Received: 24 August 2020 | Revised: 12 August 2021 | Accepted: 14 September 2021

DOI: 10.1002/cpe.6663

RESEARCH ARTICLE

WILEY

Multi-GPU room response simulation with hardware raytracing

Peter Thoman[1] | **Markus Wippler**[1] | **Robert Hranitzky**[2] | **Philipp Gschwandtner**[3] | **Thomas Fahringer**[1]

[1]Distributed and Parallel Systems, Department of Computer Science, University of Innsbruck, Innsbruck, Tyrol, Austria

[2]tofmotion GmbH, Vienna, Austria

[3]Research Center HPC, Department of Computer Science, University of Innsbruck, Innsbruck, Tyrol, Austria

Correspondence
Peter Thoman, Distributed and Parallel Systems, Department of Computer Science, University of Innsbruck, Innsbruck, Tyrol, Austria.
Email: petert@dps.uibk.ac.at

Summary

Time-of-flight camera systems are an essential component in 3D scene analysis and reconstruction for many modern computer vision applications. The development and validation of such systems require testing in a large variety of scenes and situations. Accurate room impulse response simulation greatly speeds up development and validation, as well as reducing its cost, but large computational overhead has so far limited its applicability. While the overall algorithmic requirements of this simulation differ significantly from 3D rendering, the recently introduced hardware raytracing support in GPUs nonetheless provides an interesting new implementation option. In this article, we present a new room response simulation method, implemented in a vendor-independent fashion with Vulkan compute shaders and leveraging NVIDIA VKRay hardware raytracing. We also extend this method to multi-GPU computation with asynchronous streaming and introduce a domain-specific high-performance compression scheme in order to overcome the limitations of on-board GPU memory and PCIe bandwidth when simulating very large scenes. Our implementation is, to the best of our knowledge, the first ever combined application of Vulkan hardware raytracing and multi-GPU compute in a non-rendering simulation setting. Compared to a state-of-the-art multicore CPU implementation running on 12 CPU cores, we achieve an overall speedup factor of up to 20 on a single consumer GPU, and 71 on four GPUs.

KEYWORDS

asynchronous streaming, floating point compression, GPU computing, hardware raytracing, multi-GPU, simulation, time of flight, Vulkan

1 | INTRODUCTION

Many modern computer vision applications depend on fast and accurate 3D scene analysis. One promising hardware option to provide these data is time-of-flight (ToF) camera systems.[1] These systems generate pulses of modulated light of a specific wavelength—generally outside of the human visible spectrum—and monitor the response on an image sensor. The 3D environment is reconstructed based on the timing of reflected impulses as they arrive on the image sensor, with the result of that process being illustrated in Figure 1.

In order to evaluate the variety of design trade-offs in the creation of ToF hardware and software systems, and to validate their safety, for example, for industrial applications, testing in a large set of distinct scenes is required. These scenes vary in geometric complexity and surface

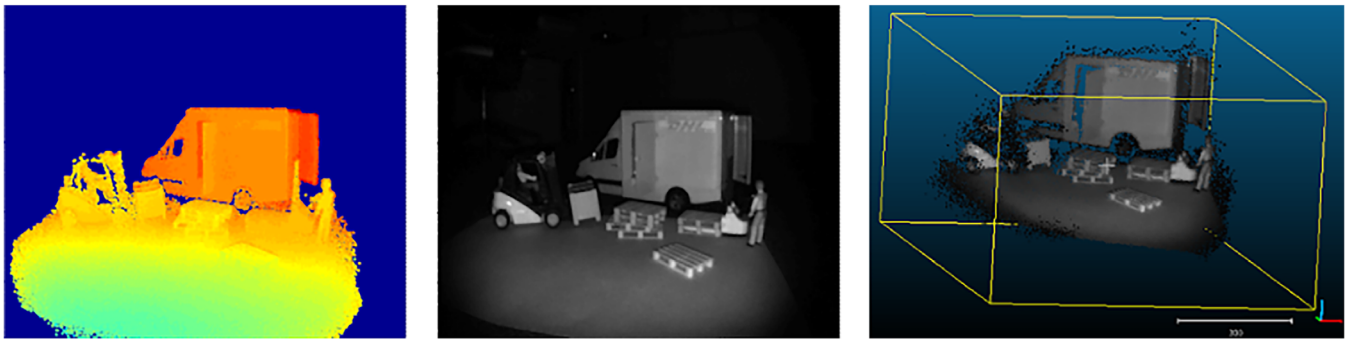


FIGURE 1 Time-of-flight sample scene. Left: Reconstructed depth; center: reflectivity; right: point cloud

properties. In order to speed up development and reduce costs, *room impulse response simulation*[2] is a promising alternative and precursor to real-world testing.

While the overall computational process and requirements of this simulation differ quite significantly from raytracing (RT) as it is traditionally performed for computer graphics,[3] and even its GPU-accelerated variants,[4] there are nonetheless several important steps in the simulation which benefit from fast ray/scene queries. NVIDIA recently introduced the Turing family of GPUs, which feature hardware acceleration for bounding volume traversal and ray/triangle intersection.[5] In this work, we have investigated the potential of this architecture—and future hardware raytracing architectures with similar capabilities—for room response simulation in the context of ToF cameras. We have also investigated various methods of GPU-friendly application-specific data compression for this workload, as we have demonstrated that the size of the mutual visibility matrix is the most significant limiting factor for large-scale simulations. Our concrete contributions include:

- A novel hardware-accelerated multi-GPU room response simulator—*M-RTX-RSim*—based on Vulkan compute[6] with SPIR-V[7] shaders generated from GLSL, and the NVIDIA raytracing extension.[8]
- An asynchronous multi-GPU streaming and computation scheme which allows arbitrarily-sized simulations independent of limited local GPU memory, and across multiple GPUs in a system, with near-optimal bandwidth utilization.
- A study of the compression rate and performance as well as simulation accuracy implications of custom application-specific lossy compression schemes for essential intermediate scene visibility data, which grows with the square of the geometric input scene complexity and is thus a central limiting factor for overall performance.
- An in-depth evaluation of M-RTX-RSim performance on a multi-GPU system, with comparisons to multi-core CPU versions across five scenes, including an overhead analysis for streaming and data synchronization, and a multi-GPU scalability study.

To the best of our knowledge, this work is the first to use Vulkan compute, across multiple GPUs, and raytracing using NVIDIA RTX in scientific computing, and one of the first to use hardware raytracing in a scientific simulation context in general. We previously published an early version of this work at a workshop.[9] This publication expands on our original findings with a focus on extending the algorithm for multi-GPU support, and the design, implementation and evaluation of various means of application-specific compression. We have also performed a far more extensive evaluation including real-world test cases, and provide a performance comparison of our Vulkan compute based simulation with an equivalent CUDA version, as the latter is far more commonly used in HPC.

The remainder of this article is structured as follows. Section 2 provides an overview of the general room response simulation algorithm, its individual phases, and the involved datasets. In Section 3, we describe our Vulkan compute and hardware raytracing accelerated room response simulation method. Section 4explores application-specific compression schemes, while asynchronous data streaming and multi-GPU parallelization are summarized in Section 5. Section 6 analyzes the results of our in-depth performance evaluation. Finally, an overview of related work in Section 7 precedes the conclusion of this article.

2 | ROOM RESPONSE SIMULATION

2.1 | Overview and motivation

Figure 2 provides a schematic overview of room response simulation in the context of this work. Scene geometry (**G**) is presented as a set of triangle meshes. Some light source (**L**) generates a pulse of modulated light, which is reflected in **G** until it either reaches back to a sensor (**S**)—usually in close proximity to the light source—or until a given time interval elapses.

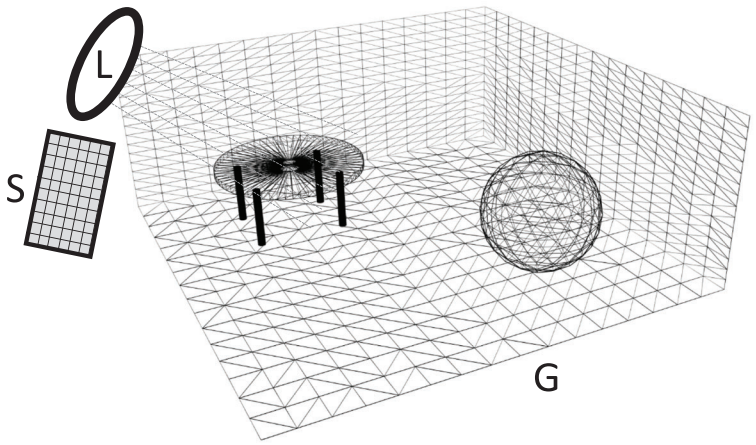


FIGURE 2 Schematic illustration of RSim

Unlike traditional image rendering, the *timing* properties of light propagation are crucial for this type of simulation. The goal of the simulation is to compute a radiosity time series for every geometric primitive (i.e., triangle), including the sensor pixels. Based on this time series, which simulates the actual photons received by a ToF camera sensor, scene depth can be reconstructed and an approximate 3D representation can be derived. With RSim, since the exact depth is known, different scenes and reconstruction schemes can be easily evaluated.

2.2 | Computational algorithm

The mathematical background for room impulse simulation has been studied in previous work.[2] For our contribution, a numerical view of the algorithm, centered on the most resource-intensive operations, is required. As such, we will only briefly outline the overall algorithm here, and then discuss its most performance-relevant parts in more detail from a computational perspective.

The main phases of the RSim algorithm are as follows:

1. Reading input data, including geometric primitives (G), their surface material information (ρ), and initial impulse information.
2. Pre-computation of the per-triangle area (A_i).
3. Mutual signal delay computation, storing the signal delay for each triangle pair (g_i, g_j) in τ_{ij}.
4. Mutual visibility computation, evaluating the energy transfer between each triangle pair stochastically and storing in K_{ij}. Triangle pairs which do not transfer energy or are not mutually visible are represented by 0.
   This phase optionally includes building and using a geometric acceleration structure.
5. For each timestep $t \in [0, T)$:
   Propagate radiosity, computing $rad_{t,i}$ for each triangle g_i in all pairs (g_i, g_j) based on K_{ij} and $rad_{t-\tau_{ij},j}$.
6. Compute the distance from the light/sensor position to each triangle g_i, based on rad.

2.3 | Performance analysis and algorithmic considerations

In order to guide the hardware accelerated implementation of RSim, it is important to understand the basic performance of the individual phases of the algorithm. In this context, it is very advantageous to study the computational complexity of each phase.

Phases 1 and 2 are readily identified as $O(N)$, with N being the number of triangles, and can thus be assumed to be largely irrelevant for total execution time. Phase 3 is $O(N^2)$, as each pair of triangles is considered. However, the fixed factor is low, and compared to the remaining phases, even N^2 complexity is largely negligible. The final distance computation in phase 6 is based on cross-correlation across per-timestep radiosity for each geometric primitive, and results in a complexity of $O(N * T^2)$. T is the number of timesteps that are simulated. It is generally much smaller than N, and the fixed factor is small for this computation as well.

TABLE 1 CPU execution times of individual RSim phases (in seconds)

Phase	Small room		Medium room		Large room		Traffic		Industry	
Mutual visibility	8.6	17.7%	50.9	12.3%	560.4	10.4%	445.2	10.5%	775.5	7.2%
Radiosity simulation	38.8	79.6%	361.0	87.0%	4829.1	89.3%	3764.5	88.9%	9993.3	92.6%
Other	1.4	2.8%	3.3	0.8%	11.0	0.2%	13.4	0.3%	18.2	0.2%

2.3.1 | Mutual visibility computation

The first of the two remaining phases, mutual visibility computation 4, requires stochastically evaluating the visibility between every pair of triangles in the scene, with each such check in turn involving one or more raycasts. A naïve implementation of raycasting without additional data structures requires a ray-triangle intersection check against all other $N - 2$ triangles in the scene. With S stochastic samples this results in an overall complexity of $O(N^3 * S)$.

Since this is not a scalable approach, a significant amount of effort has been invested into optimizing ray-scene intersection computations, primarily in the graphics industry. Most approaches use *geometric acceleration structures*, which subdivide the scene recursively into progressively smaller sub-regions, which are stored in some form of tree structure. The most commonly used types of acceleration structures are bounding volume hierarchies (BVHs),[10] octrees,[11] and KD-trees.[12] While each of them induces specific tradeoffs concerning build and memory overheads as well as lookup performance, the crucial point for our purposes is that they reduce average-case query complexity from $O(N)$ to $O(\log(N))$. The current CPU-based RSim implementation uses octrees, and thus the actual expected complexity for the mutual visibility computation phase is $O(N^2 * \log(N) * S)$.

2.3.2 | Radiosity simulation

Finally, phase 5 of the algorithm implements the actual simulation of radiosity propagation over time. It requires the previously computed signal delay τ_{ij} and mutual visibility information K_{ij}, as well as the previous radiosity up to the currently computed timestep *rad*. The radiosity in this context consists of an array of structs with four FP32 values as shown in Table 2, namely B, the actual radiosity, Z the importance, E the emission/source, and R the receiver.[2] Indexing into the *rad* array is based on the signal delay for each pair of triangles examined: for each timestep t and each pair (g_i, g_j), energy is propagated between triangles in the pair from time $t - \tau_{ij}$ according to the mutual visibility K_{ij} as well as their surface properties ρ_i and ρ_j. Note that at this point, all necessary geometric information about the scene is encoded in K_{ij} and A_i, and therefore G is no longer necessary.

Due to the nature of this process, which needs to consider each triangle pair in each time step, the computational complexity for this phase is bounded by $O(N^2 * T)$.

2.3.3 | Measured performance

While complexity categories provide a good baseline for what to expect, actual implementations might occasionally behave in unexpected ways due to, for example, large differences in constant factors, or caching effects. Therefore, in order to verify our performance analysis, we measured the multi-core CPU runtime of individual phases of RSim. The same system and scenes as in our performance evaluation were used, the details of which are described in Section 6.1.

The measurements in Table 1 demonstrate that real-world performance largely matches the expectations set by our theoretical complexity analysis. Mutual visibility computation and radiosity simulation dominate the performance at all problem sizes, and by an increasing margin for larger test cases.

3 | VULKAN COMPUTE AND RAYTRACING METHOD

In this section, we describe our Vulkan-based implementation of RSim: RTX-RSim. In order to alleviate implementation overhead regarding the basics of Vulkan API interaction, setting up raytracing, and compute shader invocation, we leveraged the open-source Raygun engine.[13]

Note that currently the only API options for accessing raytracing hardware are Microsoft DXR, NVIDIA OptiX, and Vulkan VKRay, with no direct support in OpenCL and CUDA. Of the three available choices, the first is exclusive to Microsoft Windows and the second is entirely

proprietary. Therefore we chose VKRay for our implementation, which is required to be able to run on Linux-based headless server hardware, and should maintain portability to future raytracing hardware.

3.1 | Data management

For our massively parallel Vulkan implementation, we take a data-centric view of the RSim algorithm. As outlined in Section 2.2, there are several intermediate results which need to be prepared before starting the main time series simulation loop, as well as storage which needs to be allocated for storing radiosity information over time, and the final distance result.

Table 2 summarizes the data involved in the complete RTX-RSim computation. N is the number of triangles in the simulation geometry, S sample points are evaluated for each triangle pair, and T is the number of timesteps computed in the simulation. In this context, it is important to note that in any realistic large-scale RSim use case $S \lll T \ll N$ holds.

Triangles are the actual scene geometry, which is stored in a Vulkan indexed vertex buffer format for efficient use with hardware raytracing, while ρ stores per-triangle material information. The *Raytracing Buffers* table entry represents buffers which store opaque internal data used by the hardware during raytracing, including top- and bottom-level acceleration structures as detailed in Section 3.2.1. *Sample Coordinates* is a read-only buffer storing 2D coordinates to allow sampling uniformly distributed points of any triangle, which are used during mutual visibility computation. The result of the mutual visibility evaluation step, K_{ij}, stores the relative visibility between each pair (g_i, g_j) of triangles in the scene and the resulting energy transfer factor.

Radiosity is computed during the main simulation, and contains light transport information for each timestep. More specifically, it comprises four FP32 values, and we have evaluated both an array-of-structures (AoS) and a structure-of-arrays approach for this data. Both approaches are within 10% of each other across our experiments, with the AoS approach offering slightly better performance. We therefore chose it as our default implementation.

Finally, the *Distance* buffer stores the computed distance of each triangle from the light/sensor position, based on *Radiosity* results.

3.1.1 | Memory size optimization

For discussing memory size and storage optimization, it is important to remember that $S \lll T \ll N$ holds, and as such the triangle pair relation K_{ij} dominates memory requirements. For this reason, and since this memory footprint translates directly to a PCIe streaming performance bottleneck when the on-board GPU storage capacity is exceeded, we have implemented and evaluated several application-specific minimum overhead compression schemes. Our findings and choice for implementation in M-RTX-RSim are detailed in Section 4.

A notable difference between the data structures listed in Table 2 and the quantities used in the algorithmic description in Section 2.2 is that no data structure exists for storing τ_{ij}. Similarly to K_{ij}, this quantity would need to be stored for each pair of triangles, with increasing geometric complexity resulting in very large memory requirements. However, unlike K_{ij} which encodes geometric visibility information about the scene which is complex to recompute, τ_{ij} is the result of a basic distance calculation modified by a fixed propagation speed factor. As such, we can rematerialize

TABLE 2 Data buffers (size in number of elements)

Contents	Format	Size
Triangles (G)	Indexed vertex buffer	N
ρ	$3 * \text{FP32}^a$	N
Raytracing buffers	Internal/opaque	$O(N)$
Sample coordinates	$2 * \text{FP32}$	S
K_{ij}	FP16^b | UINT8^c LUT^d index	N^2
Radiosity	$4 * \text{FP32}$	$N * T$
Distance	FP32	N

Note: K_{ij} format depends on compression strategy explained in Section 4.1.
[a]32-bit floating point.
[b]16-bit floating point.
[c]8-bit unsigned integer.
[d]Look-up-table.

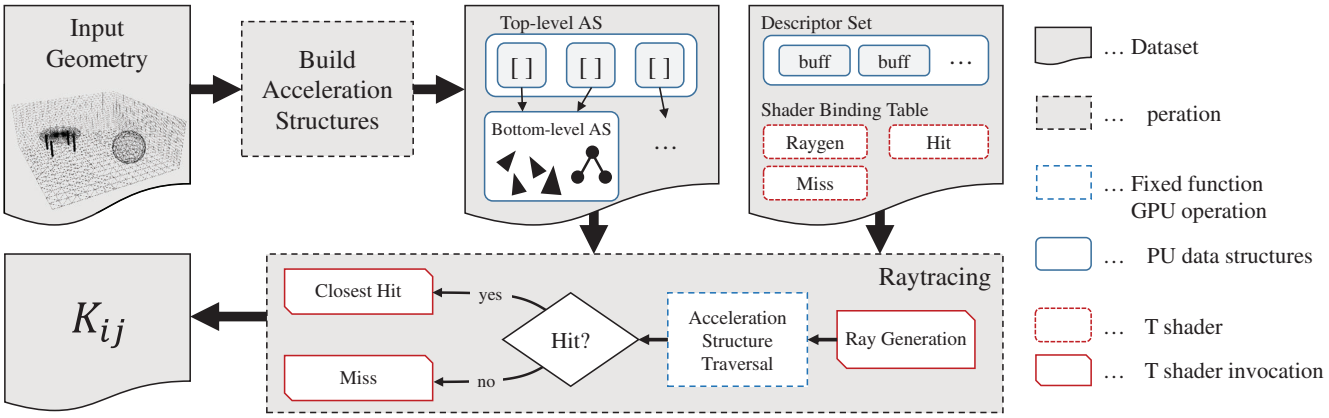


FIGURE 3 Hardware raytracing overview for RTX-RSim K_{ij} computation

these τ values as required, saving memory space, and actually achieving equal performance, as on modern GPUs the minor computation overhead is negligible compared to increased cache pressure for accessing a large data structure.

3.2 | Hardware raytracing for mutual visibility

In order to perform hardware raytracing with the VKRay API, three main ingredients need to be prepared: *acceleration structures* encompassing geometry information, a *shader binding table* which contains all the shader programs used during the raytracing process, and a *descriptor set* enabling access to resources required by these shaders. An overview of the hardware raytracing process with NVIDIA VKRay as it is leveraged by RTX-RSim is shown in Figure 3.

3.2.1 | Acceleration structure building

For efficient scene traversal and updating, the hardware raytracing API distinguishes two types of acceleration structures. The top-level acceleration structure contains instances of objects, each of which has its own transformation matrix and a reference to some bottom-level acceleration structure containing the actual geometry. These bottom-level structures can potentially be reused by multiple instances.

As geometry is entirely static in the current RTX-RSim use case, we can instruct the raytracing API to build acceleration structures optimized for traversal—rather than allowing efficient updating. This is accomplished during build information setup by disabling `VK_BUILD_ACCELERATION_STRUCTURE_ALLOW_UPDATE_BIT_NV`. We also tested the fast trace bit,* but could not verify any performance difference in our evaluation scenes.

3.2.2 | Descriptor set and shader binding table

In terms of data access, our raytracing shaders require read-only access to G, ρ, and the *Sample Coordinates*buffer, as well as write access to K_{ij} in order to store the resulting energy transfer factors. Shader binding is relatively simple compared to most real-time rendering use cases: we only require a ray generation shader, and a single hit and miss shader. Both the any-hit and intersection shader stages are unbound in RTX-RSim, as this use case does not require any-hit materials, and all geometry is triangle-based at this point. The reason only a single hit shader is needed is that ρ encodes all the per-triangle material information required to compute K_{ij}.

3.2.3 | Raytracing invocation

From the host program perspective, hardware raytracing is invoked by calling `traceRaysNV` on a grid of a given size in up to three dimensions. In RTX-RSim, we use a 1D grid and launch one thread per required triangle pairing. Since the matrix is symmetric and triangles do not need to compute visibility to themselves, the number of actually required pairs is $(N^2 - N)/2$.

*`VK_BUILD_ACCELERATION_STRUCTURE_PREFER_FAST_TRACE_BIT_NV`

Thus, for each index in the range $[0, (N^2 - N)/2)$, the ray generation shader stage is invoked. Shader 1 contains a simplified excerpt of the most relevant portion of the RTX-RSim ray generation shader program. We omitted some boilerplate code including extension specifications and index computation, which maps from the linear 1D index in the execution range provided by `gl_LaunchIDNV.x` to the 2D triangle pair indices `srcTriIdx` and `dstTriIdx`.

Shader 1: Simplified RTX-RSim raygen GLSL code.

```
1  for(uint j = 0; j < ubo.numSamples; ++j) {
2    const vec2 triPoint = trianglePoints.p[j];
3    const vec3 from = mapPointFromUnitTriangle(srcTri, triPoint);
4    const vec3 to = mapPointFromUnitTriangle(dstTri, triPoint);
5    const vec3 ray = to - from;
6
7    if(dot(normalize(ray), srcNormal) < COPLANAR_EPSILON) continue;
8
9    payload.dst = dstTriIdx;
10
11   traceNV(topLevelAS, rayFlags, cullMask,
12           0 /*sbtRecordOffset*/,
13           0 /*sbtRecordStride*/,
14           0 /*missIndex*/, from, tmin,
15           ray, length(ray) + DISTANCE_EPSILON,
16           0 /*payload*/);
17
18   if(payload.visible) {
19     const float cosPhiDst = receiverCosPhi(cosPhi(dstNormal, -ray), cosR);
20     const float cosPhiSrc = sourceCosPhi(cosPhi(srcNormal, ray), cosS);
21
22     if(cosPhiDst > 0.0f && cosPhiSrc > 0.0f) {
23       kijSum += (cosPhiSrc * cosPhiDst) / (PI * squaredNorm(ray));
24       ++n;
25     }
26   }
27 }
28
29 if(n > 0) {
30   const float kij = kijSum / float(n);
31   writeToBuffer(kij_t(kij), gl_LaunchIDNV.x);
32 }
33 else {
34   writeToBuffer(kij_t(0.0f), gl_LaunchIDNV.x);
35 }
```

The stochastic sampling loop (Line 1), which encompasses most of the shader code performs `ubo.numSamples` iterations, corresponding to S in our previous algorithmic discussions. For each sample, a source and destination point inside the respective triangles is computed (`mapPointFromUnitTriangle`, Lines 3–4). If they are not coplanar (Line 7), `traceNV` (Line 11) traces a ray from the source to the destination.

Our raytracing *payload*—structured data allocated per thread and propagated through the raytracing stages—is very simple. It contains only two values: the index of the expected target triangle (`payload.dst`, Line 9) and a flag indicating visibility (`payload.visible`, Line 18). Both the closest hit and miss shaders are trivial: the miss shader simply sets `payload.visible = false`, while the hit shader checks whether the expected target triangle was hit (via `gl_PrimitiveID`) and sets the `visible` value accordingly.

If the payload was visible, K_{ij} is computed based on the material properties of the source and destination triangles (`sourceCosPhi`, Line 20, and `receiverCosPhi`, Line 19), which access and take into account the respective material information stored in the ρ buffer. Once all S samples were processed in this fashion, the resulting value `kij` is cast to the appropriate type and written to the output buffer (Line 31). When all threads of the given ray generation grid have been processed, K_{ij} is fully computed and thus the mutual visibility computation phase of the algorithm discussed in Section 2.3.1 is complete.

3.3 | Radiosity simulation with compute shaders

Besides mutual visibility computation, which is accomplished with hardware raytracing, the second compute-intensive phase of the RSim algorithm as identified in Section 2.2 is *radiosity simulation*. In order to fully leverage GPU performance for this phase, and integrate seamlessly with the data structures used in hardware raytracing, we have implemented the entire simulation with Vulkan compute shaders. Shader 2 contains a simplified excerpt of GLSL code for a single radiosity simulation step.

After initializing some values (Lines 1–4), the algorithm loops over all destination triangles (Line 6). For each of them, τ_{ij} is computed (Line 10), and based on this propagation time, the previous emission at the correct time point is retrieved from the radiosity buffer `radBuffer` (Line 16). Finally, K_{ij} is read (Line 21) and applied in the radiosity computation (Lines 24–25). After the loop, the surface properties of the source triangle are applied (Lines 29–30), and the final radiosity value for the given timestep and triangle index is stored (Line 31).

Shader 2: Simplified radiosity simulation GLSL code.

```
1  const int N = ubo.numTriangles;
2  radBuffer.r[timestep * N + srcTriIdx].B = 0.0f;
3  radBuffer.r[timestep * N + srcTriIdx].Z = 0.0f;
4  const mat3 srcTri = getTriangle(srcTriIdx);
5
6  for(uint dstTriIdx = dstTriStart; dstTriIdx < dstTriEnd; ++dstTriIdx) {
7    if(srcTriIdx == dstTriIdx) continue;
8
9    const mat3 dstTri = getTriangle(dstTriIdx);
10   const int tauij = calcTauij(srcTri, dstTri);
11
12   // Skip if wave has not yet propagated between triangle i and triangle dstTriIdx.
13   if(timestep < tauij) continue;
14
15   // Use radiosity from the point in time where emission actually took place.
16   const Radiosity link = radBuffer.r[(timestep - tauij) * N + dstTriIdx];
17
18   if(link.B == 0.0f && link.Z == 0.0f) continue;
19
20   const uint index = (dstTriIdx - dstTriStart) * N + srcTriIdx;
21   const kij_t kij = curKijBuffer.f[index];
22   if(kij == 0.0f) continue;
23
24   radBuffer.r[timestep * N + srcTriIdx].B += clamp01(kij * calcArea(dstTri)) * link.B;
25   radBuffer.r[timestep * N + srcTriIdx].Z += clamp01(kij * calcArea(srcTri)) * link.Z;
26 }
27
28 Radiosity rad = getRadiosity(timestep, srcTriIdx);
29 rad.B = rhoBuffer.r[srcTriIdx].rho * rad.B + rad.E;
30 rad.Z = rhoBuffer.r[srcTriIdx].rho * rad.Z + rad.R;
31 setRadiosity(timestep, srcTriIdx, rad);
```

An important point of note for these simulation steps is that they are only parallelized across the given number of N triangles, rather than the full set of pair relations as is the case for mutual visibility computation (Section 3.2.2). This limits the efficiency of the parallelization for very small N, as the number of threads in flight will not be enough to fully saturate a large GPU. We also tested a 2D parallelization, however, as the radiosity from each triangle to all other visible triangles needs to be summed up in each time step, either atomic operations or some form of reduction is required. This induces a significant overhead compared to sequential per-triangle summation. Furthermore, the main goal of the GPU implementation is to perform medium- to large-scale simulations with 10,000 triangles or more, at which point a 1D parallelization which loops over destination triangles—as implemented in the current version of M-RTX-RSim—is far more efficient. This parallelization approach also lends itself naturally to workload distribution across multiple GPUs and streaming of the K_{ij} buffer, as discussed in Section 5.

4 | MUTUAL VISIBILITY INFORMATION COMPRESSION

The storage size of the mutual visibility matrix K_{ij} dominates the overall memory requirements in all practical RSim applications, for the reasons outlined in Section 3.1.1. While all other data required by the simulation can be kept resident in GPU local memory and is relatively small in any reasonable target use case, K_{ij} easily grows to several tens of gigabytes in size in real-world simulations. In the initial CPU-based RSim implementation, individual K_{ij} values are stored in FP32 format, and a large shared memory system is assumed for high-performance simulations, with a fallback to memory-mapped I/O.

For our GPU-based implementation, after validating that it does not affect result quality for common values of S (the number of stochastic samples taken per triangle pair), we initially chose to store K_{ij} in a 16-bit floating point format.[14] This is sufficiently precise in the usual range of these factors, which is [0.0, 10.0] (with relatively few elements larger than 10).

Of course, even when using FP16, scaling with N^2 still limits simulation size significantly, which motivates our multi-GPU streaming approach described in Section 5. Streaming solves the primary issue of GPU memory capacity limiting the types of experiments which can be simulated by M-RTX-RSim. However, with common PCIe 3.0 16x links to each GPU, even at 16-bits per element, the radiosity simulation step described in Section 3.3 is limited by streaming bandwidth and cannot fully leverage the compute capabilities of high-end GPUs.

To improve the performance of M-RTX-RSim, we investigated several ways to further reduce the storage space and bandwidth required for handling K_{ij}. For this use case, we have to consider the following constraints which need to be satisfied by a given compression or storage scheme in order to be used in the context of our simulation:

1. The compression needs to be *fixed-rate*, as the overhead of dealing with variable-sized sub-blocks on the GPU on the main path of the simulation negates the performance benefits of reduced bandwidth requirements.
2. When employing lossy compression, the quality of the final simulation result is more important than the quality of the representation of K_{ij}.
3. The compression step is done only once per block, while decompression is performed T times. As such, compression speed, while important, is not as critical as decompression speed.
4. There is no inherent spatial locality in the K_{ij} data that could be exploited.

A natural choice which would cover the first three points very well is using GPU texture compression supported in hardware.[15] However, all texture compression methods rely on *spatial* locality as their main working principle, which is generally not the case in K_{ij} data. Nonetheless, the data has some other application-specific features which can be leveraged.

4.1 | Value distribution and application-specific compression

Figure 4 depicts a fine-grained histogram of the value distribution for K_{ij} in one of our smaller test samples (see Section 6.1 for details on all samples). Two important characteristics of this data are visible, which are consistent across most real-world use cases: it is distributed logarithmically with a peak close to 1 and a long tail toward higher values, and the values are clustered in a number of groups. This is a result of different objects in the space being simulated, each consisting of several triangles, and those triangles having similar distances and visibility to those making up other individual objects in space.

In order to utilize this structure while maintaining very fast decompression on the GPU and allowing per-element addressing, we decided to store K_{ij} as 8-bit indices with an individual per-block lookup table (LUT). The compression challenge then amounts to constructing a high-quality LUT for a given block's K_{ij} distribution. Note that in the context of the full simulation, compression can be performed per-block in a streaming manner, and that access to the entirety of K_{ij} at the same time is not required.

The first choice from existing well-known algorithms for this purpose would be performing k-means clustering,[16] however, even with optimized approaches, generating 2^8 clusters from >10 GB of data is not feasible in a time frame which is reasonable for our purposes.

To rapidly generate a suitable LUT, we designed an algorithm which sorts the data points and then performs a recursive refinement of data ranges. Listing 1 contains the most relevant excerpt of the implementation of this algorithm. The `LUTRange` type (Lines 1–14) represents a value range ultimately mapped to a single LUT entry, which is ordered by (Lines 11–13) an error metric derived from the total range of values stored within the index range (Line 7). The actual LUT generation then starts by sorting the data (Line 18), and creates a single range spanning its entirety (Lines 19–20). In the `while`loop, the range with the highest error metric is chosen (Line 23) and refined by splitting (Lines 26–28). The execution time for this algorithm is dominated by `std::sort`, and thus $O(N * \log(N))$ in the number of elements per block being compressed.

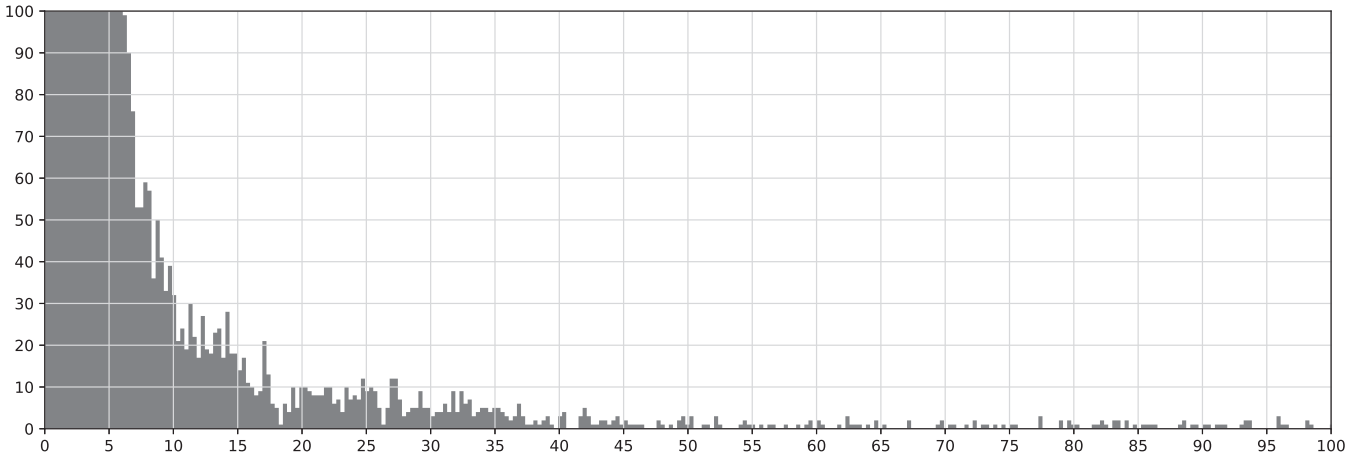


FIGURE 4 K_{ij} histogram (5000 bins) for the *Small room* scene. *Y*-axis capped at 100 elements for better visibility of small clusters and *X*-axis capped at 100, because 99.99% of all elements are less than 100

Code Listing 1: C++ Compression *"lutrange"* LUT Generation Code Excerpt

```cpp
struct LUTRange {
  int64_t start_idx, end_idx;
  float error_metric;

  LUTRange(const std::vector<float>& sorted, int64_t start_idx, int64_t end_idx)
        : start_idx(start_idx), end_idx(end_idx) {
    error_metric = (sorted[end_idx] - sorted[start_idx]);
    // Prevent further splitting of ranges that already have minimal size
    if(end_idx - start_idx == 1) error_metric = 0;
  }
  bool operator<(const LUTRange& rhs) const {
    return error_metric < rhs.error_metric;
  }
};

// ... Actual LUT Generation
auto sorted = std::vector(first, last);
std::sort(sorted.begin(), sorted.end());
std::priority_queue<LUTRange> rangeQ;
rangeQ.emplace(sorted, start_idx, (int64_t)sorted.size() - 1);

while(rangeQ.size() < LUT_SIZE - 1) {
  auto r = rangeQ.top();
  rangeQ.pop();

  int64_t mid_idx = (r.start_idx + r.end_idx) / 2;
  rangeQ.emplace(sorted, r.start_idx, mid_idx);
  rangeQ.emplace(sorted, mid_idx, r.end_idx);
}
```

While this LUT generation scheme is quite efficient, with consideration for the structure of typical K_{ij} data as presented in Figure 4, we also evaluated an even simpler approach: directly mapping values using a logarithmic transformation of the per-block value range. In terms of complexity class, at $O(N)$ this is as fast as any algorithm which accomplishes the purpose of mapping all N values could possibly be.

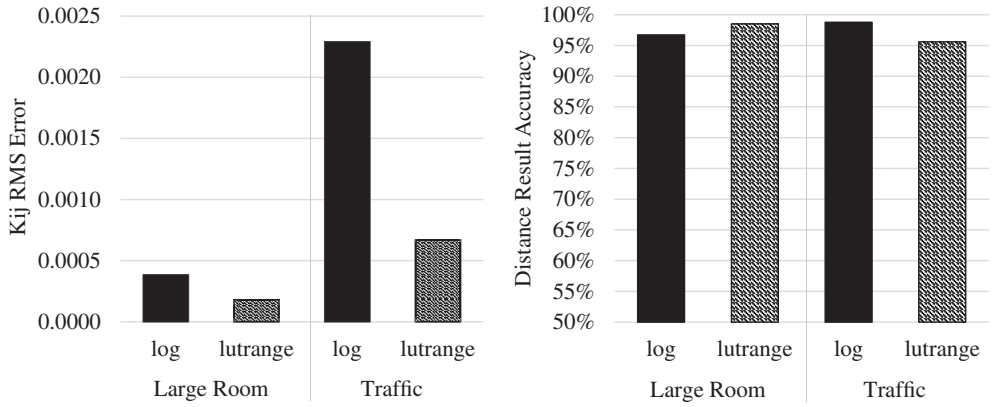
The logarithmic mapping works by linearly interpolating every $\ln(K_{ij})$ value into an 8-bit range. Because $\ln(0)$ is not defined, special care must be taken when handling zeros. We decided to map a zero value in the input to a zero value in the output, and then linearly interpolate every other value from the range $[\min(\ln(K_{ij})), \max(\ln(K_{ij}))]$ (where 0 is not considered for the min) into the range [1,255]. The lookup is performed analogously using the exp function. Note that K_{ij} can either be the whole 2D matrix, or one streaming chunk as discussed in Section 5.2, depending on the size of the input scene.

4.2 | Compression quality

We have analyzed the quality impact of both of these approaches on the *Large room* and *Traffic* data sets, which are described in more detail in Section 6.1. Figure 5A compares the root mean square (RMS) error of the K_{ij} values directly. Both approaches show small errors in this metric, but the error for the simple logarithmic mapping ("log") is two to four times as large as for the recursive "lutrange" approach. However, the RMS error of the K_{ij} representation does not correlate directly with the accuracy of the simulation, which is the ultimately relevant metric. In particular, the RMS error metric at different scales of the absolute energy transport value is weighted differently with regard to the extent to which inaccuracies at those scales influence the simulation result. As such, in Figure 5B, we show the relative accuracy of the simulation when performed with K_{ij} data compressed using either of the two approaches. The accuracy metric is the percentage of computed distance results which are within a small delta of the result provided by a simulation performed at full FP32 precision. In the *Large room* sample, the accuracy of both approaches is very similar, while in the *Traffic* sample, the logarithmic mapping approach is actually noticeably more effective. The reason for this result, despite the significantly worse RMS error in the K_{ij} data, is that the absolute error in the representation of the K_{ij} matrix is not as important for the quality of the final result as the relative error and how it maps to the scene geometry.

For this reason, as well as its lower computational complexity and the ability to easily parallelize the compression step across all available CPU cores, we chose to use LUT-based compression with the "log" LUT generation method in M-RTX-Rsim.

Given the good results of both the "log" and the "lutrange" approaches, an interesting question is whether further (sub-8-bit) compression is viable, at least from a quality perspective. In terms of performance, such compression approaches would have the inherent disadvantage of requiring several bit-wise operations per operand to extract the individual values from the data stream, which will have a significant performance cost on GPU architectures. However, for streaming workloads particularly 4-bit-per-value compression might still be viable if quality can be maintained, as alignment issues would be minimized by fitting two 4 bit values per byte.



(A) RMS errors of K_{ij} representations. **(B)** Accuracy of simulation result.

FIGURE 5 Comparison of compression quality with recursive "lutrange" algorithm and direct logarithmic mapping

TABLE 3 Data for comparison of compression quality with recursive "lutrange" algorithm and direct logarithmic mapping

	K_{ij} RMS error		Distance result accuracy	
	Large room	**Traffic**	**Large room**	**Traffic**
log	0.000385	0.002290	96.71%	98.75%
lutrange	0.000180	0.000671	98.50%	95.59%
log4	0.008197	0.117436	67.82%	83.90%
lutrange4	0.001965	0.077244	54.14%	65.19%

In order to evaluate this possibility, we extended our compression implementation and quality analysis with "log4" and "lutrange4" algorithms. These implement the same principles as their 8 bit counterparts, but with only 4 bits per value. Table 3 depicts the error metrics of these approaches. Clearly, there is a substantial and unacceptable drop in our accuracy metric in most cases. Even worse, when looking at the actual error magnitude rather than just the percentage of sufficiently accurate results, the mean distance error is 0.3 m in the best case and 8.2 m in the worst case. Given the real-world and industrial use cases for M-RTX-Rsim, even 0.3 m is an unacceptable level of error, and as such 8 bits per K_{ij} value is the smallest viable storage strategy while using a per-element compression method which allows arbitrary indexing.

5 | MULTI-GPU PARALLELIZATION AND ASYNCHRONOUS STREAMING

5.1 | Overview and motivation

As described in Table 2 the memory requirements for K_{ij} grow with the square of the triangles in the scene. This results in not having sufficient GPU memory for large input scenes. Since host memory is usually larger than GPU memory we implemented an asynchronous streaming algorithm to allow for larger problem sizes. Such large scenes often require significant simulation time even with GPU acceleration, and therefore we extended our simulation scheme to allow for multi-GPU parallelization. This requires some unavoidable additional synchronization due to the properties of the algorithm, but as the radiosity data being produced by the simulation during each timestep is relatively small ($O(N)$) compared to the input data required for its computation ($O(N^2)$), we can still achieve meaningful multi-GPU scaling. Detailed performance results are available in Section 6.

5.2 | Streaming for radiosity simulation

Figure 6 illustrates the complete data flow and parallelization strategy for multi-GPU radiosity simulation in M-RTX-RSim. Before the start of timestep computation, in step ❶, the K_{ij} matrix is chunked into C partitions and each chunk is compressed by applying the algorithm detailed in

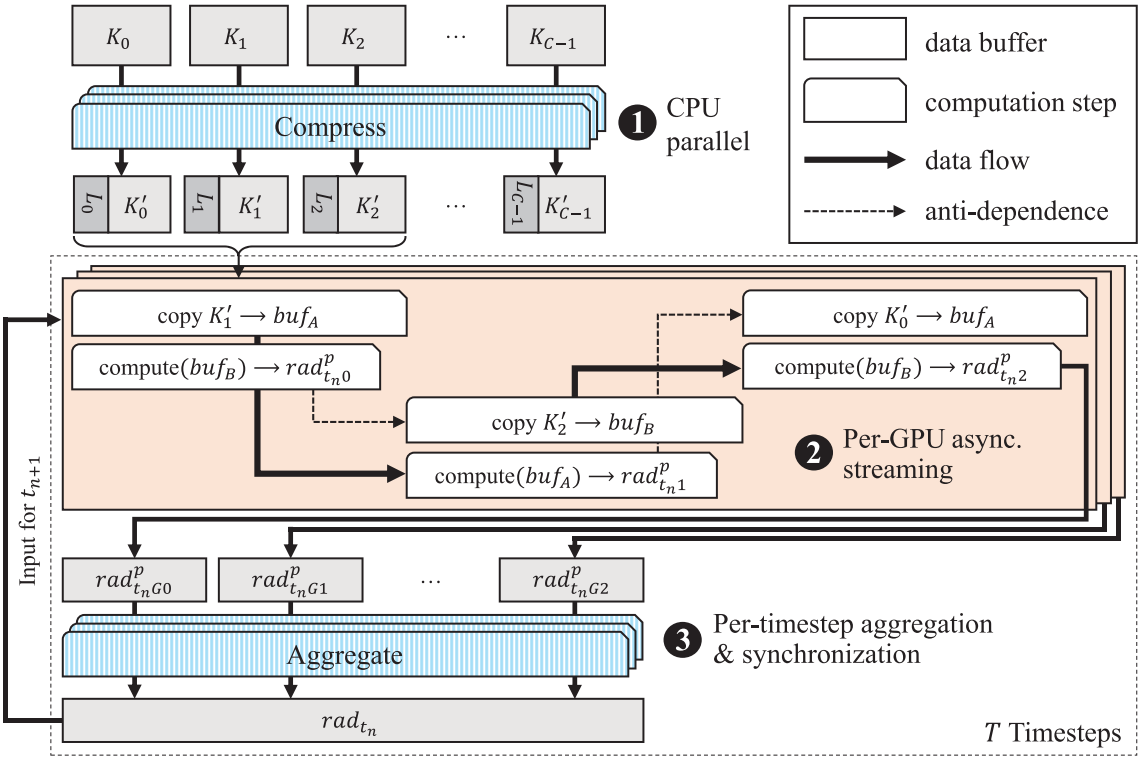


FIGURE 6 Data and compute dependency visualization for streaming

TABLE 4 Additional data buffers for streaming (size in number of elements)

Contents	Format	Size
buf_A	FP16 | UINT8 LUT index	$chunk * N$
buf_B	FP16 | UINT8 LUT index	$chunk * N$

Note: Format depends on compression strategy.

Section 4. This results in the compressed chunks $K'_0, \ldots, K'_{C-1}$, each storing indices into their associated LUTs $L_0, \ldots, L_{C-1}$. Note that the compression is implemented in a CPU-parallel fashion across all cores.

After this preparatory work, T radiosity propagation timesteps are simulated. Each of these timesteps consists of a GPU-parallel compute phase 2 with optional asynchronous streaming, followed by a CPU-parallel aggregation and synchronization phase 3. Both of these phases will be discussed in more detail in the remainder of this section.

5.2.1 | Additional data requirements

In addition to the data buffers described in Table 2 our streaming implementation needs two intermediate buffers which are listed in Table 4. The two buffers buf_A and buf_B are used for double buffering the data chunks of K_{ij} in GPU memory. Each buffer holds *chunk* number of rows of the total visibility matrix K_{ij}, where *chunk* is the chunk size. In our implementation, the desired chunk size in bytes can be specified as an argument, while the actual chunk size is dynamically determined to be as many rows of K_{ij} as necessary to closely fit the desired size.

For this streaming approach the memory size optimization described in Section 3.1.1 of reducing the N^2 data requirement of K_{ij} to $(N^2 - N)/2$ is not suitable, because this representation cannot be sensibly divided into contiguous chunks of data necessary for work on a contiguous range of output elements. Foregoing this optimization incurs some host memory overhead, but yields a significant performance advantage.

For our multi-GPU implementation the chunks are distributed among all devices as shown in Figure 7. Each device has a known set of chunks associated with it, which are streamed from the host as illustrated in Figure 6. The device chunks are static and determined at the start of the algorithm to provide an approximately uniform distribution of K_{ij} among all devices.

5.2.2 | Streaming algorithm

The streaming algorithm, in terms of computation, differs only slightly from the non-streaming version. The major differences are the data layout of K_{ij} (as described in Figure 5.2.1) and the fact that one timestep computation is split into multiple compute shader dispatch calls.

Figure 6 illustrates the behavior of the streaming algorithm for each GPU for an arbitrary timestep t_n in the box associated with overall computation step ❷. The number of chunks for this illustration is chosen to be three, denoted as 0, 1, and 2. During the computation associated with each chunk the data required by the next chunk (e.g., K_1 during *chunk* 0) is copied from the host buffer to the device local buffer $buf_{A|B}$. At the same time the previously copied chunk of K_{ij} is used to compute the radiosity, for example, $rad_{t_n 0}$ is computed during *chunk* 0 in timestep t_n based on data copied during *chunk* 2 in timestep t_{n-1}.

Latency hiding optimization

Utilizing the ability to concurrently copy data from host to GPU memory and also do computation, we optimized our streaming to hide as much latency as possible. Figure 6 shows the data dependencies that exist. Since there is no dependency between copying the next chunk of K_{ij} and computing the radiosity for the current chunk, this is done in parallel. Double buffering is used to facilitate this, where buf_A and buf_B are the two buffers which store chunks of K_{ij}. Each chunk alternates between these buffers, concurrently using one for copying and the other for computation. Synchronization between chunks is guaranteed by using Vulkan *semaphores*, making sure both the copy and compute operations have completed before starting work on the next chunk.

5.2.3 | Multi-GPU

For efficient multi-GPU execution, the primary challenge is minimizing the data transfers and synchronization required. To that end, all buffers shown in Tables 2 and 4 are duplicated on every device, with the notable exception of K_{ij} which is distributed among all devices. Note that this reduces the need for K_{ij} streaming, as the available memory on all devices can contribute to storing the visibility matrix. As such, if a scene with N triangles can be computed without resorting to streaming on a single GPU, then $\sim 2N$triangles can be simulated on 4 GPUs without streaming.

In terms of the radiosity simulation compute kernel, the extension to multi-GPU does not require significant changes, as the splitting of work into chunks already provides the main mechanism required for distributed computation. However, crucially, the computed radiosity of each timestep needs to be synchronized between all GPUs as shown in step ❸ of Figure 6. The value $rad^p_{t_n G0}$, for example, represents the partial radiosity of timestep t_n, computed on the first GPU device. These partial radiosities store the contribution of their device chunk to the radiosity of all triangles in a given timestep. Therefore, all partial radiosities need to be accumulated to get the total radiosity result for this timestep, which is in turn required for subsequent computation. Consequently there is some synchronization and computation overhead compared to the single-GPU case, which cannot be eliminated as the computation depends on the radiosity contribution of all chunks. It is important to note in this context that the radiosity result

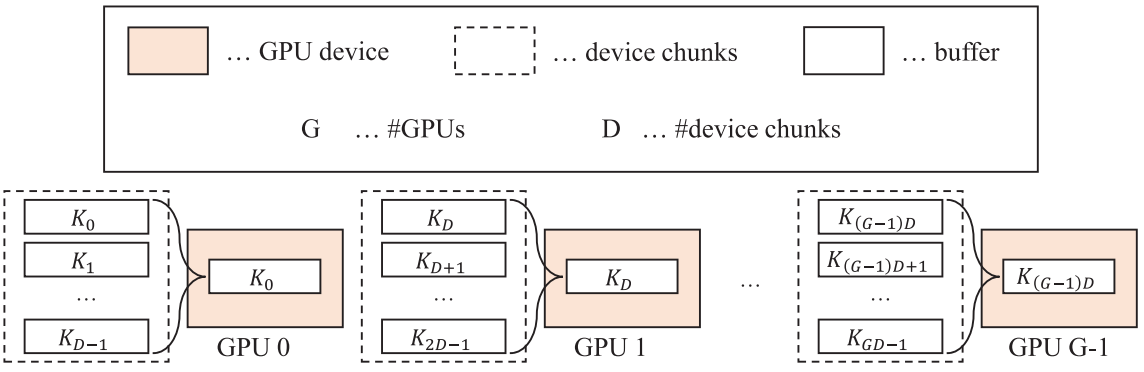


FIGURE 7 Multi-GPU K_{ij} data distribution

is relatively small in comparison to the mutual visibility matrix evaluated in each timestep, and as such the multi-GPU synchronization overhead only has a limited impact on overall speedup.

5.3 | Streaming for raytracing

Implementing streaming for the raytracing phase of M-RTX-RSim is generally significantly simpler than doing the same for the radiosity propagation phase. The reasons for this are the lack of synchronization or data dependencies, there is no need to compute multiple timesteps, and all elements of K_{ij} can in principle be computed entirely in parallel. The only dependency of the entire computation is on the previous generation of the required acceleration structures, but since the execution time for this step is negligible—in the range of milliseconds even for large scale scenes—it can easily be replicated across multiple GPUs.

In theory, it would be possible to also distribute the computation of K_{ij} among multiple GPUs, analogous to Section 5.2.3. However, we currently leave this as future work, as it requires significant implementation effort, and the majority of execution time in large scenes is spent in the radiosity simulation phase.

5.3.1 | Implementation

Since the raytracing Shader 1 calculates the mutual visibility matrix K_{ij}, and the radiosity simulation computation uses this matrix, the same memory limitations apply for both. The shader code modifications needed to implement streaming are generally trivial. An additional entry in the *uniform buffer object* that stores the offset for the current shader dispatch is all that is needed.

On the host side some additional modifications are necessary. For our non-streaming implementation we require only a single dispatch call over the linearized elements of the K_{ij} matrix as described in Section 3.2.3. The streaming version splits this into multiple chunks, setting the proper offset before each call. The chunk size is limited by the available GPU memory and determined automatically (analogous to Section 5.2.1).

Latency hiding optimization

Because there is no dependency between calculating chunks of K_{ij}, we implemented the same latency hiding optimization described in Section 5.2.2.1. The buffers buf_A and buf_B shown in Table 4 are also used to facilitate double buffering when computing K_{ij}.

Data layout conversion

To avoid unnecessarily evaluating the visibility twice for each triangle pair the raytracing Shader 1 only computes the upper triangle matrix of K_{ij}. For the reasons given in Section 5.2.1 the host must store the entire N^2 representation of K_{ij} for efficient multi-GPU streaming. The conversion between the linearized chunk that is computed on the GPU and the full 2D matrix is done during the copy operation from GPU to host memory.

6 | PERFORMANCE EVALUATION

6.1 | Testing setup

We evaluated our implementation on five differently sized geometry samples, described in Table 5. The three *Room* sample sizes are artificial scenes chosen to provide good coverage of common problem sizes, ranging from small to large, with increasing geometric complexity at each step. The small and medium room samples exhibit manageable memory requirements, whereas the other examples do not fit into the device memory of a single GPU and therefore require our asynchronous streaming implementation described in Section 5. *Traffic* and *Industry* represent two real-world use cases, featuring a traffic and warehouse scene respectively, and they are shown in Figure 8.

For consistency and comparability we chose a fixed number of timesteps (1000) for all samples, since this is large enough for a valid room response result, while not resulting in unnecessary computations. The number of stochastic samples S for visibility computation was chosen to

TABLE 5 Samples used for simulation

	Small room	Medium room	Large room	Traffic	Industry
# triangles	4963	11,509	34,801	39,665	52,885

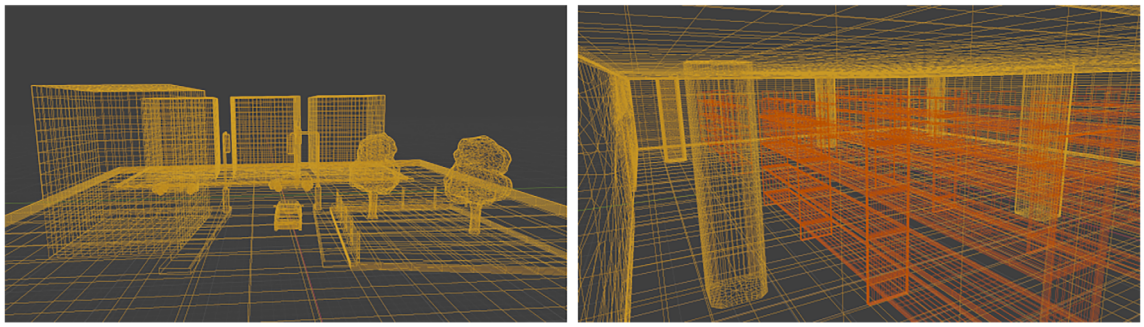


FIGURE 8 Test scene geometry examples. Left: *Traffic*; right: *Industry*

TABLE 6 Hardware configuration of testing system

Component	Model	Memory	PCIe
CPU	Ryzen TR 2920X	32 GiB	–
GPU 0	GeForce RTX 2070	8 GiB	8
GPU 1	GeForce RTX 2070	8 GiB	16
GPU 2	GeForce RTX 2070	8 GiB	8
GPU 3	GeForce RTX 2070	8 GiB	16

TABLE 7 Testing system software

Component	Name	Version
OS	Ubuntu LTS	18.04.1
Compiler	gcc	8.3.0
Vulkan SDK	LunarG SDK	1.1.126.0
GPU Driver	NVIDIA Driver	440.33.01

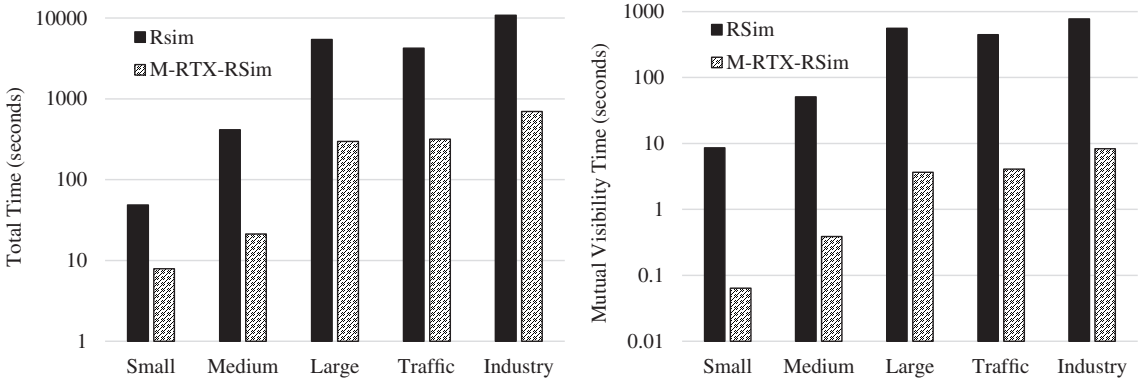
be 16 for every scene—the default value for the existing CPU RSim implementation, which delivers good results in practice—and we performed five repetitions of each measurement for statistical evaluation. Note that we do not report individual standard deviation values in our subsequent analysis, which is based on the median result across all repetitions, since none of our measurements feature a deviation of more than 1.1% across runs.

The hardware setup for the benchmarks presented in this section is described in Table 6 and consists of a mid-range workstation CPU and one to four consumer GPUs in a Linux server configuration. The CPU RSim implementation runs in parallel on all 24 hardware threads of the 2920X. Note that two of the GPUs are connected by 8 PCIe 3.0 lanes each, while the other two GPUs can leverage the full 16 lanes each. Our software stack is described in Table 7 and is based on Ubuntu LTS. All benchmarks were performed on the same machine, with no other load present on the system.

6.2 | Single-GPU M-RTX-RSim performance versus CPU RSim

Before analyzing individual properties of M-RTX-RSim, such as the effectiveness of our streaming and compression mechanisms and the efficiency of multi-GPU scaling, we want to provide an overview of the performance of our hardware accelerated Vulkan implementation compared to the state-of-the-art multi-core CPU implementation of the algorithm (RSim).

The CPU and GPU implementations are algorithmically identical and make use of the same—or analogous—data structures as outlined in Table 2. The only significant exception is the K_{ij} computation, which is different by necessity, because the GPU version utilizes NVIDIA RTX hardware raytracing, whereas the CPU version is based on a manually implemented raytracing algorithm using an octree-based acceleration structure. The CPU and GPU raytracing algorithms do not handle all edge-cases identically, however, we have verified that this does not affect the final results in any



(A) Comparison of total simulation time. **(B)** Comparison of mutual visibility phase.

FIGURE 9 Comparison between multi-core CPU (RSim) and single GPU (M-RTX-RSim) performance

meaningful way. Additionally, given the same K_{ij} matrix, the CPU and GPU implementations compute the exact same result, apart from minor floating point rounding differences. Figure 9A illustrates the total simulation time of both implementations. Note that the *Y* axis is log-scaled to provide a more meaningful comparison.

Three important observations can be made based on this data. First, the M-RTX-RSim implementation provides a very significant speedup for the overall simulation compared to the multi-core CPU implementation across the board, between a factor 6 for the small sample size and a factor of 20 at medium size. Second, at *Small* size, the GPU implementation is not as effective as at larger sizes. This is primarily due to two factors: overhead induced by GPU initialization and warmup for very short per-timestep kernels, and the parallelization strategy chosen for the radiosity simulation, which limits parallelism for small problem sizes. Finally, the *Traffic* scene takes slightly longer than *Large room* on the GPU—in line with the relative triangle count—while it is actually faster on the CPU despite consisting of slightly more triangles. This is a result of the CPU acceleration structure implementation being particularly effective for the *Traffic*scene.

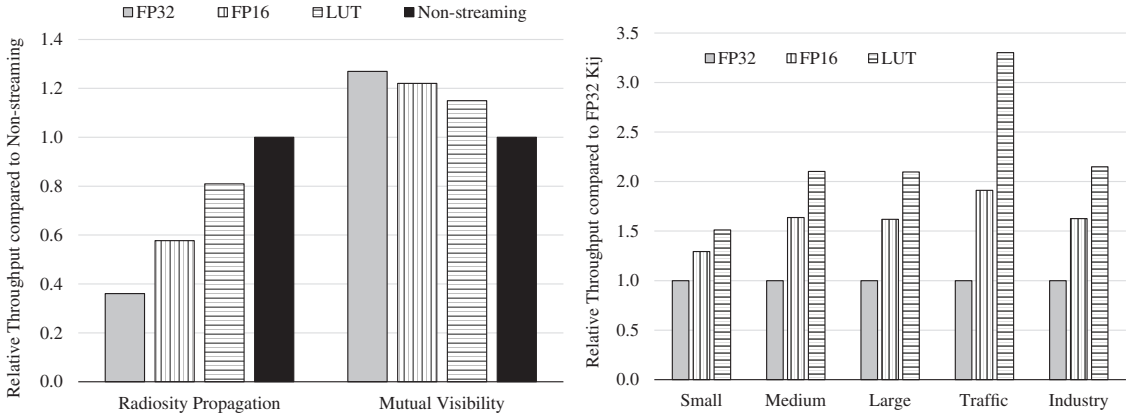
One of the most unique aspects of our work is leveraging hardware raytracing acceleration, and as such studying the mutual visibility computation phase in detail is of particular interest. Figure 9B shows a comparison of this individual phase. With hardware-accelerated raytracing in a nearly-ideal setting—with a very small per-ray payload and simple first-hit shader semantics—we measure a speedup of more than factor 100 compared to the state-of-the-art parallel CPU implementation. Note that this measurement includes memory transfer and setup times. The speedup is slightly lower with very small and very large problem sizes, since the former do not allow the GPU to fully utilize its parallelism, while the latter benefit from the much larger per-thread cache on the CPU. In this context it is interesting to note that for the best-performing sample tested (*Traffic*), we reach a measured 3.44 GRays/s, while the absolute theoretical maximum for this metric provided by NVIDIA in marketing materials for the RTX 2070 GPU is 6 GRays/s.[†] Though our results do not quite reach this theoretical limit, they are closer to it than the performance independently reported in examinations of RTX for traditional ray-traced rendering workloads,[17] in which a maximum of 1.1 GRays/s is reached in the best case.

6.3 | Compression and streaming performance impact

In order to evaluate the impact of our asynchronous streaming implementation on simulation performance, we need to compare it to performance without streaming, which is not possible at the large scene sizes which actually necessitate a streaming approach. The default implementation decides whether or not to perform streaming based on the available GPU memory and scene size. To acquire the data in this section, we changed this implementation to force streaming regardless of data size, and performed all measurements with this modified version.

Figure 10A illustrates the impact of utilizing streaming during mutual visibility computation and radiosity propagation. The number reported is the throughput relative to the standard version which does not use streaming for this problem size, that is, 1.0 would mean that streaming has no performance impact, while 0.5 would indicate that the execution time with streaming is doubled. Analyzing these results, we first note that the impact of streaming for the raytracing phase is relatively low, while there is some significant slowdown in the simulation phase. The reason for this lies in the amount of work required per transferred data element. During the mutual visibility evaluation phase, for every value in K_{ij} transferred over the PCI express bus (i.e., 1–4 bytes), *S* rays need to be cast into the scene, in addition to some computation. This takes more than enough time to fully mask the transfer overhead. One important point to note about this result is that it confirms that our asynchronous implementation of streaming

[†]NVIDIA RTX 2070 product page—https://www.nvidia.com/en-us/geforce/graphics-cards/rtx-2070/



(A) Relative throughput with streaming compared to no streaming, across different compression (*Medium room* sample).

(B) Relative throughput, with streaming, of different K_{ij} compression schemes across test scenes.

FIGURE 10 Evaluation of streaming and compression effectiveness

TABLE 8 Overhead of compressing K_{ij} data using LUT relative to total execution time

	Small room	Medium room	Large room	Traffic	Industry
Compression overhead	5.86%	5.10%	3.34%	0.73%	0.63%

is very effective at overlaying communication with computation, as forcing streaming actually increases performance by up to 27% as the buffer transfer overhead is partially masked by computation.

During the radiosity simulation phase, the amount of computation per transferred data volume is significantly smaller, resulting in a larger negative impact on performance. In fact, with full FP32 precision, throughput drops to only 36% of the computation speed achievable without streaming, even though we have confirmed that we are using the available hardware PCIe bandwidth near-optimally. This is the main incentive for our investigation of various compression schemes, which improve matters considerably: using FP16 storage, we achieve 58% of the non-streaming performance, and with our LUT-based compression the performance loss when streaming is necessary is only 19%.

In Figure 10B, we evaluate the effectiveness of all three K_{ij} storage schemes across all test scenes. The *Large room* and *Industry* scenes behave very similarly to *Medium room*. *Small room* and *Traffic* are more noteworthy: in the former case, the total data volume is smaller and thus this case benefits less from reduced storage size. *Traffic* is a more interesting case, where the gains are significantly larger. Here, the spatial structure of the scene results in only a small subset of triangles actually transmitting radiosity during the simulation timeframe, which means that the early exit condition in Shader 2 is hit more frequently, making the overall computation time even more dependent on K_{ij} streaming performance.

Table 8 shows the overhead associated with LUT-based compression using the "log" compression scheme. Since this happens only once and has $O(N)$ complexity, and can be parallelized over chunks of K_{ij}, the overhead decreases as the total execution time increases. But even for the smaller input scenes, where the overhead is more pronounced, the performance gain due to reduced memory transfers outweighs this overhead as shown in Figure 10B. Also note that FP16 has no overhead associated with it, as casting from FP32 to FP16 is effectively free and the conversion occurs in-band during the generation of K_{ij}.

6.4 | Multi-GPU performance scaling

Figure 11 illustrates the multi-GPU scaling for each scene on our testing system, depicting throughput relative to the single-GPU result. Note that, in these experiments, the GPUs connected by PCIe 16x are used before those connected by PCIe 8x. In practice, the GPUs are added in the order #1, #3, #0, #2.

Of all the results, *Medium room* follows the expected behavior most closely, with a parallel speedup of 1.9, 2.8, and 3.6 with 2, 3, and 4 GPUs respectively. This scene fits entirely in the memory of a single GPU and needs no streaming, leading to near-linear scaling at low GPU counts, dropping off slightly with 4 GPUs due to limited overall size and thus parallelism. *Small room* also behaves as expected, with worse scaling due to the very small total computation size and consequently limited parallelism.

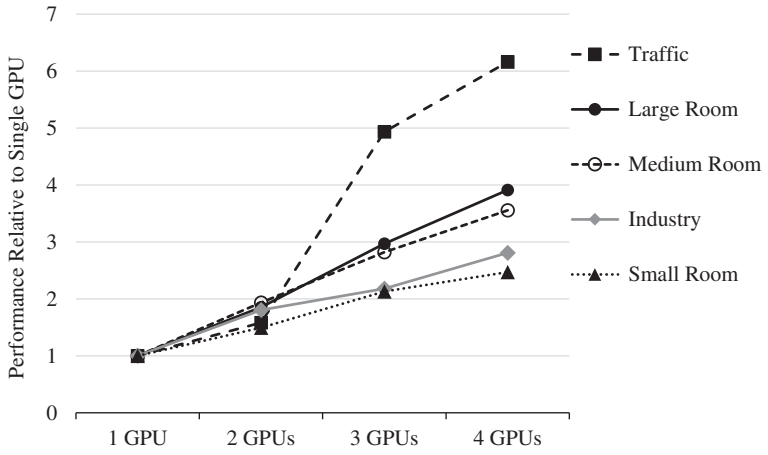


FIGURE 11 Performance scaling across multiple GPUs, using LUT K_{ij} compression

TABLE 9 Computation and synchronization overhead for exchanging radiosity values between GPUs (in seconds)

	Small room	Medium room	Large room	Traffic	Industry
Simulation	6.7	23.6	123.1	102.4	254.1
Radiosity exchange	1.4	3.7	14.4	10.9	19.7
Percentage	21%	16%	12%	11%	8%

The *Industry* scene is sufficiently large to always require streaming. It scales well to 2 GPUs, but then drops off sharply in efficiency. This is due to the third and fourth GPUs which are added to the simulation (GPUs #0 and #2, respectively, in Table 6) being limited by their PCIe 8x connection, as described in Section 6.1. *Medium room* and *Traffic* both show an increase in multi-GPU efficiency with 3 GPUs compared to 2, with *Traffic* even scaling superlinearly. This unexpected increase in efficiency is due to the previously explained behavior of M-RTX-RSim only choosing to stream K_{ij} when required: for these two scenes, the entirety of the mutual visibility matrix fits on 3 and 4 GPUs, but not 1 or 2. The effect of not requiring streaming is more pronounced in the *Traffic* scene due to its lesser computational load, as discussed in Section 6.3.

Table 9 lists the total and relative time required for radiosity exchange in the multi-GPU execution scenario. These values represent the full, *effective* overhead in the 4 GPUs case, determined by pre-computing and pre-loading the radiosity buffer and running the full simulation without any radiosity exchange. As expected, the relative time requirement is much more significant for the smaller compute scenarios, while it drops to 12% and below for the larger and more realistic real-world test cases. For these cases, especially the largest *Industry* case, the impact of K_{ij} streaming bandwidth limitations on multi-GPU scaling in our test system is clearly far more pronounced than the impact of radiosity exchange—following the expected behavior given that the latter scales with N while the former scales with N^2.

Overall, the multi-GPU efficiency of the simulation is high at the relatively low GPU counts which are viable in a single node, and the synchronization overhead is generally masked by meaningful computation in all but the *Small Room* sample. For networked large-scale simulations, the overhead of the per-timestep synchronization and aggregation would likely become more relevant even for larger scenes.

6.5 | Vulkan compute versus CUDA performance

As Vulkan compute is still far less widely used in HPC applications compared to NVIDIA CUDA, but offers the potential advantage of cross-vendor compatibility, one additional contribution of this work is investigating its performance vis-a-vis CUDA in a real-world simulation application. For this purpose, we implemented the radiosity propagation algorithm in CUDA and measured its performance.

Figure 12 shows the total execution time for 1000 timesteps of radiosity simulation in the *Medium room*test case. Three versions are shown: *Vulkan*, the implementation discussed in the majority of this article; *CUDA*, which is a direct port of the Vulkan implementation to CUDA, with the same kernel code barring minor API differences, and the same block sizes; and *CUDA-tune* which is the result of some manual tuning of the CUDA version. Thirty measurements of each version were performed on the system described in Table 6 and contribute to the box plot. The CUDA versions were compiled with `nvcc 11.3`.

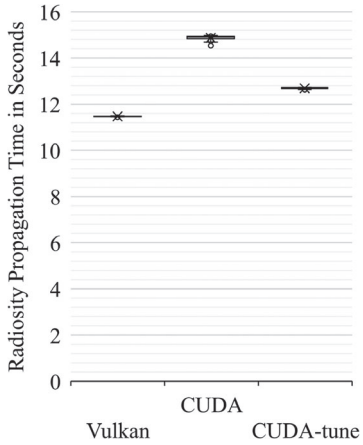


FIGURE 12 Radiosity propagation performance comparison between Vulkan and CUDA

As illustrated in the figure, the direct conversion of the algorithm to CUDA was 26% slower than the Vulkan baseline. This was a very surprising result which prompted us to investigate the causes for it in-depth, by studying the PTX output, NVIDIA Nsight Compute profile, and the exact code generation options and defaults for both CUDA and Vulkan compute. We have identified three contributing factors: (i) by default, the floating point rounding modes used differ in minor ways between the Vulkan compute shader and CUDA compilation pipelines; (ii) the CUDA toolchain makes different function inlining decisions, which result in one function call for computing τ_{ij} not being inlined by default, unlike the Vulkan version; and (iii) the underlying (closed-source) driver and runtime system make different block and group scheduling decisions. For the *CUDA-tune* version, we were able to mitigate points (i) and (ii) by using specific compiler options regarding floating point handling and pragmas to force inlining, however, we cannot directly affect point (iii). As such, we retuned our block sizes specifically for the CUDA version. With these changes, the performance difference drops below 10%, but we were not able to entirely eliminate it.

Both versions achieve a very similar GPU DRAM bandwidth utilization ratio of roughly 42%, and are not limited by global memory throughput. For smaller samples which fit into GPU global memory, performance is latency-bound—and could potentially be further optimized by software pipelining of the radiosity propagation loop—while for larger and more realistic scenes we are limited by PCIe streaming bandwidth.

Overall, the Vulkan compute performance is clearly very competitive in this particular application, while also easing cross-vendor portability. However, programmability suffers compared to CUDA, which offers a more succinct API and more mature tooling.

7 | RELATED WORK

To the best of our knowledge, there has not been any published research so far which employs hardware raytracing and multiple GPUs for room response simulation. However, our work touches on several active research areas, including the more general topic of room response simulation, the use of the Vulkan API and its compute shaders for scientific simulation, high-speed floating point compression suitable for GPUs, and the emerging field of leveraging raytracing hardware for non-rendering tasks.

7.1 | Room response simulation

As it has been an active research topic for several decades, we can only provide a brief overview of room response simulation here. The goal of our work is not to improve on the algorithmic state of the art in this field, but rather to demonstrate the adjustments required for and performance potential of a multi-GPU implementation of a well-established algorithm.

Room response simulation has multiple distinct use cases which also require different algorithmic approaches. One of the most studied fields is in audio, where several methods for sound propagation simulation are applied.[18,19] Some of these include ray-based models, and while the constraints and time scales are very different from ToF simulation it seems likely that some of our approach, and raytracing hardware in general, could be leveraged in this domain.

Another related field is the analysis of signal propagation in indoor channels.[20,21] Again, ray-tracing is a commonly used simulation model, but no hardware-accelerated implementations have been studied so far.

7.2 | Floating point compression for GPUs

Over the years, there has been significant interest in the study of both lossless and lossy compression of floating point data.[22] While some very high-throughput GPU implementations exist,[23] and some methods are even supported in hardware,[15] these generally depend on spatial similarity structure in the ND array or texture data they compress.

Other methods, such as the algorithm by Patel et al.[24] achieve good compression results even for data without obvious spatial similarity, but do not provide performance suitable for our use case. As such, to fit the structure of our data and fulfill the requirements of our use case, we implemented a relatively simple LUT-based fixed-rate lossy compression scheme.

7.3 | Vulkan compute

While GPU computing in general is now well-established in scientific computation and simulation,[25] the vast majority of implementations are based either on CUDA and other proprietary technologies, or OpenCL. Vulkan is a very promising alternative, with the potential for broader hardware support than OpenCL (e.g., on System-On-Chip platforms with limited driver support). Vulkan also provides a more predictable development platform due to standardized conformance testing. Among others, Vulkan implementations of fluid simulation[26] and an analysis of the general applicability of Vulkan for compute tasks[27] have been published. Our contribution demonstrates the applicability of Vulkan in a new domain and its use with multiple GPUs, while also providing evidence that asynchronous computation and PCIe streaming at near-optimal bandwidth is feasible.

7.4 | Hardware raytracing

Consumer raytracing hardware was introduced only relatively recently, and as a result the application of it to non-rendering computational problems is still rather rare. RTX hardware has been leveraged for Monte Carlo particle transport,[28] Tet-Mesh point location,[29] in positron emission tomography,[30] and in the simulation of 2D sound propagation in water.[31] All of these applications are very different from our use case, both in the scientific algorithms studied as well as in implementation. Crucially, they all leverage NVIDIA's OptiX[32] higher-level raytracing framework, and commonly interact with CUDA for general computation. Conversely, our work is based entirely on Vulkan for both raytracing and general compute workloads.

8 | CONCLUSION

M-RTX-RSim demonstrates that leveraging raytracing hardware for room response simulation is viable and effective, and that radiosity simulation can be efficiently scaled across multiple GPUs. For the raytracing-heavy mutual visibility computation phase of the algorithm, we measured a speedup of more than factor 100 on a single consumer GPU compared to a multithreaded CPU implementation. Furthermore, Vulkan compute shaders have proven well suited to implementing the radiosity simulation phase of the algorithm, reaching a speedup of factor 20 in a medium-sized test scene.

In order to overcome the physical memory size limitations of GPUs, we have explored various application-specific high-speed data compression schemes and implemented an asynchronous multi-GPU streaming method. We were able to fully overlap computation with streaming data transfers, and reached close to the maximum available bandwidth over the PCIe bus. The performance drawback of this streaming approach could be reduced from over 60% to less than 20% using our LUT-based compression method, while maintaining a result accuracy of more than 95%.

In terms of multi-GPU scaling, we have demonstrated near-linear throughput improvements for scenes which are sufficiently large to saturate the available parallelism, with the potential for superlinear scaling in cases where the combined device memory offered by multiple GPUs allows the simulation to proceed without the need for streaming. A potential avenue for further improvements, especially in scaling to large GPU clusters, is exploring methods to mitigate the per-timestep synchronization and aggregation overhead, for example by performing optimistic, per-chunk asynchronous computations combined with a rollback strategy. In regards to application-specific compression strategies for this use case, approaches which resort the scene triangles based on an analysis of the spatial structure of a scene could be explored. This could result in local clustering and similarities in the mutual visibility matrix which could be leveraged by either hardware-accelerated compression or run length encoding.

ACKNOWLEDGMENTS

This work was partially funded by the FFG (Austrian Research Promotion Agency) project INPACT, project number 868018.

DATA AVAILABILITY STATEMENT

The data that support the findings will be available in the MRTX-RSim repository on Github following an embargo from the date of publication to allow for commercialization of research findings.

ORCID

Peter Thoman https://orcid.org/0000-0002-4028-7451
Philipp Gschwandtner https://orcid.org/0000-0002-7774-0344

REFERENCES

1. Kolb A, Barth E, Koch R, Larsen R. Time-of-flight cameras in computer graphics. *Comput Graph Forum*. 2010;29(1):141-159.
2. Hranitzky R. *A Scalable multi-DSP System for Room Impulse Response Simulation*. PhD thesis. Technical University of Graz; 1997.
3. Glassner AS. *An Introduction to Ray Tracing*. Elsevier; 1989.
4. Purcell TJ, Buck I, Mark WR, Hanrahan P. Ray tracing on programmable graphics hardware. Proceedings of the ACM SIGGRAPH 2005 Courses; 2005:268; ACM.
5. NVIDIANVIDIA Turing GPU architecture whitepaper; 2018. https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf
6. Sellers G, Kessenich J. *Vulkan Programming Guide: The Official Guide to Learning Vulkan*. Addison-Wesley Professional . 2016.
7. Kessenich J, Ouriel B, Krisch R. *SPIR-V Specification*. Vol 3. Khronos Group; 2018.
8. Werness E, Lele A, Stepinski R. VK_NV_ray_tracing Vulkan 1.2 Specification. NVIDIA; 2018. https://www.khronos.org/registry/vulkan/specs/1.2-extensions/html/vkspec.html#VK_NV_ray_tracing
9. Thoman P, Wippler M, Hranitzky R, Fahringer T. RTX-RSim: accelerated vulkan room response simulation for time-of-flight imaging. In: McIntosh-Smith S, ed. *IWOCL'20*. ACM; 2020:1-11. https://dl.acm.org/doi/proceedings/10.1145/3388333
10. Karras T, Aila T. Fast parallel construction of high-quality bounding volume hierarchies. Proceedings of the 5th High-Performance Graphics Conference; 2013:89-99; ACM.
11. Glassner AS. Space subdivision for fast ray tracing. *IEEE Comput Graph Appl*. 1984;4(10):15-24.
12. Hapala M, Havran V. Kd-tree traversal algorithms for ray tracing. *Comput Graph Forum*. 2011;30(1):199-213.
13. Hirsch A, Thoman P. Running on Raygun; 2020. https://arxiv.org/abs/2001.09792.
14. Ho NM, Wong WF. Exploiting half precision arithmetic in Nvidia GPUs. Proceedings of the 2017 IEEE High Performance Extreme Computing Conference (HPEC); 2017:1-7; IEEE.
15. Nystad J, Lassen A, Pomianowski A, Ellis S, Olson T. Adaptive scalable texture compression. Proceedings of the 4th ACM SIGGRAPH/Eurographics Conference on High-Performance Graphics; 2012:105-114.
16. Hartigan JA, Wong MA. Algorithm AS 136: a k-means clustering algorithm. *J Royal Stat Soc Ser C (Appl Stat)*. 1979;28(1):100-108.
17. Sanzharov V, Gorbonosov A, Frolov V, Voloboy A. Examination of the Nvidia RTX. Technical report; 2019.
18. Vorländer M. Simulation of the transient and steady-state sound propagation in rooms using a new combined ray-tracing/image-source algorithm. *J Acoust Soc Am*. 1989;86(1):172-178.
19. Habets EA. Room impulse response generator. Technical report. Technische Universiteit Eindhoven; 2006, 2(2.4):1.
20. Lopez-Hernandez FJ, Perez-Jimenez R, Santamaria A. Ray-tracing algorithms for fast calculation of the channel impulse response on diffuse IR wireless indoor channels. *Opt Eng*. 2000;39(10):2775-2781.
21. Rodríguez Pérez S, Pérez Jiménez R, López Hernández FJ, González Hernández OB, Ayala Alfonso AJ. Reflection model for calculation of the impulse response on IR-wireless indoor channels using ray-tracing algorithm. *Microw Opt Technol Lett*. 2002;32(4):296-300.
22. Lu T, Liu Q, He X, et al. Understanding and modeling lossy compression schemes on HPC scientific data. Proceedings of the 2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS); 2018:348-357; IEEE.
23. O'Neil MA, Burtscher M. Floating-point data compression at 75 Gb/s on a GPU. Proceedings of the 4th Workshop on General Purpose Processing on Graphics Processing Units; 2011:1-7.
24. Patel RA, Zhang Y, Mak J, Davidson A, Owens JD. Parallel lossless data compression on the GPU; 2012:1-9; IEEE.
25. Navarro CA, Hitschfeld-Kahler N, Mateu L. A survey on parallel computing and its applications in data-parallel problems using GPU architectures. *Commun Comput Phys*. 2014;15(2):285-329.
26. Gunadi SI, Yugopuspito P. Real-time GPU-based SPH fluid simulation using vulkan and OpenGL compute shaders. Proceedings of the 2018 4th International Conference on Science and Technology (ICST); 2018:1-6.
27. Gkeka MR, Bellas N, Antonopoulos CD. Comparative performance analysis of vulkan implementations of computational applications; 2019:1.
28. Lewis Salmon J, McIntosh Smith S. Exploiting hardware-accelerated ray tracing for Monte Carlo particle transport with OpenMC; 2019.
29. Wald I, Usher W, Morrical N, Lediaev L, Pascucci V. RTX beyond ray tracing: exploring the use of hardware ray tracing cores for TET-mesh point location. *Proceedings of High Perform Graphics*. 2019;7–13.
30. Bialas P, Sagało P, Nowakowski K. Simulating the J-PET detector on NVidia ray tracing hardware. *Acta Phys Polon B*. 2020;51(1).
31. Ulmstedt M, Stålberg J. *GPU Accelerated Ray-tracing for Simulating Sound Propagation in Water*. Master's thesis. Linköping University; 2019.
32. Parker SG, Bigler J, Dietrich A, et al. OptiX: a general purpose ray tracing engine. *Acm Trans Graph*. 2010;29(4):1-13.

How to cite this article: Thoman P, Wippler M, Hranitzky R, Gschwandtner P, Fahringer T. Multi-GPU room response simulation with hardware raytracing. *Concurrency Computat Pract Exper*. 2022;34(4):e6663. doi: 10.1002/cpe.6663