Future Generation Computer Systems 175 (2026) 108109

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

Out-of-core aware multi-GPU rendering for large-scale scene visualization

Milan Jaros, Lubomir Riha, Petr Strakos *, Tomas Kozubek

IT4Innovations, VSB - Technical University of Ostrava, Ostrava, Czech Republic

ARTICLE INFO

Keywords:
Multi-GPU path tracing
NVLink
X-Bus
CUDA unified memory
Data distributed path tracing
Distributed shared memory path tracing
Out-of-core

ABSTRACT

We introduce an out-of-core method for multi-GPU path tracing of large-scale scenes, leveraging memory access analysis to optimize data distribution. Our approach enables efficient rendering on multi-GPU systems even when the combined GPU memory is smaller than the total scene size. By partitioning the scene between GPU and CPU memory at the memory management level, our method strategically distributes or replicates scene data across GPUs while storing the remaining data in CPU memory. This hybrid memory strategy ensures efficient access patterns and minimizes performance bottlenecks, facilitating high-quality rendering of massive scenes. The main contribution of this rendering approach is that it enables GPUs to perform accelerated path-tracing for massive scenes that would otherwise be rendered only by CPUs. Previous work on this topic has enabled the rendering of massive scenes that are distributed among memories of all GPUs on one server. However, in that method, it is not possible to render scenes larger than the total size of all GPU memories. In our paper, we show how to break this barrier and how to combine the capacity of GPU memories with CPU main memory to render massive scenes with only a minor impact on the performance. When applied to a small system with 4xGPUs, the results are equivalent to a more powerful system with 16xGPUs if using the same number of GPUs as on the smaller system. Therefore, users can use even small systems to render massive scenes.

1. Introduction

Although GPU memory capacity continues to grow annually, it remains insufficient to meet the high demands of the film industry. As a result, most production rendering still relies on CPUs due to their greater memory availability. To overcome this limitation, we introduce an out-of-core rendering approach that enables GPU-based rendering even when the total available GPU memory is insufficient. Our method leverages an advanced distribution strategy based on memory access patterns and statistical analysis. Previously, a memory management-level approach [1] replicated frequently accessed scene data on all GPUs to optimize performance. Our new approach enhances data distribution by organizing memory into chunks that can now be efficiently offloaded to system RAM, further improving scalability and rendering efficiency.

The primary contribution of this paper is a unified solution for distributed and out-of-core GPU rendering that efficiently adapts to various hardware architectures integrating CPUs and GPUs. Our approach is compatible with specific high-speed interconnects such as NVLink [2] and X-Bus [3], ensuring efficient data transfer between processing units. Using a unified memory management strategy, our method remains effective in different hardware configurations. It enables seamless rendering of both small scenes and massive datasets that exceed the combined memory capacity of all GPUs, making it a versatile and scalable solution for GPU-based rendering. The small scenes are straightforward since they can be fully duplicated in the memory of each GPU. The solution for massive scenes is to keep part of the scene data out-of-core in the main memory of the CPU. We demonstrate that even a standard bus, such as PCI-Express Gen 3 and 4, provides sufficient bandwidth for our proposed method. Although high-speed interconnects like NVLink offer advantages in tightly coupled GPU configurations, our approach remains effective even when relying on alternative data transfer mechanisms, such as the X-Bus interconnect. Another important technology that we utilize for the proposed solution is the CUDA Unified Memory [4], which we use to manage the data placement among the memories of GPUs as well as data placement in the system memory.

The organization of the paper is as follows. In Section 2, we describe work related to the topic of this contribution. In Section 3, the specification of the implemented method is presented in detail. Section 4 provides a performance evaluation of the method with a specification of the tested scenes and HW architectures that were used. Section 5 summarizes the results and brings the conclusion.

* Corresponding author.
E-mail address: petr.strakos@vsb.cz (P. Strakos).

https://doi.org/10.1016/j.future.2025.108109
Received 10 February 2025; Received in revised form 20 June 2025; Accepted 29 August 2025
Available online 1 September 2025

2. Related work

Cinema-quality rendering of computer-generated scenes relies on global illumination techniques such as path tracing to produce high-quality images. As image quality demands increase, so do the requirements for scene complexity, including larger geometric structures and high-resolution textures. This complexity significantly impacts memory usage, particularly for large-scale outdoor environments, which can quickly exceed the available memory of a single computing device. In production renderers [5–9], scene sizes often surpass 256 GB-far beyond the capacity of even the most powerful GPUs. This limitation makes it impractical to render such scenes on a single device or to use multiple GPUs while redundantly duplicating the entire scene across each device. To address the constraints of local memory, rendering techniques must employ one of the following strategies: (i) distributed rendering across multiple devices, (ii) out-of-core rendering that utilizes additional system memory, or (iii) a hybrid approach combining both methods.

2.1. Distributed rendering

In distributed rendering, methods can be categorized into two fundamental approaches based on when the transformation from scene object coordinates to image coordinates occurs during the rendering process. This classification, originally introduced by Molnar et al. [10], distinguishes between sort-first and sort-last techniques, each determining how rendering workloads are distributed across multiple computing units.

In sort-first distributed rendering, the transformation to image coordinates occurs at the beginning of the rendering process. The image is divided into distinct regions, with each rendering device assigned a specific portion of the scene data based on a predefined or adaptive partitioning strategy. This approach aims to balance workload distribution while minimizing redundant data processing across devices. During the rendering process, portions of the scene are accessed and moved according to whether they are required to render a particular part of the image. Appropriate preprocessing must ensure that the scene data is distributed so that the maximum amount of data required to render a part of the image on a particular device is held locally and the remaining data is distributed to other rendering devices. One extreme case, the simplest one, describes the situation where all scene data can be duplicated to every device, and no data has to be placed remotely. This approach is commonly used for small scenes and represents an embarrassingly parallel approach. For large-scale scenes, efficient data distribution and rapid access to remote data are crucial factors in achieving high-performance rendering. Several approaches have been proposed to handle on-demand data movement in large scene rendering, as explored in [11–14]. These solutions aim to optimize data transfers, reduce latency, and ensure efficient memory utilization across distributed systems.

In sort-last distributed rendering, pixel composition occurs at the final stage of the rendering pipeline. Unlike sort-first methods, the scene data is initially distributed across all devices and remains fixed throughout the rendering process. When necessary, individual rays in the path tracing algorithm are exchanged between devices to ensure correct shading and visibility. This approach has been widely adopted in production rendering [15,16] and large-scale scientific visualization [17], where efficiently handling vast datasets is crucial for performance and scalability.

Hybrid approaches that combine elements of sort-first and sort-last rendering have also been explored. For instance, dynamic ray scheduling, as proposed in [18,19], adapts rendering workloads based on real-time data access patterns. This method has been effectively applied to distributed memory systems, including supercomputers, to optimize rendering performance by dynamically balancing computational load and memory usage across multiple nodes.

2.2. Out-of-core rendering

In out-of-core rendering, scene data is typically loaded dynamically as needed [5,20]. This approach is particularly beneficial for GPU rendering, where memory constraints often prevent storing the entire scene in GPU memory. Instead, less frequently accessed portions of the scene are offloaded to system RAM, significantly reducing GPU memory requirements. However, to maintain performance, an efficient data management system must be implemented to ensure low-latency data transfers between RAM and the GPU, allowing seamless retrieval of out-of-core data when required for rendering.

Several approaches have been developed to address out-of-core rendering in both single and multi-GPU environments. Budge et al. [21] introduced a path tracing method for heterogeneous architectures composed of multiple CPU and GPU nodes, incorporating out-of-core data management. A similar strategy was later refined by Son et al. [22].

For single-GPU rendering, Garanzha et al. [23] proposed an effective out-of-core solution, while Wang et al. [24] extended this approach to support both geometry and lights. Harada et al. [25] developed a framework designed to facilitate the conversion of in-core GPU path tracing implementations into out-of-core implementations, utilizing a software virtual memory system to manage page requests. However, their method considers out-of-core memory to be read-only.

In hybrid rendering approaches, Mattausch et al. [26] introduced a ray-tracing technique that integrates seamlessly with a streaming rasterization pipeline, supporting both dynamic and out-of-core scenes. Kim et al. [27] developed an interactive global illumination technique using heterogeneous computing resources (CPU and GPU), reducing costly CPU-GPU data transfers by implementing separate, decoupled data representations for each processor.

For large-scale ray tracing, Hunter et al. [28] presented a parallel architecture for efficiently distributing geometry and constructing acceleration structures, enabling the rendering of massive models within an out-of-core memory framework. Their method was applied to radiative transfer modeling (infrared imaging).

Several works focus specifically on multi-GPU out-of-core rendering. Zhou et al. [29] developed a system for handling large textures across multiple GPUs, while Pantaleoni et al. [30] introduced an out-of-core multi-GPU ray tracing algorithm. Eilemann et al. [31] proposed a parallel rendering framework that supports both multi-GPU and out-of-core techniques using a hierarchical data structure. However, their method relies on sequential traversal, resulting in significant memory overhead.

More recently, Dong et al. [32] presented a promising multi-GPU out-of-core rendering technique that optimizes data transfer between the CPU and GPUs. However, their approach is tailored specifically for rasterization-based rendering, limiting its applicability to ray-tracing workloads.

2.3. Distributed shared memory systems

To contextualize our proposed solution within large-scale rendering, we employ a distributed GPU rendering approach that utilizes the sort-first method to distribute workloads efficiently. Additionally, we extend the total available memory for scene data by leveraging system RAM, following an out-of-core strategy. A key component of our method is the use of NVIDIA Unified Memory (UM) [4,33], a hardware/software technology that creates a single unified memory address space across all devices in the system, including CPUs, GPUs, or hybrid architectures.

The concept of Unified Memory builds on principles of Distributed Shared Memory (DSM) systems, which were widely explored in the 1990s when they first became commercially available [34]. DSM systems provide a logical shared-memory abstraction over physically distributed memory, allowing applications to function as if they were operating in a traditional shared-memory environment [35]. However, such systems are inherently affected by Non-Uniform Memory Access

(NUMA), where data is distributed across multiple devices with varying access speeds, leading to higher latency for remote data access.

To mitigate NUMA-related performance bottlenecks, data locality plays a critical role in DSM-based systems. Prior research has explored data replication and migration techniques to optimize memory access patterns and improve performance [36]. These principles have also been carefully considered in the design of our solution to ensure efficient data distribution, minimal memory access overhead, and improved overall rendering performance.

Our approach further extends the work in [37] by Jaros et al., which introduced an out-of-core method for multi-GPU path tracing using memory access analysis to efficiently distribute scene data across GPU and CPU memory. Their approach enables the rendering of large scenes on multi-GPU systems by partitioning scene data at the memory management level, either replicating or distributing data among GPUs and offloading the remaining portion to CPU memory. Here, we perform a more in-depth analysis of the out-of-core method and confirm its general applicability by testing it on another multi-GPU system as well as focusing on a suitable solution that can tackle not only static frames but also animations.

3. Multi-GPU path tracing with out-of-core data distribution

In this section, we build upon the method proposed by Jaros et al. [1], introducing enhancements that reduce the negative impact of remote memory accesses at the algorithmic level while maximizing performance and scalability in multi-GPU path tracing. Our approach optimizes data distribution to efficiently utilize all available memory resources, including both GPU memory and system RAM, enabling the rendering of massive scenes that exceed the total GPU memory capacity. By strategically managing memory access patterns and minimizing costly data transfers, our method significantly improves load balancing and computational efficiency across multiple GPUs.

Our method has been implemented and evaluated within Blender Cycles [38], a production-grade path tracing renderer chosen for its open-source availability and adaptability to High-Performance Computing (HPC) clusters. However, the choice of renderer is not a prerequisite for our approach. As our method operates at the memory management level, it remains independent of any specific rendering engine, making it highly versatile and applicable to a wide range of GPU-based rendering frameworks. This universality allows seamless integration into various rendering pipelines without requiring fundamental modifications to the renderer itself.

3.1. Out-of-core algorithm based on memory access pattern

CUDA Unified Memory [39] supports several modes for memory management and the placement of data structures in GPU memory, system memory, or a combination of both. The *cudaMallocManaged* function provides memory allocation within the unified memory model, while the recommended function for specific memory placement is *cudaMemAdvise*.

In our approach, we utilize *cudaMemAdvise* to optimize data distribution:

- *cudaMemAdviseSetReadMostly* is applied for data replication across multiple GPUs.
- *cudaMemAdviseSetPreferredLocation* with a specified GPU device ensures targeted data distribution.
- *cudaMemAdviseSetPreferredLocation* with *CU_DEVICE_CPU* places data in main system memory when GPU memory is insufficient.

To further enhance performance, we introduce memory access counters, which track the frequency of data accesses per device during rendering. These counters allow for dynamic adjustments in data placement, ensuring that frequently accessed data remains closer to the GPU that requires it. This strategy helps to minimize costly remote memory accesses and improve overall rendering efficiency.

Blender Cycles incorporates multiple CUDA kernels for rendering, which we extended with software memory access counters to analyze memory usage patterns. However, these counters introduce additional computational overhead, reducing overall performance. To mitigate this, the modified kernels with memory access counters are used only during the initial statistics-gathering phase. Once data distribution between GPUs and main memory is optimized based on the collected access patterns, the rendering process resumes using the original, unmodified kernels, ensuring maximum efficiency without the performance penalty of continuous memory tracking.

The workflow of the out-of-core algorithm is as follows (see Fig. 1):

1. Allocate system memory using unified memory functions.
2. Load the scene from disk and store the entire scene in main memory.
3. Execute the path-tracing kernel with memory access counters to collect memory access statistics.
4. Determine the optimal placement for each chunk using Algorithm 1.
5. Redistribute the data structures across all GPUs and retain the remaining data in RAM based on the collected memory access statistics.
6. Launch the original path-tracing kernel with the redistributed data for rendering.

Algorithm 1 processes a three-dimensional array of memory access counters, denoted as $\texttt{AccessCounters}[G][S][C]$, where:

- G represents the set of GPU indices,
- S is the set of all data structures,
- C denotes the set of chunks within each data structure s.

The algorithm takes two additional input parameters: the scene size D_{scene} and the replication ratio R_r. The output of the algorithm provides an efficient heuristic placement for each chunk. A chunk can be assigned to one of the following memory locations:

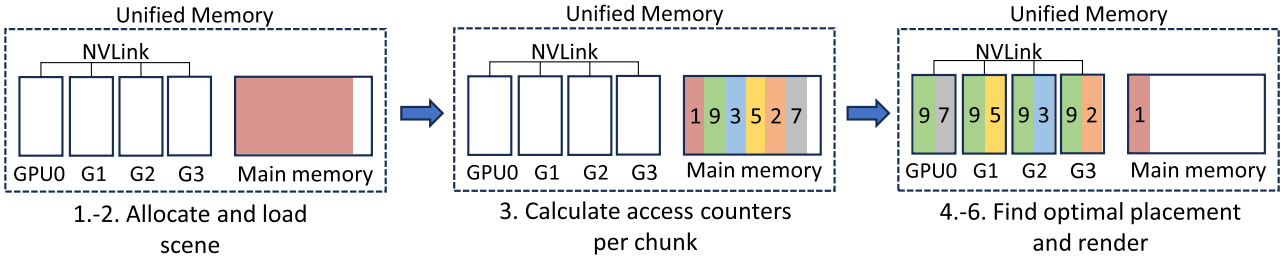

Fig. 1. The workflow of the out-of-core algorithm. The process begins with scene allocation and loading into main memory. Next, a specialized kernel tracks memory accesses during path tracing using memory access counters, collecting statistics on data usage (e.g., a value of 7 indicates the number of accesses for the last chunk). Based on these statistics, the optimal data placement is determined: frequently accessed chunks (e.g., chunk 9, the most accessed, is replicated across all GPUs), while less frequently accessed chunks (e.g., chunk 1, which lacks space on the GPUs) remain in main memory. Finally, the original path tracing kernel is executed using the redistributed scene data to ensure efficient rendering.

Algorithm 1: Out-of-core algorithm.

Input : Access counters `AccessCounters`$[G][S][C]$, where G is a set of the GPU indices, S is a set of all data structures, C is a set of the chunks per data structure s; D_{scene} is a scene size; R_r is a replication ratio.
Output: Chunk distribution $c_{place} \in$ {*"Replicated", "Distributed", "CPU"*} with optimal placement

$i = 0$
foreach $c \in C, s \in S$ **do**
  $a_{sum} = \sum_{g \in G}$ `AccessCounters`$[g][s][c]$ For chunk c in data structure s, sum accesses to it by all GPUs and save it to a_{sum}.
  $\mathtt{H}_{comb}[i{+}{+}] = (a_{sum}, s, c)$ Put all per chunk counter values a_{sum} to one array so that it can be sorted by number of accesses.
end
$\mathtt{H}_{>} =$ ***sort***$(\mathtt{H}_{comb})$ **by** a_{sum} Descending sort by a_{sum}.
$m_t = R_r * D_{scene}$ Find replication memory threshold m_t from the replication ratio R_r and scene size D_{scene}.

$c_{sum} = 0$ The size of all processed chunks.
foreach $(a_{sum}, s, c) \in H_{>}$ Process all chunks from the highest number of accesses to the lowest one.
do
  $G_f = \{g_f |$ GPU g has a free memory $m_f > c_{size}\}$ Find GPUs that have free memory m_f for chunk c with size of c_{size}.
  if $c_{sum} < m_t$ ***and*** $|G| == |G_f|$ ***and*** $a_{sum} > 0$ **then**
    c_{place} = *"Replicated"* When the memory threshold m_t has not been exceeded and memories of all GPUs have a space for chunk c, the chunk is replicated.
  else if $a_{sum} > 0$ ***and*** $|G_f| > 0$ **then**
    $g_d = \max_{g \in G_f}$ `AccessCounters`$[g][s][c]$ Find a GPU g_d that has the highest number of accesses to chunk c in s and has free memory.
    c_{place} = *"Distributed"* The chunk is marked as distributed.
  else if $a_{sum} == 0$ ***and*** $|G_f| > 0$ **then**
    $g_d = g_{rr} \in G_f$ Chunks with zero accesses are distributed in a round robin fashion among GPUs.
    c_{place} = *"Distributed"*
  else
    c_{place} = *"CPU"*. When memories of all GPUs are full, place the chunk into the CPU memory.

  $c_{sum} = c_{sum} + c_{size}$ Add the current chunk size c_{size} to the sum of the processed chunk sizes c_{sum}.

  if $c_{place} ==$ *"Replicated"* **then**
    set chunk c of data structure s as replicated
  if $c_{place} ==$ *"Distributed"* **then**
    set chunk c of data structure s as distributed and set its preferred location to GPU g_d
  if $c_{place} ==$ *"CPU"* **then**
    set chunk c of data structure s as out-of-core and set its preferred location to CPU
end

- *Replicated* - The chunk is copied to all GPU memories, allowing fast access across multiple devices.
- *Distributed* - The chunk is placed in the memory of a specific GPU with an assigned ID g_d, meaning it is stored on a single GPU to optimize memory usage.
- *CPU* - The chunk remains in system RAM, typically for less frequently accessed data or when GPU memory is insufficient.

In the first part of Algorithm 1, the GPU access counters are summed to compute the total number of accesses a_{sum} for each chunk $c \in C$ within each data structure $s \in S$. These results are represented as tuples (a_{sum}, s, c), where:

- a_{sum} denotes the total number of accesses to a specific chunk,
- s represents the data structure to which the chunk belongs,
- c is the chunk index within the data structure.

All tuples (a_{sum}, s, c) are then inserted into a single 1D array $\mathtt{H}_{comb}$. This array is subsequently sorted in descending order based on a_{sum}, prioritizing chunks with the highest number of memory accesses. The sorted array is stored in $\mathtt{H}_{>}$.

In the second part of Algorithm 1, we define the replication memory threshold m_t based on the scene size D_{scene} and the replication ratio R_r. The placement of each chunk is determined as follows:

- If the total size of all processed chunks, denoted as c_{sum}, is less than the replication memory threshold m_t, the corresponding chunk is set as *Replicated*, meaning it is copied to all GPU memories.
- If c_{sum} exceeds the memory threshold m_t, the chunk is set as *Distributed*, and it is assigned to the GPU g_d with the highest number of accesses to that chunk.
- If the access counter value $a_{sum} = 0$ (i.e., the chunk is not accessed by any GPU), it is distributed in a round-robin fashion among available GPUs to balance memory usage.
- If no GPU has sufficient memory to store a given chunk, the chunk is placed in system RAM to ensure all scene data remains accessible for rendering.

4. Performance

In this section, an evaluation of the performance of the proposed method is conducted on four systems with different architectures, utilizing NVIDIA V100 and A100 GPUs.

Table 1
The key parameters of GPU systems used for the tests.

	DGX-2	Karolina
# of GPUs	16× V100	8× A100
GPU memory	16× 32 GB	8× 40 GB
CPU	Intel8168[a]	AMD[b]
System memory	1.5 TB	1 TB
	Barbora	M100
# of GPUs	4× V100	4× V100
GPU memory	4× 16 GB	4× 16 GB
CPU	Intel6126[c]	IBM[d]
System memory	192 GB	256 GB

[a] Intel® Xeon® 8168
[b] AMD EPYC™ 7763
[c] Intel® Xeon® 6126
[d] IBM® POWER9 AC922

Table 2
The measured bandwidths of GPU systems used for the tests.

Bandwidth	DGX-2	Karolina
Local CPU memory	76 GB/s	92 GB/s
Local GPU memory	790 GB/s	1193 GB/s
GPU to GPU	145 GB/s	255 GB/s
CPU to GPU	12 GB/s	24 GB/s
	Barbora	M100
Local CPU memory	82 GB/s	108 GB/s
Local GPU memory	740 GB/s	715 GB/s
GPU to GPU	48 GB/s	70/33 GB/s
CPU to GPU	12 GB/s	68 GB/s

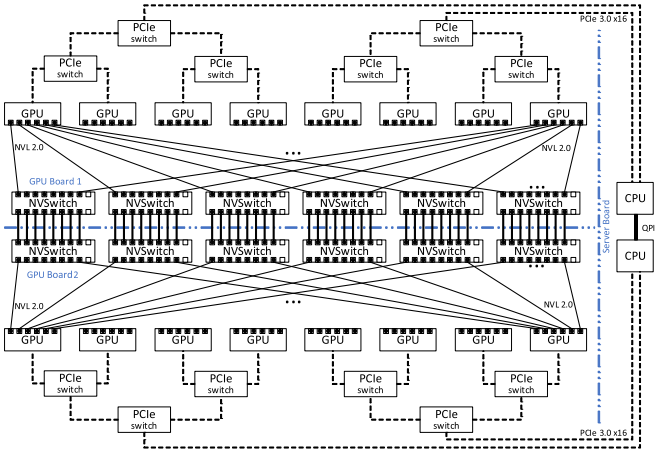

Fig. 2. NVLink GPU interconnect with Intel QPI in the DGX-2 system.

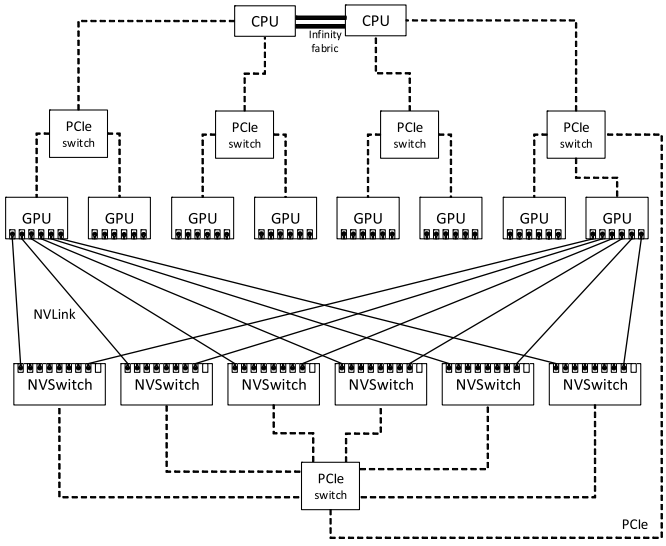

Fig. 3. NVLink GPU interconnect with Infinity fabric in the Karolina GPU server.

4.1. Multi-GPU systems

In this paper, we use four platforms with multiple GPUs (see Table 1): DGX-2 System, Karolina GPU Server, Barbora GPU Server, and M100 GPU Server.

4.1.1. DGX-2 system

The NVIDIA DGX-2 [40] is a high-performance computing server designed for processing massive scenes with up to 2 TB of unified memory (1.5 TB of CPU RAM and 512 GB of GPU memory). It features 16 Tesla V100 GPUs, each equipped with 32 GB of memory, and utilizes NVLink connectivity for high-speed GPU communication.

A key advantage of the DGX-2 is its NVSwitch interconnect architecture [41], which enables high-bandwidth connectivity between all 16 GPUs (see Fig. 2). The NVSwitch-based link facilitates a one-way bandwidth of 150 GB/s and a bidirectional bandwidth of 300 GB/s between any two GPUs. This is achieved by leveraging all six NVLink 2.0 connections per GPU. Each GPU maintains a total measured bandwidth of 145 GB/s across all peer GPUs. Additionally, the local memory bandwidth reaches 790 GB/s, which is 5.4× higher than the remote memory bandwidth (see Table 2).

For comparative performance evaluations, we conducted tests using both the full 32 GB memory capacity per GPU and a restricted 16 GB per GPU configuration. The unrestricted measurements are labeled *DGX-2*, while the memory-limited tests are denoted as *DGX-2 (16 GB)*.

4.1.2. Karolina GPU server

Karolina GPU Server is equipped with DGX-A100 nodes, which represent the next generation of NVIDIA's DGX-2 system and are designed for high-performance computing workloads. Each DGX-A100 node features 8 Tesla A100-SXM4 GPUs. Every Tesla A100 includes 12 NVLink 3.0 ports, each providing a theoretical bidirectional bandwidth of 25 GB/s. The DGX-A100 incorporates 6 NVSwitches, with each GPU connected to all NVSwitches via two NVLinks (see Fig. 3). This architecture enables the A100 GPU to achieve 300 GB/s unidirectional and 600 GB/s bidirectional peer-to-peer throughput over NVLink.

Benchmark results using STREAM [42] show that the effective remote memory bandwidth between any pair of GPUs is approximately 255 GB/s. The local memory bandwidth is approximately 1193 GB/s.

4.1.3. Barbora GPU server

The BullSequana X410-E5 NVLink-V [43] is a blade server equipped with four Tesla V100 GPUs, each featuring 16 GB of HBM2 memory. This system is part of the Barbora HPC cluster, located at the IT4Innovations national supercomputing center [44], and is referred to as *Barbora* in our study.

The Barbora server features two Intel Xeon Gold 6126 processors and 192 GB of RAM, with all components interconnected via PCI-Express 3.0. A key component of this architecture is the NVLink 2.0 interconnect, which enhances GPU-to-GPU communication (see Fig. 4). Each Tesla V100 GPU includes six NVLink 2.0 ports, each providing a theoretical bandwidth of 25 GB/s per direction. As a result, each GPU is connected to three other GPUs, with a total measured bandwidth of 48 GB/s across all connections.

In terms of memory access efficiency, the local memory bandwidth reaches 740 GB/s, which is 15.4× higher than the remote memory bandwidth (see Table 2). The Barbora server provides up to 256 GB of unified

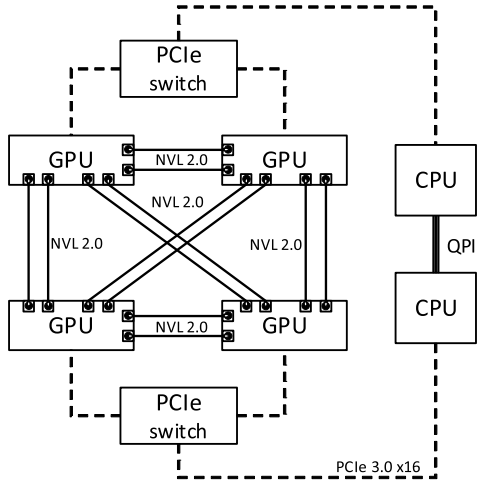

Fig. 4. NVLink GPU interconnect with Intel QPI in the Barbora GPU server.

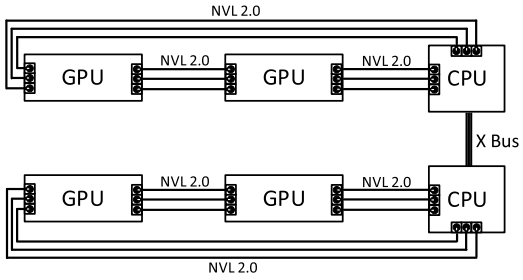

Fig. 5. NVLink GPU interconnect with POWER9 and X-Bus interconnect in the M100 GPU server.

memory (64 GB GPU memory and 192 GB RAM), supporting efficient memory management for large-scale GPU workloads.

4.1.4. *M100 GPU server*

The IBM POWER9 [3] platform, referred to as *M100*, is part of the Marconi 100 (M100) cluster at Cineca, Italy's largest supercomputing center [45]. It features four NVIDIA Tesla V100 GPUs, each with 16 GB of memory, and provides a total of 320 GB of unified memory (64 GB GPU memory and 256 GB RAM).

Unlike the Barbora GPU server, which uses a fully interconnected NVLink 2.0 topology, the M100 server connects its four GPUs in pairs via NVLink, with inter-pair communication handled by the X-Bus interconnect (see Fig. 5). This architecture results in a measured bandwidth of 70 GB/s between directly connected GPUs. However, communication between non-neighboring GPUs via X-Bus is 33 GB/s, which is 2.1× lower than direct NVLink bandwidth and 1.5× lower than the Barbora server's inter-GPU communication speed (see Table 2).

Despite these interconnect limitations, M100's high-capacity unified memory and heterogeneous architecture provide an efficient platform for large-scale GPU workloads.

4.2. *Benchmark scenes*

We selected a diverse set of interior and exterior scenes with varying geometry-to-texture size ratios to evaluate our approach (see Fig. 6). The benchmark includes:

- Moana Island Scene [46] (*Moana 169 GB*)
- Natural History Museum [47–49] (*Museum 124 GB*)
- Agent 327: Operation Barbershop [50] (*Agent 167 GB*)
- Spring [51] (*Spring 137 GB*)
- Galaxy (*Galaxy 137 GB*) – an example of a Milky Way-type galaxy, as described in Wissing et al. [52].

Table 3
Scene parameters used for scalability testing and performance evaluation.

Scene	*Moana 169 GB*	*Museum 124 GB*
Memory used for path-tracing	169 GB	124 GB
Image resolution	5120 × 2560	5120 × 2560
Geometry size	90 GB	69 GB
Triangles count	673 M	610 M
Total textures size	58 GB	46 GB
Number of textures	3421	22
Scene	***Agent 167 GB***	***Spring 137 GB***
Memory used for path-tracing	167 GB	137 GB
Image resolution	5120 × 2560	5120 × 2560
Geometry size	57 GB	48 GB
Triangles count	468 M	395 M
Total textures size	101 GB	74 GB
Number of textures	118	96
Scene	***Galaxy 137 GB***	
Memory used for path-tracing	137 GB	
Image resolution	5120 × 2560	
Geometry size	0	
Triangles count	0	
Total textures size	137 GB	
Number of textures	1	

Detailed parameters of these scenes are provided in Table 3. The scene sizes range from 124 GB to 169 GB, exceeding the memory capacity of a single GPU on any of the evaluated systems.

On both the *Barbora* and *M100* systems, the total available GPU memory is insufficient to store any of these benchmark scenes entirely. As a result, an out-of-core rendering approach is required to efficiently offload excess scene data to system memory, ensuring successful rendering of these large-scale environments.

4.3. *Preprocessing steps evaluation*

By preprocessing steps, we specifically mean evaluation of steps 3-5 from the main algorithm workflow (see Section 3). Assessment of these steps includes:

- 1spp pre-pass (CPU): Time required to render the scene with 1 sample per pixel to gather initial statistics.
- Algorithm runtime: Time taken by Algorithm 1 to compute the final scheduling or distribution strategy based on the gathered statistics.
- Chunk redistribution: Time required to redistribute the scene chunks across GPUs.

The cost of this preprocessing is shown in Table 4, which presents the breakdown of preprocessing times for five large-scale scenes (*Moana 169 GB*, *Museum 124 GB*, *Agent 167 GB*, *Spring 137 GB*, and *Galaxy 137 GB*) on two systems for reference: DGX-2 (16 GPUs) and Karolina (8 GPUs).

The DGX-2 system uses a chunk size of 64 MB with a replication ratio of 10 %, while the Karolina system uses a chunk size of 2 MB with a replication ratio of 10 %.

4.4. *Scalability evaluation*

The scalability of the proposed approach was evaluated on all four servers (DGX-2, Karolina, Barbora, and M100).

The scalability results are shown in Figs. 7, 8, 9, 10, and 11. For each of the evaluated systems, we found a pattern for chunk sizes and replication ratios (see Table 5). This pattern provides the best rendering times for the selected number of GPUs.

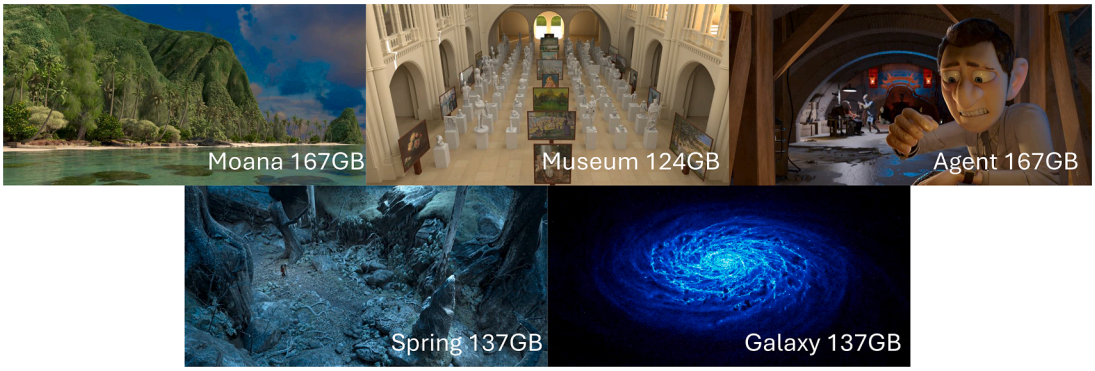

Fig. 6. Rendered examples of benchmark scenes using our modified version of the Blender Cycles path tracer.

Table 4
The preprocessing was performed on two systems with the following configurations: DGX-2 system with 16 GPUs, a chunk size of 64 MB and a replication ratio of 10 %; Karolina GPU server, using 8 GPUs, a chunk size of 2 MB, and a replication ratio of 10 %.

Scene	Moana 169 GB	Museum 124 GB	Agent 167 GB
Preprocessing on DGX-2			
1spp pre-pass (CPU)	10.1 s	7.7 s	9.3 s
Algorithm runtime	0.18 s	0.01 s	0.02 s
Chunk redistribution	67.1 s	28.6 s	51.9 s
Preprocessing on Karolina			
1spp pre-pass (CPU)	6.7 s	2.5 s	4.5 s
Algorithm runtime	0.18 s	0.01 s	0.02 s
Chunk redistribution	31.4 s	25.2 s	28.4 s
	Spring 137 GB	Galaxy 137 GB	
Preprocessing on DGX-2			
1spp pre-pass (CPU)	28.9 s	9.2 s	
Algorithm runtime	0.02 s	0.02 s	
Chunk redistribution	32.9 s	31.8 s	
Preprocessing on Karolina			
1spp pre-pass (CPU)	9.7 s	5.8 s	
Algorithm runtime	0.02 s	0.02 s	
Chunk redistribution	25.3 s	23.4 s	

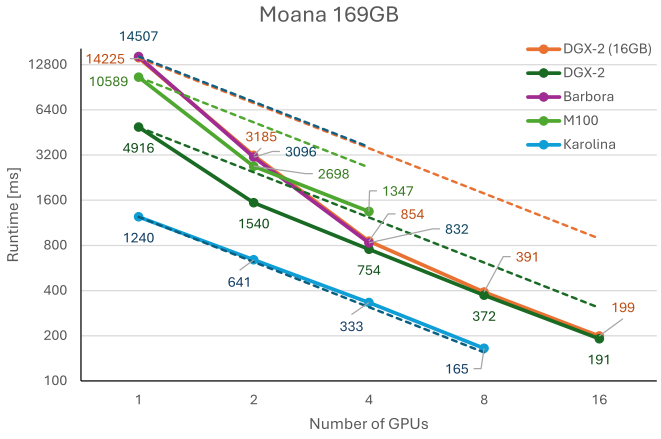

Fig. 7. Scalability of the Blender Cycles path tracer on the DGX-2 system, Barbora GPU server, M100 server and Karolina GPU server for *Moana 169 GB* scene. The rendering time is for 1 sample and the resolution is 5120×2560. Linear scalability for each server is presented as dashed lines.

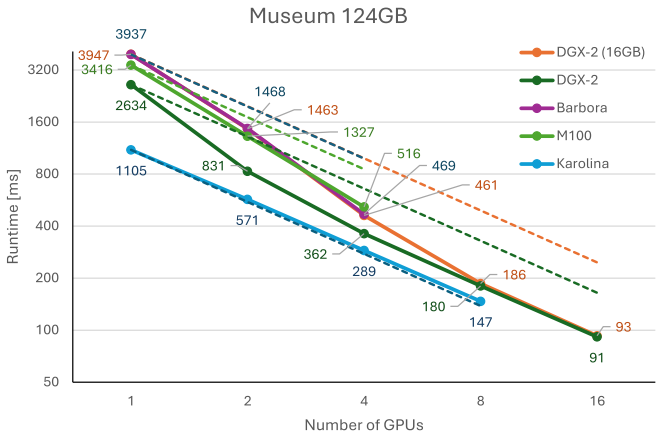

Fig. 8. Scalability of the Blender Cycles path tracer on the DGX-2 system, Barbora GPU server, M100 server and Karolina GPU server for *Museum 124 GB* scene. The rendering time is for 1 sample and the resolution is 5120×2560. Linear scalability for each server is presented as dashed lines.

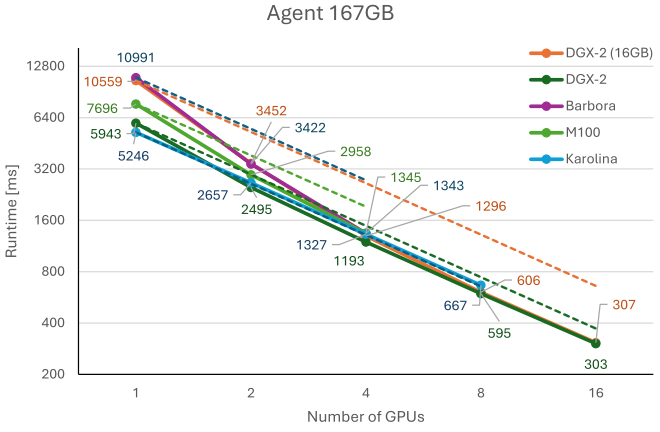

Fig. 9. Scalability of the Blender Cycles path tracer on the DGX-2 system, Barbora GPU server, M100 server and Karolina GPU server for *Agent 167 GB* scene. The rendering time is for 1 sample and the resolution is 5120×2560. Linear scalability for each server is presented as dashed lines.

Scalability is linear on the Karolina system, which has the fastest NVLink of the tested systems, while other systems exhibit superlinear behavior. In superlinear cases, adding more GPUs not only increases computing power but also expands the available GPU memory, which is a critical resource. This reduces data transfer between the CPU and GPU and increases communication between GPUs. Since GPU

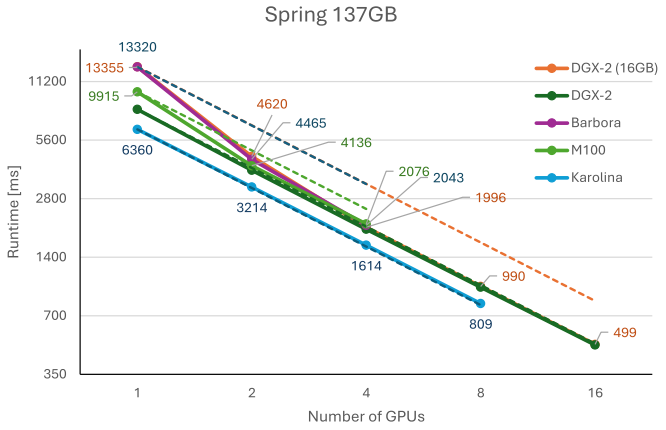

Fig. 10. Scalability of the Blender Cycles path tracer on the DGX-2 system, Barbora GPU server, M100 server and Karolina GPU server for *Spring 137 GB* scene. The rendering time is for 1 sample and the resolution is 5120×2560. Linear scalability for each server is presented as dashed lines.

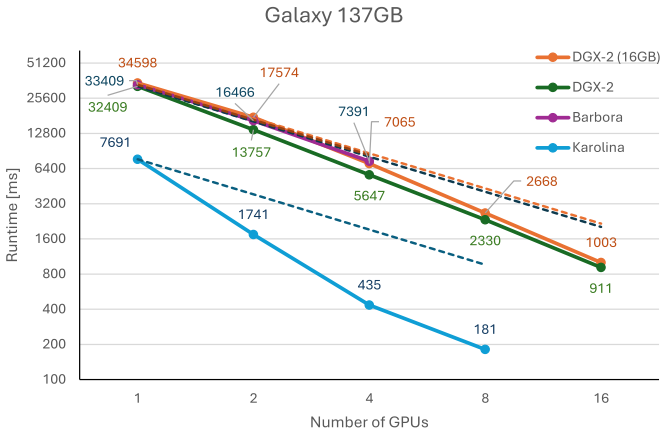

Fig. 11. Scalability of the Blender Cycles path tracer on the DGX-2 system, Barbora GPU server, and Karolina GPU server for *Galaxy 137 GB* scene. The rendering time is for 1 sample and the resolution is 5120×2560. Linear scalability for each server is presented as dashed lines.

interconnects offer significantly higher bandwidth, this shift leads to increased performance.

The pattern in Table 5 shows that Barbora and DGX-2 (16 GB) have the same settings (chunk size, replication ratio). In Figs. 7, 8, 9, 10, and 11, we can see that rendering times and even the scalability were the same for both platforms as well. On the Karolina system, a chunk size of 2 MB is used in all cases, as the A100 GPU features a newer architecture optimized for unified memory. This proves the repeatability of the proposed concept on similar architectures.

In Table 6, we present the speedup evaluation for each platform based on the number of GPUs utilized. The baseline for comparison is the runtime of *DGX-2 (16 GB)* using a single GPU for each scene separately. The baseline runtimes for the benchmark scenes are as follows:

- *Moana 169 GB*: 14 225 ms
- *Museum 124 GB*: 3947 ms
- *Agent 167 GB*: 10 559 ms
- *Spring 137 GB*: 13 355 ms
- *Galaxy 137 GB*: 34 598 ms

These values serve as reference points for assessing the scalability and performance improvements across different platforms and GPU configurations.

Table 5
Chunk size in MB and replication ratio used for the scalability test for all scenes. It is set up for scalability test of the Blender Cycles path tracer on DGX-2, Karolina, Barbora, and M100.

#GPUs	DGX-2 (16 GB)	DGX-2	Karolina	Barbora	M100
Chunk size [MB]					
1	2	2	2	2	2
2	2	32	2	2	2
4	32	32	2	32	16
8	32	32	2	–	–
16	32	64	–	–	–
Replication ratio					
1	0 %	0 %	0 %	0 %	0 %
2	0.1 %	2 %	5 %	0.1 %	0.1 %
4	0.5 %	5 %	7 %	0.5 %	2 %
8	2 %	7 %	10 %	–	–
16	5 %	10 %	–	–	–

Table 6
Speedup for various GPU configurations in the scalability test. The baseline for speedup calculation is the runtime of *DGX-2 (16 GB)* with a single GPU.

#GPUs	DGX-2 (16 GB)	DGX-2	Karolina	Barbora	M100
Moana 169 GB					
1	1.0	2.9	11.5	1.0	1.3
2	4.5	9.2	22.2	4.6	5.3
4	16.7	18.9	42.7	17.1	10.6
8	36.4	38.2	86.2	–	–
16	71.5	74.4	–	–	–
Museum 124 GB					
1	1.0	1.5	3.6	1.0	1.2
2	2.7	4.8	6.9	2.7	3.0
4	8.6	10.9	13.7	8.4	7.6
8	21.2	21.9	26.9	–	–
16	42.5	43.1	-	–	–
Agent 167 GB					
1	1.0	1.8	2.0	1.0	1.4
2	3.1	4.2	4.0	3.1	3.6
4	8.1	8.9	8.0	7.9	7.8
8	17.4	17.7	15.8	–	–
16	34.4	34.8	–	–	–
Spring 137 GB					
1	1.0	1.7	2.1	1.0	1.3
2	2.9	3.4	4.2	3.0	3.2
4	6.7	6.8	8.3	6.5	6.4
8	13.5	13.6	16.5	–	–
16	26.8	27.0	–	–	–
Galaxy 137 GB					
1	1.0	1.1	4.5	1.0	–
2	2.0	2.5	19.9	2.1	–
4	4.9	6.1	79.6	4.7	–
8	13.0	14.8	191.2	–	–
16	34.5	36.4	–	–	–

By further analyzing the scalability results across different architectures (Table 6), we observe the following:

- The Karolina system consistently demonstrates excellent scalability in all tested scenarios. This behavior is primarily attributed to two key architectural advantages: high-bandwidth NVLink interconnects between A100 GPUs and larger GPU memory (40 GB per GPU). These factors enable efficient data sharing between GPUs and reduce host memory dependency, significantly increasing performance. For scenarios such as *Galaxy 137 GB* and *Moana 169 GB*, Karolina exhibits

near-perfect or even superlinear scaling (e.g., speedup 191× on 8 GPUs for Galaxy against *DGX-2 (16 GB)* with one GPU), indicating that not only is computational scalability good, but also data locality and GPU memory availability reduce the overhead of data transfers.

- The *Barbora* server and *DGX-2 (16 GB)* exhibit significantly lower performance compared to the full *DGX-2* when using only one or two GPUs. This is primarily due to their limited GPU memory capacity (16 GB per GPU), whereas *DGX-2* provides 32 GB per GPU. Additionally, both systems suffer from a narrow 12 GB/s bandwidth between the GPU and CPU, which becomes a bottleneck as most scene data remains in system memory during rendering.
- If we compare the speedup on a single GPU on the DGX-2 system and the M100 server, we can see that the effect of larger memory has a greater impact than the higher bandwidth (12 vs 68 GB/s) between CPU and GPU. The described behaviour is observed also if the second GPU is used for rendering.
- Interesting change appears between M100 and Barbora architecture when 4 GPUs are brought to play. We can observe that in that case, Barbora outperforms M100. This is mainly caused by the fact that GPUs at M100 are forced to communicate over the X-Bus when accessing the second pair of GPUs (see Fig. 5 in the Section 4.1.4) in contrast to NVLink interconnect topology on the Barbora server that connects all GPUs directly (see Fig. 4 in the Section 4.1.3).

4.5. Performance evaluation

In this section, we analyze the impact of the replication factor on rendering performance and examine performance trends under varying replication ratios. Additionally, we compare the performance of our proposed algorithm with the approach introduced in [1], highlighting key differences in data distribution and efficiency.

All results are measured for rendering at 1 sample per pixel with a resolution of 5120×2560. The corresponding performance results are presented in Figs. 12, 13, 14, 15, and 16. As discussed in the scalability evaluation (Section 4.4), performance is significantly influenced by the choice of chunk size. Table 7 provides an overview of the optimal chunk size for different replication ratios, demonstrating its impact on rendering efficiency.

The following conclusions can be made from the results shown in Figs. 12, 13, 14, 15, and 16:

- The *DGX-2 (16 GB)* system with 16 GPUs gives from 2 % to 5 % of the replication ratio the optimal performance for all evaluated scenes.

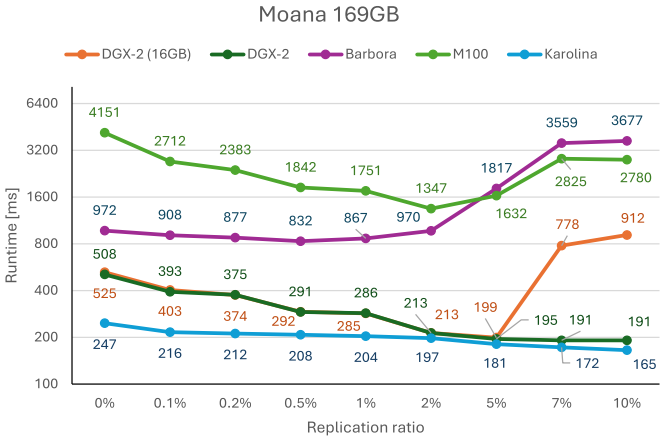

Fig. 12. Comparison of path tracing times for *Moana 169 GB* scene running on 16 GPUs of the DGX-2 system, on 4 GPUs of the Barbora GPU node, on 4 GPUs of the M100 node, and on 8 GPUs of the Karolina GPU node (*Note.* different colour represents a different system with specific chunk size from Table 7).

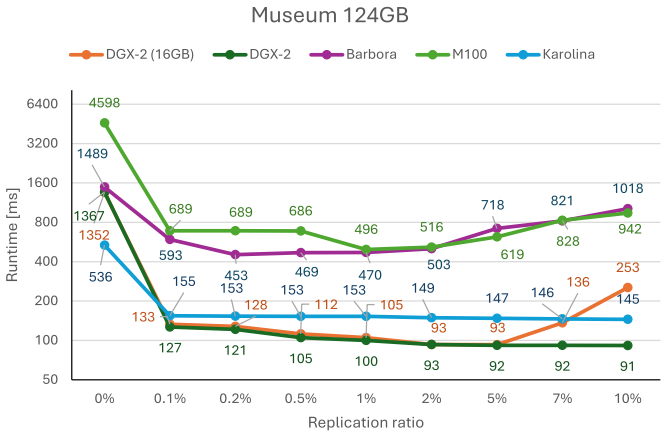

Fig. 13. Comparison of path tracing times for *Museum 124 GB* scene running on 16 GPUs of the DGX-2 system, on 4 GPUs of the Barbora GPU node, on 4 GPUs of the M100 node, and on 8 GPUs of the Karolina GPU node (*Note.* different colour represents a different system with specific chunk size from Table 7).

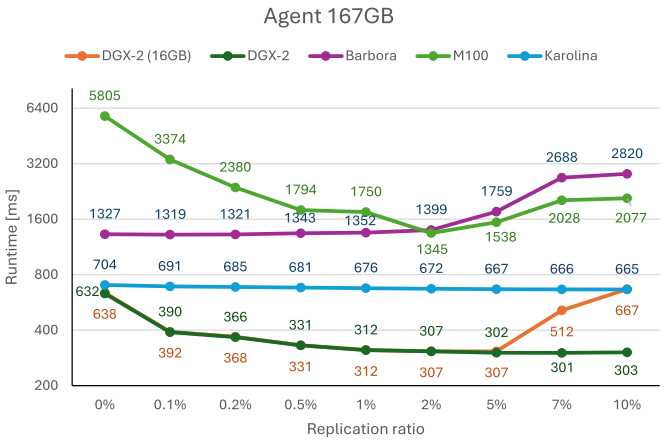

Fig. 14. Comparison of path tracing times for *Agent 167 GB* scene running on 16 GPUs of the DGX-2 system, on 4 GPUs of the Barbora GPU node, on 4 GPUs of the M100 node, and on 8 GPUs of the Karolina GPU node (*Note.* different colour represents a different system with specific chunk size from Table 7).

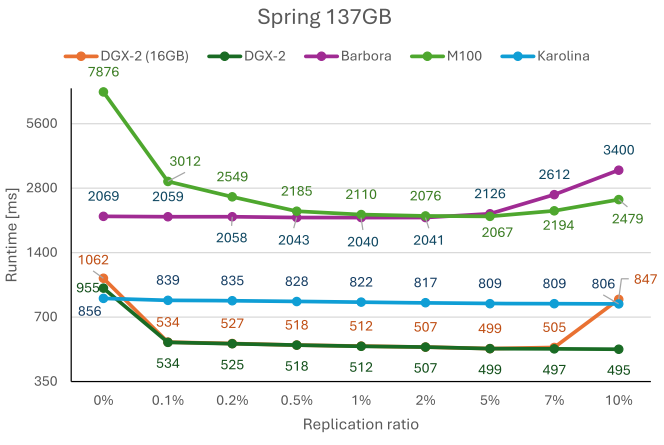

Fig. 15. Comparison of path tracing times for *Spring 137 GB* scene running on 16 GPUs of the DGX-2 system, on 4 GPUs of the Barbora GPU node, on 4 GPUs of the M100 node, and on 8 GPUs of the Karolina GPU node (*Note.* different colour represents a different system with specific chunk size from Table 7).

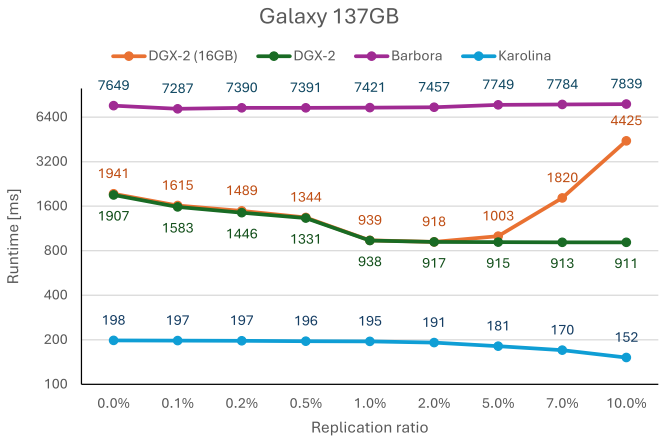

Fig. 16. Comparison of path tracing times for *Galaxy 137 GB* scene running on 16 GPUs of the DGX-2 system, on 4 GPUs of the Barbora GPU node, and on 8 GPUs of the Karolina GPU node (*Note.* different colour represents a different system with specific chunk size from Table 7).

Table 7
Chunk size in MB for all scenes for performance evaluation. It is set up for the comparison of path tracing times running on 16 GPUs of DGX-2, on 8 GPUs Karolina, on 4 GPUs of Barbora and on 4 GPUs of M100.

Replication ratio	DGX-2 (16 GB)	DGX-2	Karolina	Barbora	M100
0 %	2	2	2	32	2
0.1 %	4	4	2	32	2
0.2 %	8	8	2	32	2
0.5 %	16	16	2	32	4
1 %	32	32	2	32	16
2 %	32	32	2	32	16
5 %	32	32	2	2	16
7 %	2	64	2	2	2
10 %	2	64	2	2	2
#GPUs	16	16	8	4	4

- The *DGX-2* system with 16 GPUs gives from 2 % to 10 % of the replication ratio the optimal performance for all evaluated scenes.
- The *Karolina* server with 8 GPUs gives from 0.1 % to 10 % of the replication ratio the optimal performance for all evaluated scenes.
- The *Barbora* server with 4 GPUs gives from 0.1 % to 1 % of the replication ratio the optimal performance for all evaluated scenes.
- The *M100* server with 4 GPUs and 2 % of the replication ratio gives the optimal performance for all evaluated scenes. The rendering time on the M100 server after a certain replication value is close to the rendering time on the Barbora server.

Fig. 13 highlights a unique characteristic of the *Museum 124 GB* scene. Unlike other test cases, most of its triangles are densely concentrated in the statues at the center of the room. This spatial distribution benefits significantly from our approach, as the BVH tree and geometry of each statue are efficiently stored in the memory of the specific GPU responsible for rendering it. Given that the BVH tree is one of the most frequently accessed structures during rendering, this targeted data placement minimizes memory access overhead and enhances overall performance.

The rendering time is strongly dependent on where the chunks are placed. Figs. 17, 18, 19, 20 show the proportion of placement of individual chunks by replication ratio. In the case of 0 % replication (full distribution) on *Barbora* and *M100*, 72 % of the *Moana 169 GB* scene is stored in RAM, while in the case of *Museum 124 GB* it is 61 % of the scene, in the case of *Agent 167 GB* it is 71 % of the scene and in the case of *Spring 137 GB*, it is 65 % of the scene in RAM. On the DGX-2 system

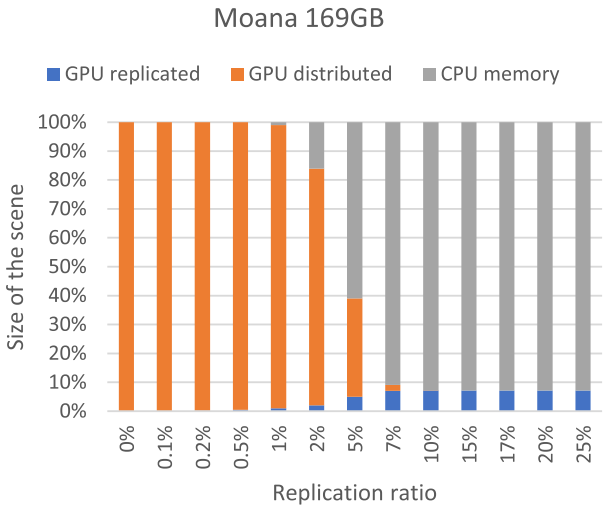

Fig. 17. The proportion of placement of individual chunks by replication ratio for DGX-2 (16 GB) with 16 GPUs. The chunk size was set to 32 MB.

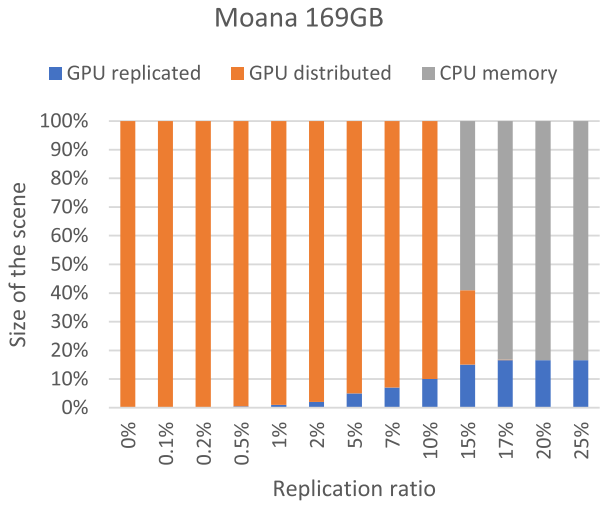

Fig. 18. The proportion of placement of individual chunks by replication ratio for DGX-2 with 16 GPUs. The chunk size was set to 32 MB.

and on the Karolina GPU server with the fully distributed scene, all data is stored only in the memory of the GPUs.

The distribution of chunk placements based on the replication ratio provides insight into when rendering performance slows down (see Figs. 12, 13, 14, 15, and 16) and when significant changes in chunk size occur (Table 7).

For the *Moana 169 GB* scene on *DGX-2 (16 GB)*, a performance slowdown occurs at a replication ratio of 7 %. This is due to excessive data replication and insufficient data distribution, leading to frequent memory accesses from system RAM, which significantly impacts performance.

On the full *DGX-2* system, the slowdown occurs later, at 17 % replication ratio, as the larger GPU memory helps accommodate more data before excessive RAM access becomes a bottleneck.

For both *Barbora* and *M100*, the *Moana 169 GB* scene experiences a slowdown at 5 % replication ratio, indicating that their lower overall GPU memory capacity reaches its limit more quickly, forcing an increased reliance on system RAM.

To further assess performance, we compared each platform against the *DGX-2 (16 GB)* system. Table 8 presents the relative performance per single GPU, revealing the following insights:

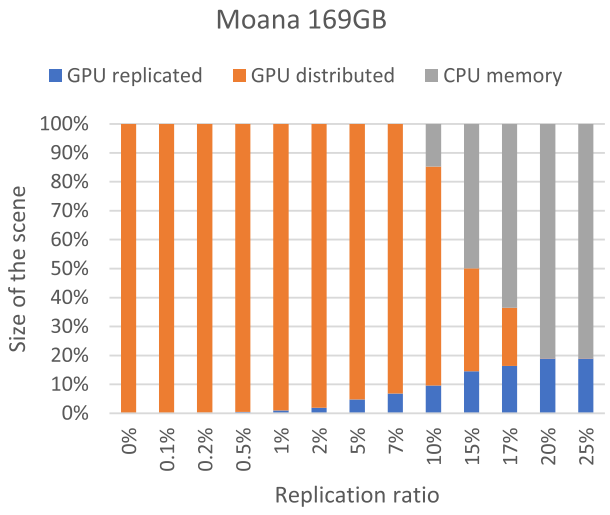

Fig. 19. The proportion of placement of individual chunks by replication ratio for Karolina with 8 GPUs. The chunk size was set to 2 MB.

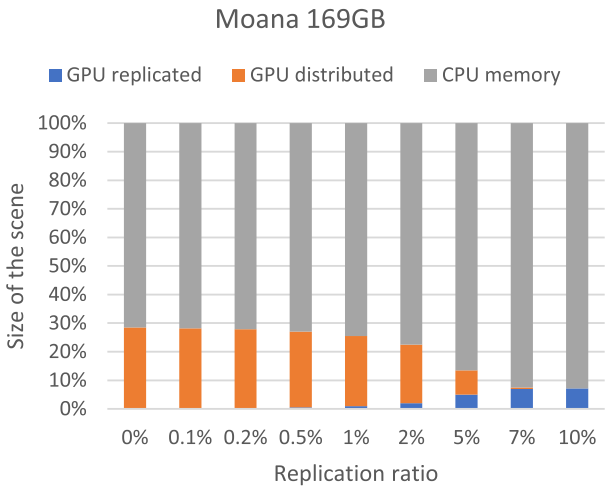

Fig. 20. The proportion of placement of individual chunks by replication ratio for Barbora and M100 with 4 GPUs. The chunk size was set to 32 MB.

Table 8
The relative performance per GPU wrt. DGX-2 (16 GB) for all platforms.

Relative performance	DGX-2	Karolina	Barbora	M100
Moana 169 GB	1.04	2.41	0.96	0.59
Museum 124 GB	1.01	1.26	0.79	0.72
Agent 167 GB	1.01	0.92	0.91	0.91
Spring 137 GB	1.01	1.23	0.98	0.96
Galaxy 137 GB	1.05	11.08	0.54	–
#GPUs	16	4	4	8

- If we compare *DGX-2* with *DGX-2 (16 GB)*, where the only difference between these configurations is GPU memory capacity. However, the results indicate negligible speedup, suggesting that the additional GPU memory does not significantly impact performance under these conditions.
- In the case of the Barbora server, the relative performance remains above or equal to 0.91 across all tested scenes, except for *Museum 124 GB* and *Galaxy 137 GB*. These scenes are the most memory-bound, leading to a significant slowdown on *Barbora*. The primary cause of this slowdown is frequent memory accesses from system RAM, which limits rendering performance.

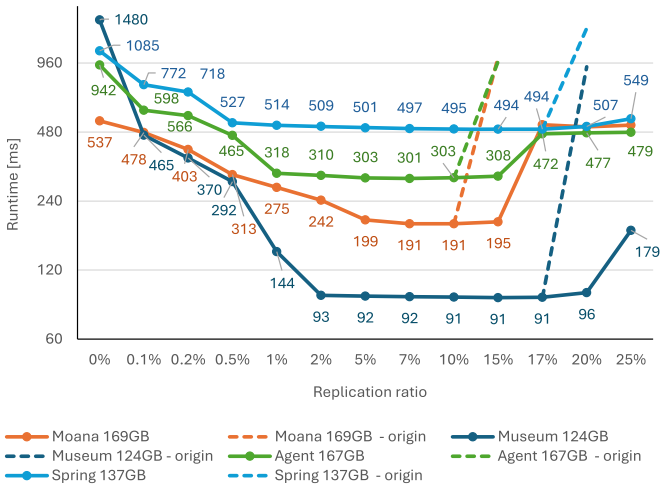

Fig. 21. Comparison of the behaviour of the new modified algorithm with the original algorithm. Path tracing times are for four scenes running on 16 GPUs on the DGX-2 system. The chunk size is 64 MB for all replication ratios.

- In the case of *M100*, it can be seen that the relative performance is the same as the *Barbora* server for the *Agent 164 GB* and *Spring 137 GB* scenes because these scenes are compute-bound. There is a slowdown due to the communication via X-Bus interconnect for the *Moana 169 GB* and *Museum 124 GB* scenes.
- In the case of the *Karolina* server, the relative performance per GPU significantly exceeds that of *DGX-2 (16 GB)* in multiple scenes, particularly for *Moana 169 GB* and *Galaxy 137 GB*, where the speedup reaches 2.41× and 11.08×, respectively. This highlights Karolina's efficiency in handling memory-intensive and complex scenes, thanks to its optimized memory bandwidth, faster interconnects, and efficient workload distribution across 8 GPUs.

We compared the performance and behavior of our proposed algorithm with the approach introduced in [1]. The results for DGX-2 are presented in Fig. 21. The key observations from the comparison are as follows:

- In the case of the original algorithm, for the *Moana 169 GB* and *Agent 167 GB* scenes, a replication ratio of 15 % or higher causes the scene to exceed the available GPU shared memory. As a result, chunks are continuously swapped between GPU and CPU memory, leading to multiple rendering slowdowns depending on the volume of data being transferred. The new approach mitigates this issue by integrating CPU memory as an additional memory tier within the total memory pool, effectively preventing the massive slowdowns observed in the original algorithm. As shown in Fig. 18, when the replication ratio reaches 17 % or higher, there is no distributed data, and the majority of the scene is stored in RAM, leading to only a minor rendering slowdown compared to the previous algorithm.
- A similar pattern is observed for the *Museum 124 GB* and *Spring 137 GB* scenes. When the replication ratio reaches 20 %, a significant performance slowdown occurs due to frequent chunk swapping between GPU and CPU memory. As in the previous case, the new out-of-core algorithm optimizes data placement by offloading less frequently accessed data to RAM, reducing unnecessary memory transfers. When the replication ratio increases to 25 %, data distribution is no longer required, and the rendering performance is only slightly affected, demonstrating the efficiency of the proposed approach.

4.6. Animation

We have extended the evaluation of our Algorithm 1 on rendering animations (see Fig. 22). In Fig. 23, 24, and 25 we present a comparison

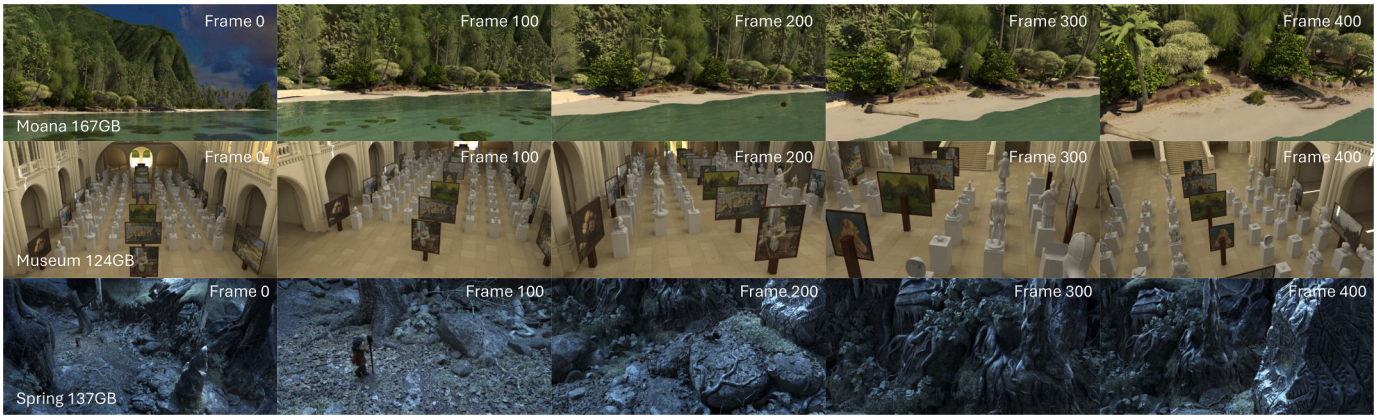

Fig. 22. Rendered frames (0, 100, 200, 300, and 400) from animated camera paths through three complex 3D scenes used in our experiments: *Moana 167 GB*, *Museum 124 GB*, and *Spring 137 GB*.

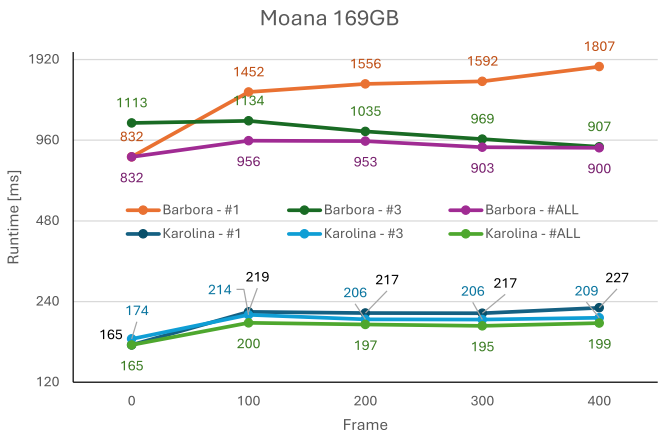

Fig. 23. Comparison of path tracing runtimes for the *Moana 169 GB* scene rendered with 1 sample at a resolution of 5120×2560. The results capture performance across different camera positions at frames 0, 100, 200, 300, and 400, and under three statistical data collection configurations: cases #1, #3, and #ALL. The scene was rendered on the Barbora and Karolina GPU nodes.

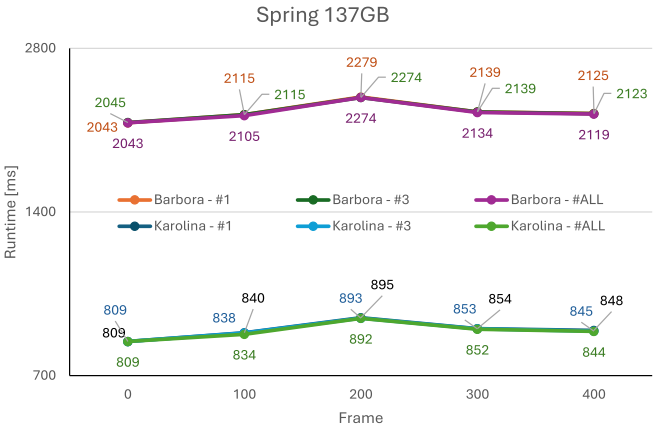

Fig. 25. Comparison of path tracing runtimes for the *Spring 137 GB* scene rendered with 1 sample at a resolution of 5120×2560. The results capture performance across different camera positions at frames 0, 100, 200, 300, and 400, and under three statistical data collection configurations: cases #1, #3, and #ALL. The scene was rendered on the Barbora and Karolina GPU nodes.

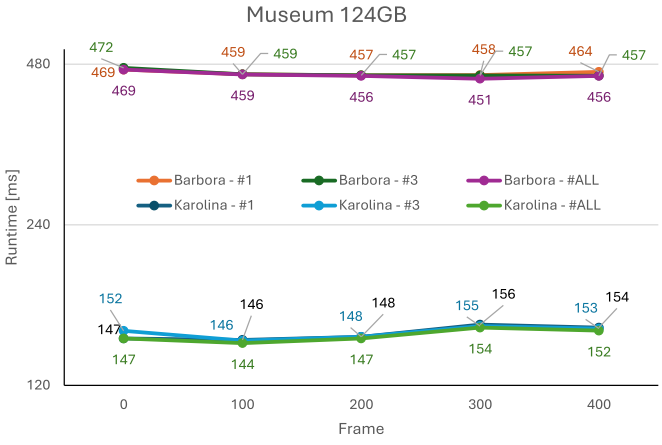

Fig. 24. Comparison of path tracing runtimes for the *Museum 124 GB* scene rendered with 1 sample at a resolution of 5120×2560. The results capture performance across different camera positions at frames 0, 100, 200, 300, and 400, and under three statistical data collection configurations: cases #1, #3, and #ALL. The scene was rendered on the Barbora and Karolina GPU nodes.

of path tracing runtimes for three complex scenes *Moana 169 GB*, *Museum 124 GB*, and *Spring 137 GB*, rendered with 1 sample per pixel at a resolution of 5120×2560. The performance is evaluated across various camera positions (frames 0, 100, 200, 300, and 400) and under three different methods of statistical data collection, referred to as cases #1, #3, and #ALL, using the Barbora and Karolina GPU nodes. Each figure illustrates the impact of when and how statistical data is collected. In the first case #1, statistical calculations are performed exclusively for the initial frame 0. As a result, this approach provides optimal performance for frame 0 but suboptimal performance for other frames. This is because there is no additional statistical data that would allow adaptation to new camera views. In the case #ALL, the statistical data undergoes a recalculation process that is performed independently for each frame. This approach provides optimal runtime for all frames but introduces the highest overhead costs for preprocessing due to repeated complete statistical calculations. The third case study #3 (our recommended solution for animations) presents a new strategy for accumulating statistics. In this experiment, we calculate statistics in frame 0 with 1 spp, then move the camera to frame 200 (middle frame), render it with 1 spp, and continue accumulating statistics. We then move to frame 400 (end frame) and repeat the above process. These accumulated statistics are then used as input for Algorithm 1 for rendering at all camera positions (frames 0,100,200,300,400). The results show the effectiveness of case #3 and verify its applicability. A notable finding is that the approach

Table 9
Comparison to geometry and texture streaming method running on Karolina with 8 GPUs.

#GPUs	Streaming method	Ours method
Moana 169 GB		
2	8.76 s	**0.64 s**
4	4.75 s	**0.33 s**
8	2.88 s	**0.17 s**
Museum 124 GB		
2	4.94 s	**0.57 s**
4	3.29 s	**0.29 s**
8	2.24 s	**0.15 s**
Galaxy 137 GB		
2	18.71 s	**1.74 s**
4	9.33 s	**0.44 s**
8	4.77 s	**0.18 s**

achieves the most significant performance increase in the *Moana 169 GB* scene, indicating its potential for optimizing rendering in highly complex, memory-intensive datasets.

4.7. Comparison to geometry and texture streaming method

We have compared our method with the geometry and texture streaming approach using the page-swap technique described by Garanzha et al. [23]. We have implemented this method in our code. For easier implementation, we used unified memory, where the geometry (triangles and vertices) and textures reside in CPU memory, while all other parts of the scene, including the BVH, are stored entirely in GPU memory. The results are presented in Table 9, showcasing the absolute dominance of our approach over the streaming (page-swap) approach.

5. Conclusions

In this work, we introduced a multi-GPU out-of-core path tracing method that effectively extends GPU memory by utilizing CPU system memory, enabling the rendering of massive scenes that exceed the combined memory capacity of all GPUs. Our approach builds upon previous methods by minimizing performance penalties associated with out-of-core data access while efficiently distributing and replicating scene data across available memory resources. By leveraging NVIDIA Unified Memory and memory access statistics, our algorithm dynamically determines the optimal placement of scene chunks, significantly reducing data transfer bottlenecks between GPUs and the CPU.

Through scalability and performance evaluations conducted on multiple high-performance computing platforms, including *DGX-2*, *Barbora*, *M100*, and *Karolina*, we demonstrated that our approach achieves linear or superlinear scalability as additional GPUs not only increase compute power but also expand total GPU memory, reducing the dependency on slower CPU memory. Our results show that memory-aware chunk distribution and replication play a critical role in determining overall rendering efficiency. By intelligently placing frequently accessed data in GPU memory and offloading less critical data to system RAM, our method maintains high performance even for extremely large and complex scenes.

Compared to previous approaches, including the data replication method of Jaros et al. [1], the streaming technique of Garanzha et al. [23], and the out-of-core method of Jaros et al. [37], our algorithm achieves excellent rendering times on all tested platforms. This is primarily due to the integration of CPU memory as a controlled part of the memory hierarchy and intelligent block placement decisions based on empirical access behavior. The algorithm prevents excessive memory overhead by moving infrequently used data to system RAM, while ensuring that frequently used data remains in the local memory of each GPU. We have also evaluated our method on multiple frames of a movie and suggested a suitable solution for acquiring access statistics that can deliver low rendering times for all frames of a movie.

Furthermore, the analysis of per-GPU relative performance showed that our approach maintains efficiency across different hardware configurations. On *Barbora* and *M100*, we observed a higher impact of interconnect limitations (such as X-Bus in M100), whereas *DGX-2* and *Karolina* demonstrated better memory bandwidth utilization due to its NVSwitch architecture. Despite these hardware differences, our method adapted effectively to each system, showing only minor slowdowns when GPU memory was exhausted and RAM was used instead.

In terms of general applicability, the proposed approach does not require any major changes to the rendering engine and can be integrated into any path tracer that supports CUDA Unified Memory. This architectural independence, combined with the ability to handle both static and animated tasks, makes this method a strong candidate for integration into existing GPU-accelerated production pipelines.

In summary, our memory-aware out-of-core path tracing algorithm provides a scalable, efficient, and adaptable solution for rendering massive scenes on multi-GPU systems. By combining intelligent data replication and distribution strategies, we successfully mitigate performance losses associated with out-of-core rendering. Our results indicate that even small-scale multi-GPU systems can achieve rendering performance comparable to significantly larger clusters, making high-quality path tracing more accessible for production rendering and scientific visualization applications.

Future work could explore further optimizations in data prefetching, compression techniques, and improved scheduling strategies to further enhance performance for even larger and more complex scenes.

CRediT authorship contribution statement

Milan Jaros: Writing – original draft, Visualization, Methodology, Investigation, Conceptualization; **Lubomir Riha:** Writing – review & editing, Validation, Methodology, Funding acquisition, Conceptualization; **Petr Strakos:** Writing – original draft, Validation, Methodology, Investigation, Conceptualization; **Tomas Kozubek:** Writing – review & editing, Validation, Funding acquisition.

Data availability

Data will be made available on request.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgement

This work was supported by the Ministry of Education, Youth and Sports of the Czech Republic through the e-INFRA CZ (ID:90254). This work was supported by the SPACE project under grant agreement No. 101093441. The project is supported by the European High-Performance Computing Joint Undertaking and its members (including top-up funding by the Ministry of Education, Youth and Sports of the Czech Republic ID: MC2304). This work was supported by the FALCON project - the European Union's Horizon Europe research and innovation programme under grant agreement No. 101138305. The authors would like to thank Disney for releasing the Moana Island Scene, 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. The Agent scene from the Agent 327: Operation Barbershop movie is copyright (CC) by Blender Foundation based on original characters (C) by Martin

Lodewijk and the Spring scene from Spring movie is copyright (CC) by Blender Foundation. We would like to thank Cineca for the access to Marconi 100 cluster and for the availability of high-performance computing resources and support.

References

[1] M. Jaros, L. Riha, P. Strakos, M. Spetko, GPU accelerated path tracing of massive scenes, ACM Trans. Graph. 40 (2) (2021).

[2] A. Li, S.L. Song, J. Chen, J. Li, X. Liu, N.R. Tallent, K.J. Barker, Evaluating modern GPU interconnect: PCIe, NVLink, NV-SLI, NVSwitch and GPUDirect, IEEE Trans. Parallel Distrib. Syst. 31 (1) (2020) 94–110. https://doi.org/10.1109/TPDS.2019.2928289

[3] R. Nohria, G. Santos, V. Haug, IBM power system AC922: technical overview and introduction, 2018. https://www.redbooks.ibm.com/redpapers/pdfs/redp5494.pdf.

[4] M. Harris, Unified Memory for CUDA beginners, 2017, (https://devblogs.nvidia.com/unified-memory-cuda-beginners/).

[5] B. Burley, D. Adler, M.J.-Y. Chiang, H. Driskill, R. Habel, P. Kelly, P. Kutz, Y.K. Li, D. Teece, The design and evolution of Disney's hyperion renderer, ACM Trans. Graph. 37 (3) (2018).

[6] P. Christensen, J. Fong, J. Shade, W. Wooten, B. Schubert, A. Kensler, S. Friedman, C. Kilpatrick, C. Ramshaw, M. Bannister, et al., RenderMan: an advanced path-tracing architecture for movie rendering, ACM Trans. Graph. 37 (3) (2018).

[7] L. Fascione, J. Hanika, M. Leone, M. Droske, J. Schwarzhaupt, T. Davidovič, A. Weidlich, J. Meng, Manuka: a batch-shading architecture for spectral path tracing in movie production, ACM Trans. Graph. 37 (3) (2018).

[8] I. Georgiev, T. Ize, M. Farnsworth, R. Montoya-Vozmediano, A. King, B.V. Lommel, A. Jimenez, O. Anson, S. Ogaki, E. Johnston, et al., Arnold: a Brute-Force production path tracer, ACM Trans. Graph. 37 (3) (2018).

[9] C. Kulla, A. Conty, C. Stein, L. Gritz, Sony pictures Imageworks Arnold, ACM Trans. Graph. 37 (3) (2018).

[10] S. Molnar, M. Cox, D. Ellsworth, H. Fuchs, A sorting classification of parallel rendering, IEEE Comput. Graph. Appl. 14 (4) (1994) 23–32. https://doi.org/10.1109/38.291528

[11] S. Parker, P. Shirley, Y. Livnat, C. Hansen, P.-P. Sloan, Interactive ray tracing for Ssosurface rendering, in: Proceedings of the IEEE Visualization Conference, 1998, pp. 233–238.

[12] I. Wald, C. Benthin, P. Slusallek, Distributed interactive ray tracing of dynamic scenes, in: PVG 2003, Proceedings - IEEE Symposium on Parallel and Large-Data Visualization and Graphics 2003, 2003, pp. 77–85.

[13] D.E. Demarle, C.P. Gribble, S. Boulos, S.G. Parker, Memory sharing for interactive ray tracing on clusters, Parallel Comput. 31 (2) (2005) 221–242. Cited By :13, www.scopus.com.

[14] A. Keller, C. Wächter, M. Raab, D. Seibert, D. van Antwerpen, J. Korndörfer, L. Kettner, The Iray light transport simulation and rendering system, 2017. arXiv:1705.01263

[15] T. Kato, H. Nishimura, T. Endo, T. Maruyama, J. Saito, P.H. Christensen, Parallel rendering and the quest for realism: the Kilauea massively parallel ray tracer, in: Alan Chalmers, Practical Parallel Processing for Today's Rendering Challenges. SIGGRAPH 2001 Course Note #40, 2001, pp. IV–1 to IV–59.

[16] I. Wald, S.G. Parker, Data parallel path tracing with object hierarchies, Proc. ACM Comput. Graph. Interact. Tech. 5 (3) (2022). https://doi.org/10.1145/3543861

[17] W. Usher, I. Wald, J. Amstutz, J. Günther, C. Brownlee, V. Pascucci, Scalable ray tracing using the distributed FrameBuffer, Comput. Graph. Forum 38 (3) (2019) 455–466. https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.13702. https://doi.org/10.1111/cgf.13702

[18] P.A. Navrátil, D.S. Fussell, C. Lin, H. Childs, Dynamic scheduling for large-scale distributed-memory ray tracing, in: H. Childs, T. Kuhlen, F. Marton (Eds.), Eurographics Symposium on Parallel Graphics and Visualization, The Eurographics Association, 2012. https://doi.org/10.2312/EGPGV/EGPGV12/061-070

[19] P.A. Navrátil, H. Childs, D.S. Fussell, C. Lin, Exploring the spectrum of dynamic scheduling algorithms for scalable distributed-memory ray tracing, IEEE Trans. Visual. Comput. Graph. 20 (6) (2014) 893–906.

[20] M. Pharr, C. Kolb, R. Gershbein, P. Hanrahan, Rendering complex scenes with memory-coherent ray tracing, in: Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH '97, ACM Press/Addison-Wesley Publishing Co., USA, 1997, p. 101–108. https://doi.org/10.1145/258734.258791

[21] B. Budge, T. Bernardin, J.A. Stuart, S. Sengupta, K.I. Joy, J.D. Owens, Out-of-core data management for path tracing on hybrid resources, Comput. Graph. Forum 28 (2) (2009) 385–396.

[22] M. Son, S.-E. Yoon, Timeline scheduling for out-of-core ray batching, in: Proceedings of High Performance Graphics, HPG 2017, 2017.

[23] K. Garanzha, A. Bely, S. Premoze, V. Galaktionov, Out-of-core GPU ray tracing of complex scenes, in: ACM SIGGRAPH 2011 Talks, SIGGRAPH'11, 2011.

[24] R. Wang, Y. Huo, Y. Yuan, K. Zhou, W. Hua, H. Bao, GPU-based out-of-core many-lights rendering, ACM Trans. Graph. 32 (6) (2013).

[25] T. Harada, A framework to transform in-core GPU algorithms to out-of-core algorithms, in: Proceedings – 20th ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games, I3D 2016, 2016, pp. 179–180.

[26] O. Mattausch, J. Bittner, A. Jaspe, E. Gobbetti, M. Wimmer, R. Pajarola, CHC+RT: coherent hierarchical culling for ray tracing, Comput. Graph. Forum 34 (2) (2015) 537–548.

[27] T. Kim, X. Sun, S. Yoon, T-ReX: interactive global illumination of massive models on heterogeneous computing resources, IEEE Trans. Visual. Comput. Graph. 20 (3) (2014) 481–494.

[28] R.H. Hunter, B.C. White, R.R. Patel, J.R. Ballard, Partitioning terabyte-scale faceted geometry models for efficient parallel ray tracing using out-of-core memory, in: ACMSE 2020 – Proceedings of the 2020 ACM Southeast Conference, 2020, pp. 256–259.

[29] K. Zhou, Q. Hou, Z. Ren, M. Gong, X. Sun, B. Guo, RenderAnts: interactive REYES rendering on GPUs, ACM Trans. Graph. 28 (5) (2009). ACM SIGGRAPH Asia Conference 2009, Yokohama, JAPAN, DEC 16-19, 2009. https://doi.org/10.1145/1618452.1618501

[30] J. Pantaleoni, L. Fascione, M. Hill, T. Aila, PantaRay: fast ray-traced occlusion caching of massive scenes, ACM Trans. Graph. 29 (4) (2010). https://doi.org/10.1145/1778765.1778774

[31] S. Eilemann, D. Steiner, R. Pajarola, Equalizer 2.0-convergence of a parallel rendering framework, IEEE Trans. Visual. Comput. Graph. 26 (2) (2020) 1292–1307. https://doi.org/10.1109/TVCG.2018.2870822

[32] Y. Dong, C. Peng, Multi-GPU multi-display rendering of extremely large 3D environments, Vis. Comput. 39 (12) (2023) 6473–6489. https://doi.org/10.1007/s00371-022-02740-7

[33] N. Sakharnykh, Unified memory on Pascal and Volta, in: GPU Technology Conference (GTC), 2017. http://on-demand.gputechconf.com/gtc/2017/presentation/s7285-nikolay-sakharnykh-unified-memory-on-pascal-and-volta.pdf.

[34] J. Protic, M. Tomasevic, V. Milutinovic, A survey of distributed shared memory systems, in: Proceedings of the Twenty-Eighth Annual Hawaii International Conference on System Sciences, 1, IEEE, 1995, pp. 74–84.

[35] J. Protic, M. Tomasevic, V. Milutinovic, Distributed shared memory: concepts and systems, IEEE Parallel Distrib. Technol. Syst. Appl. 4 (2) (1996) 63–71.

[36] V. Soundararajan, M. Heinrich, B. Verghese, K. Gharachorloo, A. Gupta, J. Hennessy, Flexible use of memory for replication/migration in cache-coherent DSM multiprocessors, in: Proceedings. 25th Annual International Symposium on Computer Architecture (Cat. No. 98CB36235), IEEE, 1998, pp. 342–355.

[37] M. Jaros, L. Riha, P. Strakos, T. Kozubek, Multi-GPU accelerated rendering of massive scenes with out-of-core support for CPU memory, in: PPAM 2024: 15th International Conference on Parallel Processing and Applied Mathematics, Lecture Notes in Computer Science (LNCS), Springer, Bialystok, Poland, 2024. To appear.

[38] B. Foundation, Cycles open source production rendering, 2018, (https://www.cycles-renderer.org/).

[39] NVIDIA, Programming guide: CUDA toolkit documentation, 2022. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#um-unified-memory-programming-hd.

[40] NVIDIA, DGX-2/2H SYSTEM user guide, 2019, (https://docs.nvidia.com/dgx/pdf/dgx2-user-guide.pdf). DU-09130-001_v08.1.

[41] NVIDIA, NVIDIA NVSWITCH technical overview, 2018, (https://images.nvidia.com/content/pdf/nvswitch-technical-overview.pdf).

[42] J.D. McCalpin, Memory bandwidth and machine balance in current high performance computers, IEEE Comput. Soc. Tech. Committee Computer Architect. (TCCA) Newslett. (1995) 19–25.

[43] Atos, BullSequana X410 E5 dense GPU-accelerated compute node, 2017, (https://atos.net/wp-content/uploads/2017/11/FS_BullSequana_X410E5_en1-web.pdf).

[44] IT4Innovations, Barbora supercomputer cluster, 2019, (https://docs.it4i.cz/barbora/introduction/).

[45] Cineca, UG3.2: MARCONI100 userguide, 2020. https://wiki.u-gov.it/confluence/pages/viewpage.action?pageId=336727645.

[46] W.D.A. Studios, Moana island scene, 2018, (https://www.disneyanimation.com/resources/moana-island-scene/).

[47] J. Birn, 3dRender.com: lighting challenges, 2015, (http://www.3drender.com/challenges/).

[48] Threedscans, Three D scans, 2020, (https://threedscans.com).

[49] T.A. I.o. Chicago, Discover art & artists, 2020, (https://www.artic.edu/collection).

[50] B. Studio, Agent 327 – blender cloud, 2020, (https://cloud.blender.org/films/agent-327).

[51] B. Studio, Spring – blender cloud, 2020, (https://cloud.blender.org/films/spring).

[52] R. Wissing, S. Shen, Numerical dependencies of the galactic dynamo in isolated galaxies with SPH, Astron. Astrophys. 673 (2023) A47. https://doi.org/10.1051/0004-6361/202244753