiples of island size, but generally speaking leads to useful gains.

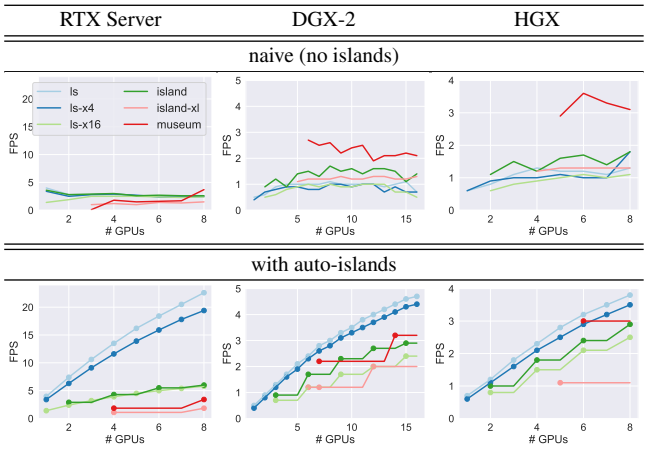

Figure 5: *Performance of our method over different hardware and models, using different numbers of GPUs; once without (top) and one with (bottom) island parallelism. Solid circles indicate that the method switched to a different island configuration.*

5.2. Scalability in Model Size

We also implemented a method that lets us artificially grow a given scene by any user-defined factor: E.g., for a factor of 12 we would get a model that has an expected $\sim 12\times$ the number of triangles in each mesh, plus 12 copies of all the model's instances on a 4×3 grid; for a total of $144\times$ the number of instantiated triangles.

Figure 6 shows the render time for different growth scales of the *landscape* model—starting at $1\times$ (the original model) up to $120\times$ the number of triangles and instances (i.e., $14,400\times$ the number of instanced triangles). Automatic island parallelism was enabled; the ticks on the x axis also show when our method switched to different island sizes. Actual data parallel rendering does not even kick in until $16\times$, where our method first uses more than one GPU per island. Overall, we see render time increase from 44 ms/frame to 1 s/frame (i.e., by $25\times$), for a range of $120\times$ in model size and $14,400\times$ in instanced model size, and from data-parallelism going from fully data replicated (IS=1) to all data parallel (IS=8).

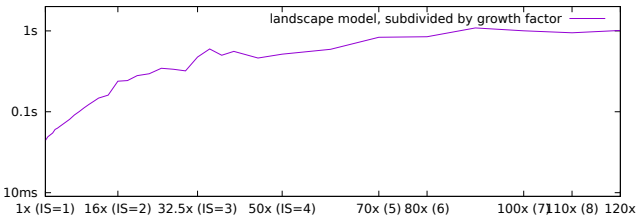

Figure 6: *Time per frame (using an RTX Server) over a scale of artificially up-sampled variants of the* landscape *model; from* $1\times$ *(the original model) up to one that is roughly* $120\times$ *as big in number of triangles, instances, and overall memory consumption.*

5.3. Timing Profile

To illustrate how our framework behaves over time we also acquired a timing profile for one frame of the *museum* model (see Figure 7). To reduce measurement noise and improve readability we used a high resolution of $8k\times8k$ pixels and 8 paths/pixel (smaller resolutions would affect all kernels equally, and show overall similar behavior). In this graph, light green boxes include trace and transfer time, dark green are shading, and background means idle.

The most important observation to make in this graph is that generally speaking, all GPUs are busy almost all the time: GPUs 1, 3, and 4 have less work than the others, likely because they received geometry that is mostly off-screen and/or easier to trace against—but overall there is no single hot-spot where everything bottlenecks.

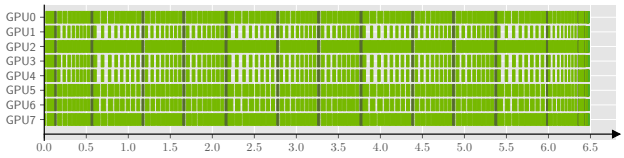

Figure 7: *Timing profile for one frame of the* museum *model, using* $8k\times8k$ *pixels and 8 paths/pixel, wave-front merging is enabled. Dark green indicates shading, light green indicates trace and cycle.*

5.4. Qualitative Factors: Generality and Ease of Use

So far, our evaluation has concentrated on the obvious "hard" criteria for evaluating a parallel system—like performance and scalability. However, the main goal of our method was not to always be *faster* than any competing technique, but instead to be, above all else, *practical* in the sense that it has fewer restrictions—and is as easy to use, employ, and extend as any other *non*-data parallel technique. This is not something that can easily be proven, and where every case in point will necessarily be subjective. However, we want to briefly sketch two such case-in-point examples.

Experimental Blender Integration. We prototypically integrated our sample implementation into Blender (see Figure 8). We do this using a Blender plugin that intercepts the data Blender would usually have given to Blender's own *Cycles* renderer, and writes that in our renderer's scene format. This plugin then remotely starts our renderer on a different node in the data center, and waits for this to connect back using a TCP socket—at which point the plugin can interact with our renderer, change render settings, display the resulting pictures in the Blender GUI, etc. Though crude, this integration is not unlike *Cycles* itself; in particular, that plugin does not even have to be aware that our renderer is data-parallel: model partitioning happens on the fly during loading, everything is fully automatic, and the data-parallel rendering across multiple GPUs just so happens without Blender even being aware of it.

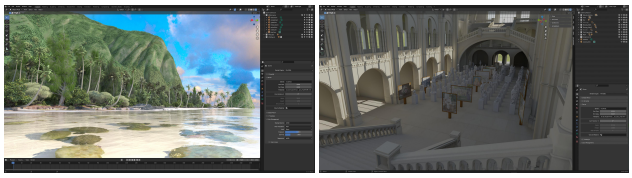

Figure 8: *The* island-XL *and* museum *scenes in our experimental Blender integration, interactively rendered with our method on a DGX-2, while controlled from, and displayed in, Blender.*

Extension to Other Data Types. To evaluate how *general* our method is we also experimentally extended it to a totally different type of data: volume data. This required lots of different changes in terms of data loaders, user interface, applying transfer functions, etc., but conceptually was trivially easy: We load different parts of the volume to different GPUs, then use Woodcock tracking [WMHL65] to compute intersections between rays and volume, and a 3D-DDA for traversing the volume. All these components are completely compatible with the rest of the system, meaning we can also mix surface and volume data on the same GPUs, or load some GPUs with volume data while others use surface data; we can also run that inside our Blender plugin, etc. Two examples of this are shown in Figure 9; extensions to other types of data—both volumetric and/or surface based—should be similarly easy. We also observe that these case-in-point examples, too, are fairly large models of hundreds of Gigabytes.

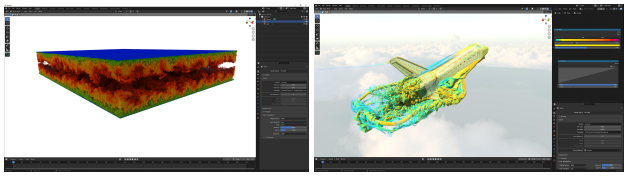

Figure 9: *Proof-of-concept application of our method to volume data (both inside our Blender plugin). Left: the* $10,240 \times 7,680 \times 1,536$ DNS *data set (483 GB). On a DGX-2 with 16 GPUs of 32 GB each our method renders this at 9 frames per second (*1625×930 *resolution, 16 paths per pixel). Right: The* Space Shuttle *data set featuring both surface data and a 218 GB volume data set.*

6. Summary and Discussion

In this paper, we have presented a novel approach to single-node/multi-GPU data-parallel path tracing that we termed *ray queue cycling*. Our method employs ray-forwarding, and primarily aims for practicability and generality. Each ray is sent to—and traced on—each GPU, requiring the presence of interconnect such as fully switched PCIe, or preferably NVLink; however, on typical cloud or data center hardware where such is available our method is significantly simpler to employ than other recently proposed methods. Our method does not require a user to be familiar with MPI or clusters, nor does it require any sophisticated (and possibly precarious) data-partitioning to work well: instead, data can be assigned to different GPUs on the fly during the load time, with which GPU gets what part of the data is largely irrelevant.

We have shown a sample implementation of our method to be able to interactively render even massively complex models with hundreds of Gigabytes of data, including both surface- and volume data, including non-trivial path traced shading, and including a prototypical integration into Blender.

While our implementation makes heavy use of NVIDIA-specific technologies (and will benefit from both ray tracing hardware and NVLink where available) we have shown it also works without ray tracing cores, and on vendor-agnostic PCIe. Similarly, we make heavy use of NVIDIA APIs such as OptiX and CUDA, but the core ideas are not specific to those APIs.

6.1. Comparison to Alternative Approaches

One obvious question is how our framework refers to other methods, in particular those by Wald and Parker [WP22], and Jaros et al. [JRSS21]. For the former, the authors report 7.9 FPS for *island*, using four nodes of two RTX 8000 cards each. For the same model and camera position, in our best configuration we achieve 2.9 FPS on a DGX-2 and an HGX, respectively, and 6 FPS on an RTX Server—but each using 8 times as many paths per pixel. To get an even closer comparison we also used a desktop PC with two RTX 8000 cards, which is virtually identical to the render nodes used in *BriX*. On this machine, our framework renders the island model at 12.3 FPS. This is faster than *BriX*—at $4\times$ less nodes.

Comparing to Jaros et al. is harder because the underlying path tracers are different. With this in mind, Jaros et al. report 181 seconds for *island-XL*, and 126 seconds for *museum*, each using 1000 paths per pixel. On identical hardware and render settings our renderer takes 172 seconds and 98 seconds, respectively. Unlike Jaros', our method can make full use of vendor-supplied accelerated ray tracing frameworks, and is less dependent on NVLink.

Though such performance comparisons have to be taken with a grain of salt we believe this proves our method to be at least competitive with regards to performance (and often faster). This is important because the *real* advantage of our method was never supposed to be performance, but ease of use and generality: *BriX*, for example, relies on a costly offline preprocessing step to render these models at all, plus some amount of expert knowledge in selecting the right parameters, a much higher hardware hurdle, etc.—whereas our method can load and render these models on the fly, in a fully automatic way, and without needing any knowledge of how to employ clusters via MPI. Jaros' system is easier to use than *BriX*, but also requires some offline preprocessing. More importantly that method depends on knowing a priori which view the user will want to render (ours has no such constraints), cannot make use of ray tracing cores where available, and requires the user write and maintain his own ray tracing back-end.

6.2. Remaining Issues and Limitations

Arguably the biggest limitation of our method is that it assumes the renderer to be using a Disney Principled BRDF (or similar) that can be embedded in the ray—this may not work for every renderer. For renderers that use *shader networks* it may still be possible to compile a given ray-scene intersection's instantiation of that shader network down to a similar set of BRDF parameters (in which the same approach would then again work by each GPU doing that on its local closest hit); but how exactly this may or may not work would depend on the specific renderer.

On the performance side, the most obvious issue with our current implementation is that it is necessarily limited to how many GPUs can be found in a single machine. In theory, our argument about more GPUs also providing more bandwidth and compute applies to multi nodes as well, and with some network technologies now also reaching order 50 GB/s our technique might eventually also work for some small networked setups as well—but this requires more investigation.

Another issue is that our method relies heavily on island parallelism for performance scalability, but it is not clear that this might always be available. For example, if geometry gets created on the fly it might not be possible to compute an island size up front. In that case, our method would still work, but not scale beyond the performance of a single GPU.

Yet another limitation is that though our method can make use of ray tracing cores, most data center hardware today does not have those. Our method is still competitive even without those (and still saves the user from having to write his own ray tracing back-end), but it still means that our method cannot use its full potential on current hardware. Also, even with our method the best way of dealing with data larger than GPU memory is to avoid this situation in the first place: if techniques like streaming, LOD, etc. can be used to not go over the GPU memory limit in the first place, then it will always be more (cost-)efficient to use that.

Finally, our renderer is still but a proof of concept prototype, and integration into a real production renderer may well raise some issues we have not yet encountered.

7. Conclusion

In this paper, we have argued for a new approach to data-parallel rendering on single-node, multi-GPU hardware. The core idea of our method is to *not* try and minimize which rays get sent or traced where, and instead, to make full use of all GPUs' aggregate compute and bandwidth to simply trace each ray on every GPU. We have shown that against intuition this approach is both feasible and desirable: It is feasible, because adding more GPUs will increase the total aggregate available bandwidth by exactly the same factor as tracing on all GPUs requires. This allows for model size-scaling to however many GPUs as are required to hold the model; and when combined with island parallelism, any additional GPUs can still be used for performance scaling as well. We have shown our method to not only be generally feasible, but even to be highly competitive with the best known existing techniques.

In addition to being feasible, our method is also desirable, because it comes with significantly fewer constraints than competing techniques. It can live with essentially arbitrary assignments of geometry to GPUs, without preprocessing or a priori knowledge, etc., in a fully automatic way. This we believe finally brings data-parallel rendering to where it is *practical* for more mainstream GPU renderers. Though our method is only intended for single-node setups with a necessarily limited number of GPUs, this still allows a renderer to scale supported model size by at least one order of magnitude, with virtually no restrictions as to what kind of content or exact path tracer is being used.

Acknowledgments

This work was supported by the Ministry of Education, Youth and Sports of the Czech Republic through the e-INFRA CZ (ID:90254), and by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) through grant no. 456842964.

The authors thank Alvaro Luna Bautista and Joel Andersdon for the model of Natural History, Three D Scans project for models of statues, and Art Institute of Chicago for free images (CC0) that have been used in the Museum scene. We would also like to thank Disney for making the Moana Island Scene publicly available.

We gratefully acknowledge the use of the computational facilities of the Center for Data and Simulation Science (CDS) at the University of Cologne; at the Center for High Performance Computing (CHPC) at the University of Utah; at the IT4Innovations National Supercomputing Center at the University of Ostrava; and at NVIDIA. Amir Mohammad Tavakkoli and Daniel Rushton have helped out with various measurements.

References

[Ala20] ALARCON N.: Ray Tracing Essentials Part 4: The Ray Tracing Pipeline. NVidia Technical Blog, 2020. 2

[BAC*18] BURLEY B., ADLER D., CHIANG M. J.-Y., DRISKILL H., HABEL R., KELLY P., KUTZ P., LI Y. K., TEECE D.: The design and evolution of disney's hyperion renderer. *ACM Trans. Graph. 37*, 3 (2018). 2

[Ble] Blender – The Freedom to Create. 7

[BM21] BOKSANSKY J., MARRS A.: The Reference Path Tracer. In *Ray Tracing Gems II - Next-Generation Real-Time Rendering with DXR, Vulkan, and OptiX*, Marrs A., Shirley P., Wald I., (Eds.). 2021. 2

[Bur12] BURLEY B.: Physically-Based Shading at Disney. Siggraph 2012 Course Notes., 2012. 4, 5

[Cor20] CORP. N.: NVIDIA H100 Tensor Core GPU Architecture, 2020. Available at https://resources.nvidia.com/en-us-tensor-core, Accessed: 20 March 2023. 2

[DGP04] DEMARLE D. E., GRIBBLE C., PARKER S. G.: Memory-Savvy Distributed Interactive Ray Tracing. In *5th Eurographics / ACM SIGGRAPH Symposium on Parallel Graphics and Visualization (EGPGV 2004)* (2004). 2

[Eil19] EILEMANN S.: *Parallel Rendering and Large Data Visualization*. PhD thesis, University of Zurich, 2019. arXiv:1902.08755v1. 2

[FSP*22] FOULADI S., SHAKLETT B., POMS F., ARORA A., OZDEMIR A., RAGHAVAN D., HANRAHAN P., FATAHALIAN K., WINSTEIN K.: R2E2: Low-Latency Path Tracing of Terabyte-Scale Scenes using Thousands of Cloud CPUs. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH)* (2022). (to appear). 2

[IBH11] IZE T., BROWNLE C., HANSEN C. D.: Real-Time Ray Tracer for Visualizing Massive Models on a Cluster. In *Eurographics Symposium on Parallel Graphics and Visualization* (2011). 2

[JRSS21] JAROS M., RIHA L., STRAKOS P., SPETKO M.: GPU Accelerated Path Tracing of Massive Scenes. *ACM Transaction on Graphics 40*, 2 (2021). 2, 9

[KS02] KATO T., SAITO J.: "Kilauea" – Parallel Global Illumination Renderer. In *Fourth Eurographics Workshop on Parallel Graphics and Visualization* (2002). 2

[LKA13] LAINE S., KARRAS T., AILA T.: Megakernels Considered Harmful: Wavefront Path Tracing on GPUs. In *Eurographics/ACM SIGGRAPH Symposium on High Performance Graphics* (2013). 2

[Mor11] MORELAND K.: *IceT users' guide and reference*. Tech. Rep. SAND2010-7451, US Department of Energy Office of Scientific and Technical Information, Sandia National Labs, 2011. 2

[Nav10] NAVRATIL P. A.: *Memory-Efficient, Scalable Ray Tracing*. PhD thesis, University of Texas, Austin, 2010. 2

[NCFL14] NAVRÁTIL P. A., CHILDS H., FUSSELL D. S., LIN C.: Exploring the Spectrum of Dynamic Scheduling Algorithms for Scalable Distributed-Memory Ray Tracing. *IEEE Transactions on Visualization and Computer Graphics 20*, 6 (2014). 2

[PBD*10] PARKER S. G., BIGLER J., DIETRICH A., FRIEDRICH H., HOBEROCK J., LUEBKE D., MCALLISTER D., MCGUIRE M., MORLEY K., ROBISON A., ET AL.: OptiX: A General Purpose Ray Tracing Engine. *ACM Transactions on Graphics 29*, 4 (2010). 2, 4

[PJH16] PHARR M., JAKOB W., HUMPHREYS G.: Physically Based Rendering: From Theory to Implementation (3rd ed.). 1200. 4

[PKGH97] PHARR M., KOLB C., GERSHBEIN R., HANRAHAN P.: Rendering Complex Scenes with Memory-Coherent Ray Tracing. In *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '97)* (1997). 2

[Rei95] REINHARD E.: *Scheduling and Data Management for Parallel Ray Tracing*. PhD thesis, University of East Anglia, 1995. 2

[SG89] SALMON J., GOLDSMITH J.: A Hypercube Ray-Tracer. In *C3P: Proceedings of the third conference on Hypercube concurrent computers and applications - Volume 2* (1989). 2

[Sol14] SOLOMON R.: PCI Express Basics & Background, 2014. 2

[WB19] WÄCHTER C., BINDER N.: A Fast and Robust Method for Avoiding Self-Intersection. In *Ray Tracing Gems - High-Quality and Real-time Rendering with DXR and other APIs*. 2019. 5

[WMH20] WALD I., MORRICAL N., HAINES E.: OWL – The OptiX 7 Wrapper Library, 2020. Available at https://github.com/owl-project/owl, Accessed: 27 March 2022. 2, 4

[WMHL65] WOODCOCK E. R., MURPHY T., HEMMINGS P. J., LONGWORTH T. C.: Techniques used in the GEM code for Monte Carlo neutronics calculations in reactors and other systems of complex geometry. In *Proceedings of the Conference on Applications of Computing Methods to Reactor Problems* (1965), Argonne National Laboratory. 9

[WP22] WALD I., PARKER S. G.: Data Parallel Path Tracing with Object Hierarchies. *Proceedings of the ACM on Computer Graphics and Interactive Techniques 5*, 3 (2022). 2, 7, 9

[WSB01] WALD I., SLUSALLEK P., BENTHIN C.: Interactive Distributed Ray Tracing of Highly Complex Models. In *Eurographics Workshop on Rendering Techniques* (2001). 2

[ZMWP20] ZELLMANN S., MORRICAL N., WALD I., PASCUCCI V.: Finding Efficient Spatial Distributions for Massively Instanced 3-d Models. In *Eurographics Symposium on Parallel Graphics and Visualization* (2020), Frey S., Huang J., Sadlo F., (Eds.), The Eurographics Association. 2, 4

[ZWB*22] ZELLMANN S., WALD I., BARBOSA J., DERMIC S., SAHISTAN A., GUDUKBAY U.: Hybrid Image-/Data-Parallel Rendering Using Island Parallelism. In *Proceedings of the 2022 IEEE 12th Symposium on Large Data Analysis and Visualization (LDAV)* (2022). 7