VSB TECHNICAL
UNIVERSITY
OF OSTRAVA

# Massively parallel implementation of algorithms for computer graphics

Masivně paralelní implementace algoritmů počítačové grafiky

Milan Jaroš

PhD Thesis

Supervisor: prof. Ing. Tomáš Kozubek, Ph.D.

Ostrava, 2021

## Abstrakt a přínos práce

Počítačová grafika od svého vzniku v 60. letech 20. století udělala velký pokrok. Stala se součástí každodenního života. Můžeme ji vidět všude kolem nás, od chytrých hodinek a smartphonů, kde jsou grafické akcelerátory již součástí čipů a dokáží vykreslovat nejen interaktivní menu, ale i náročné grafické aplikace, přes notebooky a osobní počítače až po výkonné vizualizační servery nebo superpočítače, které dokáží zobrazovat náročné simulace v reálném čase. V této disertační práci se zaměříme na jednu z výpočetně nejnáročnějších oblastí počítačové grafiky, a tou je výpočet globálního osvětlení. Jednou z nejpoužívanějších metod pro simulaci globálního osvětlení je metoda sledování cesty. Pomocí této metody můžeme vizualizovat např. vědecká nebo lékařská data. Metodu sledování cest lze urychlit pomocí několika grafických akcelerátorů, na které se v této práci zaměříme. Představíme řešení pro vykreslování masivních scén na více GPU. Náš přístup analyzuje vzory přístupů k paměti a definuje, jak by měla být data scény rozdělena mezi grafickými akcelerátory s minimální ztrátou výkonu. Klíčovým konceptem je, že části scény, které mají nejvyšší počet přístupů do paměti, jsou replikovány na všech grafických akcelerátorech. Představíme dvě metody pro maximalizaci výkonu vykreslování při práci s částečně distribuovanými daty scény. Obě metody pracují na úrovni správy paměti, a proto není třeba datové struktury přepracovávat. Tento nový out-of-core mechanismus jsme implementovali do open-source path traceru Blender Cycles, který jsme také rozšířili o technologie podporující běh na superpočítačích a schopné využít všechny akcelerátory alokované na více uzlech. V této práci také představíme novou službu, která využívá naši rozšířenou verzi Blender Cycles a zjednodušuje odesílání a spouštění úloh přímo z programu Blender.

## Klíčová slova

Multi-GPU Path Tracing, CUDA Unified Memory, metoda sledování cest, distribuovaná data, sdílená paměť, služba Rendering-as-a-service

## Abstract and Contributions

Computer graphics, since its inception in the 1960s, has made great progress. It has become part of everyday life. We can see it all around us, from smartwatches and smartphones, where graphic accelerators are already part of the chips and can render not only interactive menus but also demanding graphic applications, to laptops and personal computers as well as to high-performance visualization servers and supercomputers that can display demanding simulations in real time. In this dissertation we focus on one of the most computationally demanding area of computer graphics and that is the computation of global illumination. One of the most widely used methods for simulating global illumination is the path tracing method. Using this method, we can visualize, for example, scientific or medical data. The path tracing method can be accelerated using multiple graphical accelerators, which we will focus on in this work. We will present a solution for path tracing of massive scenes on multiple GPUs. Our approach analyzes the memory access pattern of the path tracer and defines how the scene data should be distributed across up to 16 GPUs with minimal performance impact. The key concept is that the parts of the scene that have the highest number of memory accesses are replicated across all GPUs. We present two methods for maximizing the performance of path tracing when dealing with partially distributed scene data. Both methods operate at the memory management level, and therefore the path tracing data structures do not need to be redesigned. We implemented this new out-of-core mechanism in the open-source Blender Cycles path tracer, which we also extended with technologies that support running on supercomputers and can take advantage of all accelerators allocated on multiple nodes. In this work, we also introduce a new service that uses our extended version of the Blender Cycles renderer to simplify sending and running jobs directly from Blender.

## Keywords

Multi-GPU Path Tracing, CUDA Unified Memory, Data Distributed Path Tracing, Distributed Shared Memory Path Tracing, Rendering-as-a-service

## Acknowledgement

I would like to thank my family for their support on this journey. First of all, I would like to thank my wife Marta and my daughters Natalia and Cecilia, who make fun every day and make me enjoy discovering new things. This dissertation is dedicated to them.

I'd also like to thank the gentlemen at the Blender Foundation and Blender Studio for the opportunity to collaborate on open-movie projects. Furthermore, I would like to thank Jaroslav Křivánek for his inspiration and motivation, Petr Strakoš for consulting projects that will benefit from the results of this work, Tomáš Kozubek for dissertation supervision and patience, and especially Lubomír Říha for his guidance, expert consultation, and new ideas, without whom this work could not have been completed.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

In this chapter we would like to introduce the main topic of this dissertation, motivation and objectives. This thesis is based on the article [1] published in ACM Transactions on Graphics (D1), a leading peer-reviewed journal in the field of computer graphics, in which I was the lead author.

## 1.1 Rendering

The progress in high-performance computing (HPC) plays an important role in science and technology. Computationally intensive simulations have become an essential part of research and development of new technologies. Many research groups in the field of computer graphics are working on problems related to the extremely time-consuming process of image synthesis of virtual scenes, also called rendering.

The purpose of rendering is to create visually realistic 2D images of the modeled 3D scenes. It is used for generating photorealistic images and developing animated movies, computer games, architectural visualization, medical imaging, etc. The Figures 1.1,1.2 show a preview of the 3D scene in the Blender's viewport and its rendered part. Due to the complex nature of light propagation used in rendering, it is a computationally extremely demanding task. Therefore, versatile hardware and software solutions are used to speed up this process.

There are several software solutions that provide scene modelling and rendering in a single package. The open-source Blender package is one such candidate. It is mainly used for creating 3D graphics and it has a broad user base and wide extensibility through C++ and Python programming. Blender includes, among other things, a 3D modelling environment and two renderers - Eevee and Blender Cycles [2, 3]. Blender Cycles is implemented to use a multi-core CPU or GPU accelerator for efficient rendering in both offline and interactive modes. Blender can also be extended with external renderers such as Mitsuba [4] or LuxRender [5] to provide additional rendering functionality. In addition to Blender, there are several other

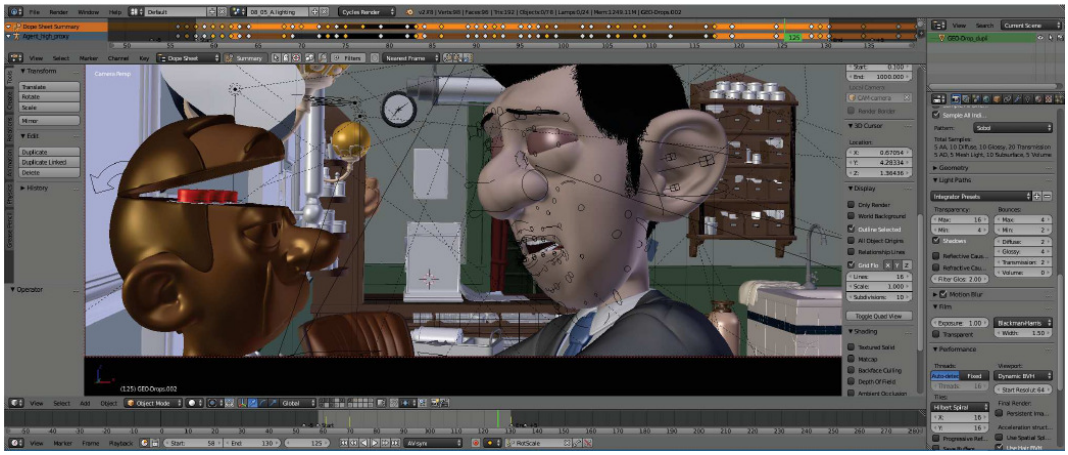

Figure 1.1: The example of Agent327 scene in the Blender's viewport

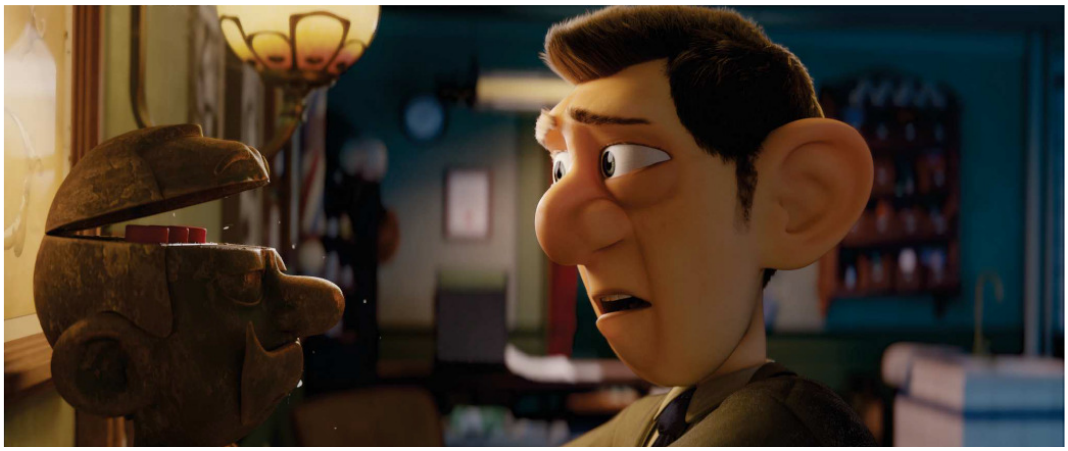

Figure 1.2: The example of Agent327 scene and its rendered image

renderers that offer both offline and interactive rendering, but few of them provide distributed rendering suitable for HPC environments. This is quite important because HPC offers high computational power that can be efficiently used for rendering. It is also important to note that essentially all rendering software usable in HPC environment, except for Blender, is commercial. Examples include Indigo [6], Maxwell [7], Redshift [8], Octane [9], Arnold [10], RenderMan [11], V-Ray [12], Iray [13], which support graphics accelerators.

In addition to using HPC clusters to accelerate computationally intensive tasks, hardware accelerators are also widely used. These can further increase the computational performance and rendering efficiency of HPC clusters. In general, two types of general-purpose hardware accelerators, namely Many Integrated Core (MIC) coprocessors, also known as Intel Xeon Phi and Graphics Processing Units (GPUs), can be used. MIC is a technology that by now has been replaced by graphics accelerators and it is no longer developed. But it is still an excellent example of upcoming many-core CPUs and therefore it is worth to focus on them in this work.

Ten years ago, Intel introduced its own accelerator technology, the Intel Many Integrated Core. The advantage of this accelerator technology was that it had more memory available compared to GPUs. To use Intel Xeon Phi technology in rendering, Intel introduced Embree [14], which is a library of photorealistic ray tracing kernels. However, the Corona renderer [15], as an example of a commercial Embree-based product, only runs on general purpose processors. Moreover, an open-source renderer OSPRay [16] developed by Intel also runs only on CPUs. Mainly for these reasons, we decided to develop and present an extension to the open-source Blender code that can use an entire cluster equipped with either Intel or AMD processors or graphic accelerators, both for offline and interactive rendering.

Direct use of a remote cluster is very complicated for inexperience users. It requires extensive knowledge of the cluster's infrastructure, operating system, installed software packages, and job schedulers. There are several solutions for using remote computers. A computer cluster that is exclusively used for rendering is also called a render farm, e.g., RebusFarm [17], Fox Render Farm [18], RNDR [19], iRender [20], GarageFarm [21]. These farms provide cloud rendering services based on the Infrastructure-as-a-Service (IaaS) model [22]. J. Ruby Annette describes the Rendering-as-a-service model for the focusing on the rendering [23]. This model provides a high-level access to the remote cluster. In this work, we present our newly developed RaaS solution based on High-End Application Execution Middleware (HEAppE Middleware, [24]). Our solution is unique in its ease of use. The controls are integrated into the Blender environment and contain two buttons, one for submitting the scene and the other for downloading. With one button, the entire cluster can be utilized. Another unique feature is the support for fast unified memory and therefore support for large scenes.Being able to take advantage of the latest technologies such as memory sharing across multiple graphic cards, this service thus allows even large scenes to be rendered.

In recent years, due to advances in GPU technology, GPU rendering has become popular for its superior performance when compared to CPU rendering [8]. On the other hand, a significant disadvantage of GPUs when compared to CPUs is the limited memory size. This is one of the key reasons why CPUs still play a dominant role in production rendering, as described by the authors of the main rendering systems used in film production [25, 26, 27, 28, 29]. For example, [27] reports that scenes from Pixar's Coco consumed up to 120 GB. Scenes of that size cannot fit in the memory of a single GPU.

An essential part of this thesis presents a solution to this problem that is based on replicating a small amount of scene data, i.e. between 1 and 5%, and intelligently distributing the rest of the data into the memory of multiple GPUs. Our approach [1] relies on two key technologies: (i) NVLink GPU interconnection, which allows multiple GPUs to efficiently share the contents of their memories due to its high bandwidth and low latency, [30], and (ii) CUDA Unified Memory (UM), which gives programmers control over the location of data in the memories of the interconnected GPUs, [31].

In 2016, NVIDIA introduced NVLink interconnect technology [32] into the world of professional computer graphics with the Quadro GP100 product [33]. With the Turing GPU hardware architecture, the technology was also released for GPUs in the gaming sector [34]. This interconnect enables multiple GPUs to share their memories due to its very high bandwidth, which is comparable to the memory bandwidth of a high-end CPU to its DDR memory, and due to its low latency. Today, in the professional computer graphics sector, two GPUs can be connected using NVLink and provide up to $2 \times 48\,\text{GB}$ of shared memory. However, in the world of High Performance Computing (HPC) and Artificial Intelligence (AI), where *"memory size matters"*, NVLink technology is already used to interconnect 4, 8, and even 16 GPUs. This means that state-of-the-art GPU servers designed for HPC and AI, like the NVIDIA DGX-2 used for this work, may have up to 512 GB of shared GPU memory [35]. In such servers, each GPU has a memory bandwidth of approximately 790 GB/s, which is $4-5$ times faster than a high-end CPU. The bandwidth for remote GPU memory over NVLink in DGX-2 is 138 GB/s which is similar to the CPU local memory bandwidth. We expect that it is merely a matter of a few years before servers like this will be equipped with GPUs with hardware acceleration for path tracing such as ray tracing (RT) cores [34].

As a proof-of-concept (using the whole cluster or rendering of the large scenes on multiple GPUs), the proposed mechanisms have been implemented in the comprehensive open-source Blender Cycles engine, which uses a path-tracing method to achieve realistic results[36].

### 1.1.1 Path to path-tracing

The history of rendering dates back to 1960, when graphic designer William Fetter needed to find a way to optimize the cockpit interior of a Boeing aircraft. His efforts led him to

a technique that started a revolution in rendering. He created a computer-generated orthographic view of the human figure that he called [37] computer graphics. Another computer graphics pioneer coming from the same period as William Fetter, Ivan Sutherland, created the first computer graphics software called Sketchpad [38]. It was the first software that allowed interactive drawing on a computer screen. At that time, 3D images were only displayed as wireframes.

For a long time, this seemed almost impossible, especially after the addition of shading and depth of field, until Henry Gouraud developed the shading model [39] in 1971, which became widely used. In practice, Gouraud's model is used to achieve illumination on surfaces with a small number of polygons without demanding computational requirements.

The final step in solving the problem of creating a realistic image was to accurately place the shadows using the ray tracing method. The first concept, developed by an IBM researcher known as Appel [40], was a method known as ray casting, where the calculation is completed after the first intersection of the ray with the object. However, computers at the time (1968) were not powerful enough to make his concept a reality. In 1980, Turner Whitted published a paper [41] on the recursive method of ray tracing, where the calculation continues after the first intersection and continues to reflect in the direction given by the material type. This paper was considered groundbreaking at the time. From the basic ray tracing method, several other variants of this method emerged, such as distributed ray tracing [42], path tracing [43], bidirectional path tracing [44, 45], and metropolis light transport [46], which are widely used today. These methods work on the same principle but differ in the number and direction of the beams. There are also other methods that are not based on pure ray tracing, such as Reyes' method [47] and photon mapping [48, 49]. In this work, we focus on the path-tracing method used by the Blender Cycles renderer.

### 1.1.2 Rendering equation

The rendering equation is the basic algorithm for all ray-tracing-based image synthesis algorithms such as path-tracing [50]. In 1986, Kajiya first introduced the rendering equation in computer graphics, [51]. One of the most recent versions ([52]) of this equation is represented as follows

$$L_o(x,\vec{\omega}_o) = L_e(x,\vec{\omega}_o) + L_r(x,\vec{\omega}_o) = L_e(x,\vec{\omega}_o) + \int_{\Omega} L_i(x,\vec{\omega}_i) f_r(x,\vec{\omega}_i,\vec{\omega}_o)(\vec{\omega}_i \cdot \vec{n}) d\vec{\omega}_i, \quad (1.1)$$

where $\omega_o$ is the direction of the outgoing beam, $L_o$ is the spectral radiation emitted by the source from point $x$ in the direction of $\omega_o$, $L_E$ is the spectral radiation emitted from point $x$ in the direction of $\omega_o$, $\omega_i$ is the direction of the incoming beam, $L_i$ is the spectral

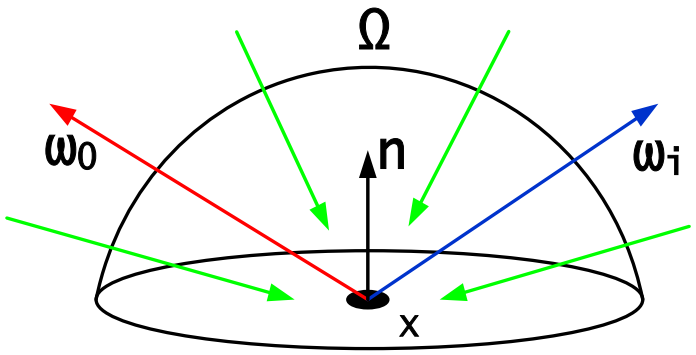


Figure 1.3: The unit hemisphere of the rendering equation in the direction of the normal vector $\vec{n}$ centered at $x$, over which we integrate.

radiation coming into $x$ in the direction of $\omega_i$, $\Omega$ is the unit hemisphere (see Figure 1.3) in the direction of the normal vector $n$ centered at $x$ over which we integrate, and $f_r(x, \omega_i, \omega_o)$ is the bidirectional image reflectance distribution function (BRDF) at $x$ from the direction $\omega_i$ to the direction $\omega_o$, $\omega_i \cdot n$ is the angle between $\omega_i$ and the surface normal.

Solving the rendering equation is computationally very demanding. The most common methods are based on numerical estimation of the integral of the [53]. One of the most used methods for numerical solution of the rendering equation is the Monte Carlo (MC) method.

The deterministic form of the Monte Carlo method is called quasi-Monte Carlo (QCM) [54]. In this method, pseudo-random numbers are replaced by quasi-random numbers that are generated by deterministic algorithms. A typical property of such numbers is that they fill the unit square more uniformly. This property is called low discrepancy (LD) [55, 56]. Well-known types of LD sequences include the Halton, Faure, Sobol, Wozniak, Hammersly, and Niederreiter sequences.

Blender uses the Sobol sequences [57, 58, 59, 60] for the path-tracing method because they suit its needs - they run well on CPU and GPU accelerators, support high path depth, and can perform adaptive sampling.

### 1.1.3 Path-tracing

Path-tracing algorithm integrates possible paths of light from the light source into a selected pixel in a rendered image. In Monte Carlo, we select random sub-pixels and then track a light ray bouncing randomly from object surfaces. Average, weighted by BRFDs, represents the pixel's intensity and color. Each ray path is represented by a high dimensional randomly generated vector, which determines the selected sub-pixel and subsequent random bounces.

It is important to bear in mind that what we see it is not light itself but a light object surface. In the Figure 1.4 you can see how light is reflected by the surface, and that this reflection depends on the surface material. There are two different types of material in the

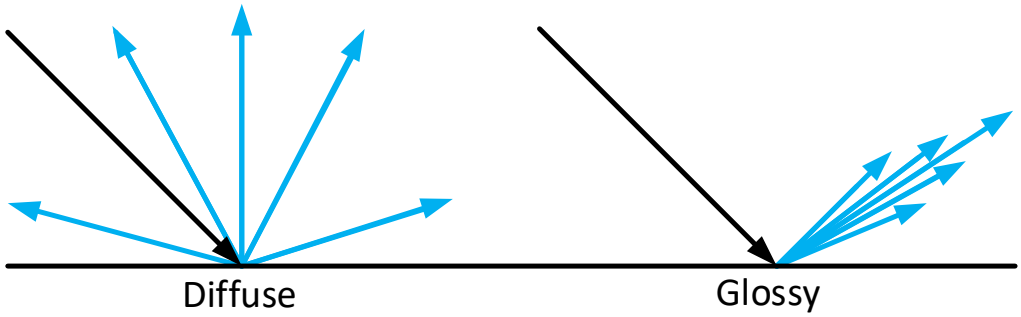


Figure 1.4: The light reflections for the diffuse material (left) and for the glossy material (right)

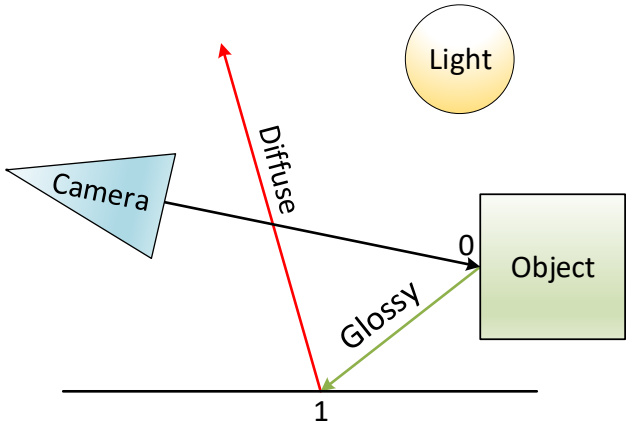


Figure 1.5: The path of the ray through the scene

picture, a glossy and diffusive one, respectively and how they change the way light is reflected. On the left, there is a diffuse shader. For this type of material all directions have the same likelihood of light being reflected. On the right, there is a glossy shader. Directions further away from the angle of reflection are less likely. The length of the arrows denotes the likelihood of a ray continuing in its direction.

Path tracing works in the following way (see Figure 1.5): For each pixel a ray is cast into the scene. It starts as a camera ray until it collides with an object. The ray is influenced by the type of a shader assigned to the surface it hits. The ray then continues as a glossy ray. Let's say after that it hits a diffuse surface. From there it bounces into a random direction. Assuming that the maximum number of bounces in your scene is not reached yet, the ray eventually hits a light source. It is discontinued and the color of the sample the ray belongs to is calculated depending on all materials the ray hits while bouncing. This process is repeated as many times as the number of samples in the render settings is reached. In the end the mean value of all samples is used for the color of the pixel (see Figure 1.6).

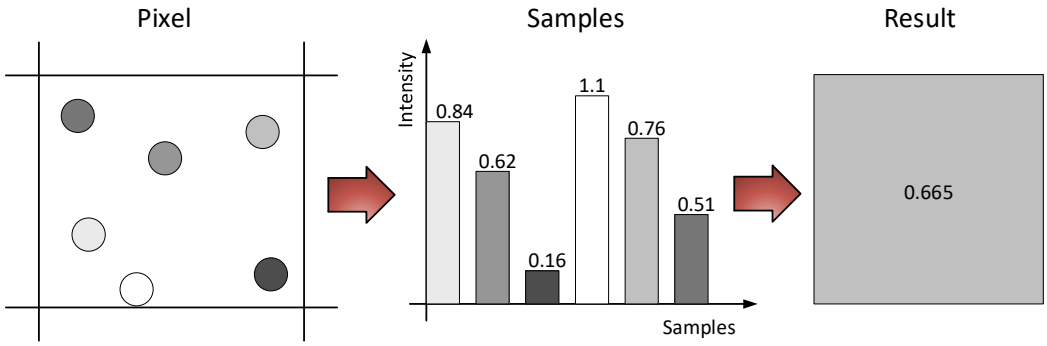


Figure 1.6: The calculation of the resulting pixel color after repeatedly casting of the ray into the scene

## 1.2 Motivation

First, I would like to mention a 1965 law by Intel co-founder Gordon Moore, called Moore's Law: the number of transistors that can be placed on an integrated circuit doubles approximately every 18 months while keeping the price the same. Such growth is called exponential. With the increasing use of new techniques such as global illumination, physical simulation, and stereo projection, more and more processor hours are needed to render still images and animations. This increase, based on Moore's Law, has been described by DreamWorks as Shrek's Law of [61]. As an example, consider the production of a DreamWorks film. The 2001 feature film Shrek 1 required 5 million CPU hours, but 2004's Shrek 2 required 10 million CPU hours, 2007's Shrek 3 required 20 million CPU hours, and 2010's Shrek Forever 3D required 50 million CPU hours. As you can see from Shrek's Law, rendering is one of the most computationally demanding tasks in computer graphics. Shrek is a commercial project, and we are particularly interested in open-source projects. One such project is the Blender project, which aims not only to develop free software but also open movie projects (see Figures 1.7,1.8,1.9,1.10).

Our main motivation for this work is to improve Blender's rendering technique and performance, and thus help Blender and its large community of users to accelerate their projects, especially those interested in HPC rendering.

However, movie rendering is only one of the domains where image synthesis of a virtual scene is needed. Medical imaging is playing an increasingly important role for refining diagnosis or disease detection. Creating photorealistic images from traditional computed tomography (CT) or magnetic resonance imaging (MRI) is called cinematic rendering [62, 63]. Cinematic rendering is a computationally very demanding task and requires a large amount of computational resources to transfer large details to the doctor's computer screen in an interactive mode, which is our other motivation (see Figures 1.11,1.12,1.13,1.14).

Figure 1.7: Blender's open movie: Cosmos Laundromat: First Cycle

Figure 1.8: Blender's open movie: Agent 327: Operation Barbershop

Figure 1.9: Blender's open movie: The Daily Dweebs

Figure 1.10: Blender's open movie: Spring

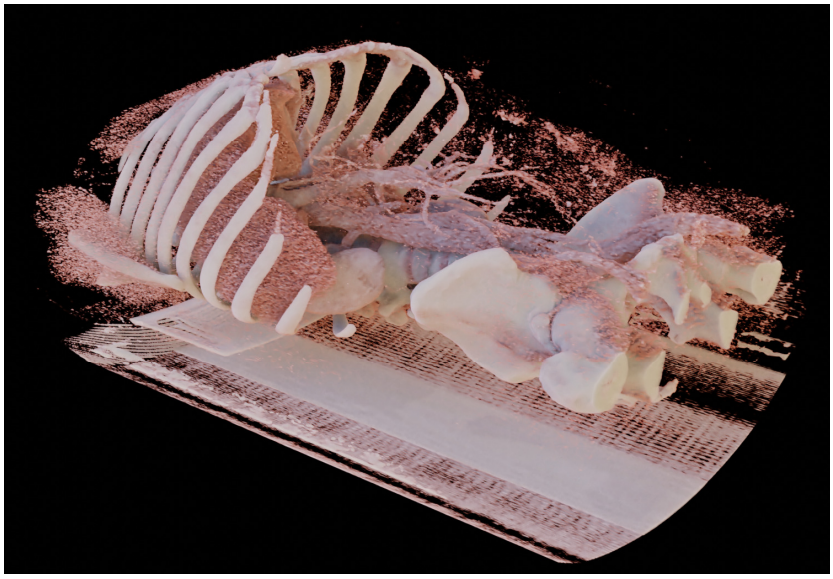

Figure 1.11: The volume rendering of computed tomography dataset using HDR environmental lighting in Blender

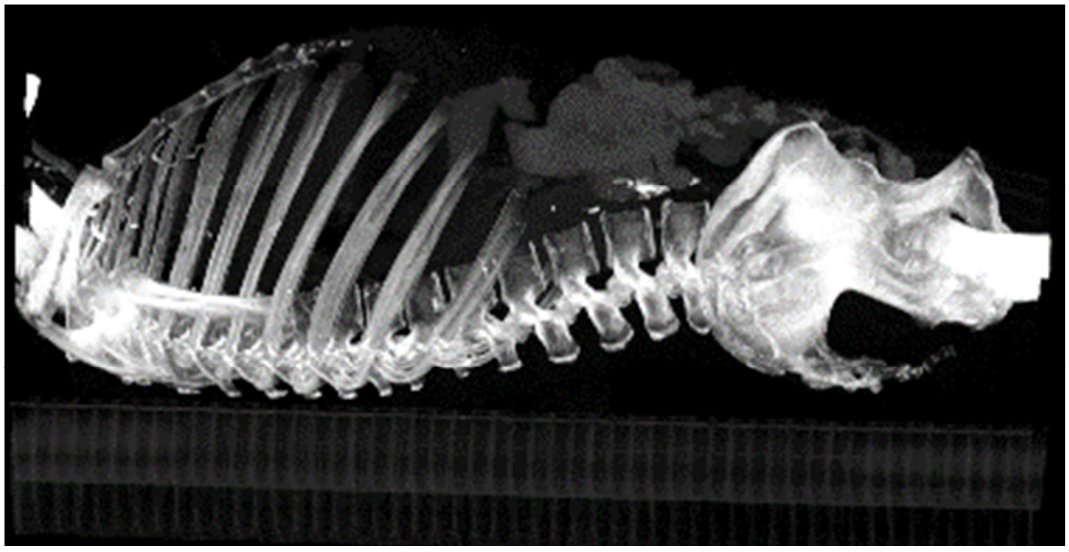

Figure 1.12: The volume rendering of computed tomography dataset using emission shader in Blender

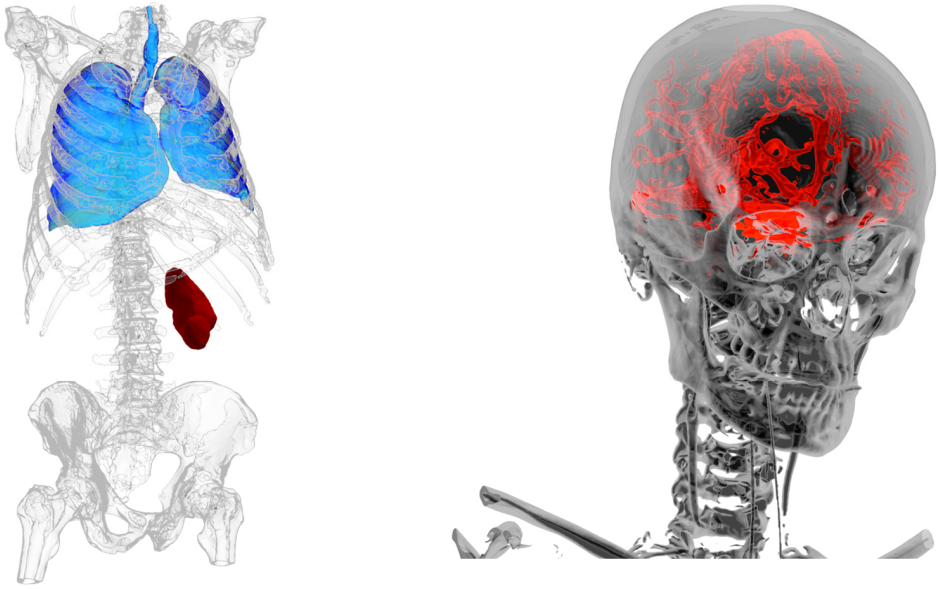

Figure 1.13: The rendering of computed tomography dataset after the segmentation method using refraction and transparent shader in Blender

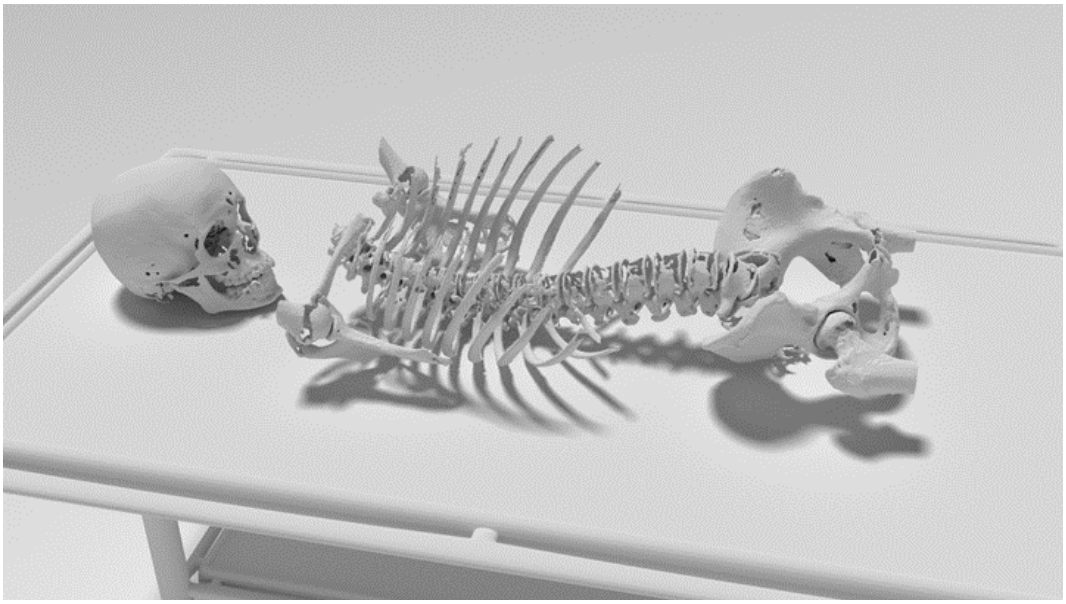

Figure 1.14: The rendering of computed tomography dataset after the segmentation method using only diffuse shader in Blender

Figure 1.15: The visualization of the computational fluid dynamics using the Covise editor and the cutting surface node in Blender

Another area where rendering is widely used is visualization of scientific data of large datasets (see Figures 1.15,1.16,1.17), which can come from simulations of various physical phenomena (e.g., fluid dynamics, structural analysis, etc.).

The purpose of our new Render-as-a-Service is to simplify the visualization process not only for researchers or students using our infrastructure but also for doctors, e.g., from the University Hospital Ostrava.

## 1.3 The goals of the thesis

The main goals of this thesis are as follows:

- Main scientific goal: development of novel method that enables rendering of massive scenes on multi-GPU systems,
- Secondary goal: development of Rendering-as-a Service toolchain based on the open-source Blender tool that enables simple usage of HPC resources to large community of Blender users.

Figure 1.16: The volume rendering of the computational fluid dynamics using the Covise editor and the isosurface node in Blender

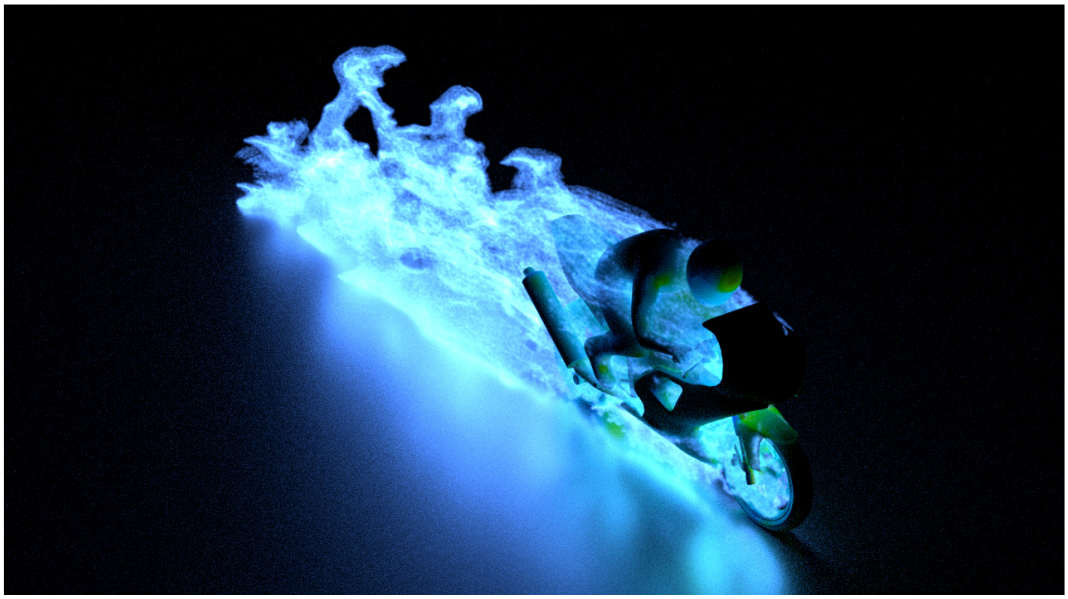

Figure 1.17: The rendering of the computational fluid dynamics using the Covise editor, Blender's particles and emission shader in Blender

# Chapter 2

# State-of-the-art

Our scientific contribution to the state-of-the-art in the area of parallel rendering is the development of the method that enables rendering of large scenes on multiple GPUs. The following sections describe the current state-of-the-art related to this topic.

## 2.1 Techniques for rendering of massive scenes on GPUs

The majority of film production rendering systems rely on path tracing. These systems almost exclusively use CPUs to deliver the final image [29, 27, 25, 28, 26] as their scenes are too large to fit in the limited memory of a GPU. There are two possible solutions for the limited local memory: (i) out-of-core rendering and (ii) distributed or parallel rendering. We map both approaches, but more attention is given to the distributed rendering since our proposed solution belongs to this area.

### 2.1.1 Out-of-core rendering

The GPU accelerated rendering of large scenes is mostly handled using out-of-core techniques. Budge et al. [64] presented an approach that dealt with out-of-core data management for heterogeneous architectures consisting of multiple CPU/GPU nodes. Son et al. [65] addressed the problem of path tracing on various sets of device configurations. Their approach supported out-of-core rendering for heterogeneous clusters with CPU/GPU nodes and network interconnection.

In terms of multi-GPU rendering, Zhou et al. [66] presented a photorealistic rendering system that used an out-of-core approach for textures together with dynamic scheduling for efficient parallelization. As it implemented REYES rendering on GPUs [47], it mainly contributed to rasterization rather than path tracing. Pantaleoni et al. [67] proposed a rendering system capable of fast lighting iterations over large scenes. It used an out-of-core GPU ray tracing algorithm for the computation of directional occlusion and spherical integrals. Wang

et al. [68] suggested a GPU rendering framework that can handle massive scenes with out-of-core geometry and complex lighting. The solution was designed to use only one GPU and applied a many-lights method instead of path tracing.

In the area of commercial GPU renderers, Redshift [8] is available as a ray tracer with a versatility of features including support for out-of-core geometry and textures. Redshift is able to use CPU memory in cases where the number of polygons or the size of the textures exceed the capacity of the GPU memory. Although it supports rendering on multiple GPUs, it is still not able to combine the memory of multiple GPUs into one large memory space.

It is important to also mention the area of ray statistics that can influence the creation of acceleration data structures in ray tracing. In [69, 70] the authors provide methods for constructing acceleration data structures, such as Bounding Volume Hierarchy (BVH), that are further compared to the standard approach of the Surface Area Heuristic (SAH). Both techniques use a priori knowledge about rays. Feltman et al. [70] focus on cost reduction by tracing only the shadow rays. Bittner et al. [69] provide a better solution compared to standard SAH where only primary rays are cast. If shadow rays and secondary rays are considered, then SAH performs better due to the rather uniform ray distribution in the scene which SAH is expecting.

### 2.1.2 Distributed rendering

There is a large body of work on distributed rendering techniques for massive scenes on distributed memory systems. In the standard classification proposed by Molnar et al. [71] these works fall into one of these categories: sort-first, which results in image-based partitioning, sort middle (related to rasterization only), and sort-last, which uses scene data distribution.

#### 2.1.2.1 Image-Parallel Rendering

Sort-first methods are based on screen-partitioning, and the workload is distributed among processors or machines per blocks of pixels of a rendered image. In the most common as well as the most efficient way, the scene data is fully replicated in all local memories, and ray tracing is embarrassingly parallel.

In the case of ray tracing complex scenes that do not fit into local memory, this approach results in on-demand scene data movement while rays remain fixed [72, 73, 74, 75]. Our proposed solution is based on scene data communication while rays never leave the GPU they are created on.

#### 2.1.2.2 Data-Parallel Rendering

Sort-last methods, also called data-parallel, distribute the workload by subdividing the scene data. In the case of distributed ray tracing, these approaches transfer ray data among proces-

sors or machines, while scene data do not move after the initial distribution. Kato et al. [76] used this approach for production rendering on a cluster of workstations. More recently, this approach was used in the field of scientific visualizations of massive data sets from scientific simulations on supercomputers [77]. Navratil et al. [78, 79] proposed a hybrid (combination of image-parallel and data-parallel) approach for distributed memory supercomputers that used dynamic ray scheduling. The proposed dynamic solutions worked well, except in situations where ray communication costs exceeded data load costs such as in high quality rendering at high resolution.

## 2.2 Hardware: Shared Memory Multi-GPU Systems

### 2.2.1 Distributed Shared Memory Systems

During the late 1990s, Shared Memory Processors (SMP) with large amounts of Distributed Shared Memory (DSM) became commercially available [80]. In such machines, each processor has a local memory with caches, and a hardware or software layer transparently creates an illusion of global shared memory for applications [81]. As multiple processors read and write from/to one shared memory, the key problem is preserving a coherent view of shared data, which in practice is solved by various cache-coherent mechanisms [82].

DSM systems exhibit Non-Uniform Memory Access (NUMA), as the latency to access remote data is considerably larger than the latency to access local data. On such machines, good data locality is therefore critical for high performance. There were several methods that improved data locality over basic cache coherency using replication/migration techniques as summarized in [83]. This is also the key concept of our proposed approach. In addition to data replication, the performance of DSM systems can also be improved by application-specific techniques to reduce memory bandwidth requirements. For ray tracing of incoherent rays, such a technique was proposed by Aila and Karras [84].

In the past, large DSM systems such as SGI Origin 2000, KSR1, and Stanford DASH were used for ray tracing of large scenes, such as in Parker et al. [85], Keates and Hubbold [86], and Singh et al. [87], respectively. In these works, authors used screen partitioning, but proposed different solutions to work distribution, load balancing, and synchronization in the case of interactive rendering.

### 2.2.2 CUDA Unified Memory for Multi-GPU systems

NVIDIA's Unified Memory (UM) [31, 88] manages communication between multiple GPUs and CPUs transparently by adopting DSM techniques. Alternative approaches have also been proposed [89, 90, 91]. UM simplifies both out-of-core processing between GPUs and CPUs

as well as multi-GPU processing[1] and combinations of both. Previously, the applications focusing on large data processing on GPUs required algorithm-specific techniques for memory handling [92, 93, 94, 95, 96, 97, 98, 99].

In terms of HW technologies, NVLink interconnect is the key enabler of DSM multi-GPU systems. Li et al. [30] provided thorough evaluation of several variants of NVLink interconnects against PCIe bus. Chien et al. [100] evaluated the performance of advanced UM features such as prefetching and user-controlled data placement [101] on two different platforms, one with PCIe 3.0 interconnect between CPUs and GPUs and one with NVLink interconnect (based on Power9 CPU).

A critical mechanism for UM is prefetching, page-eviction due to memory over subscription, and page migration between GPUs. The works of [102, 103, 104, 105, 106] proposed new algorithms to improve UM performance in the case of transparent memory management. In contrast, our approach controls the page placement and replication manually based on analysis of memory access patterns. Several works from the CG area use UM on multi-GPU systems. Christensen et al. [107] efficiently used NVLink for image composition after distributed rendering on up to 8 GPUs. Kim et al. [108] utilized multiple GPUs for scalable split frame rendering (SFR), which assigns disjoint regions of a frame to different GPUs. Finally, Xie et al. [109] used a multi-GPU system for rendering for Virtual Reality systems. They exploited the data locality of scene objects to reduce inter-GPU memory traffic.

### 2.2.3 Selected Multi-GPU HPC platforms

Most large companies have their own rendering farms such as Disney or Dreamworks, and even car manufacturers have their own computer clusters that they use for rendering. However, there are rendering farms (e.g. [17]) that are equipped with rendering programs and offer their computational time to render. Qarnot Computing [110] introduced a very interesting remote rendering solution, which installs high-performance silent computers in its radiators and uses waste heat to heat rooms. Another way to render is to use the high-performance supercomputers with accelerators available in Europe.

Accelerators are also used to speed up computationally intensive tasks. These accelerators can also increase the computing power of HPC clusters several times. There are several types of accelerators, the best known of which are graphic accelerators (GPUs), programmable gate arrays (FPGAs), and MICs (Many Integrated Core) processors. In this work, we focused specifically on Intel MIC Xeon Phi processors and NVIDIA graphic cards.

As a researcher working in the European HPC environment I had the opportunity to work not only on the Salomon and Anselm supercomputers [111], which were part of IT4Innovations infrastructure but also on other supercomputers such as the Italian Marconi machine installed

[1] All GPUs must be in a single server.

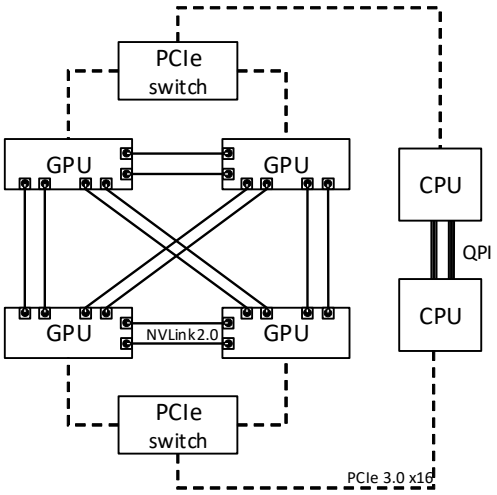


Figure 2.1: NVLink GPU interconnect in the Barbora GPU server

in Cineca [112] or the German HLRN Cray System [113]. All of these supercomputers were equipped with Intel MIC Xeon Phi processors. Salomon was equipped with hundreds of Intel MIC Xeon Phi processors codenamed Knights Corner (KNC), which we used to test Blender for the Blender Institute [2]. Cineca and HLRN were equipped with newer Intel MIC Xeon Phi processors codenamed Knights Landing (KNL).

As part of IT4Innovations, there are several clusters which contain state-of-the-art multi-GPU systems: NVIDIA DGX-2 (16x NVIDIA V100), the GPU accelerated nodes (4x NVIDIA V100) of the Barbora supercomputer, and the GPU accelerated nodes (8x NVIDIA A100) of the Karolina supercomputer. We had an opportunity to work on the DGX-A100 system in Altair Engineering GmbH. DGX-A100 contains 8x NVIDIA A100 cards, which is similar to the accelerated node configuration of Karolina.

#### 2.2.3.1 Barbora GPU Server Hardware Description

Developed for HPC infrastructures, the BullSequana X410 E5 is a dense GPU-accelerated compute node, which contains two CPUs and 4 Tesla V100 GPUs, each with 16 GB of fast HBM2 memory. For the purpose of this work, the key part is the NVLink 2.0 [114] interconnect that connects all GPUs as shown in Figure 2.1.

Tesla V100 contains 6 NVLink 2.0 ports, each capable of a theoretical bandwidth of 25 GB/s in each direction. Each GPU is connected to 3 other GPUs and there are two links between any pair of GPUs. This provides a total bandwidth of 50 GB/s. This is the theoretical bandwidth at which one GPU can read/write data from/to another GPU memory.

The actual remote memory bandwidth measured by the STREAM benchmark [115] is 48 GB/s for all combinations of GPUs, as shown in Figure 2.2. The latency is approximately

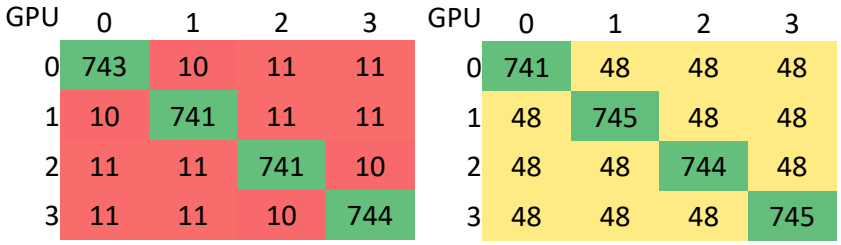


Figure 2.2: Memory bandwidth over PCI-Express (left) and over NVLink (right) in the Barbora GPU server

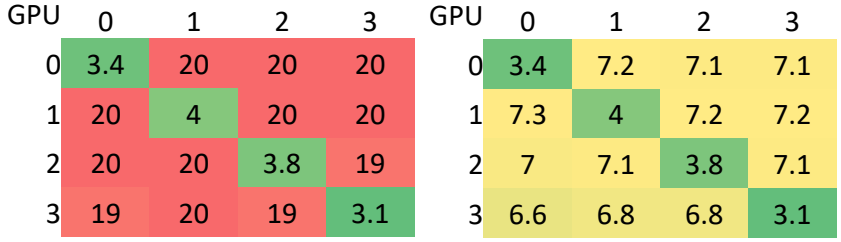


Figure 2.3: Memory access latency over PCI-Express (left) and over NVLink (right) in the Barbora GPU server

7 $\mu$s (see Figure 2.3). The local memory bandwidth is 743 GB/s, which is 15.5 times higher than the remote memory bandwidth. This work shows that even though the remote memory bandwidth is 2.9 times lower than in the case of DGX-2 (see next Section 2.2.3.2), it is still sufficient for our approach if path-tracing uses 4 GPUs only.

Comparison of bandwidth and latency for accessing remote memory over PCI-Express 3.0 (approximately 11 GB/s and 20 $\mu$s) and NVLink 2.0 (approx. 48 GB/s and 7 $\mu$s) for all combinations of GPUs on a Barbora node. Memory bandwidth and latency to local memory is approximately 743 GB/s and 3.6 $\mu$s, respectively, and is on the diagonal of the matrices.

#### 2.2.3.2 NVIDIA DGX-2 Hardware Description

There are multiple platforms on the market that are equipped with multiple GPUs connected using NVLink [114, 116]. However, currently only NVIDIA HGX-2 and DGX-2 platforms use NVSwitch [32] based interconnect to connect 16 Tesla V100 GPUs. A theoretical bandwidth between any pair of GPUs is 150 GB/s in one direction (bidirectional bandwidth is 300 GB/s).[2]

Because of its advanced interconnect, any set of 8 GPUs can communicate with the remaining 8 GPUs in parallel, and any pair of GPUs communicate at the full 300 GB/s bandwidth. This is facilitated by the non-blocking Fat tree topology of the NVSwitch network; see Figure 2.4. This unique hardware feature is also an essential one for the approach presented in this work for the following reasons: (i) it enables fast access for a GPU to the memory of

[2]This bandwidth is achieved by combining all 6 NVLink 2.0 links available at the Tesla V100.

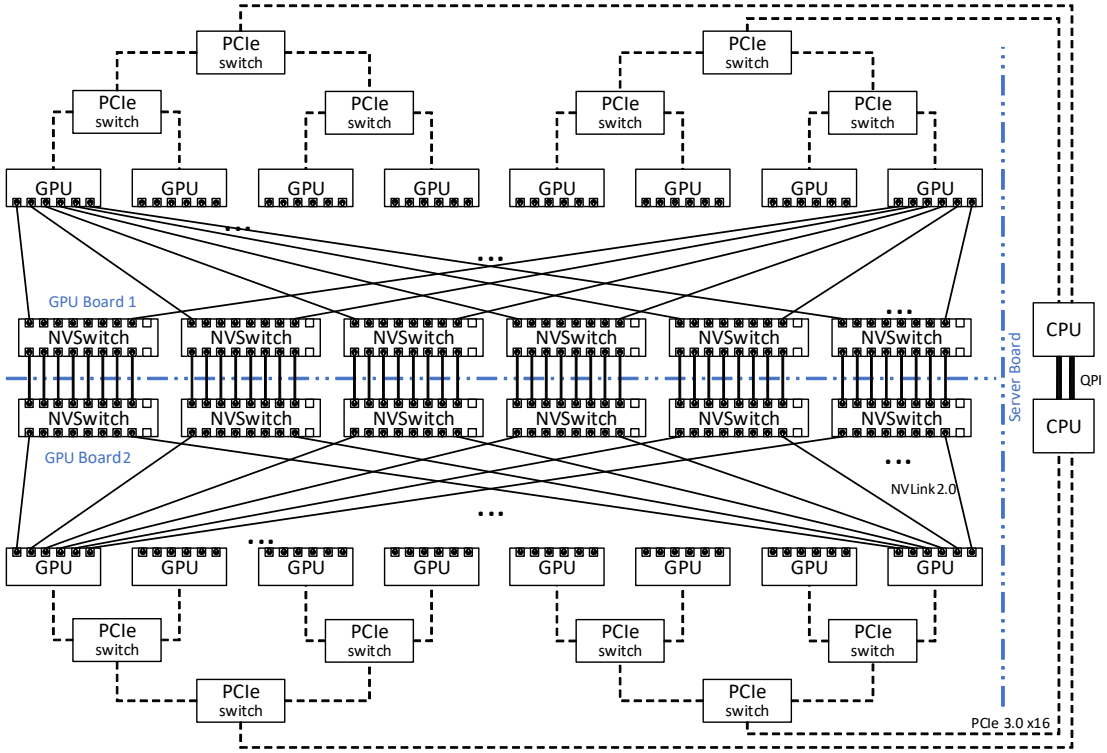


Figure 2.4: NVLink GPU interconnect in the DGX-2 system

another GPU, and (ii) with UM technology (see Section 2.2.2 for more details) it gives a code developer 512 GB of global GPU shared memory.

Real memory bandwidth and latency to the local and the remote memory was measured using a specialized benchmark [117] for both NVLink 2.0 and PCI-Express 3.0[3] buses. The results for NVLink and for PCI-Express are shown in Figures 2.5, 2.6, 2.7, 2.8. The values on the diagonal represent performance of the local memory, and off-diagonal values represent access to the remote memory. One can see that the local memory has approximately 5.7 times higher bandwidth and 3 times lower latency when compared to reading the remote memory over NVLink. Moreover, the bandwidth of NVLink is 12.5 times higher than the bandwidth of PCI-Express, and the latency is 2.5 times lower. When compared to state-of-the-art CPUs (with 8 memory channels) with a memory bandwidth which, when measured by the STREAM benchmark [115], is approximately 147 GB/s, one can see that this bandwidth is only 6 % higher than the real NVLink bandwidth [118].

These values suggest that the technology has reached the point where data distributed GPU path tracing becomes possible. However, the hardware is not a panacea. It is still necessary to minimize the negative impact of remote memory access as much as possible to

[3]Using 16 PCI-Express lanes.

| GPU | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 795 | 8 | 10 | 10 | 11 | 11 | 11 | 11 | 11 | 10 | 11 | 11 | 10 | 10 | 11 | 11 |
| 1 | 8 | 796 | 10 | 10 | 11 | 11 | 11 | 11 | 11 | 11 | 11 | 11 | 11 | 11 | 11 | 11 |
| 2 | 10 | 10 | 790 | 9 | 11 | 11 | 11 | 11 | 11 | 11 | 11 | 11 | 11 | 11 | 11 | 11 |
| 3 | 10 | 10 | 10 | 796 | 11 | 11 | 11 | 11 | 10 | 11 | 10 | 10 | 11 | 11 | 10 | 10 |
| 4 | 11 | 11 | 11 | 11 | 791 | 8 | 10 | 10 | 11 | 11 | 11 | 11 | 11 | 11 | 11 | 11 |
| 5 | 11 | 11 | 11 | 11 | 8 | 800 | 10 | 10 | 11 | 11 | 11 | 11 | 11 | 11 | 11 | 11 |
| 6 | 11 | 11 | 11 | 11 | 9 | 9 | 793 | 9 | 11 | 10 | 11 | 11 | 11 | 10 | 11 | 11 |
| 7 | 11 | 11 | 11 | 11 | 10 | 10 | 10 | 795 | 11 | 11 | 11 | 11 | 10 | 11 | 11 | 11 |
| 8 | 11 | 11 | 11 | 11 | 11 | 11 | 11 | 11 | 792 | 8 | 10 | 10 | 11 | 11 | 11 | 11 |
| 9 | 11 | 11 | 11 | 11 | 11 | 11 | 11 | 11 | 8 | 796 | 10 | 10 | 11 | 11 | 11 | 11 |
| 10 | 11 | 11 | 11 | 11 | 11 | 11 | 11 | 11 | 9 | 10 | 792 | 10 | 11 | 11 | 11 | 11 |
| 11 | 11 | 11 | 11 | 11 | 11 | 11 | 11 | 11 | 10 | 10 | 9 | 797 | 11 | 11 | 11 | 11 |
| 12 | 11 | 11 | 11 | 11 | 11 | 11 | 11 | 11 | 11 | 11 | 11 | 11 | 792 | 8 | 9 | 9 |
| 13 | 11 | 11 | 11 | 11 | 11 | 11 | 11 | 11 | 11 | 11 | 11 | 11 | 8 | 799 | 10 | 10 |
| 14 | 11 | 11 | 11 | 11 | 11 | 11 | 11 | 11 | 11 | 11 | 11 | 11 | 10 | 10 | 794 | 10 |
| 15 | 11 | 11 | 11 | 11 | 11 | 11 | 11 | 11 | 11 | 11 | 11 | 11 | 10 | 9 | 9 | 800 |

Figure 2.5: Memory bandwidth over PCI-Express bus in the DGX-2 system

achieve good path tracing performance and scalability, as shown in Section 3.4.2.

Comparison of bandwidth and latency for accessing remote memory over PCI-Express 3.0 (approximately 11 GB/s and 25 $\mu$s and NVLink 2.0 (approx. 138 GB/s and 10 $\mu$s) for all combinations of GPUs on the NVIDIA DGX-2 server. Memory bandwidth and latency to local memory is approximately 800 GB/s and 3.5 $\mu$s) respectively, and is on the diagonal of the matrices.

#### 2.2.3.3 NVIDIA DGX-A100 Hardware Description

The DGX-A100 is the new generation of DGX-2. It is equipped with 8 Tesla A100-SXM4 GPUs. Tesla A100 contains 12 NVLink 3.0 ports, each capable of a theoretical bandwidth of 25 GB/s in each direction. DGX-A100 has 6 NVSwitches. Each GPU is connected to each NVSwitch with two NVLinks (see Figure 2.9). A100 GPU achieves 300 GB/s unidirectional peer-to-peer throughput and 600 GB/s in both directions over NVLink.

The actual remote memory bandwidth measured by the STREAM benchmark [115] is 255 GB/s for all combinations of GPUs, as shown in Figure 2.10. The latency is approximately 10 $\mu$s (see Figure 2.11). The local memory bandwidth is approximately 1193 GB/s.

Comparison of bandwidth and latency for accessing remote memory over PCI-Express (approximately 17 GB/s and 27 $\mu$s) and NVLink (approx. 255 GB/s and 10 $\mu$s) for all combinations of GPUs on the NVIDIA DGX-A100 server. Memory bandwidth and latency to local memory is approximately 1193 GB/s and 4.3 $\mu$s, respectively, and is on the diagonal of the matrices.

| GPU | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 790 | 138 | 138 | 138 | 138 | 138 | 137 | 138 | 138 | 137 | 138 | 137 | 138 | 138 | 138 | 138 |
| 1 | 138 | 795 | 138 | 138 | 137 | 138 | 138 | 138 | 138 | 138 | 138 | 138 | 137 | 138 | 138 | 138 |
| 2 | 138 | 138 | 792 | 138 | 137 | 138 | 138 | 138 | 138 | 138 | 138 | 138 | 138 | 138 | 138 | 138 |
| 3 | 138 | 139 | 138 | 797 | 138 | 138 | 138 | 138 | 138 | 138 | 138 | 138 | 138 | 138 | 138 | 138 |
| 4 | 138 | 138 | 138 | 138 | 790 | 138 | 138 | 138 | 138 | 138 | 138 | 138 | 138 | 139 | 138 | 138 |
| 5 | 138 | 138 | 138 | 138 | 138 | 797 | 138 | 138 | 138 | 138 | 138 | 137 | 138 | 138 | 138 | 138 |
| 6 | 138 | 138 | 137 | 138 | 138 | 137 | 792 | 137 | 138 | 138 | 139 | 137 | 138 | 138 | 138 | 138 |
| 7 | 138 | 138 | 138 | 138 | 137 | 138 | 138 | 797 | 138 | 138 | 138 | 138 | 138 | 138 | 138 | 138 |
| 8 | 138 | 138 | 138 | 138 | 138 | 138 | 138 | 138 | 790 | 138 | 138 | 138 | 138 | 138 | 138 | 138 |
| 9 | 139 | 138 | 138 | 138 | 138 | 138 | 138 | 138 | 138 | 798 | 138 | 138 | 138 | 138 | 139 | 139 |
| 10 | 139 | 138 | 138 | 138 | 139 | 138 | 138 | 138 | 138 | 138 | 795 | 138 | 138 | 138 | 138 | 138 |
| 11 | 139 | 138 | 138 | 138 | 138 | 138 | 138 | 138 | 138 | 138 | 139 | 800 | 138 | 139 | 138 | 138 |
| 12 | 139 | 138 | 138 | 139 | 138 | 138 | 138 | 138 | 138 | 138 | 138 | 138 | 793 | 138 | 139 | 138 |
| 13 | 138 | 138 | 138 | 138 | 138 | 138 | 139 | 138 | 138 | 138 | 138 | 138 | 138 | 798 | 139 | 138 |
| 14 | 138 | 138 | 138 | 138 | 138 | 139 | 138 | 138 | 138 | 138 | 139 | 138 | 138 | 138 | 796 | 138 |
| 15 | 138 | 138 | 138 | 138 | 138 | 138 | 139 | 138 | 138 | 138 | 138 | 139 | 138 | 138 | 138 | 798 |

Figure 2.6: Memory bandwidth over NVLink bus in the DGX-2 system

| GPU | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 3,9 | 25 | 25 | 24 | 25 | 25 | 25 | 25 | 25 | 25 | 25 | 25 | 25 | 25 | 25 | 25 |
| 1 | 24 | 3,2 | 24 | 23 | 24 | 24 | 24 | 24 | 24 | 25 | 25 | 25 | 25 | 24 | 24 | 24 |
| 2 | 23 | 24 | 3,2 | 23 | 24 | 24 | 24 | 24 | 24 | 25 | 24 | 25 | 25 | 24 | 24 | 24 |
| 3 | 24 | 24 | 24 | 3,3 | 24 | 24 | 24 | 24 | 25 | 25 | 25 | 25 | 25 | 25 | 25 | 25 |
| 4 | 24 | 24 | 24 | 23 | 3,3 | 24 | 24 | 24 | 25 | 25 | 25 | 25 | 25 | 24 | 25 | 24 |
| 5 | 24 | 24 | 24 | 24 | 24 | 3,3 | 24 | 24 | 25 | 25 | 25 | 25 | 25 | 24 | 25 | 24 |
| 6 | 23 | 24 | 24 | 23 | 24 | 24 | 3,3 | 24 | 25 | 25 | 25 | 25 | 25 | 24 | 24 | 24 |
| 7 | 23 | 24 | 24 | 23 | 24 | 24 | 24 | 3,3 | 24 | 25 | 25 | 25 | 25 | 24 | 24 | 24 |
| 8 | 25 | 25 | 25 | 24 | 25 | 25 | 25 | 25 | 3,5 | 25 | 25 | 25 | 25 | 25 | 25 | 25 |
| 9 | 25 | 25 | 25 | 25 | 25 | 25 | 25 | 25 | 25 | 4,2 | 26 | 26 | 26 | 26 | 26 | 26 |
| 10 | 25 | 25 | 26 | 25 | 26 | 26 | 26 | 26 | 26 | 26 | 4,2 | 26 | 26 | 26 | 26 | 26 |
| 11 | 25 | 25 | 26 | 25 | 25 | 26 | 26 | 26 | 26 | 26 | 26 | 4,2 | 26 | 26 | 26 | 26 |
| 12 | 25 | 25 | 26 | 25 | 26 | 26 | 25 | 26 | 26 | 26 | 26 | 26 | 4,2 | 26 | 26 | 26 |
| 13 | 26 | 26 | 26 | 25 | 26 | 26 | 26 | 26 | 26 | 26 | 26 | 25 | 25 | 4,3 | 26 | 26 |
| 14 | 25 | 26 | 26 | 25 | 26 | 26 | 26 | 26 | 26 | 26 | 26 | 26 | 25 | 25 | 4,3 | 26 |
| 15 | 25 | 25 | 26 | 25 | 25 | 25 | 25 | 25 | 26 | 26 | 26 | 26 | 26 | 25 | 25 | 4,2 |

Figure 2.7: Memory access latency over PCI-Express in the DGX-2 system

| GPU | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 3,9 | 9,6 | 9,4 | 9,4 | 9,5 | 9,4 | 9,5 | 9,5 | 9,7 | 9,7 | 9,8 | 9,6 | 9,5 | 9,7 | 9,4 | 9,7 |
| 1 | 9,7 | 3,2 | 9,6 | 9,5 | 9,6 | 9,5 | 9,5 | 9,5 | 9,7 | 9,7 | 9,7 | 9,6 | 9,5 | 9,7 | 9,5 | 9,7 |
| 2 | 9,6 | 9,6 | 3,2 | 9,5 | 9,4 | 9,5 | 9,4 | 9,4 | 9,6 | 9,5 | 9,5 | 9,6 | 9,4 | 9,6 | 9,4 | 9,5 |
| 3 | 9,7 | 9,7 | 9,6 | 3,2 | 9,6 | 9,5 | 9,5 | 9,5 | 9,7 | 9,6 | 9,6 | 9,6 | 9,5 | 9,7 | 9,4 | 9,5 |
| 4 | 9,7 | 9,6 | 9,6 | 9,5 | 3,2 | 9,4 | 9,5 | 9,5 | 9,6 | 9,8 | 9,8 | 9,6 | 9,5 | 9,7 | 9,5 | 9,7 |
| 5 | 9,8 | 9,7 | 9,6 | 9,5 | 9,5 | 3,2 | 9,5 | 9,5 | 9,5 | 9,7 | 9,7 | 9,6 | 9,5 | 9,7 | 9,5 | 9,7 |
| 6 | 9,7 | 9,6 | 9,5 | 9,5 | 9,4 | 9,4 | 3,2 | 9,4 | 9,6 | 9,6 | 9,5 | 9,6 | 9,4 | 9,5 | 9,4 | 9,5 |
| 7 | 9,6 | 9,5 | 9,4 | 9,3 | 9,4 | 9,5 | 9,4 | 3,2 | 9,6 | 9,7 | 9,6 | 9,6 | 9,4 | 9,6 | 9,4 | 9,5 |
| 8 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 3,4 | 10 | 10 | 10 | 10 | 10 | 10 | 10 |
| 9 | 10 | 10 | 9,8 | 9,7 | 9,9 | 9,7 | 9,8 | 9,9 | 9,7 | 4,1 | 9,8 | 10 | 10 | 9,9 | 9,9 | 9,8 |
| 10 | 10 | 10 | 9,8 | 9,8 | 9,8 | 9,7 | 9,8 | 9,8 | 9,7 | 9,8 | 4,1 | 10 | 10 | 9,9 | 9,9 | 9,8 |
| 11 | 9,9 | 9,7 | 9,7 | 9,7 | 9,9 | 9,7 | 9,9 | 9,9 | 9,7 | 9,8 | 9,8 | 4,1 | 10 | 9,9 | 9,9 | 9,8 |
| 12 | 10 | 9,8 | 9,7 | 9,7 | 9,9 | 9,8 | 9,8 | 9,9 | 9,8 | 9,9 | 9,8 | 10 | 4,2 | 9,9 | 9.9 | 9,8 |
| 13 | 10 | 9,7 | 9,7 | 9,7 | 9,8 | 9,7 | 9,8 | 9,8 | 9,7 | 9,9 | 9,8 | 10 | 10 | 4,1 | 9,9 | 9,8 |
| 14 | 9,9 | 9,7 | 9,7 | 9,7 | 9,8 | 9,7 | 9,8 | 9,8 | 9,7 | 9,8 | 9,7 | 9,9 | 10 | 9,9 | 4,1 | 9,8 |
| 15 | 9,9 | 9,7 | 9,7 | 9,7 | 9,8 | 9,7 | 9,8 | 9,8 | 9,7 | 9,9 | 9,8 | 10 | 10 | 9,9 | 9,9 | 4,1 |

Figure 2.8: Memory access latency over NVLink in the DGX-2 system

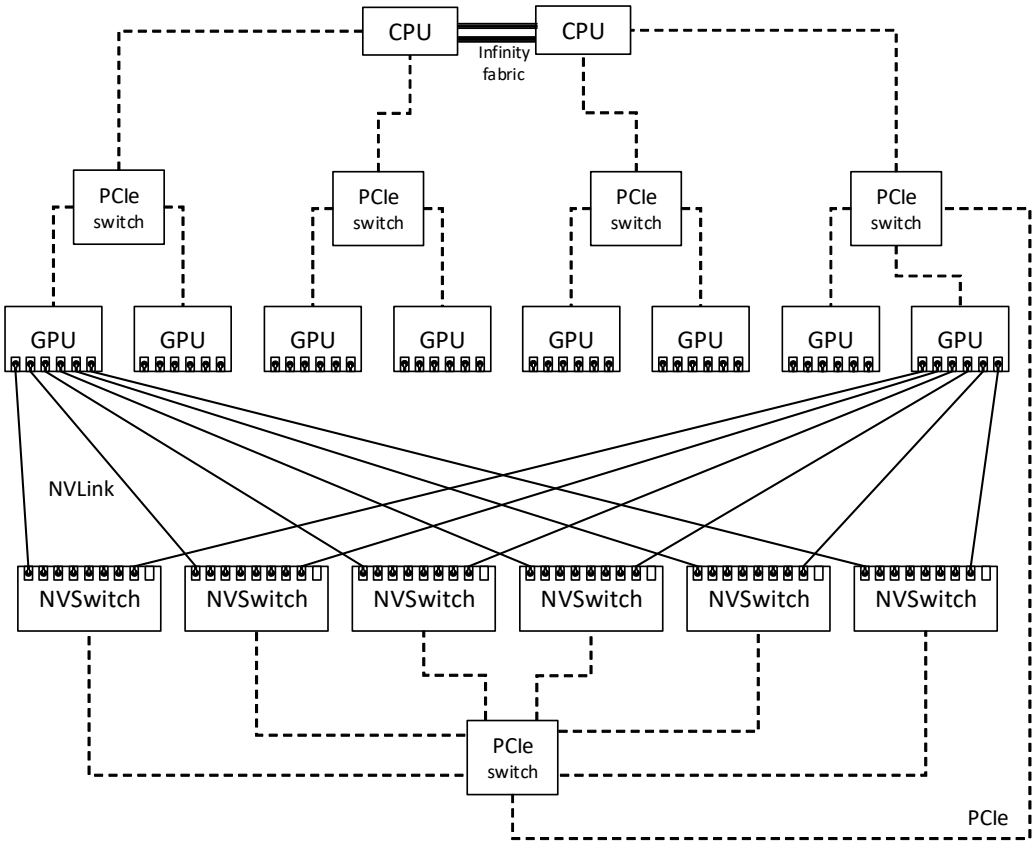


Figure 2.9: NVLink GPU interconnect in the DGX-A100 system

| GPU | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| 0 | 1136 | 16 | 18 | 18 | 16 | 18 | 17 | 18 |
| 1 | 17 | 1163 | 18 | 18 | 16 | 18 | 18 | 18 |
| 2 | 18 | 18 | 1163 | 17 | 17 | 18 | 18 | 18 |
| 3 | 18 | 18 | 17 | 1163 | 17 | 18 | 18 | 18 |
| 4 | 17 | 18 | 17 | 17 | 1152 | 15 | 17 | 17 |
| 5 | 17 | 17 | 17 | 17 | 15 | 1159 | 17 | 17 |
| 6 | 17 | 17 | 17 | 17 | 17 | 17 | 1149 | 14 |
| 7 | 17 | 18 | 17 | 17 | 17 | 17 | 14 | 1198 |

| GPU | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| 0 | 1180 | 244 | 255 | 251 | 255 | 255 | 249 | 255 |
| 1 | 251 | 1202 | 256 | 245 | 256 | 256 | 252 | 257 |
| 2 | 248 | 256 | 1195 | 255 | 252 | 255 | 255 | 248 |
| 3 | 252 | 257 | 257 | 1198 | 253 | 255 | 255 | 249 |
| 4 | 244 | 255 | 256 | 249 | 1173 | 254 | 249 | 253 |
| 5 | 251 | 256 | 255 | 251 | 256 | 1198 | 255 | 252 |
| 6 | 256 | 251 | 255 | 255 | 253 | 254 | 1195 | 248 |
| 7 | 257 | 256 | 248 | 255 | 257 | 251 | 255 | 1206 |

Figure 2.10: Memory bandwidth over PCI-Express (left) and over NVLink (right) in the DGX-A100 system

| GPU | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| 0 | 4.1 | 24.4 | 24.3 | 25.0 | 26.4 | 26.6 | 26.1 | 27.3 |
| 1 | 26.2 | 4.1 | 26.5 | 26.4 | 26.2 | 26.5 | 26.4 | 26.9 |
| 2 | 26.9 | 26.7 | 4.1 | 26.5 | 26.6 | 26.7 | 26.4 | 26.8 |
| 3 | 26.9 | 26.8 | 26.8 | 4.0 | 26.0 | 26.7 | 26.4 | 27.1 |
| 4 | 27.7 | 27.7 | 27.5 | 27.7 | 4.3 | 28.2 | 28.1 | 28.1 |
| 5 | 27.6 | 27.6 | 27.4 | 27.5 | 28.5 | 4.3 | 28.2 | 28.1 |
| 6 | 27.5 | 27.7 | 27.3 | 27.4 | 28.6 | 28.6 | 4.4 | 28.5 |
| 7 | 27.5 | 27.7 | 27.5 | 27.6 | 28.7 | 28.7 | 28.7 | 4.3 |

| GPU | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| 0 | 4.1 | 11.1 | 11.1 | 10.2 | 10.0 | 9.9 | 9.9 | 10.0 |
| 1 | 11.1 | 4.2 | 11.1 | 9.6 | 9.7 | 9.6 | 9.8 | 9.7 |
| 2 | 11.0 | 11.0 | 4.1 | 10.0 | 10.0 | 9.9 | 9.9 | 9.9 |
| 3 | 11.1 | 10.1 | 9.6 | 4.2 | 10.0 | 10.0 | 9.9 | 10.0 |
| 4 | 11.6 | 10.0 | 9.7 | 9.8 | 4.4 | 9.7 | 9.7 | 9.7 |
| 5 | 11.7 | 10.1 | 9.8 | 9.8 | 9.7 | 4.4 | 9.7 | 9.7 |
| 6 | 11.6 | 11.3 | 10.6 | 9.8 | 9.8 | 9.9 | 4.4 | 9.7 |
| 7 | 11.7 | 10.3 | 9.8 | 9.8 | 9.7 | 9.7 | 9.8 | 4.4 |

Figure 2.11: Memory access latency over PCI-Express (left) and over NVLink (right) in the DGX-A100 system

### 2.2.4 Intel Xeon Phi

The Intel Xeon Phi co-processor, in our case Knights Corner (KNC) based on the MIC architecture is composed of up to 61 low power cores in terms of both energy and performance when compared to multi-core CPUs. It can be used as a standalone Linux box or as an accelerator to the main CPU. The peak performance of the top-of-the line Xeon Phi is over 1.1 TFLOP ($10^{12}$ floating point operations per second) in double precision and over 2.2 TFLOPS in single precision. The MIC architecture could be programmed using both shared memory models such as OpenMP or OpenCL (provides compatibility with codes developed for GPU) and distributed memory models such as MPI.

Intel Xeon Phi can be used in three modes: 1) offload - one process runs on the host and the second process starts automatically on the MIC processor. Functions are called from the guest using Offload commands, 2) symmetric - this mode can be used for network communication using MPI, where we treat the MIC processor just like any other computer on the network, 3) native - compiled code can only run on Xeon Phi. For example, we can use the ssh command to connect directly to the MIC processor and run the application in the same way as on a standard Linux computer. The ray-tracing and path-tracing methods can be very easily parallelized because each pixel color can be calculated completely independently of the others. Theoretically, all points of the rendered image can be calculated simultaneously if the entire scene (all geometry, textures, etc.) is preloaded into the memory. The memory size has always been a big issue for final rendering, especially for graphic cards, where a scene can require up to 100GB of memory. However, this problem is addressed by the new Intel Xeon Phi MIC processors, codenamed Knights Landing (KNL), for which this condition is met. In addition to accelerators, we can also use network rendering to speed up the ray-tracing method, for example in HPC systems.

Salomon was equipped with 432 accelerated computing nodes, each with two Intel Xeon E5-2680v3 processors and two Intel Xeon Phi 7120P co-processors.

#### 2.2.4.1 Intel IMCI/AVX-512 Intrinsics Instructions

Optimization by vectorization can be achieved in several ways, e.g. by pragma omp simd or intrinsic instructions. The usage of OpenMP SIMD is very simple, but it only works with an Intel Compiler. The usage of intrinsic instructions on the other hand is very effective, works with a GCC compiler, but is quite difficult to provide. For such cases there is the possibility to use API (see Figure 2.12). Our CyclesPhi supports Intel Initial Many Core Instructions (IMCI) and Intel Advanced Vector Extensions 512 (AVX-512). For example, Intel Xeon Phi Knights Landing (KNL) and Skylake support AVX-512, while Intel Xeon Phi Knights Corner (KNC) supports IMCI. We have modified the API from Embree and used it in Blender. There are three main structures to work with: float (avx512f), integer (avx512i),

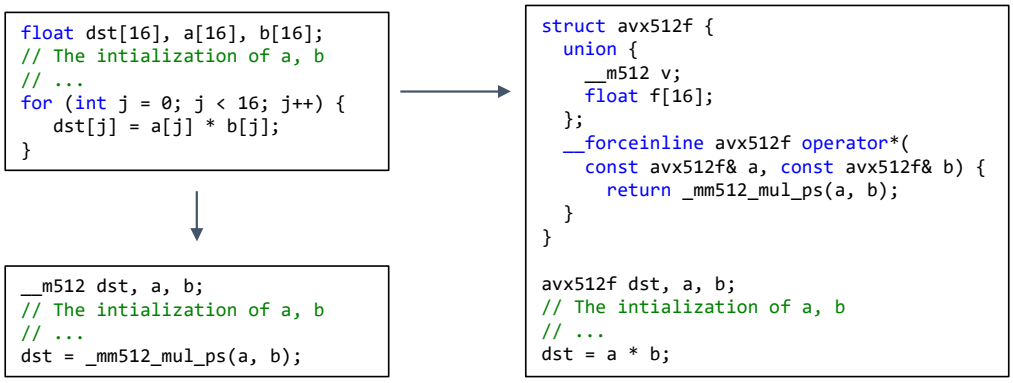


Figure 2.12: The example of overloading operators and hiding IMCI/AVX-512 intrinsic instructions

and boolean (avx512b) variables of 512 bits. SIMD vectorization is especially important for BVH traversal and helps to improve the performance of the rendering process.

# Chapter 3

# Methodology

In this chapter, we would like to introduce Blender 2.83 [2], how Blender Cycles is implemented and how we have extended Cycles to support HPC infrastructures, including support for rendering massive scenes on multi-GPU systems.

## 3.1 Blender

There are several software solutions that provide scene modeling and rendering in one package. In this work, we will focus on extending one of the most widely used tools, Blender.

Blender is the name of a project, open to everybody who wants to join and has experience with programming or 3D graphics. It has been founded by Tom Roosendaal. As part of this project, 3D open-source software has been developed. This software could be used to create pictures, movies and animations. In general, we could say that this software could be used in any discipline dealing with 3D graphics, from design through manufacturing of parts to medical applications such as visualization of 3D models of human organs.

Blender is entirely developed in Python, C and C++, which could be beneficial for developing new functions, features, or scripts or to modify the current ones. As Python is not very suitable for computationally extensive tasks such as rendering, Blender could be also extended using C++ coded extensions. This approach is used in this thesis.

Rendering is extremely time-consuming and thus parallelization is necessary. Blender uses mainly POSIX threads [119] for faster response and more user-friendly control of its GUI. Before rendering is executed, Blender creates three threads (see Figure 3.1): Blender Thread, Session Thread, and Device Threads.

The Blender Thread is the thread we are in when we receive render-engine callbacks. From this thread, we can safely access the Blender scene. The Session Thread updates the data from Blender, updates the scene, and distributes the tasks. It is located between the Device Threads that directly render the scene and the Blender Thread. The Device Threads

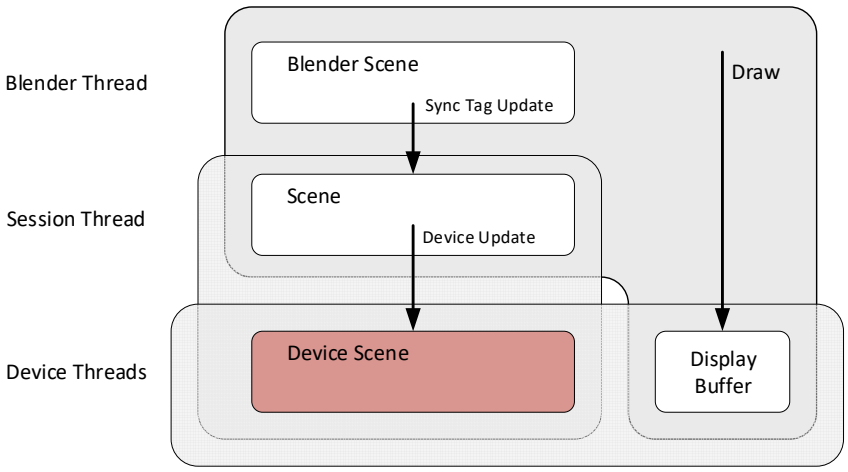


Figure 3.1: Blender's main threads: Blender Thread, Session Thread, and Device Threads

perform rendering of dedicated part of the scene and are maximally optimized for rendering performance. For CPU devices there are multiple such threads, while for GPU devices there is one thread per graphic card.

There are two main renderers in Blender, Eevee and Cycles. Eevee is a physically based real-time renderer. It uses OpenGL and the OpenGL driver takes care of GPU memory management. Eevee doesn't work very well with demanding scenes. We will focus on Cycles because this engine uses the raytracing method and state-of-the-art technologies can be used for memory management.

## 3.2 Cycles renderer

Blender has a production renderer called Cycles. It is an unbiased renderer based on unidirectional path tracing that supports CPU and GPU rendering. For ray tracing acceleration it uses a Bounding Volume Hierarchy (BVH). The BVH code is based on an implementation by NVIDIA [120, 121] with some additional code adaptation from Embree [122].

In terms of GPU rendering, Cycles supports CUDA, Optix, and OpenCL technology. CUDA and OpenCL implementations support the same features as a CPU one does, except for Open Shading Language (OSL). Optix support enables hardware acceleration of ray tracing on GPUs with RT cores, i.e., GPUs based on Turing architecture.

Blender Cycles supports multi-core processor systems or graphic accelerators (GPU) to ensure efficiency of both final (offline) and interactive rendering modes. The interactive rendering offers possibilities to change many things such as the position of a light source, geometry of the object, its material, etc., in real time without manually restarting the renderer. This

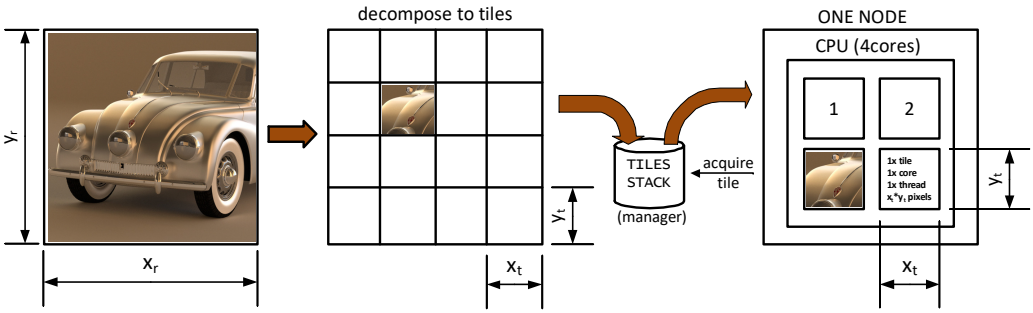


Figure 3.2: The decomposition of synthesized image with resolution x(r) x y(r) to tiles with size x(t) x y(t) by original implementation of the Blender Cycles for the CPU.

is opposite to the offline renderer, which is therefore used mainly for the final production rendering.

Cycles is directly integrated into the Blender environment and it is based on a path-tracing method that simulates the real behavior of light. Today, even the largest studios in their respective renderers, such as Disney in Hyperion [29], Pixar in RenderMan [27], and Solid Angle in Arnold [28], use the path-tracing method. Path-tracing algorithm is an embarrassingly parallel problem but only under the assumption that the entire scene description fits into a main CPU or GPU memory, for CPU or GPU based rendering, respectivelly.

The next section introduces the standard parallel mechanism in Blender where the scene is fully duplicated in memory and where tiles are used for parallelization. The way of the parallel rendering algorithm itself is executed inside a tile and it is different for each compute device. For CPU, POSIX threads are used. In contrast, OpenCL and CUDA technologies are used for GPU.

### 3.2.1 Parallelization by POSIX

The original implementation of the Blender Cycles for CPU uses POSIX threads [119] for parallelization. Parallelization is done in the following way: the synthesized image of the resolution x(r) by y(r) is decomposed into tiles of size x(t) by y(t). Each tile is then computed or rendered by one POSIX thread using one CPU core. The situation is shown in Figure 3.2.

CPU rendering is based on the CPUDevice class (see Figure 3.3) which we could divide into two parts: a part working with data (top part of the picture) and computation itself (lower part of the picture). In the middle, the computing units are depicted, in this case only CPU cores.

The rendered scene is stored in KernelData and KernelTextures structures. At first, the elementary information about the scene (for instance, the camera position or background information) is sent to the selected compute devices, in this case the CPUDevice, using const_copy_to method. After that the objects and textures are saved to the memory of

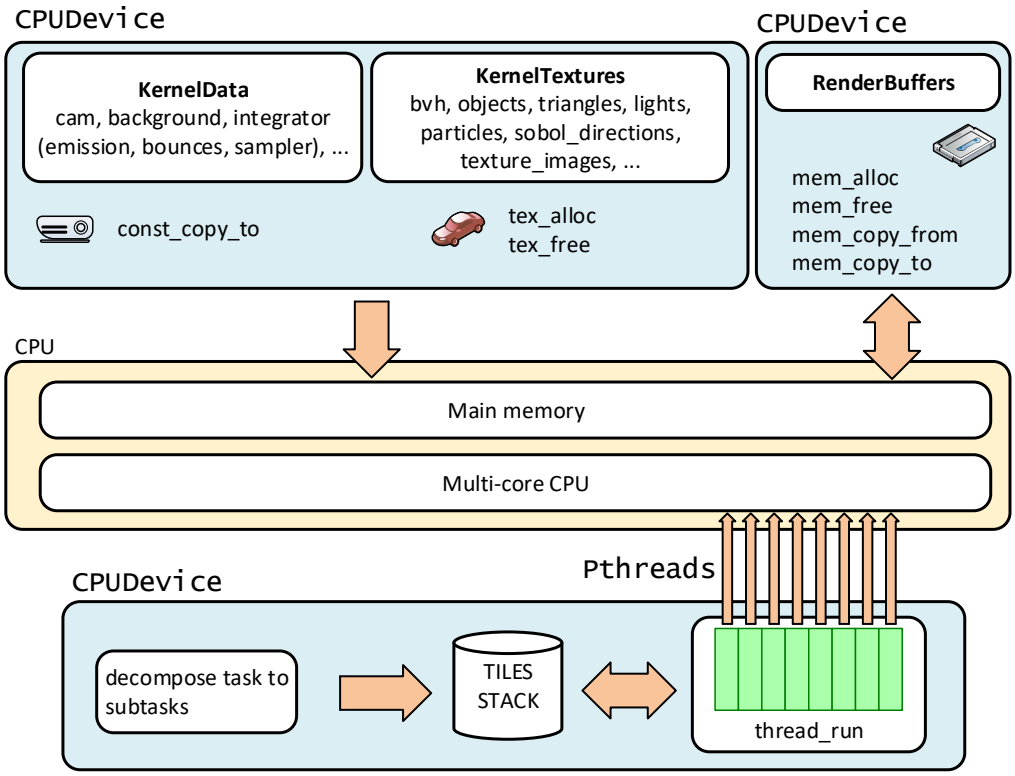


Figure 3.3: CPUDevice class for manipulating the main memory or for calling functions running on the CPU, all communication with the CPU and the main memory goes through this device interface.

the CPUDevice using tex_alloc method. In addition, one have to allocate a buffer for rendered pixels.

The next step is to decompose the rendered image into tiles. Each thread calls the path_tracing method to calculate the render equation using thread_run method. Finally, the results are stored to the buffer vector.

In the new version of CyclesX (Blender 3.0), the parallelized code was rewritten and Blender CyclesX uses Threading Building Blocks (TBB) [123] library for parallelization.

### 3.2.2 Parallelization by CUDA and OpenCL

The OpenCL compute device is used for multi-core CPUs as well as for MIC or GPU accelerators. Only one POSIX thread with a large tile for optimal performance was used. The parallelization is based on OpenCL threads where one thread is used for one pixel.

As the CUDA and OpenCL programing models are very similar, so is the decomposition. In this work CUDA technology is mainly uses. There is one POSIX thread for main computation per accelerator. On GPU a rendering kernel uses a single CUDA thread to render

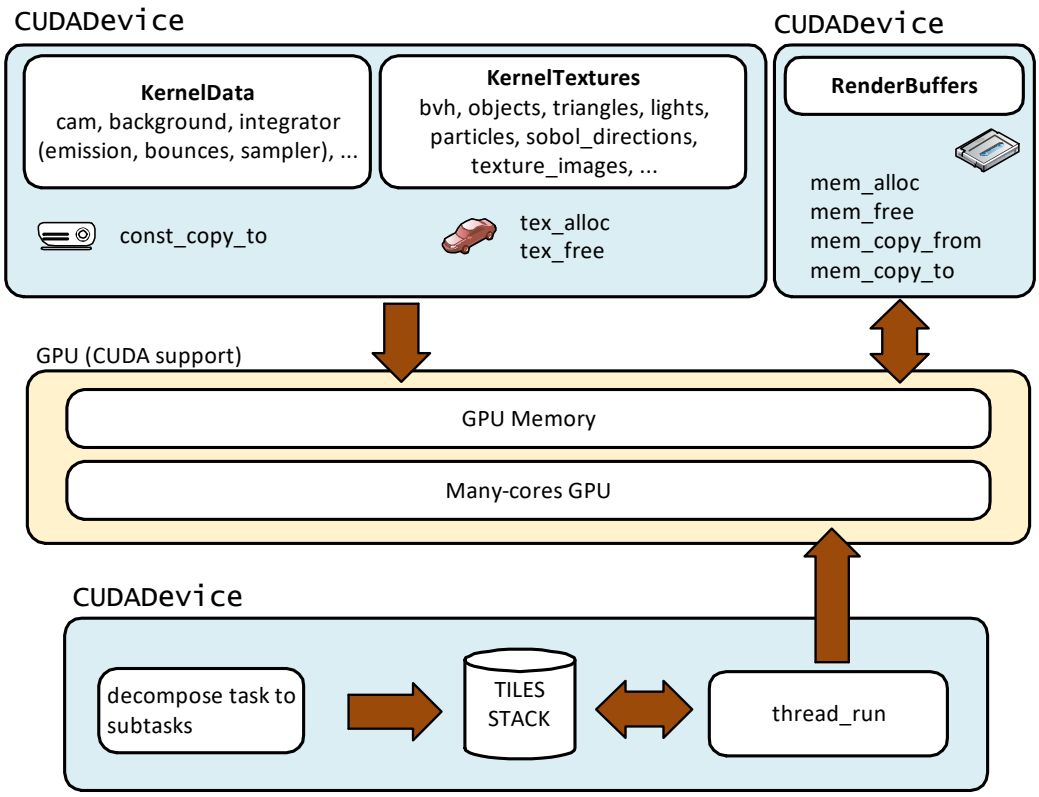


Figure 3.4: CUDADevice class for manipulating the GPU memory or for calling kernel functions running on the GPU, all communication with the GPU and its memory goes through this device interface.

a single pixel of a tile. The GPU needs a lot of threads for better performance. This is the reason why large tiles are needed.

GPU rendering is based on a CUDADevice class which could be divided into two parts (see Figure 3.4): a part working with data (the top part of the picture) and computation itself (the lower part of the picture). In the middle, computing units are depicted, and in this case we can only see one GPU in here.

The principle of GPU computing is very similar to CPU rendering in the original code. The rendered scene is stored in KernelData and KernelTextures structures. At first, the elementary information about the scene is sent to the selected compute devices, in this case the CUDADevice, using const_copy_to method. After that the objects and textures are saved to the GPU memory using tex_alloc method. In addition, one has to allocate buffer for rendered pixels.

The next step is to decompose the rendered image into tiles. There is one POSIX thread that calls the CUDA kernel which contains the path_tracing method for GPU. The results are stored to the buffer vector and sent back from GPU to CPU memory.

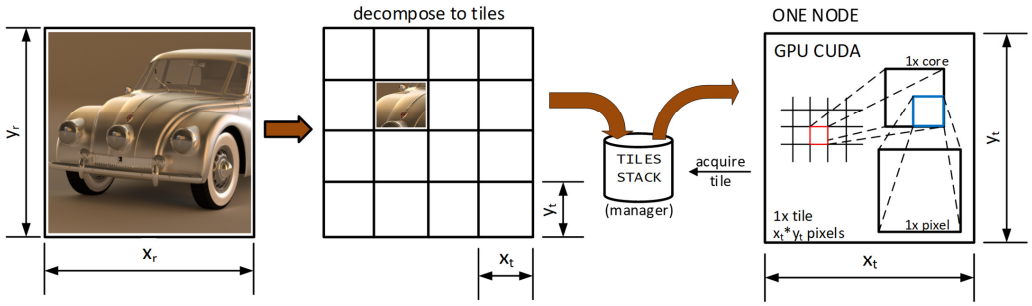


Figure 3.5: The decomposition of the synthesized image with resolution x(r) x y(r) to tiles with size x(t) x y(t) by original implementation of the Blender Cycles for the GPU.

In Figure 3.5, we can see how decomposition works in the original GPU implementation. The synthesized image with resolution x(r) by y(r) is decomposed into tiles with size x(t) by y(t). One tile is computed by one GPU device for x(t) by y(t) pixels and one CUDA thread computes one pixel. It is worth mentioning that Blender supports multiple GPUs.

In the new version of CyclesX, one big CUDA mega kernel was replaced with several smaller CUDA kernels (e.g., a kernel for initialization of camera, several kernels for intersections and shadings) for executing of path-tracing algorithm. The split kernels have better efficiency in terms of the occupancy of execution units. OpenCL is deprecated in the new version of CyclesX.

## 3.3 CyclesPhi renderer for HPC

To fully utilize the HPC cluster, a new Blender rendering client software has been created which contains a subset of the original Blender functionality. We have extended the Blender Cycles engine with OpenMP and MPI support which runs in hybrid MPI+OpenMP+CUDA mode. We named this new rendering client CyclesPhi [124, 125].

In Figure 3.6, we can see what the rendering process in CyclesPhi looks like. The basis is the CLIENTDevice class, the activity of which can be divided into the following two parts: working with data (at the top) and the calculation itself (at the bottom of the diagram). The computing means are shown in the middle. The calculation principle is very similar to the calculation principle on accelerators in the original code. The rendered scene is stored in the KernelData and KernelTextures structures. First, basic information about the scene is sent to selected compute devices, in this case CLIENTDevice, using the const_copy_to method. Objects and textures are serialized to a byte array and sent to client CyclesPhi using the tex_alloc method. In addition, we need to allocate a buffer for the rendered. As we use complex structures such as KernelData, we convert all data to the byte array before sending it to the client. The next step is to lay out the rendered image or task on the tiles (parts of

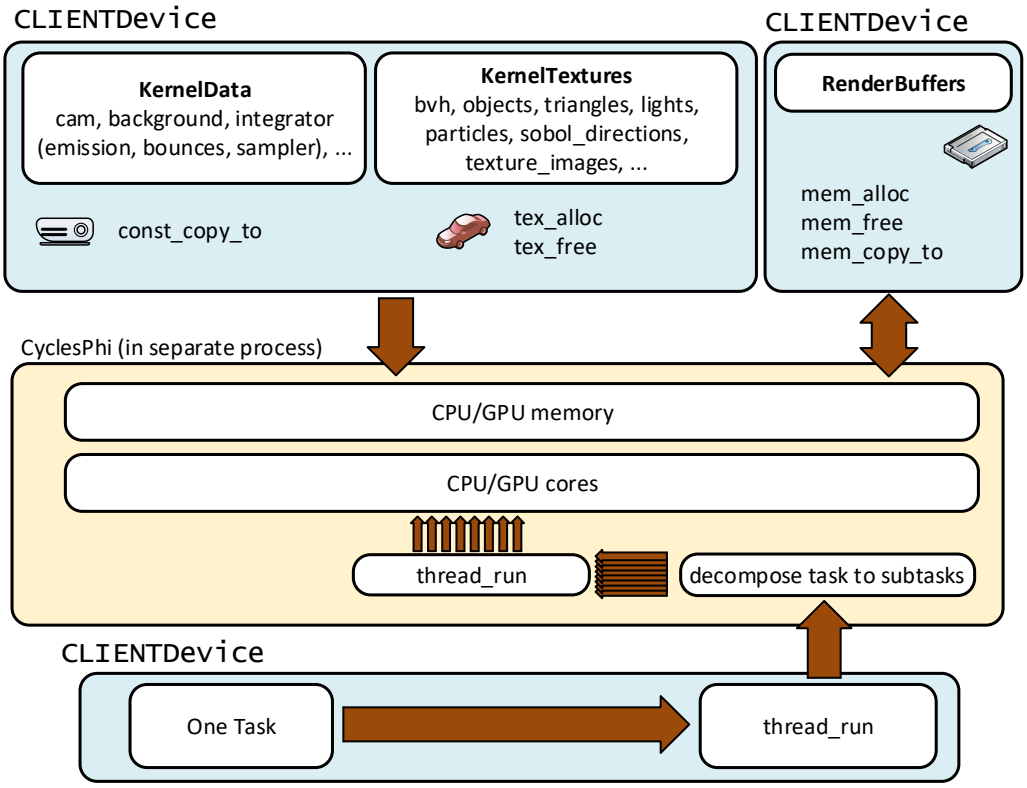


Figure 3.6: CLIENTDevice class for manipulating the main or the GPU memory on the remote node or for calling functions running on the remote computer, all communication with remote node and its memory goes through this device interface

the rendered image). Next, one POSIX thread is created that calls the path_tracing method on the client. The results are stored in a buffer vector and sent back from the client to the Blender.

### 3.3.1 OpenMP parallelization

The OpenMP (OMP) parallelization is implemented as OMP compute device. The workflow is similar to the original CPU code. The difference between original and OpenMP implementation is in the decomposition of an image (see Figure 3.7). One OpenMP thread is processing one pixel of the final image at the time. Due to the nature of the rendering algorithm, the computation time of each sub-tile can be different. To achieve an effective load balancing (see Section 3.3.3.3) we one to adjust the workload distribution by setting up the OpenMP runtime scheduler to schedule (dynamic, 1). This setup produces the most efficient work distribution among processor cores and therefore minimizes the overall processing time.

Another difference from original implementation is that only one POSIX thread is created. This thread then uses OpenMP to parallelize the loop over all pixels of the image. The

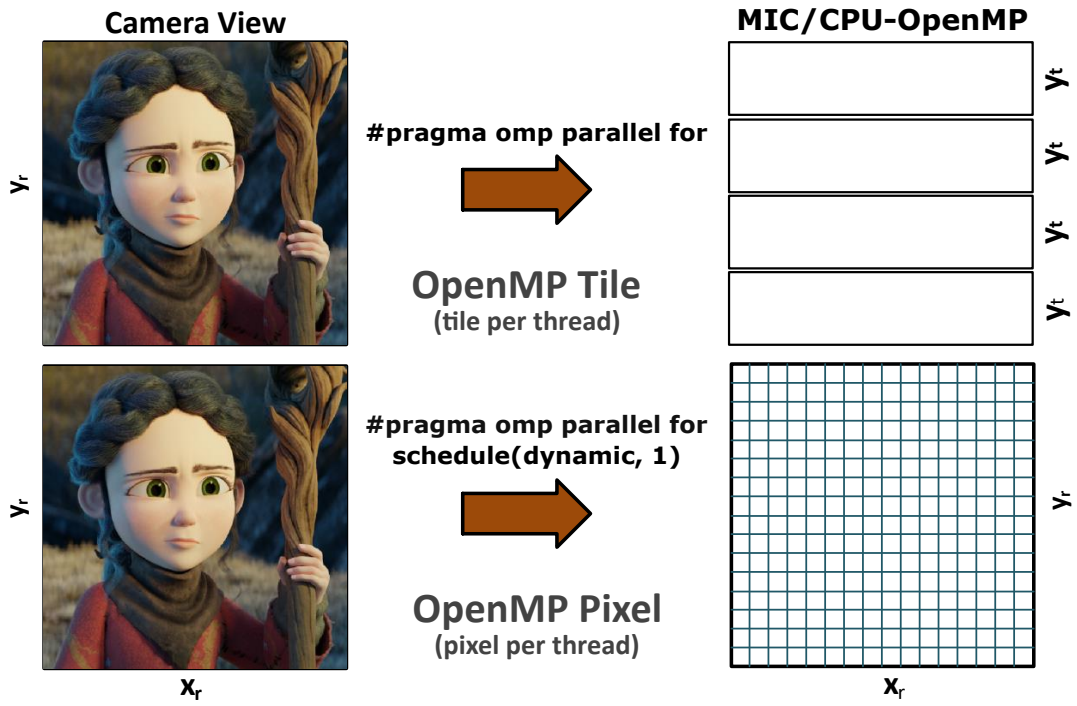


Figure 3.7: The comparison of decomposition over tiles and over pixels using OpenMP. For the decomposition of the synthesized image with resolution x(r) x y(r) the OpenMP scheduler for the dynamic load balancing performance is used.

distribution of scene data is identical to the one in the original code. This device can be used for one node only. The main difference between the original implementation and our implementation is that we moved the loop over samples (there are multiple samples per pixel) inside the loop over pixels.

Another difference is in the use of the properties of the scenes. In the case of an interactive rendering, we have moved the final color calculation of the pixel into the OpenMP loop to preserve optimal parallelization. In this case, the buffer is a vector of floats, which contains all Render Passes. Render Passes are important for the computation of the final image. In each "pass", the engine computes different interactions between objects. Everything you see in the rendered scene must be calculated for the final image. All interactions between objects in the scene, lighting, cameras, background images, world settings, etc. must be calculated separately in different passes for various reasons, such as shadow calculation. During rendering, each pixel is computed several times to ensure that it displays the correct color for the correct part of the image [126].

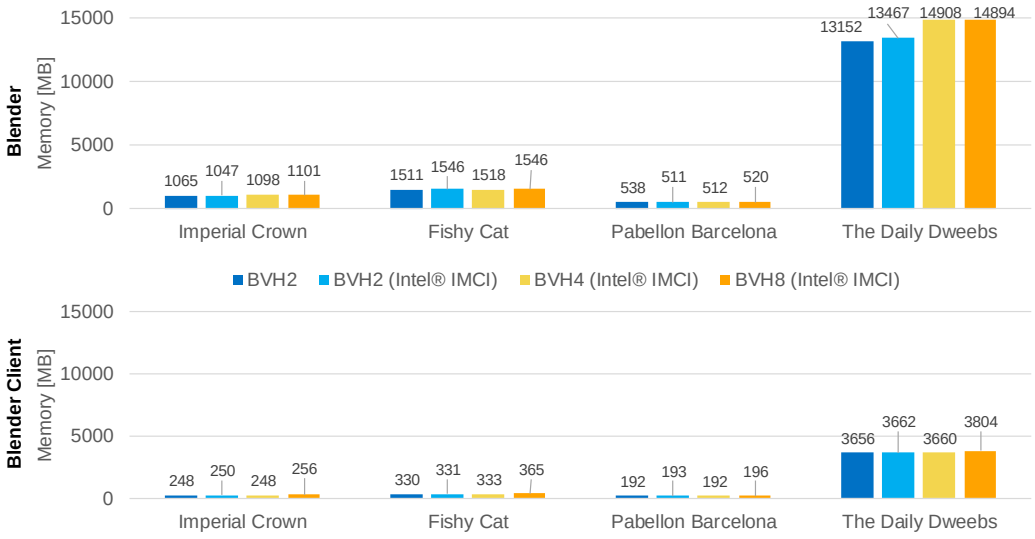


Figure 3.8: The memory usage comparison between Blender and Blender Client

### 3.3.2 Comparison of Memory Consumption

A great advantage of using Blender Client (CyclesPhi) is their low memory occupancy (see Figure 3.8). Blender with Cycles has three scene representations (for efficient synchronization), which is not very convenient in terms of memory usage. Another memory consumption is the pre-processing of data for Cycles, especially for populating KernelTextures. It contains several steps to group all the data into a single array type. For example, all triangles and all vertices are appended into one array. Another example is building a BVH tree, which requires additional memory.

The main reason for reduced memory requirements in separation of preprocessing and rendering stage, where preprocessing is the most memory demanding part.

A Blender Client contains only final data, which is stored in KernelTextures and KernelData. That means the Blender Client needs significantly less memory than the full Blender application. For example, one is able to save 75% when rendering The Daily Dweebs (see Figure 3.8).

### 3.3.3 Cluster parallelization using MPI

Another very effective method for speeding up rendering is to use distributed rendering [71, 125], which gives us the ability to use multiple computers on the network. For communication we use the Message Passing Interface (MPI) technology [127]. MPI is a communication protocol for parallel programming using multiple computers. Both point-to-point and collective

communication is supported. The MPI environment is initialized with MPI_Init_thread() when Blender is started. MPI_Finalize() is used to clean up the MPI environment and is called just before exiting Blender. Communication between nodes is primarily done using the following functions: MPI_Bcast(), MPI_Scatter(), MPI_Gather() and MPI_Reduce(). MPI_Bcast() is one of the standard techniques for collective communication. During a broadcast, one process sends the same data to all processes in the communicator. If process zero is selected as the root process and has an initial copy of the data, all other processes receive a copy of the data. MPI_Scatter() is a collective procedure that is very similar to MPI_Bcast(). MPI_Scatter() includes a designated root process that sends data to all processes in the communicator. MPI_Bcast() sends the same portion of data to all processes, while MPI_Scatter() sends portions of the array to different processes. MPI_Gather() is the inverse function to MPI_Scatter(). Instead of sending elements from one process to many processes, MPI_Gather() takes elements from many processes and collects them into one process. The MPI_Reduce() function is similar to the MPI_Gather() function. Saying that, it receives an array of input elements in each MPI process and returns an array of output elements to the root process. The output elements contain the reduced result, which in our case is the sum of all render buffers from all MPI processes.

CyclesPhi supports several modes for communication between allocated nodes, and the following section discusses these modes.

#### 3.3.3.1 CyclesPhi Runtime modes

CyclesPhi works in several modes with each being suitable for a different use case:

- Interactive Master - Client mode
- Offline Master - Client Mode
- Client only mode

In the case of Interactive Master - Client mode (see Figure3.9), Blender is used as the master and can run on a visualization server or local station, and CyclesPhi as a client runs on the cluster. This mode supports remote rendering where communication is done via TCP sockets and within the cluster via MPI.

In the case of Offline Master - Client mode (see Figure3.10), Blender as master is run from the command line without a GUI on a computed node. Blender automatically loads the selected scene and starts rendering on the client (CyclesPhi). This mode is mainly used for final rendering.

In Client Only mode (see Figure3.11), no Blender is running. The scene is already prepared in a format suitable for CyclesPhi. This mode has several advantages. We can prepare the scene on another node where more memory is available (see Section 3.3.2). The second

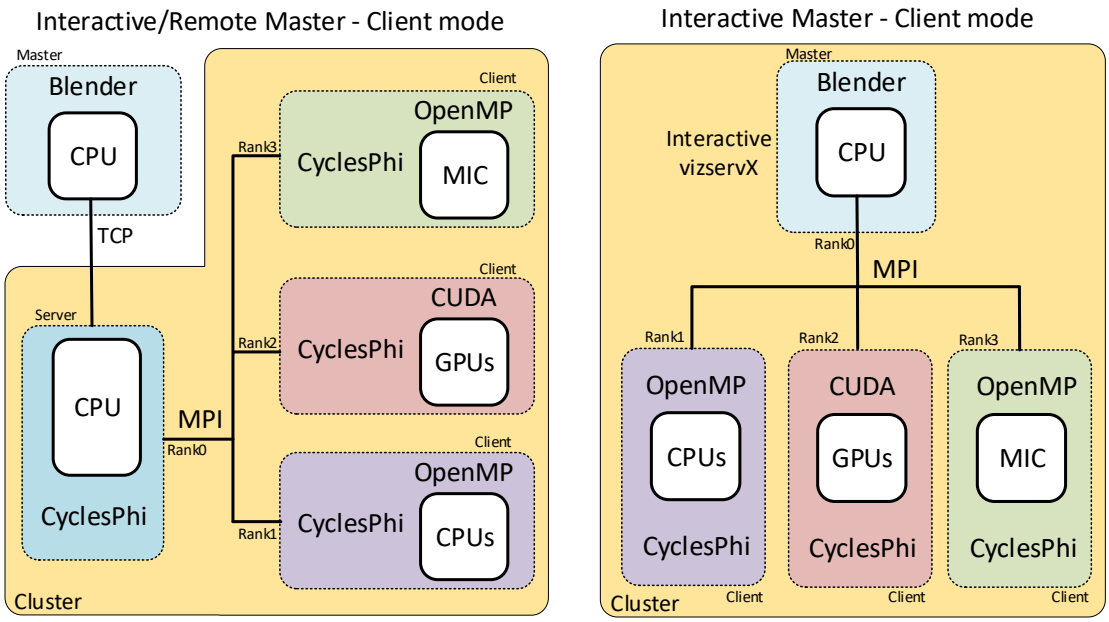


Figure 3.9: Interactive Master - Client mode

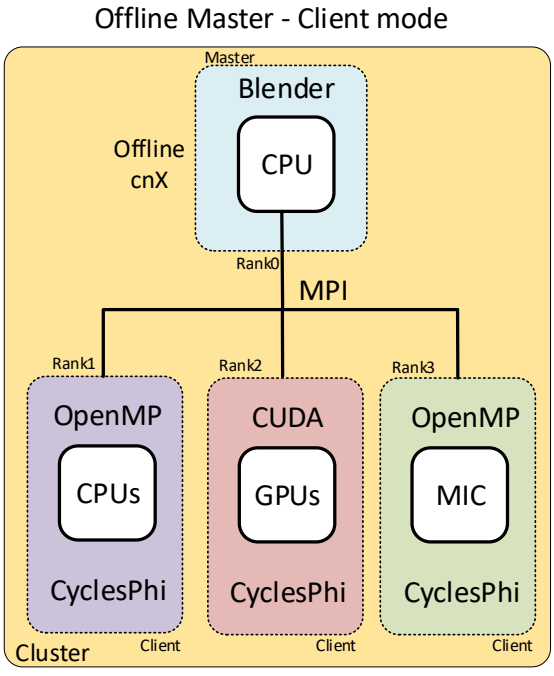


Figure 3.10: Offline Master - Client mode

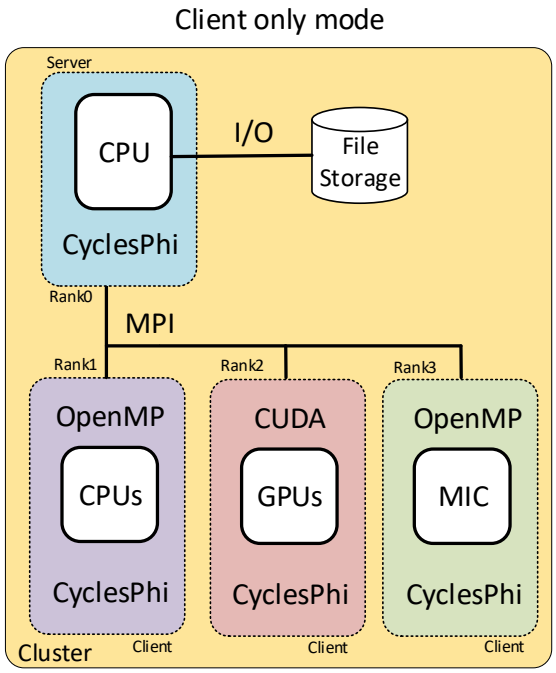


Figure 3.11: Client Only mode

advantage is the time saved in preparing and preprocessing the scene before rendering, where we can, for example, prepare the scene on a node without a graphics card, thus saving the core clock used. We can also use the pre-prepared scenes for presentation or virtual reality. Another advantage is that we can render very fast "flying" cameras through the scene. The disadvantage of the client only mode is that we can only change the camera position instead of editing the scene.

#### 3.3.3.2 Workload distribution for distributed rendering

CyclesPhi supports two types distribution for distributed rendering: distributed samples (see Figure 3.12), distributed tiles (see Figure 3.13).

When the picture is rendered in the off-line mode, the synthesized image is decomposed into samples reflecting the number of all compute devices (see Figure 3.12).

In the case of interactive rendering, the previously mentioned scheme is not convenient due to the higher communication burden. Here, the work distribution is done statically in the pre-processing phase. The synthesized image is decomposed into the tiles reflecting the number of all compute devices (see Figure 3.13).

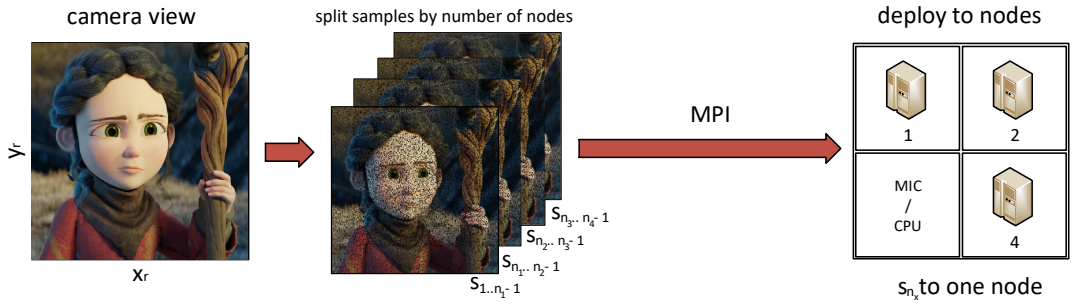


Figure 3.12: Distributed rendering over samples

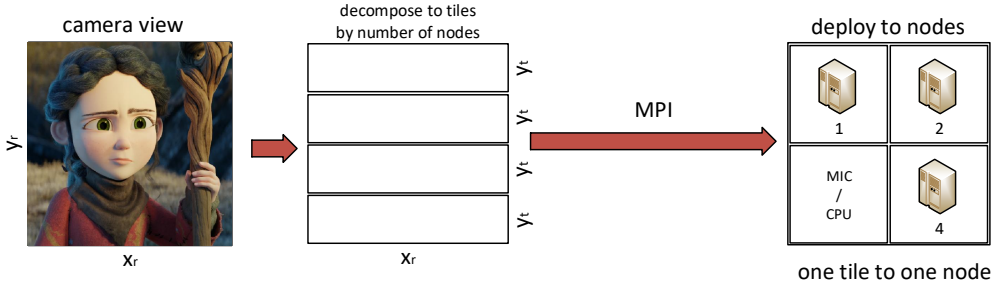


Figure 3.13: Distributed rendering over tiles

#### 3.3.3.3 Load ballancing algorithm

The main problem with distributed rendering is load balancing. The more computing devices used, the more efficiently the main task must be divided into smaller subtasks for each device, since the slowest subtask determines the overall performance. Achieving good load balancing is not trivial and can be achieved using two main strategies [128]: static and dynamic. Static load balancing is performed before the main job starts and remains the same throughout the run. A dynamic load balancing algorithm dynamically changes the load distribution during computation in accordance with the current load distribution of each subtask.

As we mentioned in the Distributed Rendering section 2.1.2, parallel rendering techniques are classified into three groups according to the classification made by Molnar et al. in [71]: sort-first, sort-middle and sort-last.

We use the sort-first rendering techniques for the communication between computation nodes and between accelerators. The workflow of the sort-first rendering techniques is continuous and independent, their communication bandwidth is smaller, which is favorable for parallel rendering, and they can theoretically realize scale extension on a class of linear costs. Currently, the sort-first systems mainly rely on the screen splitting and allocation algorithm to solve the load imbalance problem.

For the load balancing problem of rendering among multiple nodes, many researchers have given solutions and discussed. For example, Abraham and others [129] proposed a rendering

history-based load-balancing algorithm. The algorithm uses the rendering time of the previous frame to estimate the load of the next frame, which is simple and easy to implement. We use this method for dynamic load balancing both between nodes and between accelerators.

Another method for solving the load balancing problem, is to use the so-called cost-map. For example, Cosenza et al. described one way to construct this map in [130]. Cosenza et al. proposed a dynamic load balancing algorithm based on the difficulty of rendering a single pixel. First, the 3D scene is rendered using a ray tracing algorithm and in doing so, the number of intersections between rays and scene objects for each pixel is determined. Then, based on these counts, a map of the rendering cost in the frame is created. Based on the rendering history, the rendering cost map in the previous frame was used to estimate the actual distribution of rendering costs in the future frame. In our case, we create a cost map from the number of bounces for individual rows and distribute the data together with the memory access analysis mentioned in Section 3.4.2.2. This method is more efficient than the timing method (from Abraham [129]) for large scenes that do not fit in memory.

Another method for solving the load balancing problem, is to use so-called work stealing [131]. DeMarle et al. [132] introduced a decentralized load balancing scheme based on work stealing. In their implementation, task migration is performed at the beginning of the next frame and the synchronization bottleneck at the master node is hidden by asynchronous task allocation. Ize et al. [133] used a master dynamic load balancer with a job queue consisting of large tiles that are assigned to each node (the first assignment is done statically and is always the same), and each node has its own job queue that distributes sub-tasks to individual processor threads.

Our implementation of load balancing differs for offline rendering and for interactive rendering and is based on a sort-first rendering technique. For offline rendering if the entire scene fits in memory, i.e., a fully duplicated scene, we use static load balancing with redistribution according to the total number of samples per pixel and according to the total number of devices. Thus, each accelerator gets a job of size

$$work_d = H * W * \frac{S}{count_d}, \tag{3.1}$$

where $H$ is the height of the resulting image, $W$ is the width of the resulting image, $S$ is the required number of samples per pixel, and $count_d$ is the total number of all allocated accelerators.

For offline rendering and when the scene does not fit in the memory of a single graphic card, we use a combination of static and dynamic load balancing. Static load balancing via samples is used when the job is divided among the nodes as in the previous case

$$work_n = H * W * \frac{S}{count_n}, \tag{3.2}$$

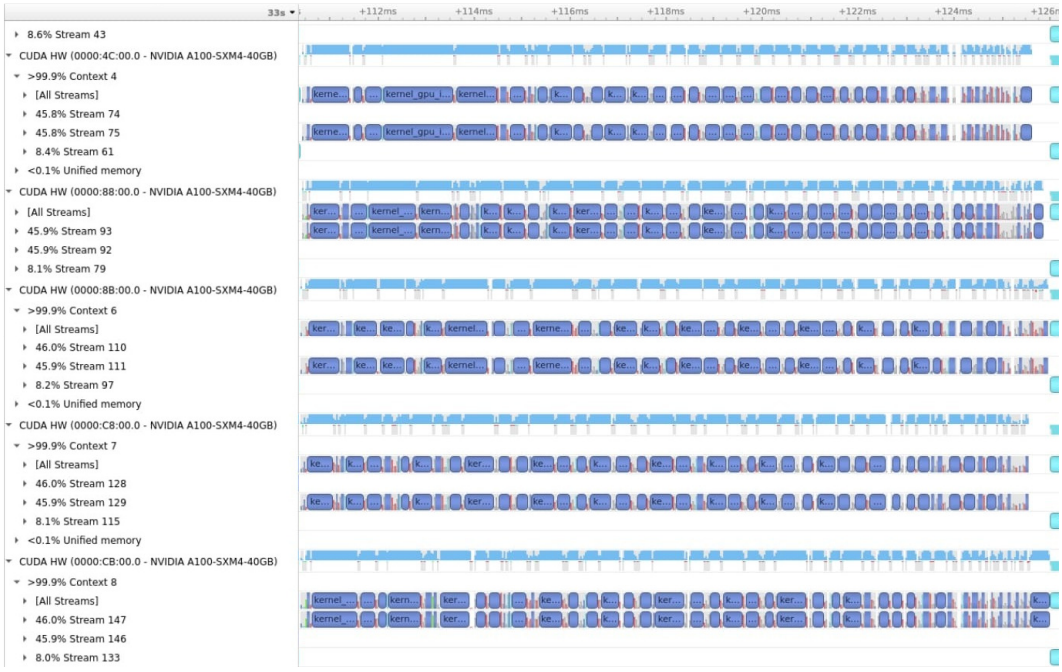


Figure 3.14: NSYS profiling of balanced scene

where $count_n$ is the number of allocated nodes. Within each node, the first step divides the image into equally sized parts and gives each accelerator the same number of rows

$$work_d = H_d * W * (\frac{S_n}{step}). \tag{3.3}$$

In subsequent steps, the number of rows is recalculated for each device according to the times of the previous step (see Algorithm 1).

**ALGORITHM 1:** Load balancing algorithm.

**if** $t_d(i) < t_d(i+1)$ **then**
- $c = (t_d(i+1) - t_d(i))/count_d/t_d(i+1)$
- $H_d(i) += H_d(i+1) * c$
- $H_d(i+1) -= H_d(i+1) * c$
- $t_d(i+1) -= t_d(i+1) * c$

**if** $t_d(i) > t_d(i+1)$ **then**
- $c = (t_d(i) - t_d(i+1))/count_d/t_d(i)$
- $H_d(i) -= H_d(i) * c$
- $H_d(i+1) += H_d(i) * c$
- $t_d(i+1) += t_d(i) * c$

The used load balancing Algorithm 1) is profiled in Figure 3.14 for calculation on multiple graphic cards (NVIDIA A100). The NSYS profiler was used.

At the end of the computation, the render buffer from all devices is collected and summed into a single buffer, from which the resulting color of each pixel is then computed.

For CPU rendering, dynamic load balancing is provided by OpenMP with a properly set scheduler - see the OpenMP section.

For distributed interactive rendering, a similar setup is used for all scene types as for offline rendering for large scenes. That means that the balancing between nodes uses a static type via samples, and within each node the load is recalculated from the previous frame time, changing dynamically the number of lines that each accelerator is responsible for. Unlike offline rendering, the image is sent continuously to the master node, which sends the image to the display device.

### 3.3.4 Extensions for Multi-GPU Support

In the previous section we introduced the Cycles to support new parallel hardware architectures (e.g., Intel Xeon Phi co-processors and processors) and other HPC technologies (e.g., distributed rendering using MPI).

To implement the multi-GPUs support with distributed data over GPUs, only a few changes need to be made to the original GPU branch of Cycles. Most importantly, the core of the path tracing CUDA kernel used for the final rendering remains unchanged. The only modifications inside this kernel concern implementations of software counters that record the memory access statistics. From the rendering point of view, the functionality of this kernel remains completely unchanged. Our modification splits each data structure used by the path tracer into chunks of a predefined size and counts the number of accesses per chunk. Since there is a performance overhead caused by this extra work, it is recommended to have these modifications in a separate kernel.

The most significant changes are made to the CPU code to support the CUDA Unified Memory, which is used to both replicate or distribute selected chunks of scene data among multiple GPUs. This is done using *cudaMallocManaged* instead of *cudaMalloc*, together with *cudaMemAdvise* data placement hints introduced in CUDA 8.0.

The CUDA UM mechanism is thoroughly described in [100, 102, 134]. For a shorter description, see Section 2.2.2.

The overall workflow is as follows:

- distribute the data structures evenly among all GPUs,
- run the kernel with memory access counters and get the memory access statistics,
- redistribute the data structures among GPUs based on the memory access statistics,
- run the original path-tracing kernel with redistributed data.

## 3.4 GPU rendering of massive scenes

### 3.4.1 Global Shared Memory programming

Global shared memory on DGX-2, DGX-A100 and Barbora is managed using the Unified Memory mechanism, which creates a unified memory address space of the physically separated memory of CPUs and GPUs. It also unifies the memory space of multiple GPUs and their physically separated memory. This greatly simplifies the GPU programming by removing the need for low-level memory management (explicit data transfers between CPU and GPU controlled by a programmer). The main goal of UM is to provide a consistent view of the data between all CPUs and GPUs within a single server. It ensures that every memory page can be accessed by one CPU or GPU thread at a time. When a GPU accesses a page that is not in its memory, a page fault occurs. The GPU that holds the requested page will release it, and the page will migrate to the GPU that requested it. The same mechanism is used between CPUs and GPUs as well as between GPUs, as thoroughly described in [100, 102, 134]. Memory trashing that continuously migrates the page from one GPU to another easily becomes a serious bottleneck. This has been addressed by hardware support for on-demand page migration in the Pascal architecture and page access counters in the Volta architecture that enable the frequency of accesses to impact decisions on page migration. For more details, see [135, 88].

This problem can also be addressed by a code developer using the data placement and movement hints introduced in CUDA 8.0. These hints are applied using cudaMemAdvise() to a specific part of the memory described by the address of the beginning and its size. It the text below, we refer to the part of the memory holding data as a chunk of data [136].

The available memory hints are:

- cudaMemAdviseSetReadMostly - communicates that the chunk will be mostly read (but can also be written),
- cudaMemAdviseSetPreferredLocation - sets the location of a chunk to the memory of a given GPU,
- cudaMemAdviseSetAccessedBy - communicates that the chunk will be accessed from a given GPU.

In addition, there is cudaMemPrefetchAsync(), which is used for prefetching (transferring) data to a particular GPU (set by the hints above) before it is used by a GPU kernel. In our proposed approach, UM is used only among GPUs connected using the NVLink bus. Due to low performance of the PCI-Express bus connecting CPU and GPU, as shown in Figures 2.3, 2.7,2.11, the CPU memory is not utilized once data are transferred to the GPU memory.

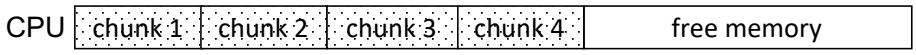


Figure 3.15: Data structure memory allocation in the CPU memory


Figure 3.16: The first way of allocation of data structure in the Unified Memory of the multi-GPU system: Fully distributed data structure

In Figure 3.15 the data structure are divided into chunks and placed in the CPU memory. Suppose there is a data structure placed in the UM. If one applies the *PreferredLocation hint on the entire data structure it will be placed in the memory of one GPU. However, different hints can be applied to different chunks of the data structure. This enables us to place different chunks to the memory of different GPUs, i.e., to distribute a data structure across the memories of all GPUs, see Figure 3.16. This is a key principle for processing data structures larger than the individual GPU memory, but it comes with the cost of non-uniform memory access. If a chunk is marked by the *ReadMostly hint, it will be replicated in the memory of all GPUs; see Figure 3.17. This results in optimal performance, as only local memory accesses are performed but leads to sub-optimal memory utilization. However, one can also combine the *PreferredLocation and *ReadMostly hints and replicate selected chunks of the data structure while distributing the rest of them; see Figure 3.18. This can be used to improve the performance of processing read only data, which is the case of path tracing using the following logic: the pages with a high number of accesses are replicated, and pages with low number of accesses are distributed and owned by the GPU with highest number of accesses.

The process of using this simple idea to enable multi-GPU path tracing with distributed scenes is described in Section 3.4.2.

### 3.4.2 Data Distributed Multi-GPU Path Tracing

In this section, we propose a method designed to minimize the negative impact of the remote memory accesses on an algorithmic level to maximize path tracing performance and scalability. We propose two approaches for splitting data structures used by a path tracer. Each approach

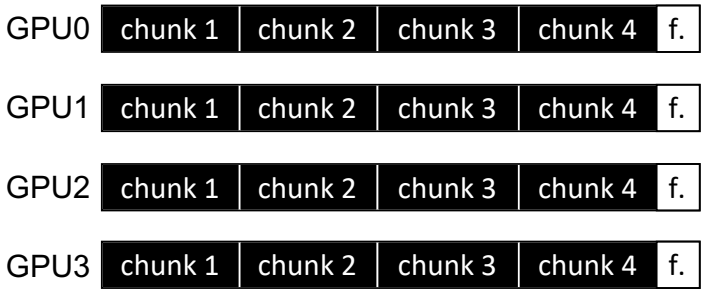


Figure 3.17: The second way of allocation of data structure in the Unified Memory of the multi-GPU system: Fully duplicated data structure

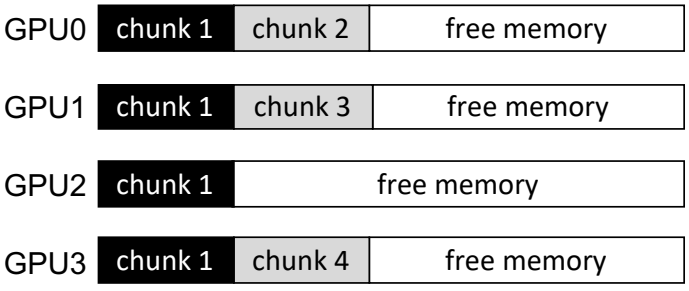


Figure 3.18: The third way of allocation of data structure in the Unified Memory of the multi-GPU system: Partially distributed data structure

works with a different granularity of data placement control. We also evaluate how different data distributions affect the overall performance.

In Section 3.4.2.1 we describe a less complex methodology that works with entire data structures. Then, in Section 3.4.2.2, we introduce an advanced methodology that works with data structures divided into chunks and controls the waythese chunks are placed in the memory of individual GPUs.

The description of the methodology in this section is demonstrated using the *Moana 12GB* and *Moana 27GB* scenes for the Barbora, DGX-2, and DGX-A100 servers. The 12 or 27 GB value refers to the overall size of all data structures used by Cycles for path tracing. The scenes of these sizes were selected because they fit into the memory of a single GPU of the respective platform, and we are therefore able to provide a full scalability evaluation from 1 to 4, from 1 to 8, or from 1 to 16 GPUs, respectively.

#### 3.4.2.1 Basic Distribution of Entire Data Structures

There are approximately 40 data structures in Cycles that describe the scene. Structures are accessed only for reading. Every data structure can be independently replicated or distributed across GPUs.

**3.4.2.1.1 Memory Access Analysis** We define the order in which data structures are replicated as a ratio of the total memory accesses to a particular data structure over its size. To be able to analyze its behavior, CyclesPhi was modified to count the number of accesses to each data structure. The analysis was done on the first sample when rendering the scenes with a resolution of $5120 \times 2560$ pixels.

The structures that have the highest number of memory accesses per byte are shown in Figures 3.19,3.20,3.21.

These are:

- *small_structures* - a set of data structures smaller than 16MB (the most important one is *svm_nodes* which stores Shader Virtual Machine (SVM) data and codes),
- *bvh_nodes* - stores the BVH tree without its leaves (leaves are stored in a separate structure),
- *prim_tri_verts* - holds coordinates of all vertices in the scene,
- *prim_tri_index* - is a set of all triangles in the scene and it contains indices to the prim_tri_verts.

As an example, we shall examine the Moana 27GB scene on the DGX-2 server. Evidently, the most important data structure is *bvh_nodes* because it is responsible for 79.6% of all

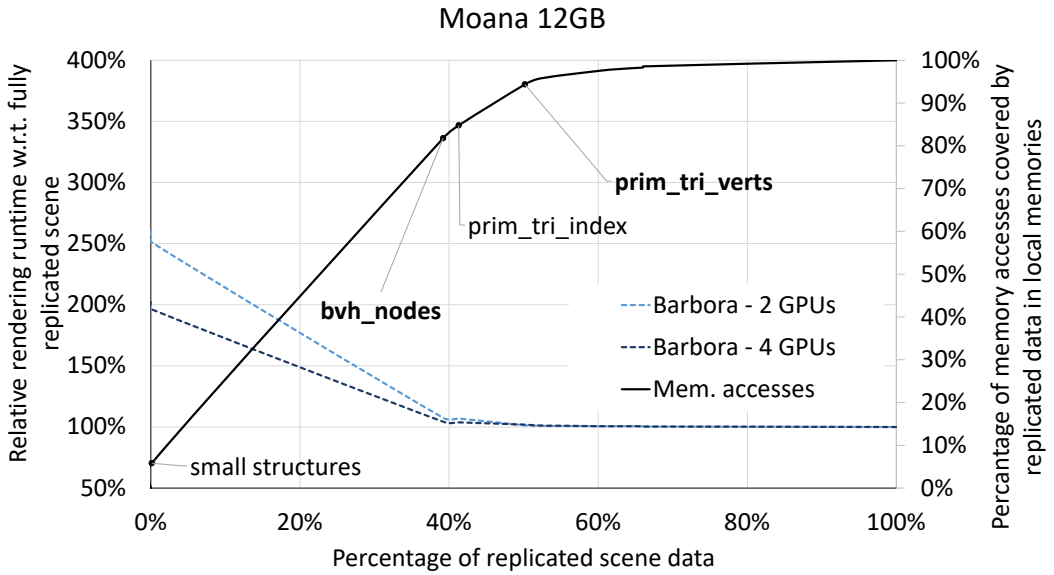


Figure 3.19: Analysis of memory accesses on the Barbora GPU system covered by the most important data structures (full lines). Impact of the data structure distribution and replication on rendering performance. Results for 0% replication represent full distribution of all data structures.

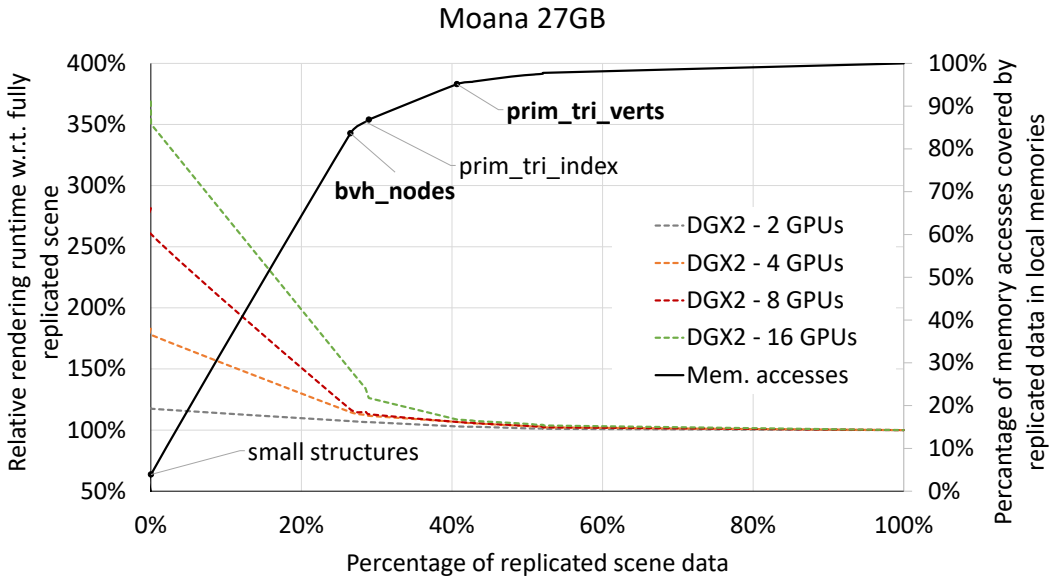


Figure 3.20: Analysis of memory accesses on DGX-2 system covered by the most important data structures (full lines). Impact of the data structure distribution and replication on rendering performance. Results for 0% replication represent full distribution of all data structures.

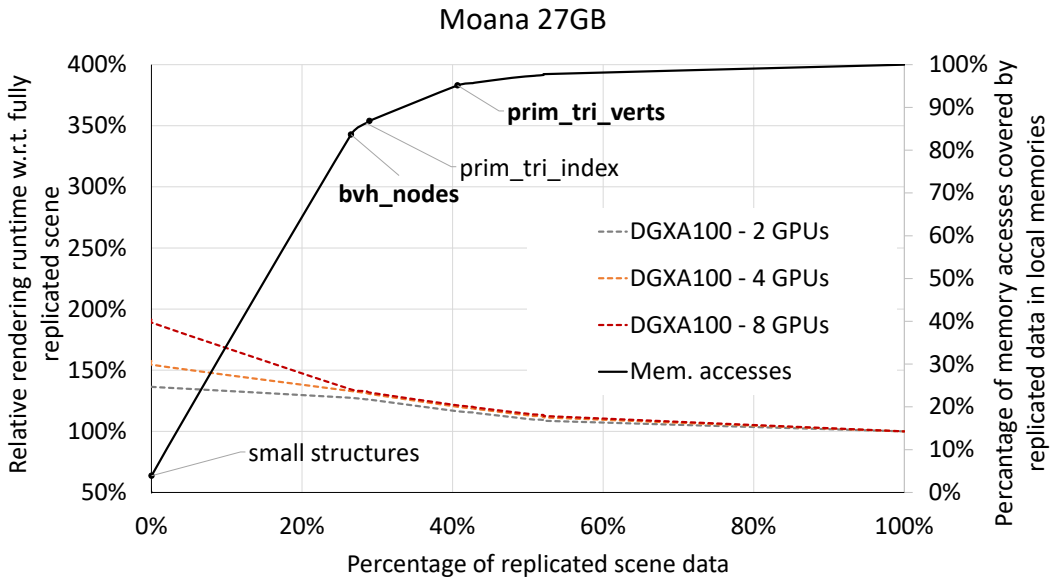


Figure 3.21: Analysis of memory accesses on the DGX-A100 system covered by the most important data structures (full lines). Impact of the data structure distribution and replication on rendering performance. Results for 0% replication represent full distribution of all data structures.

memory accesses. This means that if it is replicated in the memory of all GPUs, the 79.6% of all memory accesses will be to the local memory. See the full line in Figures 3.19,3.20,3.21. The size of this structure is 7.2GB which represents 26.5% of the entire scene size.

The remaining lines in Figures 3.19,3.20,3.21 show how the path-tracing performance is affected by the distribution and replication of these data structures. 0% scene replication means that all data structures are split into 2MB chunks, and these are distributed in a round robin fashion among all GPUs. One can see that this naive distribution leads to an almost 3.7× longer rendering time (370%) when all 16 GPUs are used. The performance penalty decreases when using a smaller number of GPUs. The path-tracing time for fully replicated scenes is used as a baseline for relative rendering time evaluation. The relative rendering time is decreased to only 149% by replicating *small_structures* and *bvh_nodes* on all 16 GPUs. If in addition to *small_structures* and *bvh_nodes*, *prim_tri_index* and *prim_tri_verts* are also replicated, then the relative rendering time is only 109% while 40.7% of the scene is replicated and the rest is distributed.

Consequently, the total memory allocation per GPU is 12.1 GB[1] instead of 27.2 GB.

**3.4.2.1.2 Performance and scalability evaluation** The scalability of the proposed approach is evaluated for four different cases:

[1] 11.1 GB replicated + (16.1 GB / 16 GPUs) distributed = 12.1 GB per GPU.

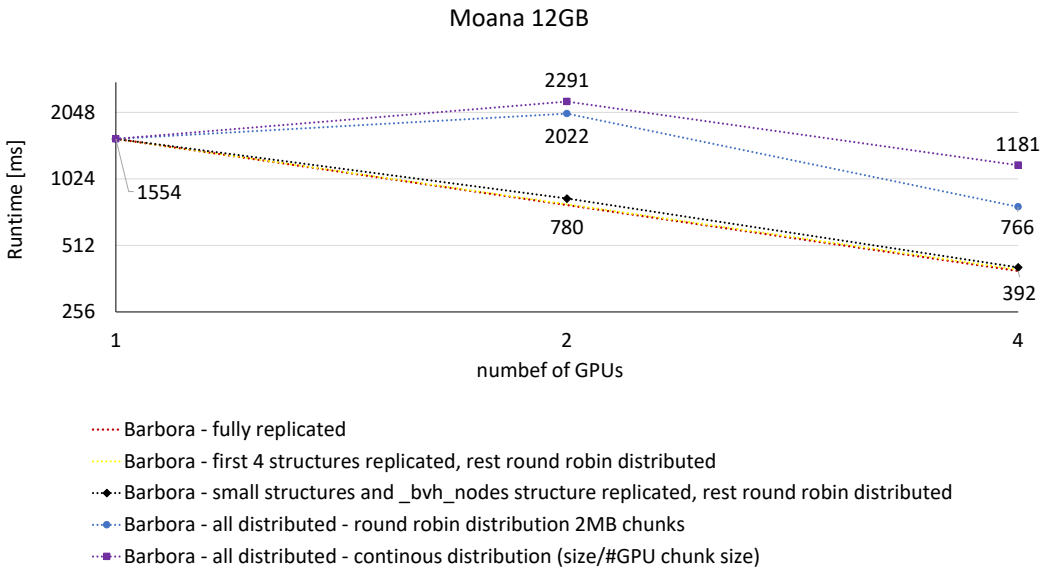


Figure 3.22: Scalability for different data distribution using entire structures on the Barbora GPU server for Moana 12GB scenes. The rendering time is for 1 sample and the resolution is 5120 × 2560.

- all data structures are replicated – this case serves as a baseline as it achieves the best performance and scalability,
- all data structures are evenly distributed,
- *small structures* and *bvh_nodes* are replicated while all other data structures are distributed,
- *small structures*, *bvh_nodes*, *prim_tri_index*, and *prim_tri_verts* are replicated while all other data structures are distributed.

For case (2), two different forms of data distribution are evaluated: (a) *continuous distribution*: the structures are divided into large chunks of a size equal to the structure size over a number of GPUs, and each GPU owns one chunk, (b) *round robin distribution*: the distributed structure is divided into chunks of 2 MB, which are distributed in a round robin fashion. The 2 MB chunk is the smallest possible size which still provides good performance and is optimal for scenes of these sizes. Smaller chunks always yield worse performance for the *SetReadMostly* memory hint which replicates the chunks over all GPUs.

The results of these tests are shown in Figures 3.22,3.23,3.24,3.25,3.26.

The rendering times are for one sample and an image resolution of 5120 × 2560 pixels. Several conclusions can be made from the presented results:

- round robin distribution of small chunks performs better than continuous distribution of large chunks, and therefore it is always used to distribute non-replicated data structures,

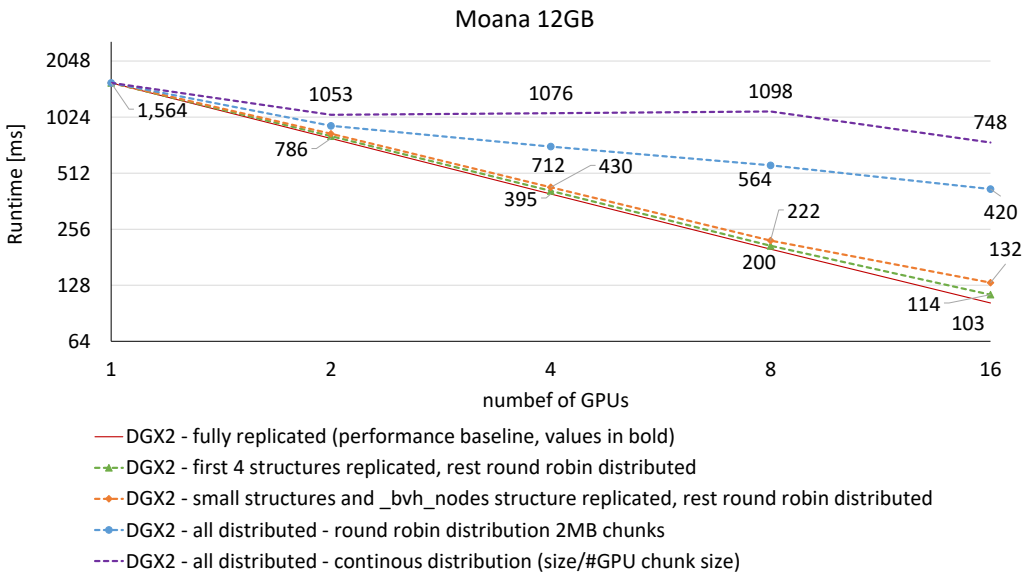


Figure 3.23: Scalability for different data distribution using entire structures on the DGX-2 system for Moana 12GB scenes. The rendering time is for 1 sample and the resolution is 5120 × 2560.

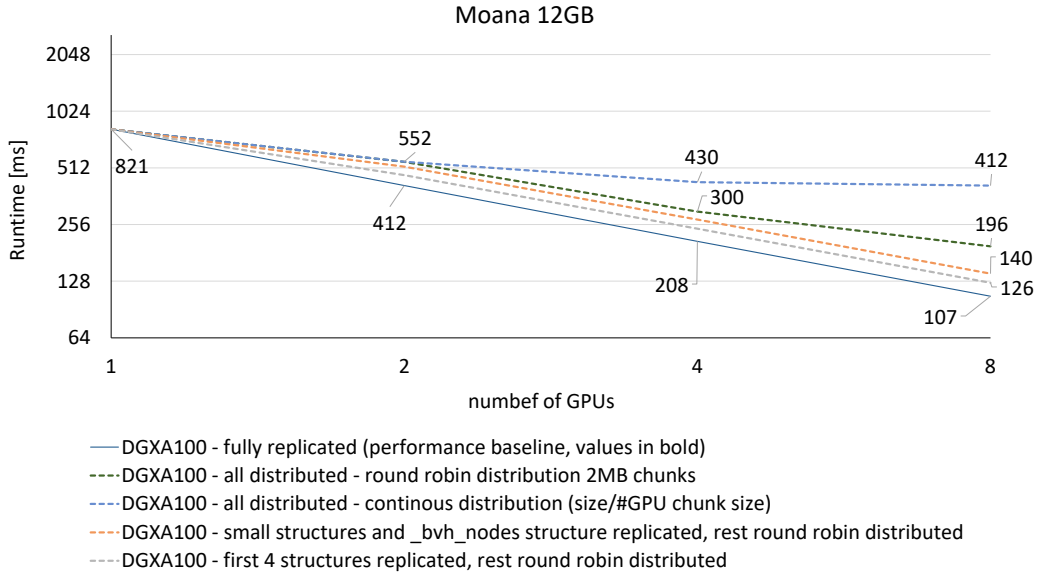


Figure 3.24: Scalability for different data distribution using entire structures on the DGX-A100 system for Moana 12GB scenes. The rendering time is for 1 sample and the resolution is 5120 × 2560.

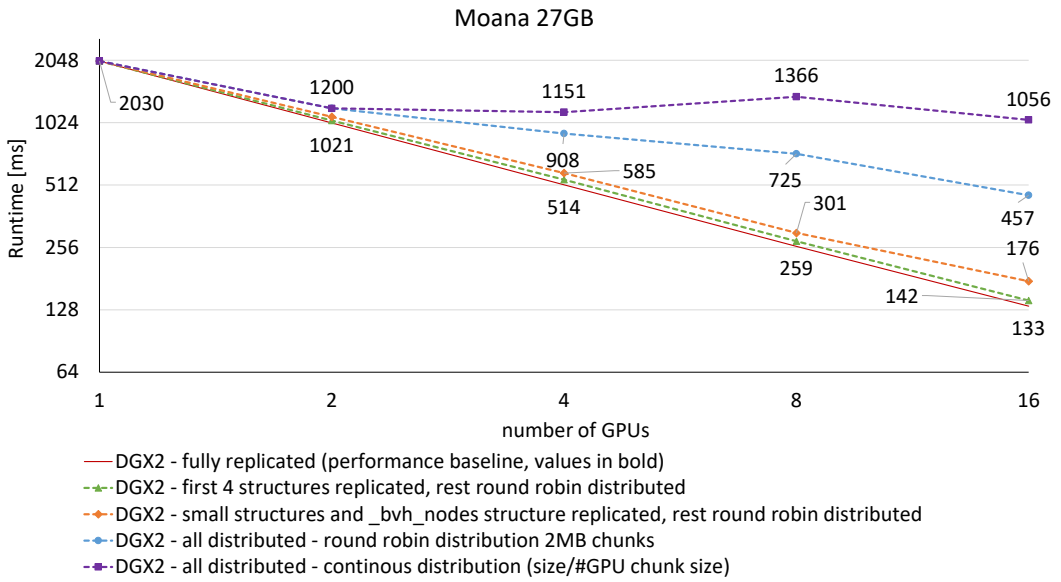


Figure 3.25: Scalability of the Cycles path tracer for different data distribution using entire structures on the DGX-2 system for Moana 27GB scenes. The rendering time is for 1 sample and the resolution is 5120 × 2560.

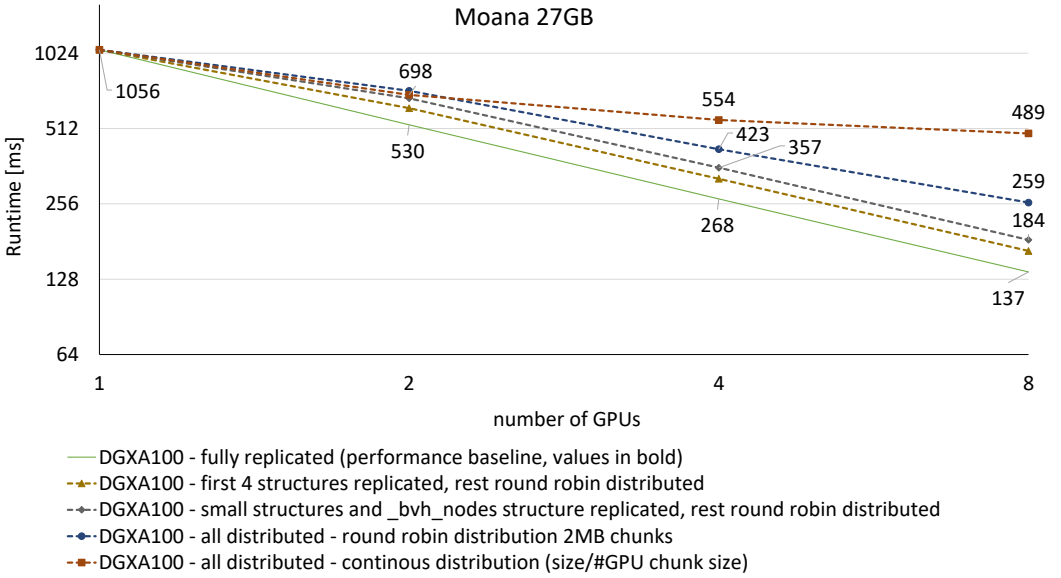


Figure 3.26: Scalability of the Cycles path tracer for different data distribution using entire structures on the DGX-A100 system for Moana 27GB scenes. The rendering time is for 1 sample and the resolution is 5120 × 2560.

- path tracing with fully distributed data structures does not scale on selected platforms (there is reasonable scalability for two GPUs on DGX-2 but not beyond that),
- if *small structures* and *bvh_nodes* are replicated, the scalability is significantly improved:
  - on Barbora the speedup is 3.8 for 4 GPUs for Moana 12GB,
  - on DGX-2 the speedup is 11.8 and 11.5 for 16 GPUs for Moana 12GB and Moana 27GB scenes, respectively,
  - on DGX-A100 the speedup is 5.8 and 5.7 for 8 GPUs for Moana 12GB and Moana 27GB scenes, respectively,
- if *small structures*, *bvh_nodes*, *prim_tri_index*, and *prim_tri_verts* are replicated, the scalability is further improved:
  - on Barbora the speedup is 3.9 for 4 GPUs for Moana 12GB,
  - on DGX-2 the speedup is 13.7 and 14.3 for 16 GPUs for Moana 12GB and Moana 27GB scenes, respectively,
  - on DGX-A100 the speedup is 6.5 and 6.3 for 8 GPUs for Moana 12GB and Moana 27GB scenes, respectively,

#### 3.4.2.2 Advanced Distribution based on Memory Access Pattern and Statistics

Now we present an advanced data placement algorithm that takes full advantage of the Unified Memory mechanism and data placement hints introduced in CUDA 8.0 (*SetReadMostly*, *SetPreferredLocation*, ...)

The data placement is done with chunks, and hints are set for each chunk individually. The optimal chunk size was identified experimentally by benchmarking the path tracer performance for chunks of sizes from 64 kB to 128 MB.

We observed that:

- for scenes smaller than 30 GB the optimal chunk size is 2 MB (smaller chunks are not recommended),
- for scenes of sizes around 40 GB the optimal chunk size is 16 MB,
- for scenes of sizes above 120 GB the optimal chunk size is 64 MB.

The workflow of this data placement strategy can be summarized by the following steps (more details are given in the subsections below):

- copy/distribute every data structure across all GPUs in a round robin fashion using chunks of an optimal size,

- run the path tracing kernel with memory access counters for 1spp to measure the statistics,
- gather the statistics on the CPU and run the proposed algorithm to get the optimal data chunks distribution,
- use *cudaMemAdvise* to migrate or replicate all chunks,
- run the original unmodified path tracing kernel.

**3.4.2.2.1 Memory Access Pattern Analysis** To identify the memory access pattern, per chunk access counters have been implemented in the GPU path tracing kernel of CyclesPhi. There are independent counters for all data structures and all their chunks, therefore, a total number of memory accesses per chunk can be recorded for each GPU.

The memory analysis starts with all data structures being evenly distributed using a round robin distribution. Then the modified path tracing kernel with memory access counters is executed on all GPUs for 1 sample.

When the kernel finishes, then for every chunk of every structure, a number of accesses from all GPUs is recorded. This data is the input for the data placement algorithm described in the next section. As is the case for final rendering, even during the analysis, the rendering workload is distributed, and each GPU works on its own part of the image. The workload is distributed among GPUs by horizontal stripes so that each GPU works on one stripe. Load balancing is done by changing the height of stripes.

The analysis of the memory accesses for the Moana scenes of different sizes is shown in Figure 3.27. The figure shows that 1% of the scene data covers between 56.7% and 74.4% of memory accesses, depending on the scene size. This is significantly better than working with entire structures. For illustration and direct comparison, we have added the dashed lines to Figure 3.27 which represent the full lines from Figures 3.19,3.20,3.21. Another important fact is that for the Moana 12GB, 27GB, 38GB, and 169GB scenes, 8.6%, 17.1%, 18.8%, and 18,2%, respectively, of the scene data chunks were not accessed at all.

The figure shows that 1% of scene data broken into 2MB chunks covers between 57% and 74% memory accesses depending on the scene size. For comparison we have added the data points from Figure 2 as dashed lines to show the advantage of this method.Comparing these results with those in Figures 3.19,3.20,3.21, the following conclusions can be drawn:

- for the Moana 27GB scene *small_structures*, *bvh_nodes*, *prim_tri_index*, and *prim_tri_verts* cover 95.1% of all memory access at the cost of 40.7% scene replication – this results in 12.1 GB[2] of data per GPU for 16 GPUs in total,

[2] 11.1 GB replicated + (16.1 GB / 16 GPUs) distributed = 12.1 GB per GPU.

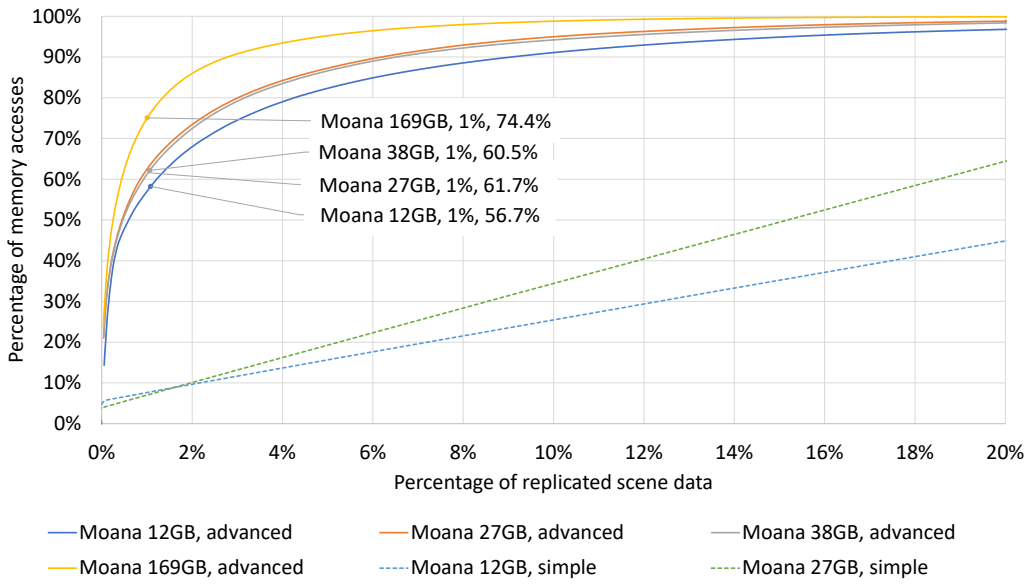


Figure 3.27: The analysis of the memory accesses of the Cycles path tracer for Moana of 12, 27, 38 and 169GB scenes.

- for the same scene, the method presented in this section covers 95.1% of memory accesses with just 10.1% of replicated data – this results in 4.2 GB[3] of data per GPU, which is 2.9 times smaller.

This analysis shows that there are clear candidates among the chunks that should be replicated on all GPUs, while a major portion of the data is accessed infrequently and can be distributed with an acceptable impact on performance. The algorithm that decides the placement of each chunk based on this analysis is described in the next section.

#### 3.4.2.2.2 Data Placement Algorithm based on Memory Access Pattern

Algorithm 2 processes the 3-dimensional array of memory access counters `Counters`$[G][S][C]$, where $G$ are GPUs, $S$ are data structures, and $C$ are chunks within a particular data structure. The output of the algorithm provides an optimal location for each chunk (the GPU in which the chunk should be placed) and decides whether the chunk should be distributed or replicated to all GPUs.

In the first step, the per GPU counters are summed to get the total number of accesses $a_{sum}$ for each chunk $c \in C$ of each data structure $s \in S$ and a $(a_{sum}, s, c)$ tuple is created. One tuple now represents one chunk of scene data. Next, all tuples are put into a single 1D array $\mathtt{H}_{\text{comb}}$. The array is sorted by $a_{sum}$ from the largest value (the highest number of accesses) to the smallest one and stored in $\mathtt{H}_{<}$ array.

[3] 2.7 GB replicated + (24.5 GB / 16 GPUs) distributed = 4.2 GB per GPU.

**ALGORITHM 2:** Data placement algorithm.

**Input :** Memory access counters $\texttt{Counters}[G][S][C]$, where $G$ are the GPU indices, $S$ are the data structures, $C$ are the memory chunks per data structure $s$; $N_{dup}$ is the maximum number of chunks that can be replicated.

**Output:** Chunk distribution with optimal placement

$i = 0$

**foreach** $c \in C, s \in S$ **do**

  $a_{sum} = \sum_{g \in G} \texttt{Counters}[g][s][c]$  For chunk $c$ in data structure $s$, sum accesses to it by all GPUs and save it to $a_{sum}$.

  $\texttt{H}_{comb}[i] = (a_{sum}, s, c)$  Put all per chunk counter values $a_{sum}$ to one array so that it can be sorted by number of accesses.

  $i = i + 1$

**end**

$\texttt{H}_{<} = \textbf{\textit{sort}}(\texttt{H}_{comb})$ **by** $a_{sum}$  Descending sort by $a_{sum}$.

$(t, s_t, c_t) = \texttt{H}_{<}[N_{dup}]$  Find replication threshold $t$.

**foreach** $(a_{sum}, s, c) \in H_{<}$  Process all chunks from the highest number of accesses to the lowest one.

**do**

  **if** $a_{sum} > t$ **then**

    set chunk $c$ of data structure $s$ as replicated

  **else if** $a_{sum} > 0$ **then**

    $G_f = \{g |$ GPU $g$ has a free memory$\}$

    $g_m = \max_{g \in G_f} \texttt{Counters}[g][s][c]$  Find a GPU $g_m$ that has free memory and the highest number of accesses to chunk $c$ in $s$.

    set chunk $c$ of data structure $s$ as distributed and set its preferred location to GPU $g_m$

  **else**

    chunk $c$ is distributed in a round robin fashion  Chunks with zero accesses are distributed among GPUs with free memory.

  **end**

**end**

The last input of the algorithm is the number of chunks that can be replicated $N_{\text{dup}}$. This value can either be set manually or automatically using the formula

$$N_{\text{dup}} = \frac{1}{C_s}\left(G_f - \frac{S_s}{N_g}\right) \tag{3.4}$$

where $G_f$ is the amount of free memory per GPU in MB available to store scene data, $S_s$ is the scene size in MB, $N_g$ is the total number of GPUs, and $C_s$ is the chunk size in MB (2-64 MB based on the scene size).

We define a threshold $t$ as the $N_{\text{dup}}$-th element in the sorted array $\texttt{H}_<$ and evaluate all tuples in the array $\texttt{H}_<$. If the counter value $a_{sum}$ is larger than $t$, the corresponding chunk will be set as *SetReadMostly*, and therefore replicated.

In the opposite case, the chunk is set as *SetPreferredLocation* and is assigned to the GPU with the highest number of accesses to this chunk. If the memory of this GPU is full, then the GPU with second, third, fourth, etc., highest number of accesses is selected until a GPU with free memory is found.

If the counter value is equal to zero (without any accesses), the corresponding chunk will be distributed in a round robin fashion across GPUs with free memory.

**3.4.2.2.3 Performance Evaluation** The performance of the proposed algorithm was evaluated for different ratios between replicated and distributed data at a 2 MB chunk level of granularity for the Moana 12GB and 27GB scenes. The range is from 0% of replicated chunks (fully distributed), all the way up to 100% of replicated chunks if possible. Please note that there is a difference in how chunks are distributed when compared to the approach presented in Section 3.4.2.1. Previously the chunks were placed in a naive round robin fashion. However, the chunks are now placed in the memory of the GPU that had the highest number of accesses to it. Chunks are placed in the GPU memories so that each GPU has the same amount of data, whilst attempting to simultaneously place chunks into the memory of the GPUs that will access them most often. This explains the different and better performance for the fully distributed scenes, i.e., 0% replication.

Figures 3.28,3.29,3.30,3.31 show the path tracing performance for 1 sample per pixel and $5120 \times 2560$ pixel resolution for both scenes and platforms and for all available GPUs. The baseline for evaluation is the runtime for 1 GPU on DGX-2, which is 1564 ms for the smaller scene and 2032 ms for the larger scene.

For the Moana 12GB scene, the speedup for a fully replicated scene on 4 GPUs on selected platforms is 3.95, and parallel efficiency is 99%. On DGX-A100, the speedup for 8 GPUs is 7.7, and the parallel efficiency is 96%. On DGX-2, the speedup for 16 GPUs is 15.5, and the parallel efficiency is 97%. For the Moana 27GB scene and 8 GPUs the speedup is 7.7 and the parallel efficiency is 96% on DGX-A100 . On DGX-2, the speedup for 16 GPUs is 15.4, and

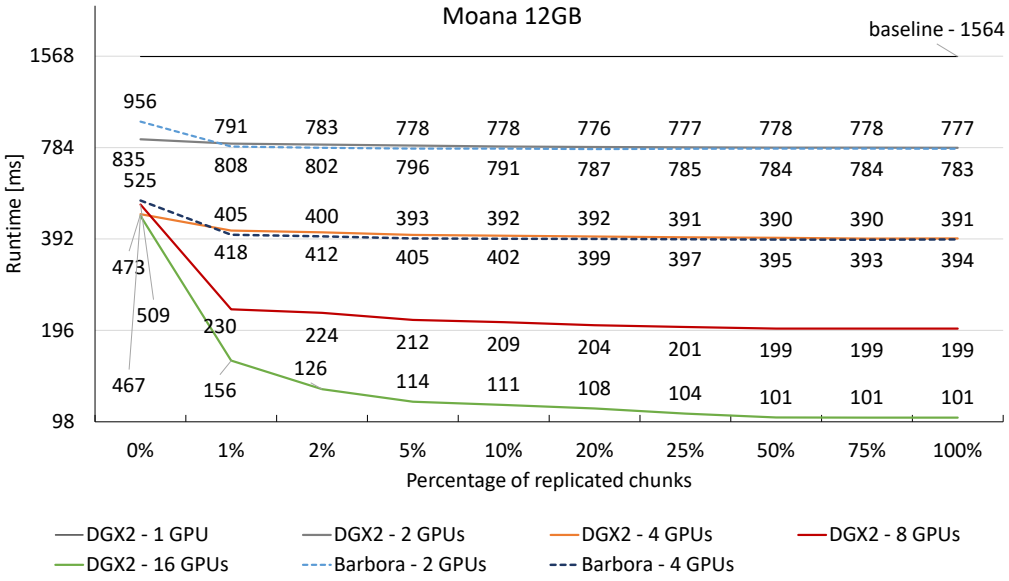


Figure 3.28: Analysis of the path tracing performance for different ratios of replicated and distributed data for the Moana 12GB scenes on the Barbora GPU server and the DGX-2 system. Runtime is for one sample per pixel.

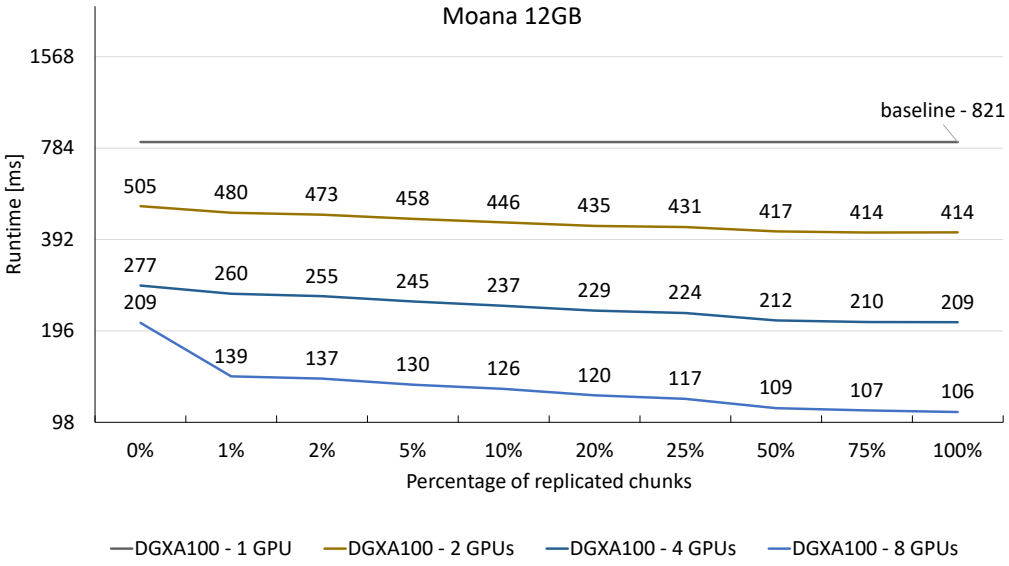


Figure 3.29: Analysis of the path tracing performance for different ratios of replicated and distributed data for the Moana 12GB scenes on the DGX-A100 system. Runtime is for one sample per pixel.

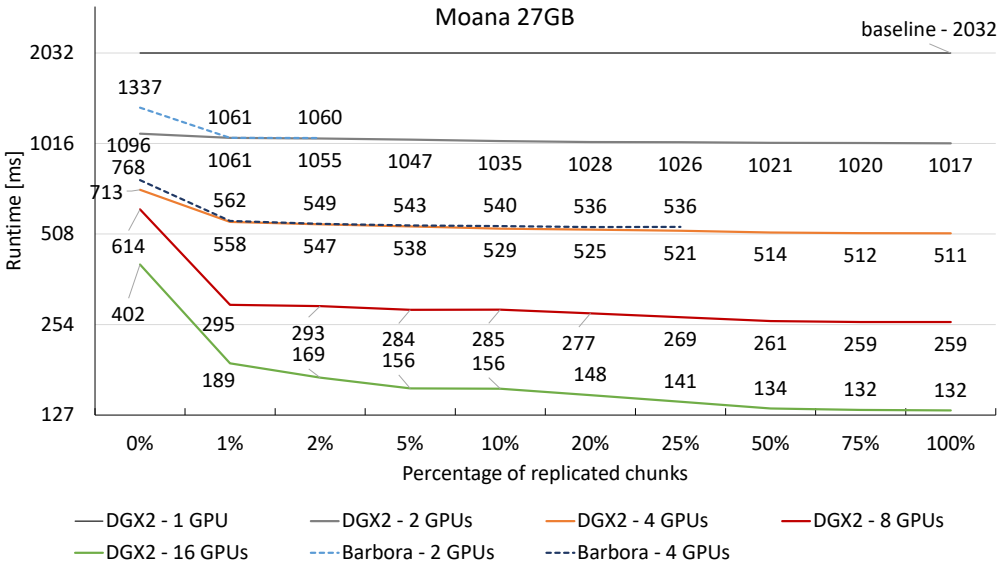


Figure 3.30: Analysis of the path tracing performance for different ratios of replicated and distributed data for the Moana 27GB scenes on the Barbora GPU server and DGX-2. Runtime is for one sample per pixel. A key observation is that the Barbora server with 4 GPUs with 16GB of memory is able to render the Moana 27GB scene as fast as the DGX-2 system with 4 GPUs with 32GB of memory in which the scene is fully replicated.

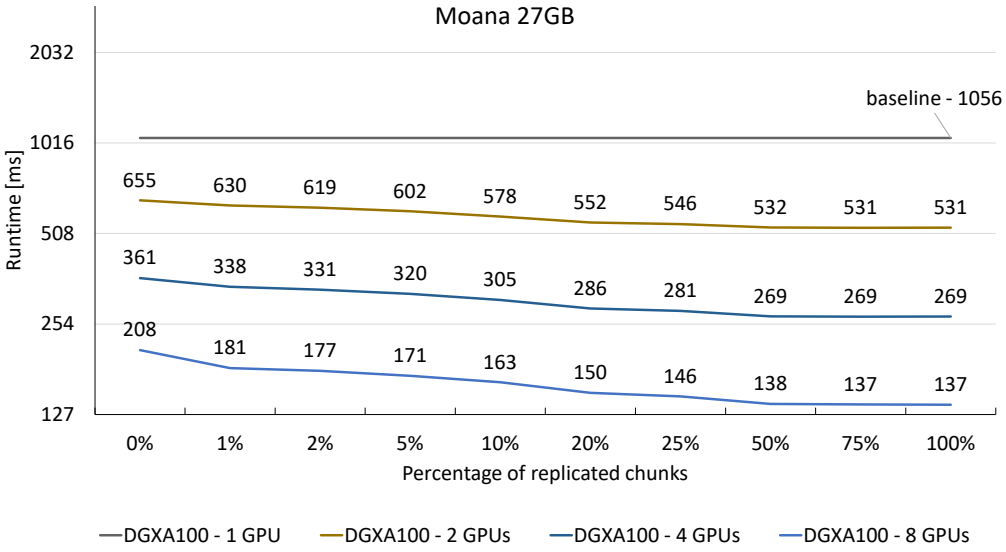


Figure 3.31: Analysis of the path tracing performance for different ratios of replicated and distributed data for the Moana 27GB scenes on the DGX-A100 system. Runtime is for one sample per pixel.

the parallel efficiency is 96%. We expect these results to be positive as in this scenario the path tracing is an embarrassingly parallel problem.

When selected platforms are compared, one can see that there is a performance difference only for fully distributed scenes. Once at least 1% of chunks are replicated, the performance is almost identical.

The Moana 27GB scene does not fit into a single GPU memory used by the Barbora platform. For 2 GPUs, only up to 2% of chunks can be replicated. However, the parallel efficiency is still as high as 96% based on the single GPU baseline measured on the GPU in DGX-2 with 32GB of memory. For 4 GPUs up to 25% of chunks can be replicated on the Barbora server, which yields 95% parallel efficiency against the baseline. For comparison, if one use 4 GPUs on DGX-2 with full scene replication, the parallel efficiency is 99%. This is the first set of results that validates that path tracing on distributed scene data works efficiently.

If the same scene is processed on DGX-2 and 8 GPUs for 2%, 10%, 20%, and 25% of scene replication, the parallel performance is 86%, 89%, 92%, and 94%, respectively. The parallel efficiency for full scene replication is 98%. For 16 GPUs the parallel efficiencies for the same scene replication ratios are 75%, 81%, 86%, and 90%. The baseline parallel efficiency is 96%.

If the Moana 27GB scene is processed on DGX-A100 and 4 GPUs for 2%, 10%, 20%, and 25% of scene replication, the parallel performance is 79%, 86%, 92%, and 94%, respectively. The parallel efficiency for full scene replication is 98%. For 8 GPUs the parallel efficiencies for the same scene replication ratios are 74%, 81%, 88%, and 90%. The baseline parallel efficiency is 96%.

**3.4.2.2.4 Maximum Scene Size Analysis** Up to this point, the scenes used for evaluation have had a size of 12 or 27 GB only. Based on the results from the previous section, the reader should now have an idea of how much of the scene data should be replicated to maintain performance. The following equation describes the maximum ratio of replicated data that fits into the memory of the GPU memory for a scene of a given size:

$$N_{max\_dup} = \frac{\left(G_f - \frac{S_s}{N_g}\right)}{\left(S_s - \frac{S_s}{N_g}\right)} \tag{3.5}$$

where $G_f$ is the amount of free memory per GPU in MB available to store the scene data, $S_s$ is the scene size in MB, and $N_g$ is the total number of GPUs.

For instance, the DGX-2 platform with 16 GPUs where each has 28 GB (out of 32GB) of memory available for the scene can render scenes of sizes of 256 GB, 179 GB, 138 GB, and 112 GB that can have up to 5%, 10%, 15%, and 20% of replicated data, respectively.

In the performance analysis section, one can see that larger scenes of around 100 GB need smaller percentages of replicated data than the smaller ones used here.

# Chapter 4

# Applications

In the previous sections we have introduced the elements from research point of view: the CyclesPhi renderer developed for HPC infrastructures and most importantly its multi-GPU extension for rendering of massive scene on GPUs. This chapter describes how these tools and algorithm are used in practical applications.

## 4.1 Rendering as a service

In both film production as well as scientific visualization a large amount of computational resources are needed to visualize the data. In most cases the scenes are developed by people who do not have sufficient background to work with HPC environment. Based on this fact, we have developed a Rendering-as-a-Service (RaaS) solution that hides the complexity of managing the computations on supercomputers and is able to utilize hundreds or thousands of compute nodes of an HPC cluster with distributed rendering. RaaS is built from three main parts:

- Frontend: Blender HEAppE add-on (BHEAppE), and
- Middleware: High-End Application Execution Middleware (HEAppE),
- Backend: modified version of Cycles production renderer for efficient rendering on HPC cluster (CyclesPhi).

We describe these three parts in the next sections.

### 4.1.1 HEAPPE

HEAppE [24] is a tool that simplifies cluster access, allocating compute nodes, and running jobs for inexperience HPC users (see Figure 4.1). It is a universally designed software architecture that enables unified access to different HPC systems through a simple object-oriented

client-server interface using the Restful Web Services standard (JSON format). This provides users with HPC capabilities without the need to manage running jobs via the command line directly on the cluster.

Each supercomputer cluster is equipped with a scheduler (e.g., PBS, SLURM) to run jobs. However, there are limits to creating and running individual jobs that restrict the rendering of a series of snapshots. The node allocation is time limited and so is the maximum number of jobs that can be created at one time. For a larger number of tasks one can use sub-tasks as extensions of the main tasks. These sub-tasks can be identified by a unique number that, in this case, is also used as a render frame number. For more demanding scenes where rendering of a single frame takes longer time than desired, distributed rendering over multiple nodes can be used.

For this type of task, it is useful to divide the whole process into three parts:

- scene preparation (pre-processing),
- rendering, and
- post-processing.

For each part, a different type of compute node can be allocated, e.g., a node with large memory can be used for pre-processing and a node with more computing power and less memory can be used for rendering. For this purpose, another scheduler function is used, namely the execution of interdependent tasks. That is, a job that performs rendering waits for a job with pre-processing and a post-processing job waits for a render job to finish, respectively. These interdependent jobs only require to have shared storage on the cluster. Based on these requirements, the original HEAppE was extended with several new features.

For security reasons, HEAppE allows its users to run only predefined sets of command templates. Each template defines a script and the type of queue under which the script should be submitted and executed. The template also contains a set of input parameters that can be passed at runtime. Users can only run predefined command templates with a predefined set of input parameters. These parameters can be changed by the user each time a job is submitted.

For each group of users (the group depends on the open call or director call project), a new web server containing an instance of HEAppE is configured. This instance is composed of three parts (see Figure 4.1): the HEAppE core (web server), the SQL Server and the SSH agent (ssh key message). A set of command templates is configured for each instance. Each template contains a script that runs CyclesPhi in the background. Thus, using a simple graphical interface, users can submit their jobs to the cluster and all job processing happens in the background.

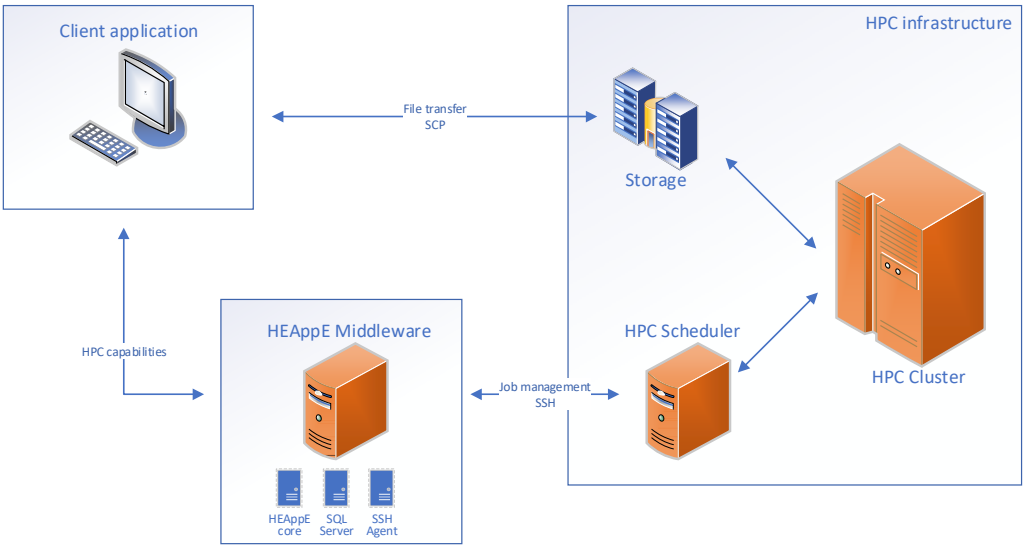


Figure 4.1: HEAppE Middleware Architecture

#### 4.1.1.1 BHEAPPE

To make the RaaS platform as user friendly as possible we have developed the BHEAppE add-on for Blender, which implements the user interaction with the platform. Our add-on is based on the Blender Cloud add-on [137] and can be activated in preferences of the Blender (see Figure 4.2).

In BHEAppE user needs to insert its the login credentials for HEAppE, specify which scp method to use when accessing the cluster storage (OpenSSH application, python paramiko module [138] and destination folder on its local computer where final images will be downloaded to. The main settings of the user job are located in the Render properties panel (see Figure 4.3). The BHEAppE panel is composed of several panels: status, message of the day, new job, list of jobs, job details and storage.

The Status panel (see Figure 4.4) is used for a basic overview of the statuses when communicating with the web server. More information including detailed logs related to the communication is printed to the Blender console.

Our platform supports cluster running Linux system. On such clusters it is common practice to use "message of the day", i.e. motd, which contains a important messages to all users. It informs e.g. about outages or planned downtimes. In BHEAppE these messages can be viewed by clicking on the Message of the day button (see Figure 4.5).

In the New Job panel (see Figure 4.6), a set of input parameters for a new job can be specified. These contains:

- cluster name (e.g. Salomon, Barbara),

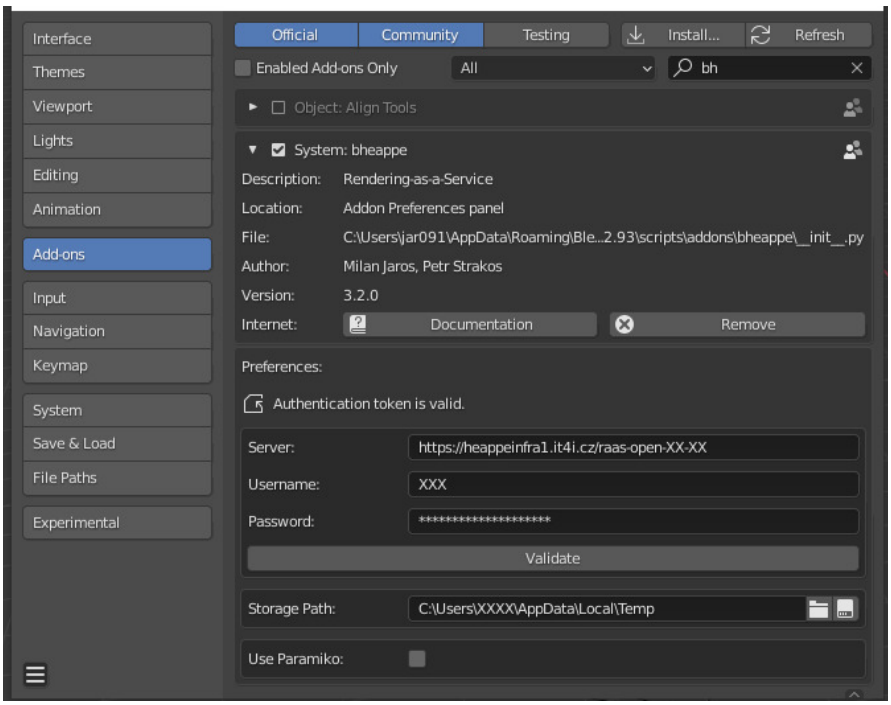


Figure 4.2: The enabling and settings of BHEAppE in Blender Preferences

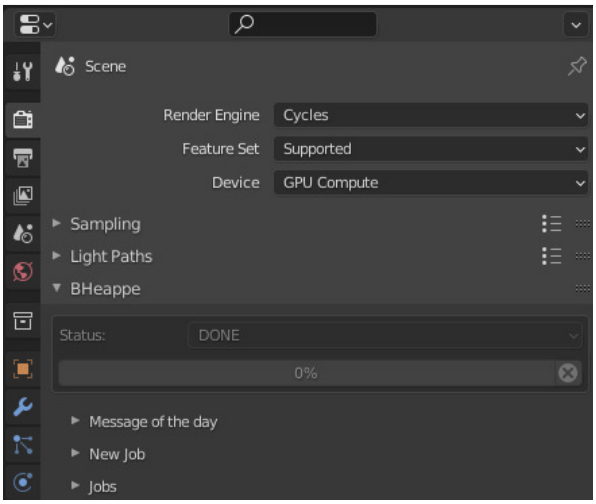


Figure 4.3: The main panel of BHEAppE in the Render properties

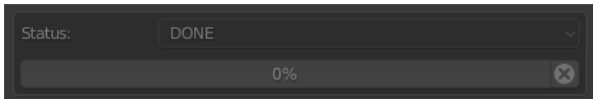


Figure 4.4: Status panel


Figure 4.5: Information about outages or planned downtimes

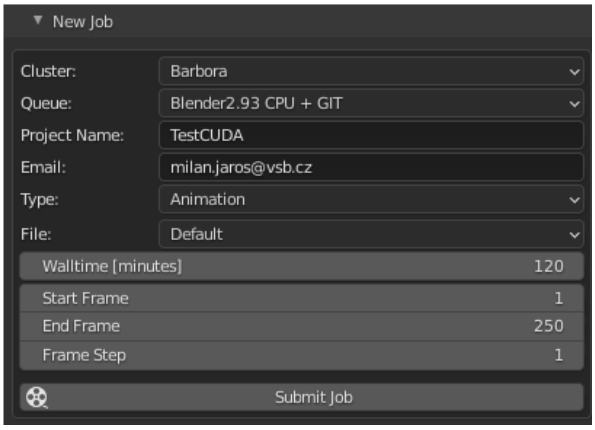


Figure 4.6: The panel for creating and submitting a new job

- Blender Cycles or CyclesPhi version with queue type (e.g. production queue, dedicated queue for nodes with accelerators, massively parallel queue),
- project name,
- email to send information about the completed job,
- number of nodes for distributed rendering, and
- number of snapshots.

A detailed description of the entire process of submitting and running a job is shown in Figure 4.7. By clicking the SubmitJob button (see Figure 4.6) add-on firstly authenticates the user (name and password) using the pre-selected authentication method. This process returns the session code, which is further used for other REST calls.

Prior to submitting the job the entire Blender scene including all dependencies is wrapped using the Blender Asset Tracer python module (BAT) [139] and prepared for transfer to cluster. Next, by calling the REST method CreateJob with all the input parameters add-on will create a JobInfo record in the SQL database for this new HEAppE job. The JobInfo, among others, also contains the unique HEAppE job ID and which is returned to the BHEAppE client.

The job ID is then used to request the FileTransferMethod information, which contains the ssh key needed for the data transfer. The packed scene data is then sent over an encrypted channel to the cluster storage. Once the data transfer is finished, a request to run the job is

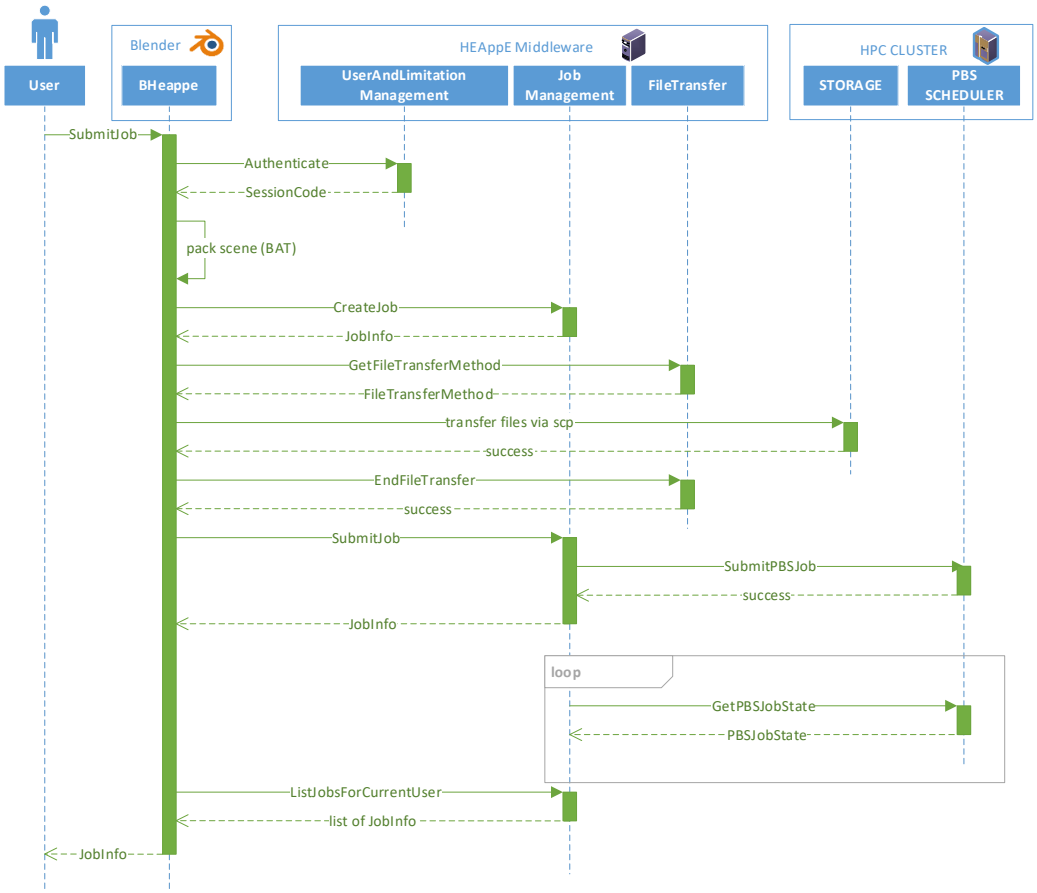


Figure 4.7: The workflow for creating and submitting a new job

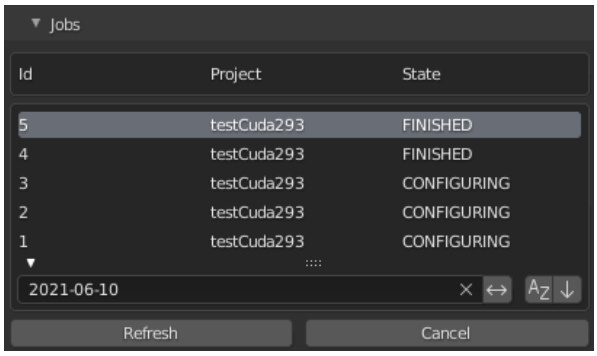


Figure 4.8: The list of submitted jobs

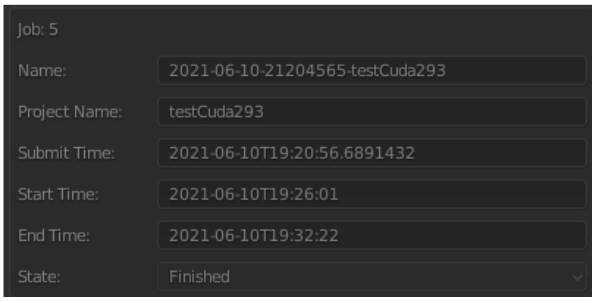


Figure 4.9: The detailed information about the submitted job

submitted to the scheduler using the REST method SubmitJob. The SQL database is updated with the status of the PBS job every 30 seconds. The In the add-on GUI the Refresh button (see Figures 4.8) can be used to get the current information of each job (see Figures 4.9).

Each job has several statuses:

- Configuring - status after job creation,
- Submitted - job was sent to the scheduler,
- Queued - scheduled job,
- Running - running job,
- Finished - completed job,
- Failed - job that did not finish or ended with an error, and
- Canceled - cancelled job by the user.

A running job can be stopped using the Cancel Job button (see Figure 4.8). The whole process is shown in Figure 4.10. After authentication and calling the REST method CancelJob, a request to stop the running PBS job is sent to the PBS scheduler. The current status of the job can be checked again using the Refresh button.

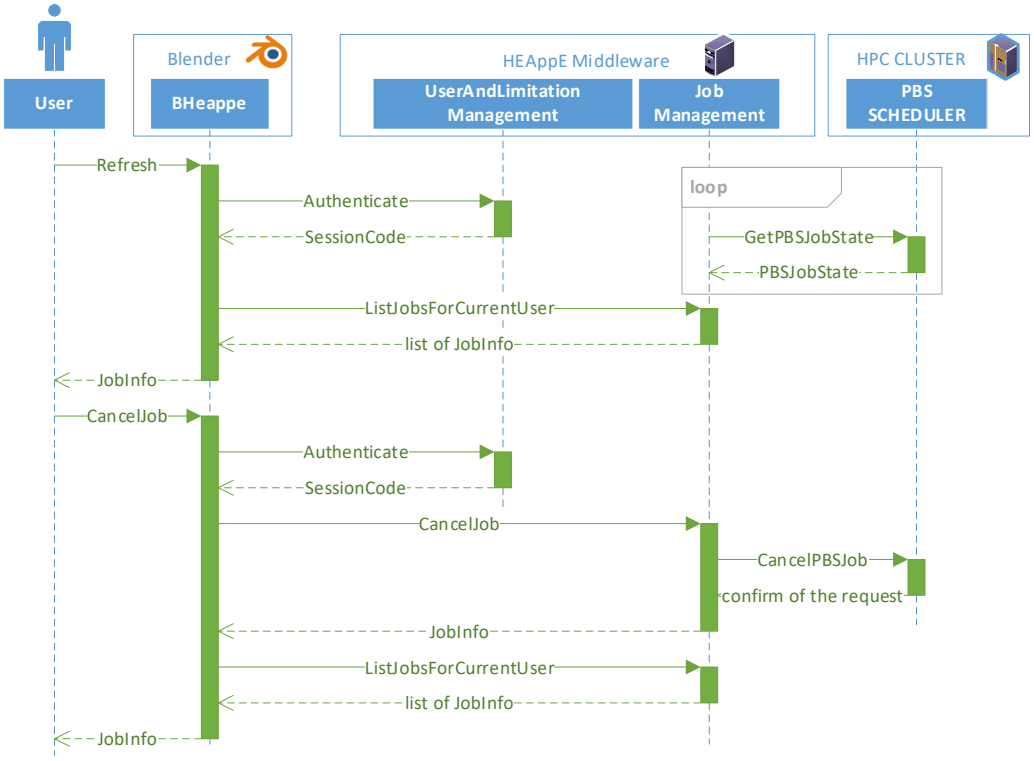


Figure 4.10: The workflow to cancel the submitted job

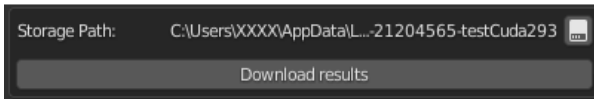


Figure 4.11: The panel for downloading results from a remote cluster to a selected local folder

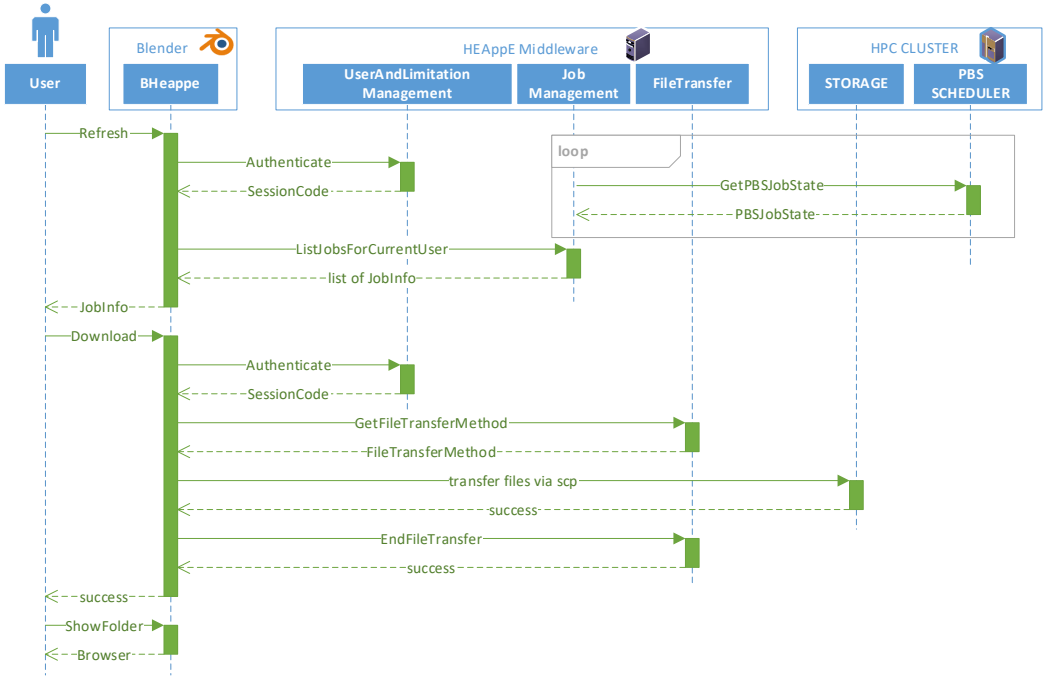


Figure 4.12: The workflow for downloading results from a remote cluster to a selected local folder

After a successful calculation, the results can be downloaded to the local computer using the Download button (see Figure 4.11). The whole process is shown in Figure 4.12. After successful authentication, the data is downloaded using the scp protocol to the folder defined in add-on preferences. The downloaded folder can be viewed using the button (icon) next to the displayed local path.

BHEAppE also supports interactive rendering mode. In this mode, an ssh tunnel is created from the user's local machine to the cluster, i.e. changes in the scene are sent directly to the operating memory of the remote compute node(s). The whole process is illustrated in Figure 4.13. After authentication, creation and sending the job to the scheduler, the PBS job is waiting to run. After the PBS job is started, the ssh key is requested using the REST method GetFileTransferMethod. After the job successfully gets the ssh key, a tunnel to the compute node is created and a TCP port is opened on the local machine. This port is used by the CyclesPhi renderer to communicate with Blender running on the local machine.

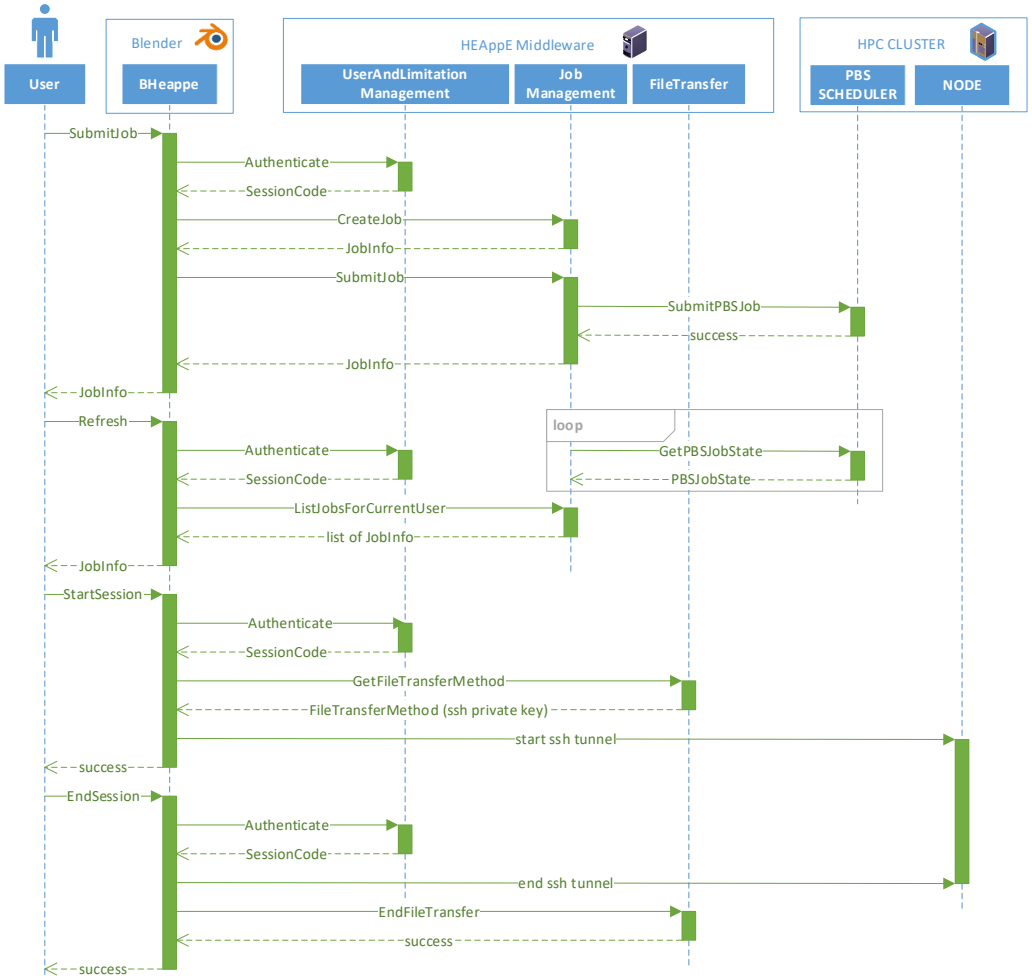


Figure 4.13: The workflow for interactive rendering mode

### 4.1.2 Backend

All jobs that are run using HEAppE contain scripts that run our modified CyclesPhi renderer, introduced in Section 3.3. That is, all of the new functionality introduced can be used in our RaaS service, such as interactive rendering, production rendering, and rendering large scenes. In this section we will describe how CyclesPhi works with incoming data from Blender and how to transfer the resulting image (render buffer) from the CyclesPhi client back to Blender for both production rendering and interactive rendering.

In Section 3.3.3.1, we introduce three modes. All modes can use the MPI technology. Figure 4.14 shows the MPI implementation of rendering using CyclesPhi. The implementation is similar to the original one, but communication thread is added in both Blender and CyclesPhi client. The principle of distributed rendering works as follows:

- Sending KernelData (scene settings) to all nodes using the MPI_Bcast message.
- Sending KernelTextures (scene content) to all nodes using the MPI_Bcast message.
- Send buffer size (for rendered pixels)
- Start rendering using the MPI_Bcast message.
- Reading results from the current node and sending the buffer to the main node using MPI_Gatherv or MPI_Reduce.
- Display results in Blender.

Figure 4.15 shows the case of the Interactive Master - Client mode with remote rendering. In this mode communication is added using TCP sockets and ssh tunneling. The principle of remote rendering works as follows:

- Send KernelData (ex. camera properties) to the socket server and from there to all nodes with Bcast
- Send KernelTextures (ex. triangles) to the socket server and from there to all nodes with Bcast
- Send the information about the size of buffer (rendered pixels) to the socket server and from there to all nodes with Bcast
- Start rendering in the socket client and distribute the command via the socket server by Bcast message
- Read the current results from the node and send the buffer to root with Gatherv or Reduce and then to the socket client.
- View results in Blender

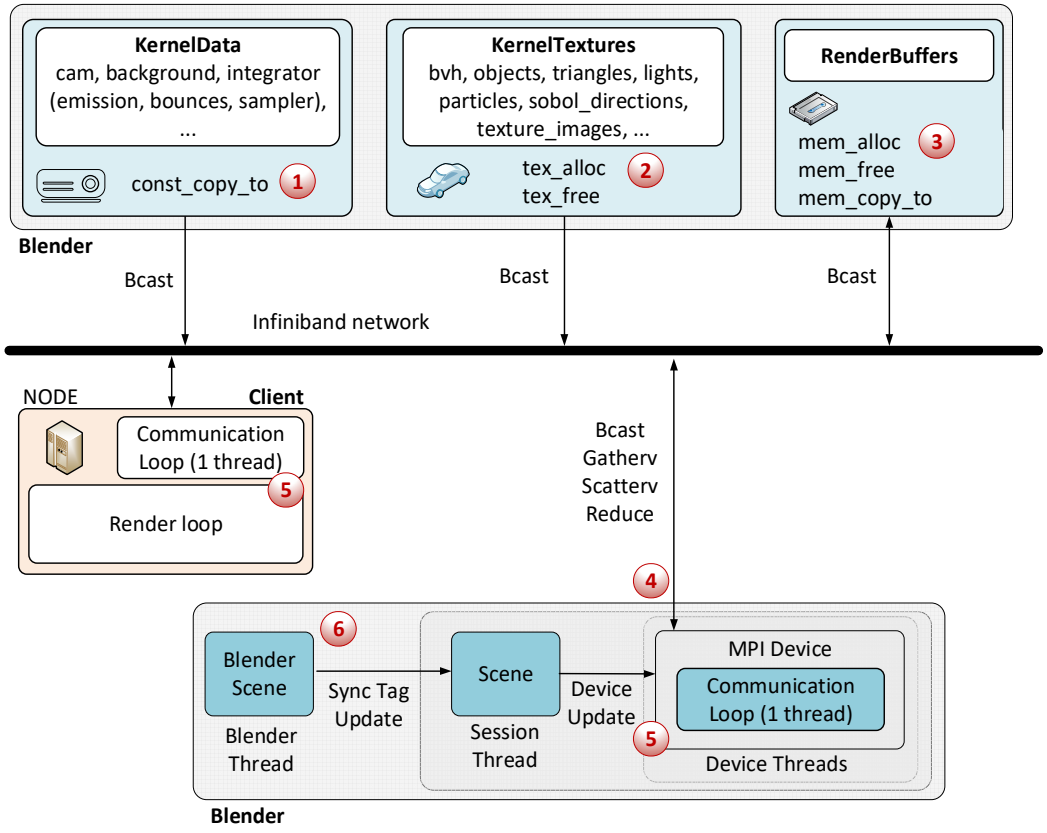


Figure 4.14: The scheme of communication in Interactive or Offline mode on a cluster using MPI

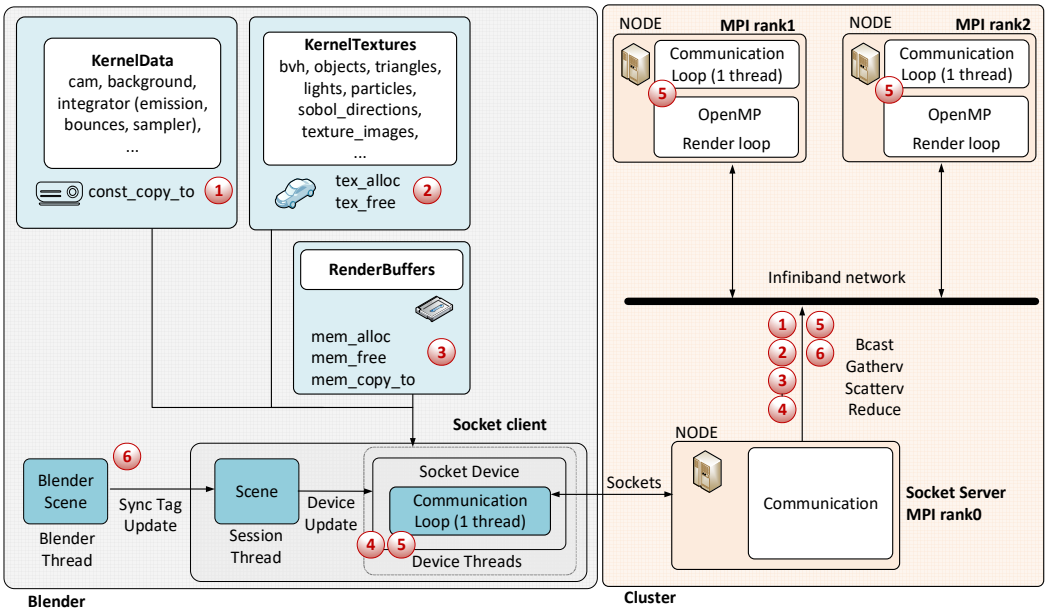


Figure 4.15: The scheme of communication in the Interactive remote mode using TCP sockets and MPI

## 4.2 Interactive Volume Rendering for Medical Visualization

In the previous section we described feeding rendered scene data from Blender and feeding the render buffer in the Interactive Master - Client mode directly to Blender. Our solution can also be used for data visualization on an external device such as a 3D projector with 4k resolution for each eye (see Figure 4.16), which is also part of our laboratory equipment. For this use case, the VRClient application, which uses OpenGL Stereo technology for displaying the image, was created. Native stereo in OpenGL, also known as Quad Buffered Stereo, is a way to provide the graphics accelerator with an image for the left and right eye in a simple way. The 3D projector uses active stereo with polarizing glasses and operates at 120Hz. Thus, each eye is rendered at 60Hz with a resolution of 3840×2400. This differs from other stereo output methods such as placing the left and right images side by side at twice the output resolution. However, the resulting 3D image is rendered and stored this way, side by side. The graphic accelerators on the compute node are divided into two groups on each allocated node. The first group renders the left eye and the second group the right eye. Here we use a common buffer in unified memory. Finally, the resulting image is fed into VRClient instead of Blender (see Figure 4.17). Our visualization lab is connected by a 100Gb/s link directly to the cluster's network, so there is no need to compress the image and thus lose quality. This way of displaying data is suitable for volumetric rendering of data obtained from computed tomography. Volumetric rendering is very computationally intensive, especially when setting

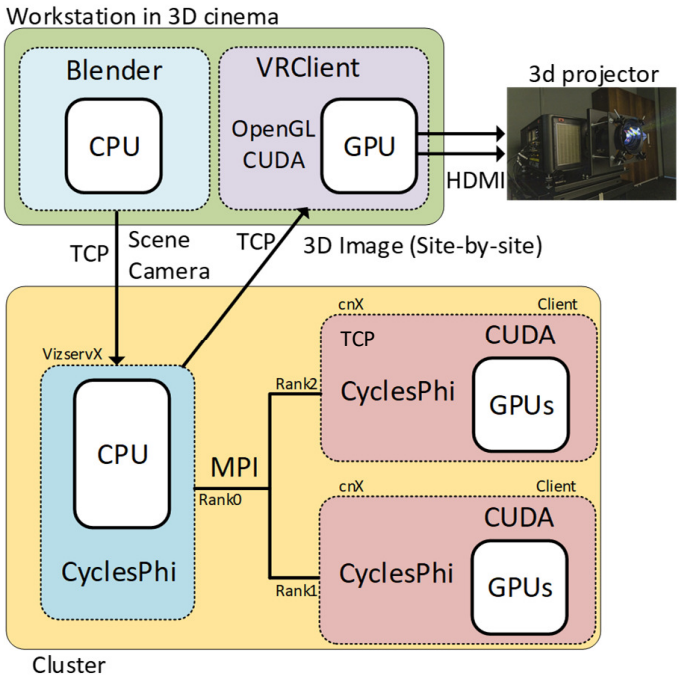


Figure 4.16: The scheme of the Interactive Volume Rendering using VRClient

the maximum detail. CT images are converted to OpenVDB format which can be used by Blender and CyclesPhi. During interactive rendering it is possible to change the shader settings and thus, for example, the colors or the density of voxels as well.

## 4.3 Real-time path tracing for Virtual Reality

In the previous section, we were dealing with interactive rendering, where we were more interested in the quality of the transmitted image than in the transmission speed itself. Another use case where our CyclesPhi extension can be used is rendering for virtual reality (see Figure 4.18). In this case we need to render and transfer data with the highest possible speed and therefore latency. When sending video over a link with lower speeds like 1Gb/s, it is necessary to compress the transmitted video. In cooperation with CESNET [140] and VRgineers [141], we have extended their Ultragrid library [142] to support VR. That is, we need to transmit not only the image for both eyes but also the position and parameters of the camera. For compression we use GPUJPEG [143] which Ultragrid can handle.

For VR, VRClient is extended with OpenVR [144] and Ultragrid libraries. The whole rendering process for VR works in the following way. VRClient receives the new camera position from the OpenVR library and passes this information to the Ultragrid library. CyclesPhi waits for the new camera position from Ultragrid and after receiving the new position, it renders

Figure 4.17: The result of the Interactive Volume Rendering for Medical Visualization using VRClient

the site-by-site image in YUV format to unified memory. This image remains in GPU memory and is passed on to Ultragrid, which compresses this image using the GPUJPEG library and the image is sent over the UDP protocol to the VRClient. The VRClient receives the decoded site-by-site image in RGBA format. This image is mapped to two OpenGL textures (one OpenGL texture per eye) and both textures are sent to OpenVR, which then sends the image to glasses that are compatible with this library, such as the HTC Vive PRO glasses (see Figure 4.19). The whole process is repeated roughly 25 to 90 times per second.

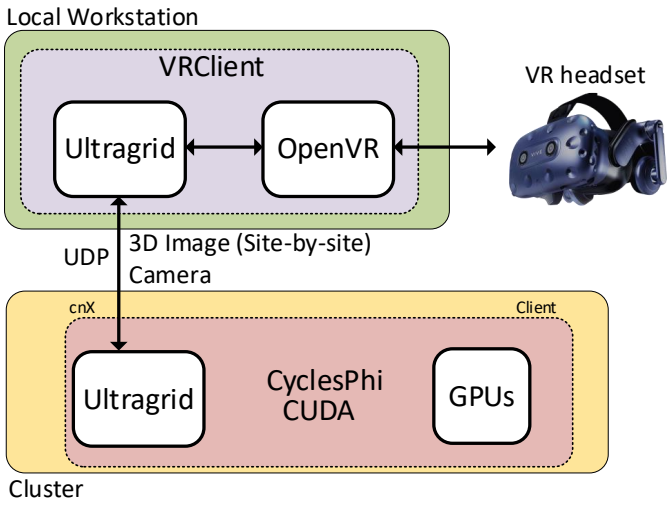


Figure 4.18: The scheme of the Real-time path tracing for Virtual Reality using VRClient

Figure 4.19: HTC VIVE Pro VR headset with wireless adapter

# Chapter 5

# Performance analysis, tests and results

In this chapter we will analyze the performance and benchmarks of the state-of-the-art accelerators introduced in chapter 2 using the methodologies described in chapter 3. The development of CyclesPhi started when Intel Xeon Phi technology was still supported, and for this reason we would like to present some measured results using this older technology.

## 5.1 Intel Xeon Phi

In this test, we demonstrate speed up of the modified version of Cycles. We will compare the rendering times of several rendered scenes using Intel Haswell, KNL, KNC, and Skylake (SKL). Another presented comparison will compare the rendering times for rendering by Cycles using original implementation of Bounding Volume Hierarchy (BVH) with Embree's BVH and with our modification of BVH.

### 5.1.1 Performance of KNC for Path Tracing in Blender Cycles

In this section, we would like to compare the performance of two technologies: MIC and CPU. Figure 5.1 compares Intel Xeon Phi two series KNC (Intel Xeon Phi 7120P) and KNL (Intel Xeon Phi 7250) with Intel Haswell (Intel Xeon E5-2680v3) and Intel SKL (Intel Xeon 8160) processors. These processors are based on x86 architecture and differ mainly in the number of OpenMP threads that were used for computation. The 244 threads were used for KNC, 12 threads for Haswell, 272 threads for KNL and 24 threads for Skylake. KNC uses BVH8. EMBREE-BVH8 is used for Haswell, KNL and SKL. In Figures 5.2, 5.3, 5.4, we can see the performance comparison of the different BVH implementations running on KNC, KNL and SKL.

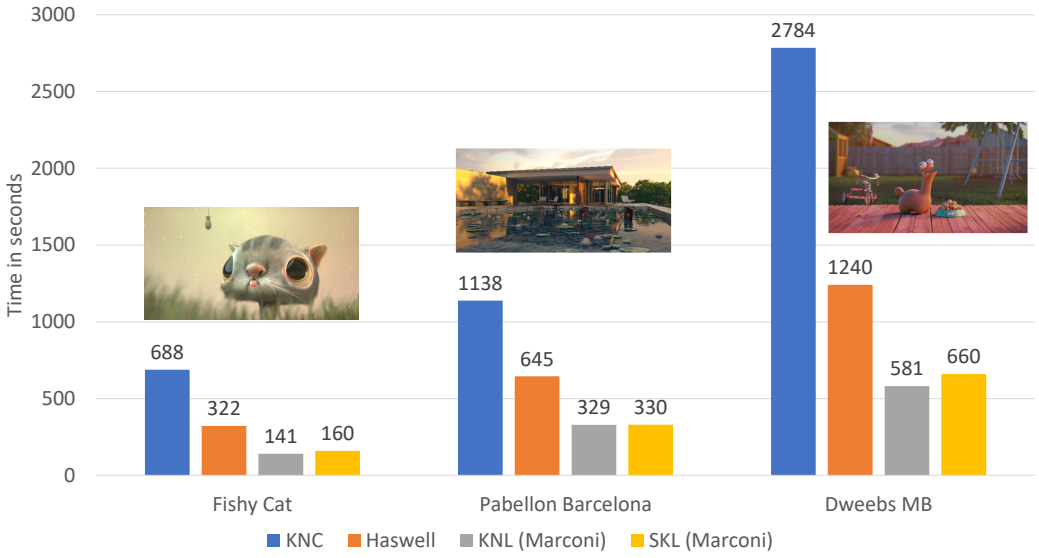


Figure 5.1: Performance comparison of rendering time using Haswell, KNL, KNC, and Skylake

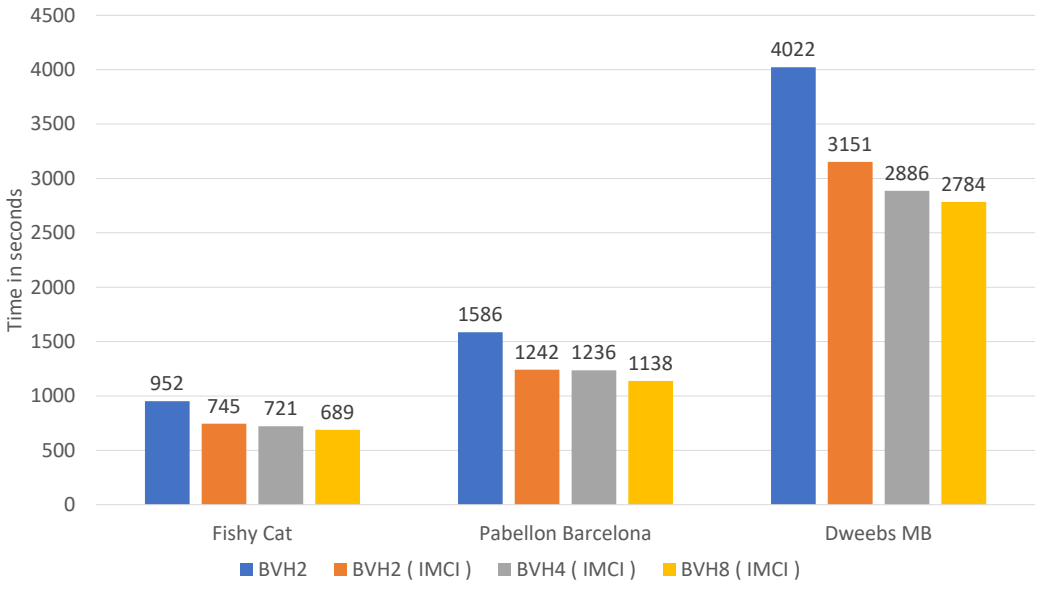


Figure 5.2: Performance comparison of BVH – KNC (Salomon). BVH2, BVH4 – ssef was replaced by avx512f. BVH8 – avxf was replaced by avx512f. BVH2, BVH4, BVH8 contain Ray Triangle Intersection with avx512f.

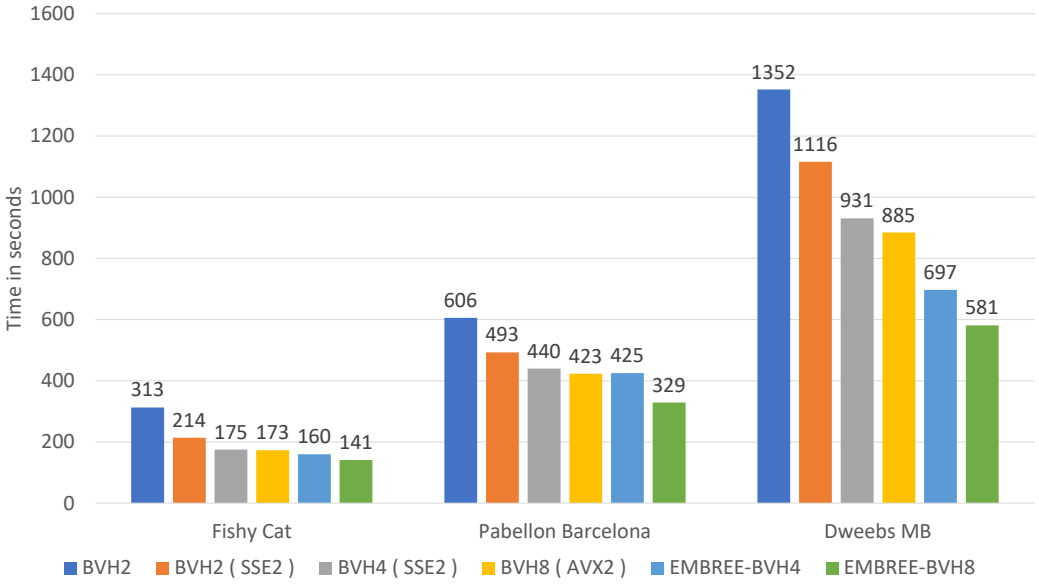


Figure 5.3: Performance comparison of BVH – KNL (Marconi)

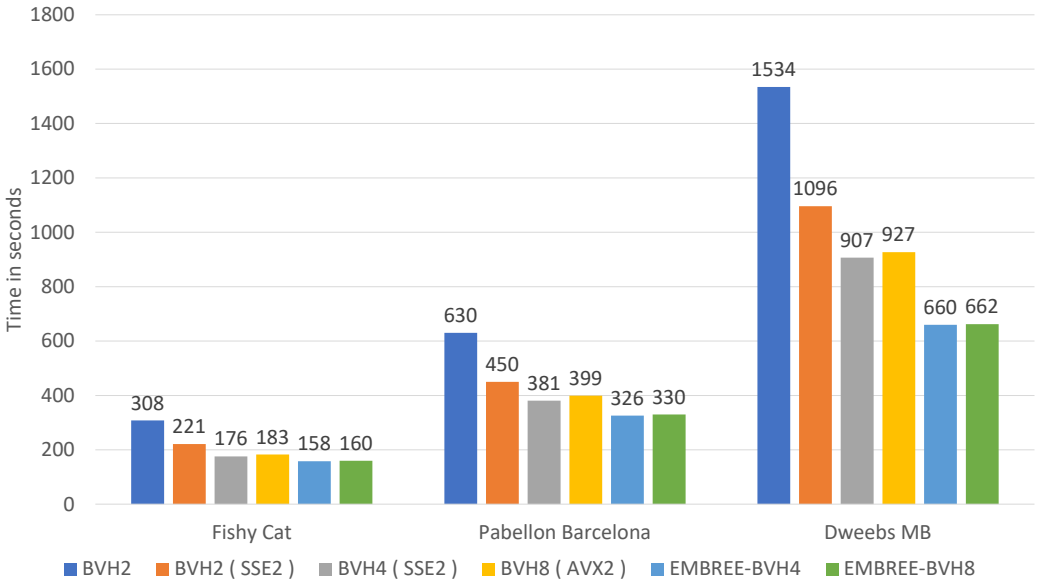


Figure 5.4: Performance comparison of BVH – SKL (Marconi)

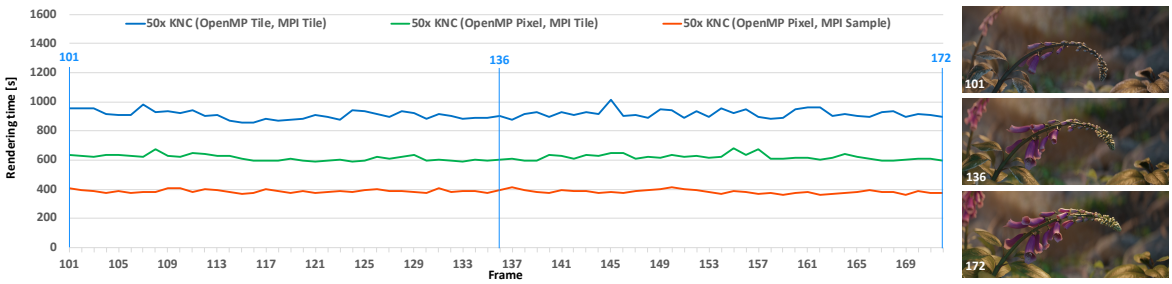


Figure 5.5: The comparison of the rendering times on 50 KNCs with different decomposition over tiles or over samples

Figure 5.6: Benchmark image: Barcelona Pavillion - archviz demo from eMirage (CC-BY), house by Claudio Andres

### 5.1.2 CPU and Intel Xeon Phi Distributed rendering

The Salomon supercomputer at IT4Innovations contained more than 800 Intel Xeon Phi Knights Corner (KNC) cards. Each KNC could be allocated independently to the node using a special queue in the PBS system. For the rendering of Spring we created many tasks where each task used 50 KNCs to render one image. In Figure 5.5, you can see the comparison of the rendering times on 50 KNCs with different decomposition over tiles or over samples.

For next test two complex scenes were chosen, see Figures 5.6,5.7. The first one is the scene with a house and a water pool. The realistic water surface increases the rendering time. The second scene is taken from Laundromat movie. The Victor scene has the grass with lots of details in it which increases the rendering time as well. The House scene consists of 40 thousand triangles with textures. Resolution of the scene is 2048x1024 and 1024 samples. The Victor scene consists of 2.4 million triangles with textures, resolution of the scene is 2048x1024 and 1024 samples.

Figure 5.7: Benchmark image: Cosmos Laundromat Demo, the Victor scene (CC-BY)

Time necessary to render the final Victor scene for various number of computational nodes is depicted in Figure 5.8. Linear scalability could be observed here, which indicates efficient parallel implementation. This means, with higher number of computing resources, the total rendering time would decrease even further. Off-line rendering time for the Victor scene depending on the number of computing nodes and different parallelization techniques. OMP24 runs on 24 CPU cores per node with parallelization via OpenMP. Offload denotes acceleration on 24 CPU cores and two Intel Xeon Phi KNC per node in offload mode. Symmetric denotes acceleration on 24 CPU cores and two Intel Xeon Phi KNC per node in symmetric mode.

Time necessary to render the preview of the House scene for various number of computational nodes is depicted in Figure 5.9. In this case, slight deflection from linear behaviour could be observed. This is most likely due to the communication overhead. On the other hand, the minimal conditions for interactive rendering (25 fps, 1 sample per pixel) are met at approximately 48 of computing nodes. Interactive rendering time for the House scene depending on the number of computing nodes and different parallelization techniques. OMP24 runs on 24 CPU cores per node with parallelization via OpenMP. Offload denotes acceleration on 24 CPU cores and two Intel Xeon Phi KNC per node in offload mode. Symmetric denotes acceleration on 24 CPU cores and two Intel Xeon Phi KNC per node in symmetric mode.

The presented implementation is able to share workload between CPU and additional Intel Xeon Phi effectively for both off-line and interactive mode. The parallel implementation exhibits almost linear strong scalability up to tested 64 computer nodes for off-line rendering mode. Almost linear strong scalability of interactive mode could be observed up to the 16 nodes. After this, linearity is lost due to communication overhead. However, we are still able to reach real-time rendering by 25fps at approximately 48 of computing nodes.

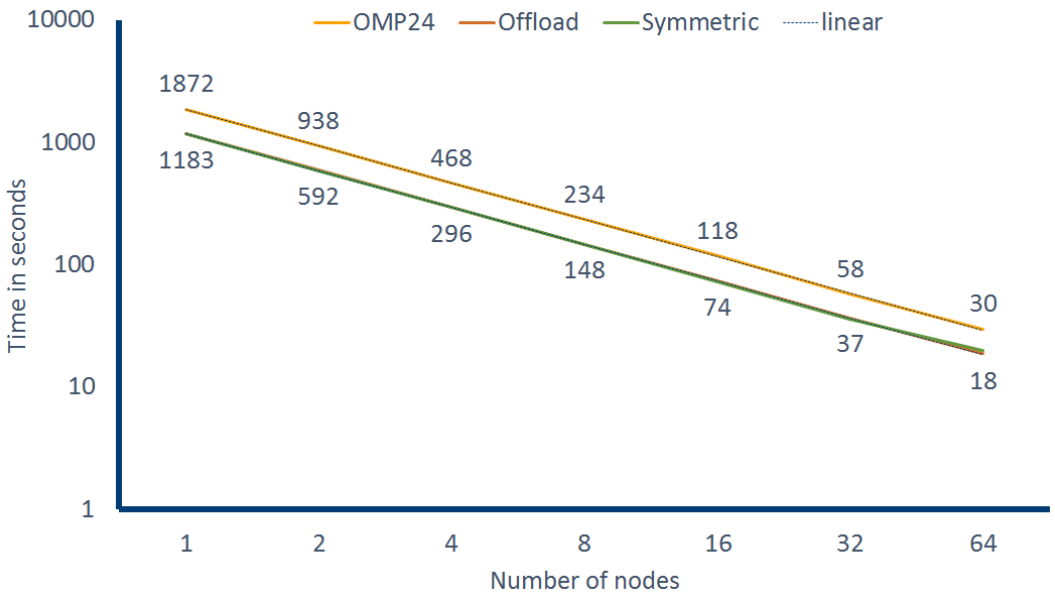


Figure 5.8: Offline rendering time for the Victor scene depending on the number of computing nodes and different parallelization techniques. OMP24 runs on 24 CPU cores per node with parallelization via OpenMP; Offload denotes acceleration on 24 CPU cores and two Intel Xeon Phi per node in offload mode; Symmetric denotes acceleration on 24 CPU cores and two Intel Xeon Phi per node in symmetric mode.

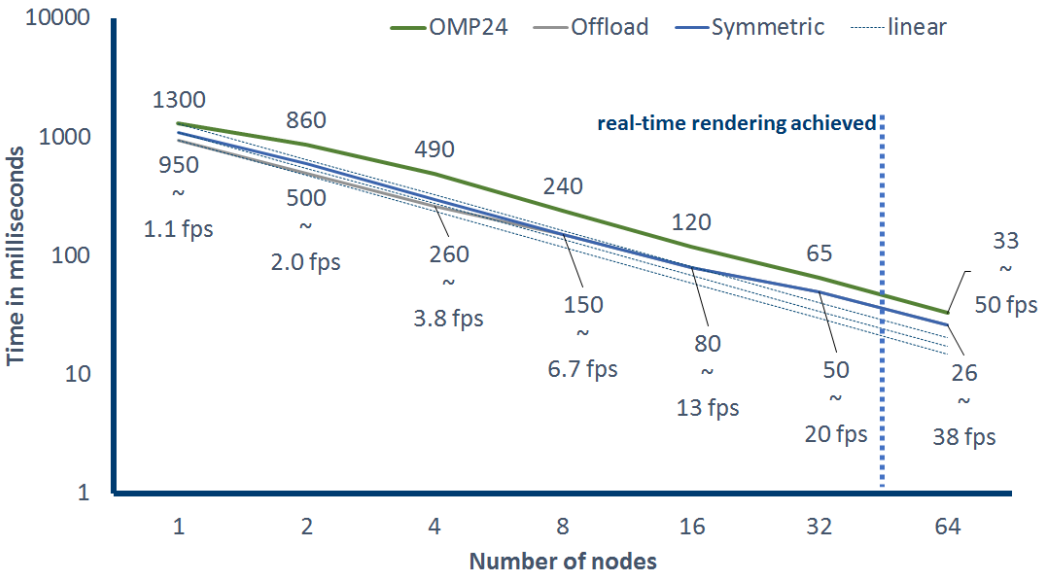


Figure 5.9: Interactive rendering time for the House scene depending on the number of computing nodes and different parallelization techniques. OMP24 runs on 24 CPU cores per node with parallelization via OpenMP; Offload denotes acceleration on 24 CPU cores and two Intel Xeon Phi per node in offload mode; Symmetric denotes acceleration on 24 CPU cores and two Intel Xeon Phi per node in symmetric mode.

## 5.2 GPU Rendering of massive scenes

### 5.2.1 Performance of Tesla V100 and Tesla A100 for Path Tracing in Blender Cycles

In this section we would like to compare the performance of NVIDIA graphics accelerators. The chart compares the Tesla V100 and Tesla A100 GPUs with the high-end consumer-class GeForce RTX 3090 card, which is based on the Ampere architecture and therefore equipped hardware acceleration for ray tracing using RT cores.

The main goal is to provide information about the path tracing performance of Tesla V100 and Tesla A100 against a well-known and widely used GPU card. The results are shown in Figure 5.10. Please, focus on the comparison of *Cycles with OptiX 7 @ GeForce RTX 3090*, which uses the Optix plug-in for Blender [145], and *CyclesPhi with CUDA @ Tesla A100*. It should be noted that Optix employs RT cores for hardware acceleration of BVH tree traversal.

### 5.2.2 Multi-GPU Benchmark Systems

In this benchmark we use three multi-GPU platforms that can perform massive scene rendering. The first one is a BullSequana X410-E5 NVLink-V blade server [116] with 4 Tesla

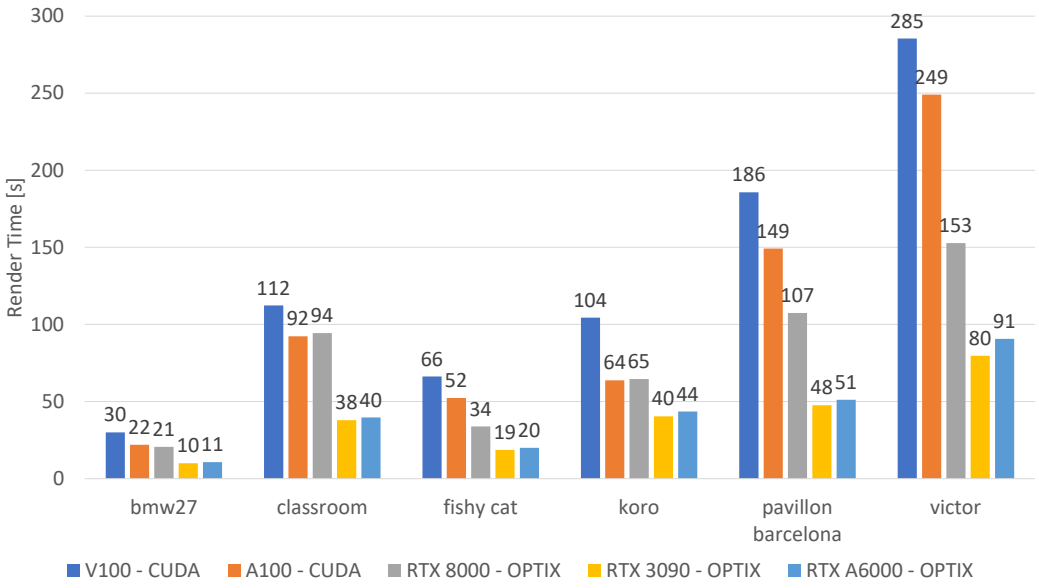


Figure 5.10: The results show that hardware acceleration in OptiX 7 boosts the performance approximately 2 ×; see the results for *Cycles w. OptiX* and *Cycles w. CUDA*, both on GeForce RTX 3090. In the rest of the work, all the results are measured using CyclesPhi with CUDA on Tesla V100 and Tesla A100.

V100 GPUs [146], each with 16GB of memory and direct NVLink interconnect. The server is installed in the Barbora HPC cluster at IT4Innovations [147] and therefore we will refer to it as *Barbora*.

The second, more advanced platform is *NVIDIA DGX-2* [35], which is able to process massive scenes of sizes up to 512 GB in the shared memory of its 16 Tesla V100 GPUs, each with 32 GB of memory. The uniqueness of this platform is the enhancement of the NVLink interconnect by using NVSwitches [32], which enable the connection of all 16 GPUs and higher bandwidth.

The third, next generation platform of DGX-2 is *NVIDIA DGX-A100* [148], which is able to process massive scenes of sizes up to 320 GB in the shared memory of its 8 Ampere A100 GPUs, each with 40 GB of memory. This platform is equipped with NVLink connectivity using NVSwitch [32] switches, which allow all 8 GPUs to be connected and therefore higher bandwidth.

Similar machines can be found in the portfolio of many cloud providers and HPC centers. The key hardware parameters of both platforms related to this benchmark are summarized in Table 5.1.

Table 5.1: Parameters of HW platforms used for validation of our proposed approach. Bandwidth and latency is measured by the STREAM benchmark.

| Server | GPUs | Local memory bandwidth & latency | Remote memory bandwidth & latency | Total GPU memory |
|---|---|---|---|---|
| Barbora | 4x V100 | 740 GB/s<br>4 $\mu$s | 48 GB/s<br>7 $\mu$s | 64 GB |
| DGX-2 | 16x V100 | 790 GB/s<br>4 $\mu$s | 138 GB/s<br>10 $\mu$s | 512 GB |
| DGX-A100 | 8x A100 | 1193 GB/s<br>4 $\mu$s | 252 GB/s<br>10 $\mu$s | 320 GB |

### 5.2.3 Benchmark Scenes

We used four scenes for benchmarking. The first one was the *Moana Island Scene* [149]. It was selected due to its uniqueness and wide acceptance by both industry and the research community as a key production grade benchmark. The second one is based on the Museum scene [150] and is extended by sculpture models [151] to increase the geometric complexity, and by paintings [152] to increase the size and number of textures.

Finally, the remaining two scenes come from the recent open movies produced by the animation studio of the Blender Institute, namely *Agent 327: Operation Barbershop* [153] and *Spring* [154].

These scenes of different sizes were created through the surface subdivision functionality and by increasing the texture resolution.

### 5.2.4 Performance for Massive Scenes

In this section, we evaluate the performance of the proposed method using two groups of scenes:

- **Group 1**: *Moana 38GB*, *Museum 41GB*, *Agent 37GB*, and *Spring 41GB* are designed to stress the Barbora GPU server with 64 GB of total GPU memory, see Figures 5.15,5.16,5.17,5.18, and
- **Group 2**: *Moana 169GB*, *Museum 124GB*, *Agent 167GB*, and *Spring 137GB* are designed to stress the DGX-2 and the DGX-A100 server with 512 GB and 320GB of total GPU memory, see Figures 5.19,5.20,5.21,5.22[1].

All scenes can be seen in Figures 5.11,5.12,5.13,5.14 and the key parameters of the second group are shown in Table 5.3. All results are presented for 1 sample per pixel, but as Table 5.2

[1]We were not able to work with larger scenes because Cycles internally uses 32 bit integers for some key data structures.


Figure 5.11: Moana Island Scene: the rendering time is for the different number of samples per pixel measured on the NVIDIA DGX-2 machine with 16 GPUs. All the scene data, including geometry, were partially distributed among the memories of all GPUs.

shows, these values can be linearly extrapolated to any number of samples per pixel. As the topic of out-of-core algorithms for textures is very extensively covered in the literature, we use scenes where both geometry and textures are larger than a single GPU memory and their ratio differs from scene to scene. The goal is to show that our approach is general, and works for textures as well as for geometry, which is the more challenging to distribute while maintaining good performance of path tracing.

The first key observation is that for different scene sizes, a different chunk size must be used to reach optimal performance, as presented in Section 3.4.2.2. For Group 1 the best performance was achieved with 16MB chunks. For Group 2 the optimal chunk size is 64MB.

The results for Group 1 are shown in Figures 5.15,5.16,5.17,5.18. The following conclusions can be made from the results:

- the performance of the Barbora server is almost identical to the performance of DGX-2 for the same amount of scene replication (up to 10%).

- DGX-2 is able to further replicate scene data up to 60%, which improves performance by 2.8% only in the case of the Moana 38GB scene (for the other scenes the performance is higher by only less than 1%[7]).

[7]Museum 41GB by 0.2%, Agent 37GB by 0,2%, and Spring 41GB by 0.7%.


*DGX-2: 21 min 49 sec* *DGX-2: 2 min 6 sec* *DGX-2: 12.6 sec*

Figure 5.12: Museum: the rendering time is for the different number of samples per pixel measured on the NVIDIA DGX-2 machine with 16 GPUs. All the scene data, including geometry, were partially distributed among the memories of all GPUs.


*DGX-2: 56 min 18 sec* *DGX-2: 5 min 37 sec* *DGX-2: 33.8 sec*

Figure 5.13: Agent327: the rendering time is for the different number of samples per pixel measured on the NVIDIA DGX-2 machine with 16 GPUs. All the scene data, including geometry, were partially distributed among the memories of all GPUs.


Figure 5.14: Spring: the rendering time is for the different number of samples per pixel measured on the NVIDIA DGX-2 machine with 16 GPUs. All the scene data, including geometry, were partially distributed among the memories of all GPUs.

Table 5.2: The preprocessing time and final rendering time for the different number of samples per pixel measured on 16 GPUs of the DGX-2 system. Chunk size is 64MB, replication ratio is 10%. Access pattern and chunk distribution is calculated from first sample only, all other samples uses the same chunk distribution.

| Scene | Moana 169GB | Museum 124GB | Agent 167GB | Spring 137GB |
|---|---|---|---|---|
| *Preprocessing* | | | | |
| 1spp pre-pass on GPUs | 0.75 s | 0.92 s | 0.74 s | 1.18 s |
| Algorithm runtime | 0.06 s | 0.01 s | 0.01 s | 0.01 s |
| Chunk redistribution | 66.1 s | 28.2 s | 50.9 s | 32.6 s |
| *Rendering* | | | | |
| 1 sample | 0.196 s | 0.125 s | 0.332 s | 0.498 s |
| 10 samples | 1.94 s | 1.25 s | 3.37 s | 5.01 s |
| 100 samples | 18.7 s | 12.6 s | 33.8 s | 50.1 s |
| 1000 samples | 181 s | 126 s | 337 s | 502 s |
| 10000 samples | 1852 s | 1309 s | 3378 s | 5016 s |

Table 5.3: Parameters of the largest scenes used for performance evaluation of the presented approach on DGX-2.

| Scene | Moana 169GB | Museum 124GB | Agent 167GB | Spring 137GB |
|---|---|---|---|---|
| GPU memory needed for path tracing | 169 GB | 124 GB | 167 GB | 137 GB |
| Image resolution | 5120 × 2560 | 5120 × 2560 | 5120 × 2560 | 5120 × 2560 |
| Geometry size | 90 GB | 69 GB | 57 GB | 48 GB |
| Triangles count | 673 M | 610 M | 468 M | 395 M |
| Total textures size | 58 GB | 46 GB | 101 GB | 74 GB |
| Number of textures | 3421 | 22 | 118 | 96 |
| Sizes of key data structures and percentage of total scene size | | | | |
| bvh_nodes | 14 GB (8 %) | 8 GB (6 %) | 6 GB (4 %) | 7 GB (5 %) |
| prim_tri_verts | 30 GB (18 %) | 27 GB (22 %) | 21 GB (13 %) | 18 GB (13 %) |
| prim_tri_index | 3 GB (2 %) | 2 GB (2 %) | 2 GB (1 %) | 2 GB (1 %) |
| svm_nodes | 2 MB (<0.1 %) | 40 kB (<0.1 %) | 4 MB (<0.1 %) | 5 MB (<0.1 %) |
| tex_image | 58 GB (34 %) | 46 GB (37 %) | 101 GB (61 %) | 74 GB (54 %) |
| Percentage of total memory access for key data structures for default bounces | | | | |
| bvh_nodes | 80.3 % | 66.8 % | 52.8 % | 62.8 % |
| prim_tri_verts | 8.1 % | 12.7 % | 8.7 % | 4.8 % |
| prim_tri_index | 2.6 % | 3.9 % | 2.6 % | 1.5 % |
| svm_nodes | 2.1 % | 3.8 % | 19.3 % | 19.2 % |
| tex_image | 0.2 % | 0.5 % | 3.4 % | 2.3 % |
| Average, across all GPUs, amount of replication per data structure for 64MB chunks and 16 GPUs. The values are for 0, 1, 2, 5, and 10 % of replicated chunks. For example, the bold value means that each GPU keeps on average 28 % of the bvh_nodes in its local memory if 2 % of the scene is replicated. | | | | |
| bvh_nodes[2] | 6, 17, **28**, 50, 62 % | 6, 16, 26, 61, 88 % | 6, 25, 33, 44, 64 % | 6, 20, 25, 31, 36 % |
| prim_tri_verts[3] | 6, 6, 6, 10, 20 % | 6, 7, 7, 7, 16 % | 6, 7, 10, 14, 20 % | 6, 7, 10, 16, 23 % |
| prim_tri_index[4] | 6, 6, 6, 27, 48 % | 6, 11, 11, 19, 57 % | 6, 16, 19, 39, 52 % | 6, 10, 17, 39, 46 % |
| svm_nodes[5] | 6, 100, 100, 100, 100 % | 6, 100, 100, 100, 100 % | 6, 100, 100, 100, 100 % | 6, 100, 100, 100, 100 % |
| tex_image[6] | 6, 6, 6, 6, 6 % | 6, 6, 7, 7, 7 % | 6, 6, 6, 7, 10 % | 6, 6, 6, 7, 11 % |

- This means that for scenes of sizes approximately up to 45GB distributed over 4 GPUs, the significantly less complex and cheaper GPU interconnect in the Barbora server is sufficient.

- For the Museum, Agent, and Spring scenes 2% of scene replication attains optimal performance. This holds for 4, 8 and 16 GPUs.

- Only the Moana scene needs higher amounts of replicated data, up to 25% for 16 GPUs.

- Scalability can be evaluated on DGX-2 for 4, 8, and 16 GPUs only. For the Moana, Museum, Agent, and Spring scenes, for 5% scene replication, the parallel efficiencies, going from 4 to 16 GPUs, are 82.7%, 97.1%, 97,9%, and 98.1%, respectively. In the case of the Moana scene, a higher replication ratio is needed to improve scalability, e.g., for 25% data replication ratio the parallel efficiency is 94,4%.

The results for Group 2 are shown in Figure 5.19,5.20,5.21,5.22. One can see that:

- The performance is affected by selecting the right chunk size, particularly for the Moana scene.

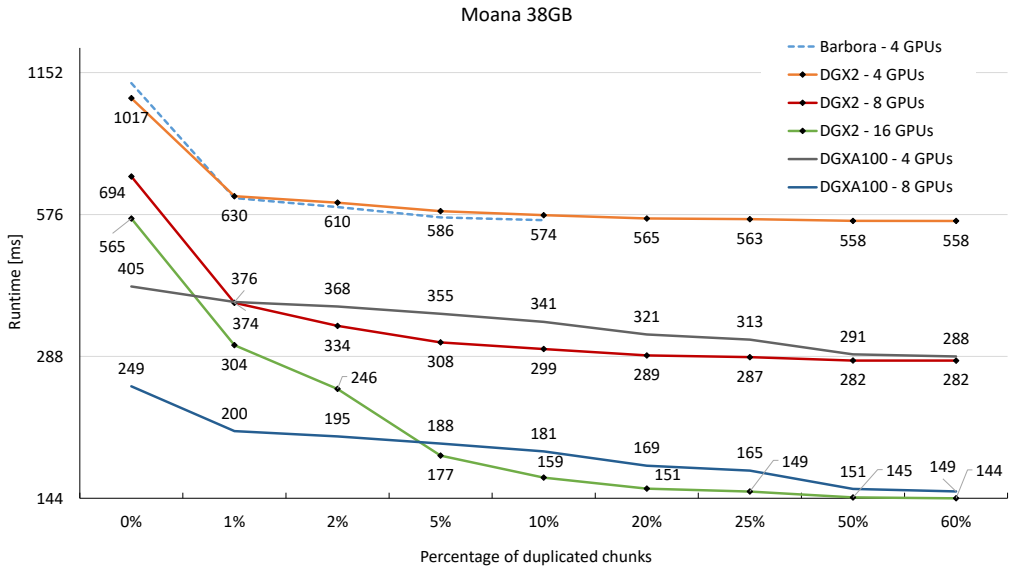


Figure 5.15: Path tracing times for the Moana 38GB scene (Group 1) running on 4 GPUs of the Barbora GPU server, 4,8, and 16 GPUs of the DGX-2 system and 4 and 8 GPUs of the DGX-A100 system. The key observation is that the Barbora server with lower performing GPU interconnect has the same performance as the DGX-2 system for 4 GPUs.

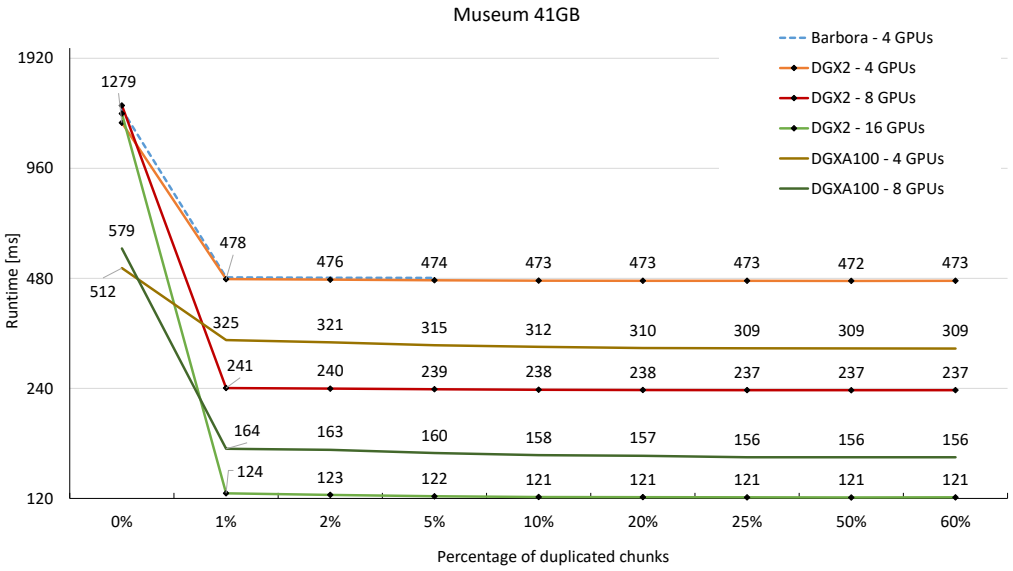


Figure 5.16: Path tracing times for the Museum 41GB scene (Group 1) running on 4 GPUs of the Barbora GPU server, 4,8, and 16 GPUs of the DGX-2 system and 4 and 8 GPUs of the DGX-A100 system. The key observation is that the Barbora server with lower performing GPU interconnect has the same performance as the DGX-2 system for 4 GPUs.

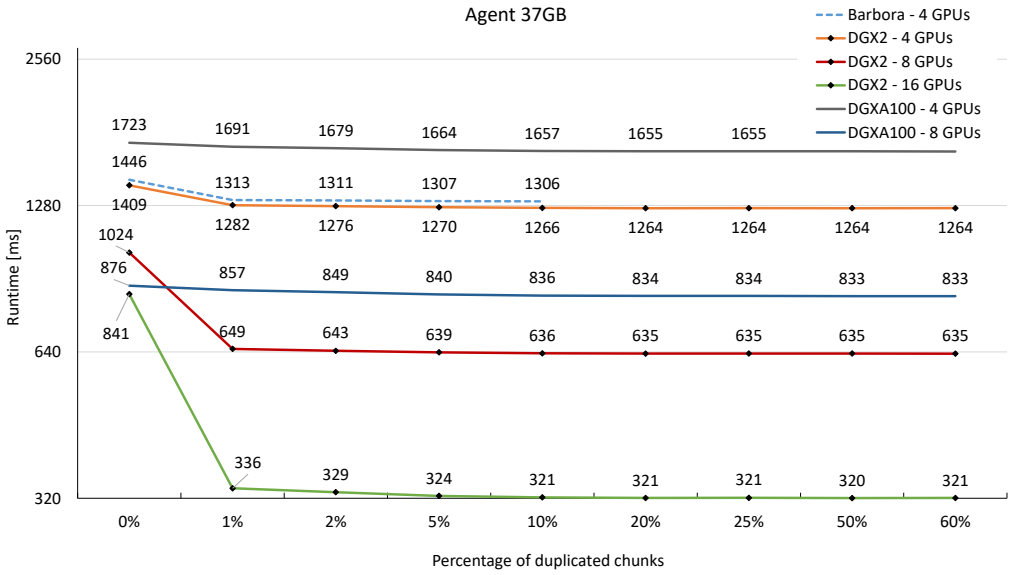


Figure 5.17: Path tracing times for the Agent 37GB scene (Group 1) running on 4 GPUs of the Barbora GPU server, 4,8, and 16 GPUs of the DGX-2 system and 4 and 8 GPUs of the DGX-A100 system. The key observation is that the Barbora server with lower performing GPU interconnect has the same performance as the DGX-2 system for 4 GPUs.

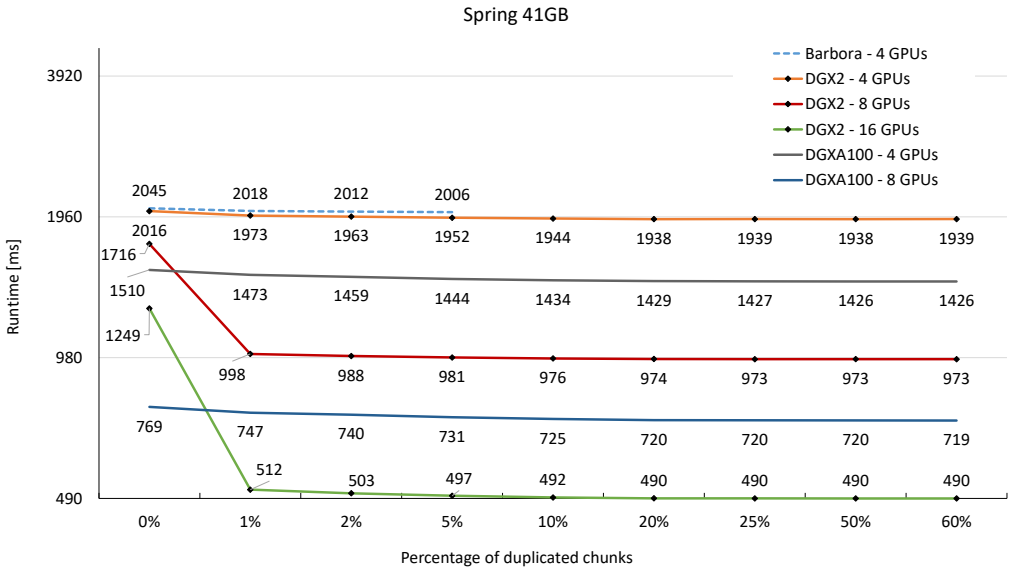


Figure 5.18: Path tracing times for the Spring 41GB scene (Group 1) running on 4 GPUs of the Barbora GPU server, 4,8, and 16 GPUs of the DGX-2 system and 4 and 8 GPUs of the DGX-A100 system. The key observation is that the Barbora server with lower performing GPU interconnect has the same performance as the DGX-2 system for 4 GPUs.

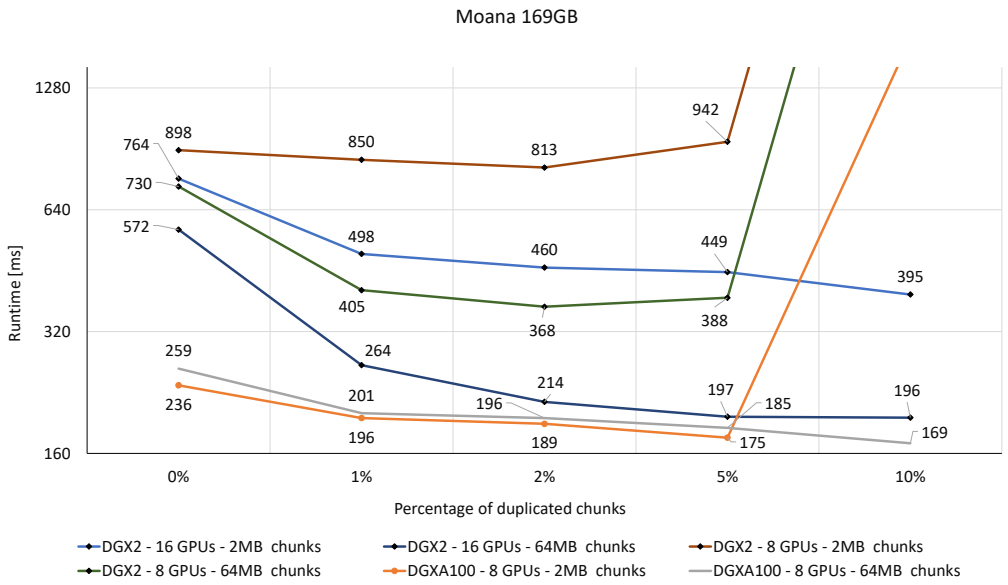


Figure 5.19: Path tracing times for the Moana 169GB scene (Group 2) running on 8 and 16 GPUs of the DGX-2 system and 8 GPUs of the DGX-A100 system. The results show how the performance is affected by changing the chunk size from 2 to 64 MB. The runtimes are for 1 sample per pixel. 0% of replicated chunks represents the fully distributed scene.

- For 8 GPUs and the Moana, Agent, and Spring scenes, if the replication ratio is 10%, the scene does not fit into the GPU shared memory anymore and chunks are swapped between the GPU and CPU memory [8] which makes the rendering several times slower depending on the number of chunks being moved back and forth. This is the point at which our approach stops working, and therefore it is crucial to correctly select the replication ratio to avoid this situation.

- For the Agent scene, 1% of scene replication gives optimal performance for both 8 and 16 GPUs. The Spring scenes needs only 0.1% for 8 GPUs and 0.25% for 16 GPUs. The Museum scene needs 1% for 8 GPUs and 2% for 16 GPUs. Finally, the Moana scene requires 2% for 8 GPUs and 5% for 16 GPUs.

- Scalability between 8 and 16 GPUs is good for all scenes. The parallel efficiencies are 93.8%, 98.8%, 98.6%, and 99.0% for the Moana, Museum, Agent, and Spring scenes, respectively.

Another important feature of the proposed approach is that we analyze memory access pattern only for one sample per pixel. Based on this pattern we distribute and replicate the chunks only once, and then use this chunk placement for all remaining samples. The cost of this preprocessing is shown in Table 5.2. We have verified that it is sufficient through a set

[8] This is the default behavior of the CUDA Unified Memory.

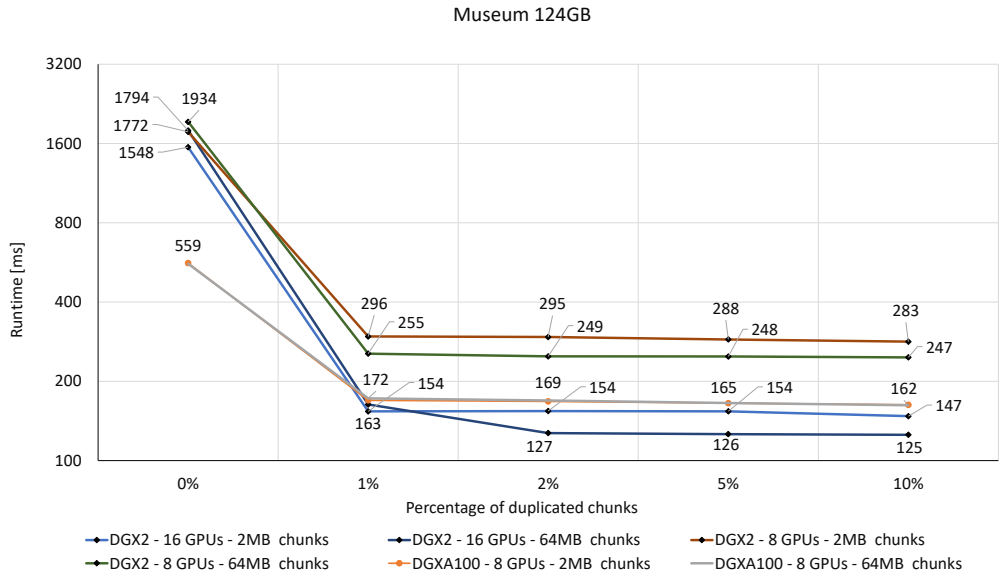


Figure 5.20: Path tracing times for the Museum 124GB scene (Group 2) running on 8 and 16 GPUs of the DGX-2 system and 8 GPUs of the DGX-A100 system. The results show how the performance is affected by changing the chunk size from 2 to 64 MB. The runtimes are for 1 sample per pixel. 0% of replicated chunks represents the fully distributed scene.

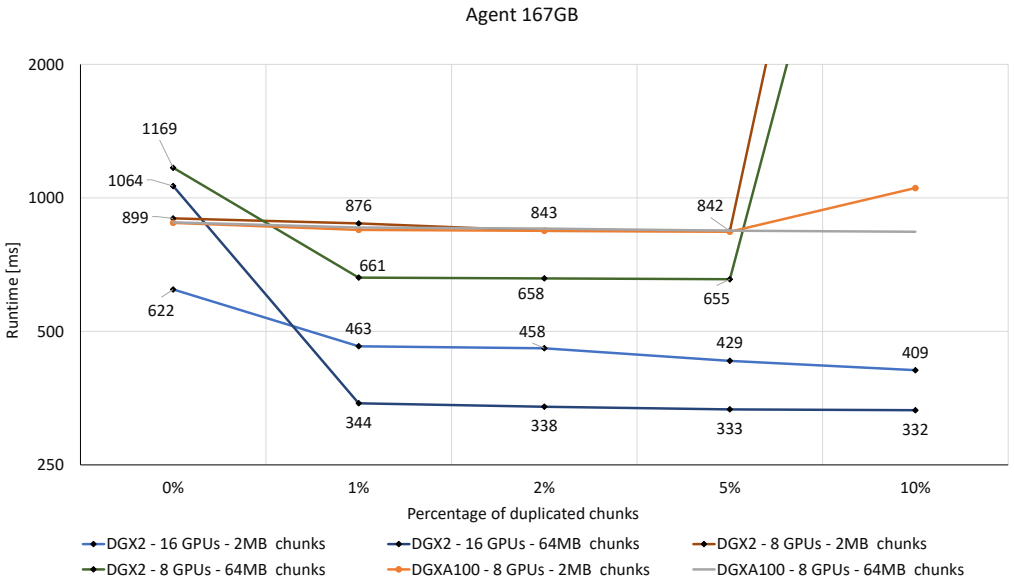


Figure 5.21: Path tracing times for the Agent 167GB scene (Group 2) running on 8 and 16 GPUs of the DGX-2 system and 8 GPUs of the DGX-A100 system. The results show how the performance is affected by changing the chunk size from 2 to 64 MB. The runtimes are for 1 sample per pixel. 0% of replicated chunks represents the fully distributed scene.

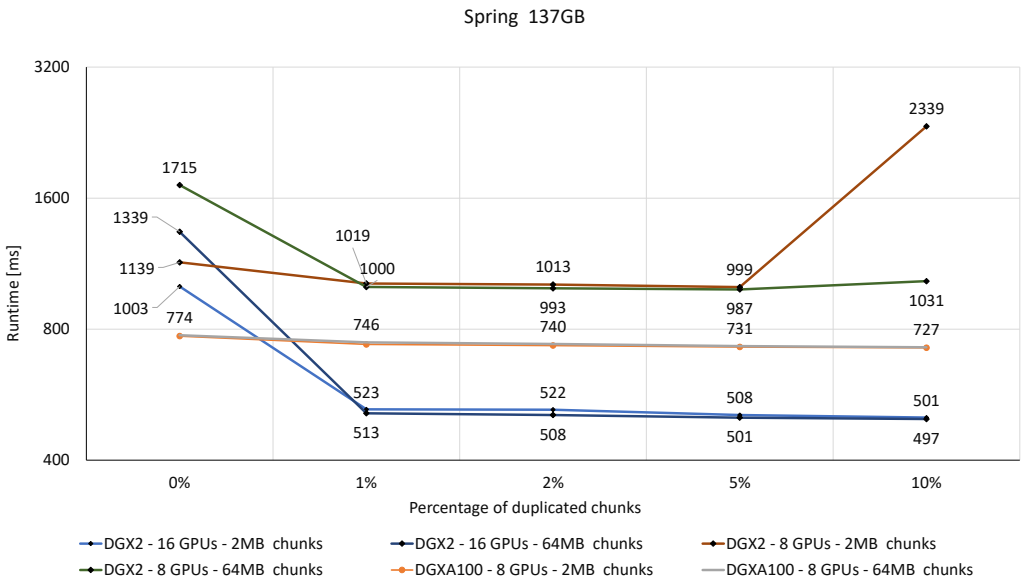


Figure 5.22: Path tracing times for the Spring 137GB scene (Group 2) running on 8 and 16 GPUs of the DGX-2 system and 8 GPUs of the DGX-A100 system. The results show how the performance is affected by changing the chunk size from 2 to 64 MB. The runtimes are for 1 sample per pixel. 0% of replicated chunks represents the fully distributed scene.

of measurements using Group2 scenes for 1 to 10 000 samples. The results are in the same table, where one can see that rendering times grow linearly with the number of samples.

### 5.2.5 GPU Accelerated Distributed rendering

Finally, for the Moana 169GB scene and for the Museum 124GB scene, we evaluated the performance of Cycles on 256 GPUs (NVIDIA A100). This strong scalability test was performed for 10 000 samples and for 5120x2560 resolution on the Karolina cluster (IT4Innovations) using up to 32 accelerated nodes. The results of this experiments are shown in Figures 5.23, 5.24. In this test, we used distributed rendering over the samples listed in Section 3.3.3.2 and the load balancing Algorithm 1 from Section 3.3.3.3. We compared the data placement Algorithm 2 using 5% duplication and 64MB chunks from Section 3.4.2.2.2 with the naive round robin fashion presented in Section 3.4.2.1.1. It can be seen that our approach works for all settings.

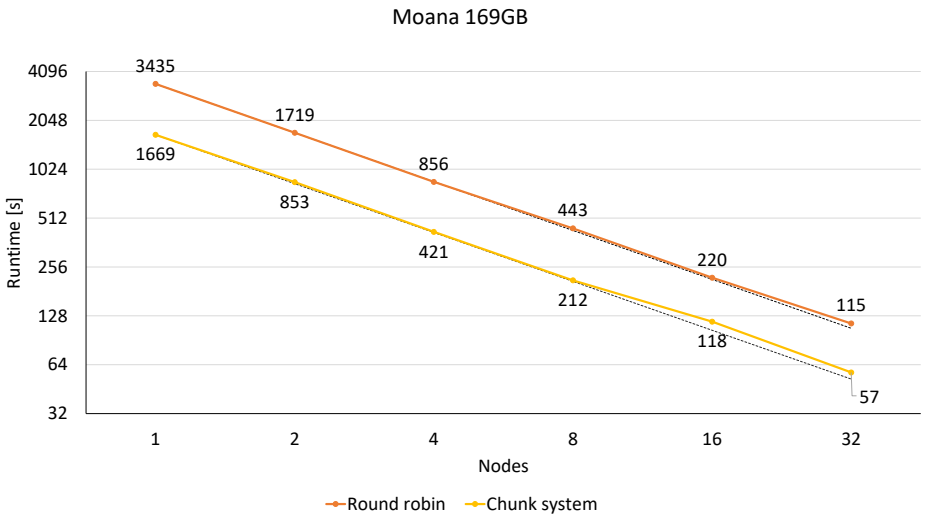


Figure 5.23: Strong scalability for the Moana 169GB scene

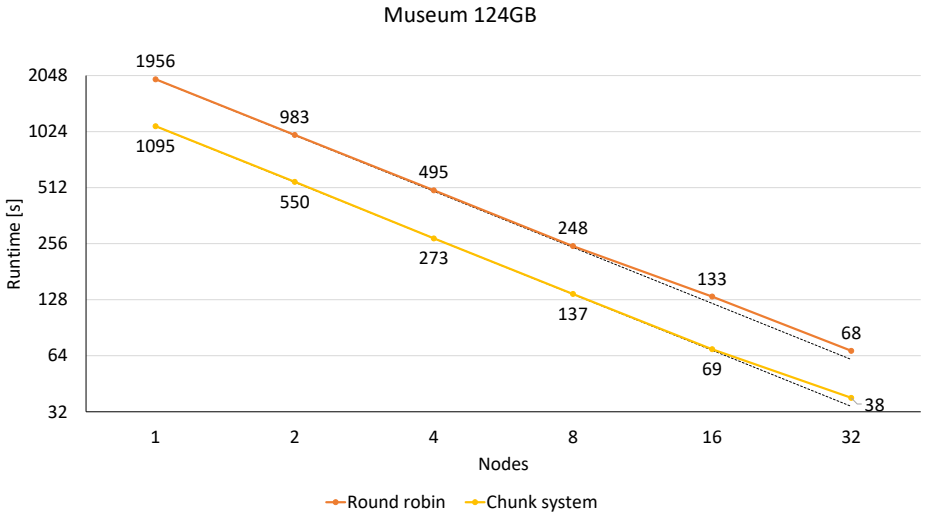


Figure 5.24: Strong scalability for the Museum 124GB scene

# Chapter 6

# Conclusion

## 6.1 Summary

In this dissertation, we presented a solution for tracking the paths of massive scenes on multiple GPUs. Our approach analyzes the memory access pattern of the path tracer and defines how the scene data should be distributed among GPUs with minimal performance loss. The key concept is that those parts of the scene data that have the highest memory access rate are replicated across all GPUs, as their distribution could have a significant negative impact on performance. We proposed two methods. Both work at the memory management level but with different granularity. Working at the memory management level means that we only manage where (on which GPU) and how (replication or distribution) the data structures are allocated, and that we do not care what is stored in the data structure. By different granularity, we mean that (i) the first approach uses the same memory management rules for the entire data structure; (ii) the second approach divides data structures into parts, and we manage the placement and/or replication of each part separately. Since we only manage memory allocations, there is no need to rework the path tracer data structures. This makes our approach applicable to other GPU-based path tracers with minor changes to their code.

We extended Blender Cycles to support the OpenMP, Intel Xeon Phi, NVIDIA CUDA technology with unified memory support, and Message Passing Interface (MPI). We called this extension CyclesPhi [125]. We used CyclesPhi mainly to test new Blender functionality in collaboration with the Blender Institute on open-movie projects (Cosmos Laundromat, Agent 327: Operation Barbershop, The Daily Dweebs). Thanks to our collaboration, dozens of bugs were found, and some important functionality was optimized. CyclesPhi has been presented at major conferences or workshops (IXPUG 2016-2018, SC 2016, Intel HPC Developer Conference 2017, BCON 2018-2019, GTC2021, SIGGRAPH2021). Another extension is the creation of a so-called bridge between the client computer and the supercomputer for remote

visualization. Interactive elevated rendering can thus be used, for example, in 3D cinema, in mobile phones or in virtual or augmented reality glasses.

All the tasks addressed in this thesis have led to the creation of a so-called visualization service. This service has the ability to visualize data from different sectors of human activity such as medicine. For example, we are able to create a 3D model from CT scans where we are able to, for example, measure the size of tumors in the liver, necessary for subsequent liver resection [155], the dimensions of the breathing tube in snoring problems, and locating fractures in the human skull. Other areas where this service can be used are computations in the areas of flow modeling or structural mechanics, where visualization of results such as in Figure 1.15, where in some cases we need to display billions of triangles in real time, is required. Last but not least, there is a need to meet the requirements of architects or artists who value photorealistic visualizations.

## 6.2 Fulfillment of the goals

In this work, we have developed an out-of-core mechanism to enable rendering of large scenes using multiple GPUs with unified memory in one of the most widely used freeware renderers, Blender Cycles. This new out-of-core mechanism is used in the newly created Rendering-as-a-service [156, 157, 158], which can be used by all current and future users of our infrastructure, opening up new possibilities to explore, for example, computational science data from simulations and cell behavior where data has been obtained from a microscope as well as to plan surgical operations more easily using analysis of images from computed tomography. From my point of view, all objectives have been met.

Author's contribution to the content of this work:

- all development and implementation of CyclesPhi introduced in Section 3.3 was done by me,
- the methods presented in Section 3.4 was designed and implemented by me and fine-tuned by Lubomír Říha and Petr Strakoš,
- the changes in HEAPPE introduced in Section 4.1.1 and the creation of the merge request were done by me and the changes, from the merge request, were integrated into the main branch by Jan Křenek,
- all the implementation of BHEAPPE introduced in Section 4.1.1.1 was done by me,
- all the measurements presented in this work were performed and processed by me.

Finally, I would like to mention a comment by Francesco Siddi, who is the COO at Blender and General Manager at Blender Studio, on the work of our team and mine as well: "IT4I has

been instrumental for the research and development process during the production of several Blender Open Movies. Having access to the Salomon cluster allowed for fast iterations during scene/performance debugging, and it also allowed the pursuit of high-end visuals, pushing rendering algorithms to the limits. The IT4I team has always supported our goals with great transparency and in the most professional way."

## 6.3 Future directions

We plan to upgrade the existing version of CyclesPhi, which is based on Blender 2.83, to Blender 3.0 with the new version of CyclesX. This version should already support the new AMD accelerators using HIP [159]. HIP is a source language that can be compiled to run on AMD platform as well as the NVIDIA platform. HIP should also support working with unified memory, which AMD refers to as HMM (Heterogeneous Memory Management), and our future goal is to apply our new memory analysis approach to AMD accelerators. We also plan to support other platforms such as POWER9 with NVIDIA accelerators, where each POWER9 CPU is equipped with NVLink 2.0. With such platforms, our approach can be extended to include an efficient out-of-core mechanism that uses CPU memory for infrequently accessed data.

# Bibliography


1. JAROŠ, Milan; ŘI'HA, Lubomı'r; STRAKOŠ, Petr; ŠPEŤKO, Matěj. GPU Accelerated Path Tracing of Massive Scenes. *ACM Trans. Graph.* 2021, vol. 40, no. 2. ISSN 0730-0301. Available from DOI: 10.1145/3447807.
2. BLENDER ONLINE COMMUNITY. *Blender – a 3D Modelling and Rendering Package*. Blender Foundation, Amsterdam, 2021. Available also from: http://www.blender.org.
3. STEINMETZ, Frederik; HOFMANN, Gottfriend. *The Cycles Encyclopedia*. blender.org, 2014.
4. JAKOB, Wenzel. *Mitsuba renderer*. 2010. http://www.mitsuba-renderer.org.
5. PHARR, Matt; HUMPHREYS, Greg; VERGAUWEN, Terrence. *LuxRender*. 2007. http://www.luxrender.net.
6. GLARE TECHNOLOGIES. *Indigo Renderer*. 2021. https://www.indigorenderer.com/.
7. NEXT LIMIT TECHNOLOGIES. *Maxwell Render*. 2021. https://maxwellrender.com/.
8. MAXON. *Redshift*. 2021. Available also from: https://www.maxon.net/en/redshift/features.
9. OTOY. *Octane Render*. 2021. https://home.otoy.com/render/octane-render.
10. AUTODESK. *Arnold Renderer*. 2021. https://www.arnoldrenderer.com/.
11. PIXAR. *RenderMan*. 2021. https://renderman.pixar.com/.
12. CHAOS GROUP. *V-Ray GPU*. 2021. https://www.chaosgroup.com/vray-gpu.
13. NVIDIA. *NVIDIA Iray*. 2021. https://www.nvidia.com/en-us/design-visualization/iray/.

14. WALD, Ingo; WOOP, Sven; BENTHIN, Carsten; JOHNSON, Gregory S.; ERNST, Manfred. Embree: A Kernel Framework for Efficient CPU Ray Tracing. *ACM Trans. Graph.* 2014, vol. 33, no. 4, 143:1–143:8. ISSN 0730-0301. Available from DOI: 10.1145/2601097.2601199.

15. CHAOS GROUP. *Corona Renderer*. 2021. https://corona-renderer.com.

16. WALD, I.; JOHNSON, G.; AMSTUTZ, J.; BROWNLEE, C.; KNOLL, A.; JEFFERS, J.; GUNTHER, J.; NAVRATIL, P. OSPRay - A CPU Ray Tracing Framework for Scientific Visualization. *IEEE Transactions on Visualization and Computer Graphics*. 2017, vol. 23, no. 1, pp. 931–940. ISSN 1077-2626. Available from DOI: 10.1109/TVCG.2016.2599041.

17. REBUSFARM. *RebusFarm - Cloud Rendering Service*. 2021. https://de.rebusfarm.net.

18. RENDERFARM, Fox. *Fox Renderfarm*. 2021. https://www.foxrenderfarm.com/.

19. OTOY. *RNDR - Rendertoken*. 2021. https://rendertoken.com/.

20. IRENDER. *iRender - GPU Cloud Platform*. 2021. https://irendering.net/.

21. GARAGEFARM.NET. *GarageFarm.NET*. https://garagefarm.net/.

22. PRODAN, Radu; OSTERMANN, Simon. A survey and taxonomy of infrastructure as a service and web hosting cloud providers. *2009 10th IEEE/ACM International Conference on Grid Computing*. 2009, pp. 17–25.

23. ANNETTE, J. Ruby; BANU, W. Aisha; CHANDRAN, P. Subash. Rendering-as-a-Service: Taxonomy and Comparison. *Procedia Computer Science*. 2015, vol. 50, pp. 276–281. ISSN 1877-0509. Available from DOI: https://doi.org/10.1016/j.procs.2015.04.048. Big Data, Cloud and Computing Challenges.

24. SVATON, Václav; MARTINOVIC, Jan; KRENEK, Jan; ESCH, Thomas; TOMANÇAK, Pavel. HPC-as-a-Service via HEAppE Platform. In: *CISIS*. 2019.

25. FASCIONE, Luca; HANIKA, Johannes; LEONE, Mark; DROSKE, Marc; SCHWARZHAUPT, Jorge; DAVIDOVIČ, Tomáš; WEIDLICH, Andrea; MENG, Johannes. Manuka: A Batch-Shading Architecture for Spectral Path Tracing in Movie Production. *ACM Transactions on Graphics*. 2018, vol. 37, no. 3.

26. KULLA, Christopher; CONTY, Alejandro; STEIN, Clifford; GRITZ, Larry. Sony Pictures Imageworks Arnold. *ACM Transactions on Graphics*. 2018, vol. 37, no. 3.

27. CHRISTENSEN, Per; FONG, Julian; SHADE, Jonathan; WOOTEN, Wayne; SCHUBERT, Brenden; KENSLER, Andrew; FRIEDMAN, Stephen; KILPATRICK, Charlie; RAMSHAW, Cliff; BANNISTER, Marc. RenderMan: An Advanced Path-Tracing Architecture for Movie Rendering. *ACM Transactions on Graphics*. 2018, vol. 37, no. 3.

28. GEORGIEV, Iliyan; IZE, Thiago; FARNSWORTH, Mike; MONTOYA-VOZMEDIANO, Ramón; KING, Alan; LOMMEL, Brecht Van; JIMENEZ, Angel; ANSON, Oscar; OGAKI, Shinji; JOHNSTON, Eric. Arnold: A Brute-Force Production Path Tracer. *ACM Transactions on Graphics*. 2018, vol. 37, no. 3.

29. BURLEY, Brent; ADLER, David; CHIANG, Matt Jen-Yuan; DRISKILL, Hank; HABEL, Ralf; KELLY, Patrick; KUTZ, Peter; LI, Yining Karl; TEECE, Daniel. The Design and Evolution of Disney's Hyperion Renderer. *ACM Transactions on Graphics*. 2018, vol. 37, no. 3.

30. LI, A.; SONG, S. L.; CHEN, J.; LI, J.; LIU, X.; TALLENT, N. R.; BARKER, K. J. Evaluating Modern GPU Interconnect: PCIe, NVLink, NV-SLI, NVSwitch and GPUDirect. *IEEE Transactions on Parallel and Distributed Systems*. 2020, vol. 31, no. 1, pp. 94–110. Available from DOI: 10.1109/TPDS.2019.2928289.

31. HARRIS, Mark. *Unified Memory for CUDA Beginners*. 2017. Available also from: https://devblogs.nvidia.com/unified-memory-cuda-beginners/.

32. NVIDIA. *NVIDIA NVSWITCH Technical Overview* [https://images.nvidia.com/content/pdf/nvswitch-technical-overview.pdf]. 2018.

33. NVIDIA. *NVIDIA Tesla P100 – Whitepaper* [https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf]. 2016. WP-08019-001_v01.1.

34. NVIDIA. *NVIDIA Turing GPU Architecture* [https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf]. 2018. WP-09183-001_v01.

35. NVIDIA. *DGX-2/2H SYSTEM User Guide* [https://docs.nvidia.com/dgx/pdf/dgx2-user-guide.pdf]. 2019. DU-09130-001_v08.1.

36. BLENDER FOUNDATION. *Cycles Open Source Production Rendering*. 2021. Available also from: https://www.cycles-renderer.org/.

37. FETTER, William. Computer Graphics at Boeing. *Print Magazine, XX:VI, November/Dezember*. 1966, p. 32.

38. SUTHERLAND, Ivan E. Sketch Pad a Man-machine Graphical Communication System. In: *Proceedings of the SHARE Design Automation Workshop*. New York, NY, USA: ACM, 1964, pp. 6.329–6.346. DAC '64. Available from DOI: 10.1145/800265.810742.

39. GOURAUD, Henri. Continuous shading of curved surfaces. *IEEE Transactions on Computers*. 1971, vol. C-20, no. 6, pp. 623–629.

40. APPEL, Arthur. Some techniques for shading machine renderings of solids. *In Proceedings of the AFIPS Spring Joint Computer Conference, volume 32*. 1968, pp. 37–45.

41. WHITTED, Turner. An improved illumination model for shaded display. *Communications of the ACM*. 1980.

42. ROBERT L. COOK, Thomas Porter; CARPENTER, Loren. Distributed ray tracing. *Computer Graphics (Proceedings of SIGGRAPH '84)*. 1984.

43. PHARR, M.; JAKOB, W.; HUMPHREYS, G. Physically based rendering: From theory to implementation: Third edition. In: 2016, pp. 1–1233. Physically Based Rendering: From Theory to Implementation: Third Edition.

44. LAFORTUNE, Eric; WILLEMS, Yves. Bi-directional path tracing. *In Proceedings of Compugraphics*. 1993, pp. 145–153.

45. VEACH, Eric; GUIBAS, Leonidas. Bidirectional estimators for light transport. *In Proceedings of the Eurographics Workshop on Rendering*. 1994, pp. 147–162.

46. AL., N. Metropolis et. Equations of state calculations by fast computing machines. *Journal of Chemical Physics*. 1953.

47. COOK, Robert L.; CARPENTER, Loren; CATMULL, Edwin. The Reyes Image Rendering Architecture. In: *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH 1987*. 1987, pp. 95–102.

48. JENSEN, Henrik Wann. Global illumination using photon maps. *In Rendering Techniques (Proceedings of the Eurographics Workshop on Rendering)*. 1996, pp. 21–30.

49. JENSEN, Henrik Wann. Realistic Image Synthesis using Photon Mapping. *A K Peters Ltd.* 2001.

50. STEINMETZ, Frederik; HOFMANN, Gottfried. *The Cycles Encyclopedia*. 2016.

51. KAJIYA, James T. The Rendering Equation. *SIGGRAPH Comput. Graph.* 1986, vol. 20, no. 4, pp. 143–150. ISSN 0097-8930. Available from DOI: 10.1145/15886.15902.

52. ŽÁRA, J.; BENEŠ, B.; SOCHOR, J.; FELKEL, P. *Moderní počítačová grafika*. Computer Press, 2004. ISBN 9788025104545. Available also from: https://books.google.cz/books?id=USQnAAAACAAJ.

53. LAFORTUNE, Eric P. *Mathematical Models and Monte Carlo Algorithms for Physcially Based Rendering*. Leuven, Belgium, 1995. PhD thesis. Katholieke University.

54. KELLER, Alexander; HEINRICH, Stefan; NIEDERREITER, Harald. *Monte Carlo and Quasi-Monte Carlo Methods 2006*. 1st. Springer Publishing Company, Incorporated, 2007.

55. MOROKOFF, William J. Generating Quasi-Random Paths for Stochastic Processes. *SIAM Review*. 1998, vol. 40, pp. 765–788.

56. NIEDERREITER, Harald. *Random Number Generation and quasi-Monte Carlo Methods*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 1992. ISBN 0-89871-295-5.

57. JOE, Stephen; KUO, Frances Y. Remark on Algorithm 659: Implementing Sobol's Quasirandom Sequence Generator. *ACM Trans. Math. Softw.* 2003, vol. 29, no. 1, pp. 49–57. ISSN 0098-3500. Available from DOI: 10.1145/641876.641879.

58. JOE, Stephen; KUO, Frances Y. Constructing Sobol Sequences with Better Two-Dimensional Projections. *SIAM Journal on Scientific Computing*. 2008, vol. 30, no. 5, pp. 2635–2654. Available from DOI: 10.1137/070709359.

59. BRATLEY, Paul; FOX, Bennett L. Algorithm 659: Implementing Sobol's Quasirandom Sequence Generator. *ACM Trans. Math. Softw.* 1988, vol. 14, no. 1, pp. 88–100. ISSN 0098-3500. Available from DOI: 10.1145/42288.214372.

60. COMMUNITY, Blender Online. *Random Number Generators*. Blender Institute, Amsterdam, 2017. http://wiki.blender.org/index.php/Dev:2.6/Source/Render/Cycles/Sobol.

61. INTEL. Animation Evolution: A Biopic Through the Eyes of Shrek. *Computer Graphic World*. 2010.

62. FELLNER, F. Introducing Cinematic Rendering: A Novel Technique for Post-Processing Medical Imaging Data. *Journal of Biomedical Science and Engineering*. 2016, vol. 9, pp. 170–175.

63. DAPPA, E.; HIGASHIGAITO, K.; FORNARO, J. et al. Cinematic rendering – an alternative to volume rendering for 3D computed tomography imaging. *Insights Imaging*. 2016, vol. 7, pp. 849–856.

64. BUDGE, Brian; BERNARDIN, Tony; STUART, Jeff A.; SENGUPTA, Shubhabrata; JOY, Kenneth I.; OWENS, John D. Out-of-Core Data Management for Path Tracing on Hybrid Resources. *Computer Graphics Forum*. 2009, vol. 28, no. 2, pp. 385–396.

65. SON, Myungbae; YOON, Sung-Eui. Timeline Scheduling for Out-of-Core Ray Batching. In: *Proceedings of High Performance Graphics, HPG 2017*. 2017.

66. ZHOU, Kun; HOU, Qiming; REN, Zhong; GONG, Minmin; SUN, Xin; GUO, Baining. RenderAnts: Interactive REYES Rendering on GPUs. *ACM Transactions on Graphics*. 2009-12, vol. 28, no. 5. ISSN 0730-0301. Available from DOI: 10.1145/1618452.1618501. ACM SIGGRAPH Asia Conference 2009, Yokohama, JAPAN, DEC 16-19, 2009.

67. PANTALEONI, Jacopo; FASCIONE, Luca; HILL, Martin; AILA, Timo. PantaRay: Fast Ray-Traced Occlusion Caching of Massive Scenes. *ACM Transactions on Graphics*. 2010-07, vol. 29, no. 4. ISSN 0730-0301. Available from DOI: {10.1145/1778765.1778774}.

68. WANG, Rui; HUO, Yuchi; YUAN, Yazhen; ZHOU, Kun; HUA, Wei; BAO, Hujun. GPU-Based Out-of-Core Many-Lights Rendering. *ACM Transactions on Graphics*. 2013, vol. 32, no. 6.

69. BITTNER, J.; HAVRAN, V. RDH: Ray Distribution Heuristics for Construction of Spatial Data Structures. In: *Proceedings - SCCG 2009: 25th Spring Conference on Computer Graphics*. 2009, pp. 51–58.

70. FELTMAN, Nicolas; LEE, Minjae; FATAHALIAN, Kayvon. SRDH: Specializing BVH Construction and Traversal Order Using Representative Shadow Ray Sets. In: *High-Performance Graphics 2012, HPG 2012 - ACM SIGGRAPH / Eurographics Symposium Proceedings*. 2012, pp. 49–55.

71. MOLNAR, Steven; COX, Michael; ELLSWORTH, David; FUCHS, Henry. A Sorting Classification of Parallel Rendering. *IEEE Comput. Graph. Appl.* 1994, vol. 14, no. 4, pp. 23–32. ISSN 0272-1716. Available from DOI: 10.1109/38.291528.

72. PARKER, Steven; SHIRLEY, Peter; LIVNAT, Yarden; HANSEN, Charles; SLOAN, Peter-Pike. Interactive Ray Tracing for Ssosurface Rendering. In: *Proceedings of the IEEE Visualization Conference*. 1998, pp. 233–238.

73. WALD, I.; BENTHIN, C.; SLUSALLEK, P. 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.

74. DEMARLE, D. E.; GRIBBLE, C. P.; BOULOS, S.; PARKER, S. G. Memory Sharing for Interactive Ray Tracing on Clusters. *Parallel Computing*. 2005, vol. 31, no. 2, pp. 221–242. Available also from: www.scopus.com. Cited By :13.

75. KELLER, Alexander; WÄCHTER, Carsten; RAAB, Matthias; SEIBERT, Daniel; ANTWERPEN, Dietger van; KORNDORFER, Johann; KETTNER, Lutz. *The Iray Light Transport Simulation and Rendering System*. 2017. Available from arXiv: 1705.01263 [cs.GR].

76. KATO, Toshiaki; NISHIMURA, Hitoshi; ENDO, Tadashi; MARUYAMA, Tamotsu; SAITO, Jun; CHRISTENSEN, Per H. 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, IV-1 to IV–59.

77. USHER, Will; WALD, Ingo; AMSTUTZ, Jefferson; GÃŒNTHER, Johannes; BROWNLEE, Carson; PASCUCCI, Valerio. Scalable Ray Tracing Using the Distributed FrameBuffer. *Computer Graphics Forum*. 2019, vol. 38, no. 3, pp. 455–466. Available from DOI: https://doi.org/10.1111/cgf.13702.

78. NAVRÁTIL, Paul A.; FUSSELL, Donald S.; LIN, Calvin; CHILDS, Hank. Dynamic Scheduling for Large-Scale Distributed-Memory Ray Tracing. In: CHILDS, Hank; KUHLEN, Torsten; MARTON, Fabio (eds.). *Eurographics Symposium on Parallel Graphics and Visualization*. The Eurographics Association, 2012. ISBN 978-3-905674-35-4. ISSN 1727-348X. Available from DOI: 10.2312/EGPGV/EGPGV12/061-070.

79. NAVRÁTIL, Paul A.; CHILDS, Hank; FUSSELL, Donald S.; LIN, Calvin. Exploring the Spectrum of Dynamic Scheduling Algorithms for Scalable Distributed-Memory Ray Tracing. *IEEE Transactions on Visualization and Computer Graphics*. 2014, vol. 20, no. 6, pp. 893–906.

80. PROTIC, Jelica; TOMASEVIC, Milo; MILUTINOVIC, Veljko. A Survey of Distributed Shared Memory Systems. In: *Proceedings of the Twenty-Eighth Annual Hawaii International Conference on System Sciences*. 1995, vol. 1, pp. 74–84.

81. PROTIC, Jelica; TOMASEVIC, Milo; MILUTINOVIC, Veljko. Distributed Shared Memory: Concepts and Systems. *IEEE Parallel & Distributed Technology: Systems & Applications*. 1996, vol. 4, no. 2, pp. 63–71.

82. HENNESSY, John; HEINRICH, Mark; GUPTA, Anoop. Cache-coherent Distributed Shared Memory: Perspectives on its Development and Future Challenges. *Proceedings of the IEEE*. 1999, vol. 87, no. 3, pp. 418–429.

83. SOUNDARARAJAN, Vijayaraghavan; HEINRICH, Mark; VERGHESE, Ben; GHARACHORLOO, Kourosh; GUPTA, Anoop; HENNESSY, John. Flexible Use of Memory for Replication/Migration in Cache-Coherent DSM Multiprocessors. In: *Proceedings. 25th Annual International Symposium on Computer Architecture (Cat. No. 98CB36235)*. 1998, pp. 342–355.

84. AILA, Timo; KARRAS, Tero. Architecture Considerations for Tracing Incoherent Rays. In: *Proceedings of the Conference on High Performance Graphics*. 2010, pp. 113–122.

85. PARKER, Steven; MARTIN, William; SLOAN, Peter-Pike J.; SHIRLEY, Peter; SMITS, Brian; HANSEN, Charles. Interactive Ray Tracing. *Proceedings of the Symposium on Interactive 3D Graphics*. 1999, pp. 119–126.

86. KEATES, M.J.; HUBBOLD, R.J. *Accelerated Ray Tracing on the KSR1 Virtual Shared-Memory Parallel Computer*. Citeseer, 1994.

87. SINGH, J Pal; GUPTA, Anoop; LEVOY, Marc. Parallel Visualization Algorithms: Performance and Architectural Implications. *Computer*. 1994, vol. 27, no. 7, pp. 45–55.

88. SAKHARNYKH, Nikolay. Unified Memory on Pascal and Volta. In: *GPU Technology Conference (GTC)*. 2017. Available also from: `http://on-demand.gputechconf.com/gtc/2017/presentation/s7285-nikolay-sakharnykh-unified-memory-on-pascal-and-volta.pdf`.

89. ALSABER, Nabeel; KULKARNI, Milind. Semcache: Semantics-Aware Caching for Efficient GPU Offloading. In: *Proceedings of the 27th international ACM conference on International conference on supercomputing*. 2013, pp. 421–432.

90. JABLIN, Thomas B; JABLIN, James A; PRABHU, Prakash; LIU, Feng; AUGUST, David I. Dynamically Managed Data for CPU-GPU Architectures. In: *Proceedings of the Tenth International Symposium on Code Generation and Optimization*. 2012, pp. 165–174.

91. GELADO, Isaac; STONE, John E; CABEZAS, Javier; PATEL, Sanjay; NAVARRO, Nacho; HWU, Wen-mei W. An Asymmetric Distributed Shared Memory Model for Heterogeneous Parallel Systems. In: *Proceedings of the fifteenth International Conference on Architectural support for programming languages and operating systems*. 2010, pp. 347–358.

92. HUYNH, Huynh P.; HAGIESCU, Andrei; WONG, Weng-Fai; GOH, Rick S. M. Scalable Framework for Mapping Streaming Applications onto Multi-GPU Systems. *ACM SIGPLAN Notices*. 2012, vol. 47, no. 8, pp. 1–10.

93. SABNE, Amit; SAKDHNAGOOL, Putt; EIGENMANN, Rudolf. Scaling Large-Data Computations on Multi-GPU Accelerators. In: *Proceedings of the International Conference on Supercomputing*. 2013, pp. 443–454.

94. JABLIN, Thomas B.; JABLIN, James A.; PRABHU, Prakash; LIU, Feng; AUGUST, David I. Dynamically Managed Data for CPU-GPU Architectures. In: *Proceedings - International Symposium on Code Generation and Optimization, CGO 2012*. 2012, pp. 165–174.

95. GELADO, Isaac; STONE, John E.; CABEZAS, Javier; PATEL, Sanjay; NAVARRO, Nacho; HWU, Wen-mei W. An Asymmetric Distributed Shared Memory Model for Heterogeneous Parallel Systems. In: *International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS*. 2010, pp. 347–358.

96. AL-SABER, Nabeel; KULKARNI, Milind. SemCache++: Semantics-Aware Caching for Efficient Multi-GPU Offloading. In: *Proceedings of the International Conference on Supercomputing*. 2015, vol. 2015-June, pp. 79–88.

97. KRIZHEVSKY, Alex; SUTSKEVER, Ilya; HINTON, Geoffrey E. ImageNet Classification With Deep Convolutional Neural Networks. In: *Advances in Neural Information Processing Systems*. 2012, vol. 2, pp. 1097–1105.

98. SEO, Hyunseok; KIM, Jinwook; KIM, Min-Soo. GStream: A Graph Streaming Processing Method for Large-Scale Graphs on GPUs. In: *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP*. 2015, vol. 2015-January, pp. 253–254.

99. SHAMOTO, Hideyuki; SHIRAHATA, Koichi; DROZD, Aleksandr; SATO, Hitoshi; MATSUOKA, Satoshi. Large-scale Distributed Sorting for GPU-based Heterogeneous Supercomputers. In: *Proceedings - 2014 IEEE International Conference on Big Data, IEEE Big Data 2014*. 2015, pp. 510–518.

100. CHIEN, Steven W. D.; PENG, Ivy B.; MARKIDIS, Stefano. Performance Evaluation of Advanced Features in CUDA Unified Memory. In: *Proceedings – International Conference for High Performance Computing, Networking, Storage, and Analysis, SC 2019*. 2020.

101. NVIDIA. *CUDA C Programming Guide*. 2018. Available also from: `https://docs.nvidia.com/cuda/archive/10.0/pdf/CUDA_C_Programming_Guide.pdf`.

102. GANGULY, Debashis; ZHANG, Ziyu; YANG, Jun; MELHEM, Rami. Interplay between Hardware Prefetcher and Page Eviction Policy in CPU-GPU Unified Virtual Memory. In: *Proceedings - International Symposium on Computer Architecture*. 2019, pp. 224–235.

103. BARUAH, Trinayan; SUN, Yifan; DINÇER, Ali Tolga; MOJUMDER, Saiful A; ABELLÁN, José L; UKIDAVE, Yash; JOSHI, Ajay; RUBIN, Norman; KIM, John; KAELI, David. Griffin: Hardware-Software Support for Efficient Page Migration in Multi-GPU Systems. In: *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 2020, pp. 596–609.

104. GANGULY, Debashis; ZHANG, Ziyu; YANG, Jun; MELHEM, Rami. Adaptive Page Migration for Irregular Data-intensive Applications under GPU Memory Oversubscription. In: *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 2020, pp. 451–461.

105. AGARWAL, Neha; NELLANS, David; O'CONNOR, Mike; KECKLER, Stephen W; WENISCH, Thomas F. Unlocking Bandwidth for GPUs in CC-NUMA systems. In: *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. 2015, pp. 354–365.

106. YOUNG, Vinson; JALEEL, Aamer; BOLOTIN, Evgeny; EBRAHIMI, Eiman; NELLANS, David; VILLA, Oreste. Combining HW/SW Mechanisms to Improve NUMA Performance of Multi-GPU Systems. In: *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2018, pp. 339–351.

107. CHRISTENSEN, Cameron; FOGAL, Thomas; LUEHR, Nathan; WOOLLEY, Cliff. Topology-Aware Image Compositing Using NVLink. In: *IEEE Symposium on Large Data Analysis and Visualization 2016, LDAV 2016 - Proceedings*. 2017, pp. 93–94.

108. KIM, Youngsok; JO, Jae-Eon; JANG, Hanhwi; RHU, Minsoo; KIM, Hanjun; KIM, Jangwoo. GPUpd: A Fast and Scalable Multi-GPU Architecture Using Cooperative Projection and Distribution. In: *Proceedings of the Annual International Symposium on Microarchitecture, MICRO*. 2017, vol. Part F131207, pp. 574–586.

109. XIE, Chenhao; XIN, Fu; CHEN, Mingsong; SONG, Shuaiwen L. OO-VR: NUMA Friendly Object-Oriented VR Rendering Framework For Future NUMA-Based Multi-GPU Systems. In: *Proceedings – International Symposium on Computer Architecture*. 2019, pp. 53–65.

110. QARNOT. *Qarnot - Green Computing*. 2017. `https://computing.qarnot.com`.

111. IT4INNOVATIONS. *IT4Innovations národní superpocítacové centrum*. 2017. `http://www.it4i.cz`.

112. CINECA. *Marconi*. 2017. `https://www.cineca.it/en/content/marconi`.

113. HLRN. *Norddeutscher Verbund fur Hoch- und Hochstleistungsrechnen (HLRN)*. 2017. `https://www.hlrn.de`.

114. SAROSH, Irani. Accelerated Computing Solutions for AI and HPC Workloads. In: *GPU Technology Conference (GTC)* [`https://developer.nvidia.com/gtc/2019/video/S9981`]. 2019.

115. MCCALPIN, John D. Memory Bandwidth and Machine Balance in Current High Performance Computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*. 1995, pp. 19–25.

116. ATOS. *BullSequana X410 E5 Dense GPU-Accelerated Compute Node*. 2017. Available also from: `https://atos.net/wp-content/uploads/2017/11/FS_BullSequana_X410E5_en1-web.pdf`.

117. LI, Ang; SONG, Shuaiwen L.; CHEN, Jieyang; LIU, Xu; TALLENT, Nathan; BARKER, Kevin. Tartan: Evaluating Modern GPU Interconnect via a Multi-GPU Benchmark Suite. In: *2018 IEEE International Symposium on Workload Characterization (IISWC)*. 2018, pp. 191–202.

118. AMD. *AMD EPYC SoC Delivers Exceptional Results on the STREAM Benchmark on 2P Servers* [`https://www.amd.com/system/files/2017-06/AMD-EPYC-SoC-Delivers-Exceptional-Results.pdf`]. 2017.

119. BUTENHOF, David R. *Programming with POSIX Threads*. Addison-Wesley, 1997. Professional Computing Series. ISBN 0201633922.

120. AILA, Timo; LAINE, Samuli. Understanding the Efficiency of Ray Traversal on GPUs. In: *Proceedings of the Conference on High Performance Graphics*. Association for Computing Machinery, 2009, pp. 145–149. ISBN 9781605586038.

121. AILA, Timo; LAINE, Samuli; KARRAS, Tero. *Understanding the Efficiency of Ray Traversal on GPUs – Kepler and Fermi Addendum*. 2012. NVIDIA Technical Report, NVR-2012-02. NVIDIA Corporation.

122. WALD, Ingo; WOOP, Sven; BENTHIN, Carsten; JOHNSON, Gregory S; ERNST, Manfred. Embree: A Kernel Framework for Efficient CPU Ray Tracing. *ACM Transactions on Graphics (TOG)*. 2014, vol. 33, no. 4, pp. 1–8.

123. PHEATT, Chuck. Intel Threading Building Blocks. *J. Comput. Sci. Coll.* 2008, vol. 23, no. 4, p. 298. ISSN 1937-4771.

124. JAROŠ, M.; RÍHA, L.; STRAKOŠ, P.; KARÁSEK, T.; VAŠATOVÁ, A.; JAROŠOVÁ, M.; KOZUBEK, T. *Acceleration of blender cycles path-tracing engine using intel many integrated core architecture*. 2015. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). Available also from: `www.scopus.com`.

125. JAROS, Milan; RIHA, Lubomir; KARASEK, Tomas; STRAKOS, Petr; KRPELIK, Daniel. Rendering in Blender Cycles using MPI and Intel (R) Xeon Phi (TM). In: NASRI, AH (ed.). *Proceedings of the 2017 International Conference on Computer Graphics and Digital Image Processing (CGDIP 2017)*. 2017. ISBN 978-1-4503-5236-9. Available from DOI: `{10.1145/3110224.3110236}`.

126. FOUNDATION, Blender. *Render Passes*. 2021. Available also from: `https://docs.blender.org/manual/en/latest/render/layers/passes.html`.

127. GROPP, William; LUSK, Ewing; SKJELLUM, Anthony. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. The MIT Press, 2014. ISBN 0262527391.

128. SIEGELL, B.S.; STEENKISTE, P. Automatic generation of parallel programs with dynamic load balancing. In: *Proceedings of 3rd IEEE International Symposium on High Performance Distributed Computing*. 1994, pp. 166–175. Available from DOI: `10.1109/HPDC.1994.340247`.

129. ABRAHAM, Frederico; CELES, Waldemar; CERQUEIRA, Renato; CAMPOS, Joao Luiz. A Load-Balancing Strategy for Sort-First Distributed Rendering. In: *Proceedings of the Computer Graphics and Image Processing, XVII Brazilian Symposium*. USA: IEEE Computer Society, 2004, pp. 292–299. SIBGRAPI '04. ISBN 0769522270.

130. COSENZA, Biagio; DACHSBACHER, Carsten; ERRA, Ugo. GPU Cost Estimation for Load Balancing in Parallel Ray Tracing. In: *GRAPP/IVAPP*. 2013.

131. BLUMOFE, Robert D.; LEISERSON, Charles E. Scheduling Multithreaded Computations by Work Stealing. *J. ACM*. 1999, vol. 46, no. 5, pp. 720–748. ISSN 0004-5411. Available from DOI: 10.1145/324133.324234.

132. DEMARLE, D.E.; PARKER, S.; HARTNER, M.; GRIBBLE, C.; HANSEN, C. Distributed interactive ray tracing for large volume visualization. In: *IEEE Symposium on Parallel and Large-Data Visualization and Graphics, 2003. PVG 2003.* 2003, pp. 87–94. Available from DOI: 10.1109/PVGS.2003.1249046.

133. IZE, Thiago; BROWNLEE, Carson; HANSEN, Charles D. Real-Time Ray Tracer for Visualizing Massive Models on a Cluster. In: *Proceedings of the 11th Eurographics Conference on Parallel Graphics and Visualization*. Llandudno, UK: Eurographics Association, 2011, pp. 61–69. EGPGV '11. ISBN 9783905674323.

134. GAYATRI, Rahulkumar; GOTT, Kevin; DESLIPPE, Jack. Comparing Managed Memory and ATS with and without Prefetching on NVIDIA Volta GPUs. In: *2019 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*. 2019, pp. 41–46. ISBN 978-1-7281-5977-5. Available from DOI: 10.1109/PMBS49563.2019.00010.

135. SAKHARNYKH, Nikolay. *Maximizing Unified Memory Performance in CUDA*. 2017. Available also from: https://devblogs.nvidia.com/maximizing-unified-memory-performance-cuda/.

136. NVIDIA. *CUDA Runtime API*. 2018. Available also from: https://docs.nvidia.com/cuda/archive/10.0/pdf/CUDA_Runtime_API.pdf.

137. FOUNDATION, Blender. *Blender Cloud Services*. 2021. Available also from: https://cloud.blender.org/services.

138. ZADKA, Moshe. Paramiko. In: *DevOps in Python: Infrastructure as Python*. Berkeley, CA: Apress, 2019, pp. 111–119. ISBN 978-1-4842-4433-3. Available from DOI: 10.1007/978-1-4842-4433-3_9.

139. BLENDER FOUNDATION. *Blender Asset Tracer*. 2021. Available also from: https://developer.blender.org/tag/blender_asset_tracer/.

140. CESNET. *CESNET*. 2021. https://www.cesnet.cz/.

141. VRGINEERS. *VRgineers*. 2021. https://vrgineers.com/.

142. HOLUB, Petr; MATELA, Jiři'; PULEC, Martin; ŠROM, Martin. UltraGrid: Low-Latency High-Quality Video Transmissions on Commodity Hardware. In: *Proceedings of the 20th ACM International Conference on Multimedia*. Nara, Japan: Association for Computing Machinery, 2012, pp. 1457–1460. MM '12. ISBN 9781450310895. Available from DOI: 10.1145/2393347.2396519.

143. HOLUB, Petr; ŠROM, Martin; PULEC, Martin; MATELA, Jiří; JIRMAN, Martin. GPU-accelerated DXT and JPEG compression schemes for low-latency network transmissions of HD, 2K, and 4K video. *Future Generation Computer Systems*. 2013, vol. 29, no. 8, pp. 1991–2006. ISSN 0167-739X. Available from DOI: https://doi.org/10.1016/j.future.2013.06.006. Including Special sections: Advanced Cloud Monitoring Systems and The fourth IEEE International Conference on e-Science 2011, e-Science Applications and Tools and Cluster, Grid, and Cloud Computing.

144. VALVE. *OpenVR*. 2021. Available also from: https://github.com/ValveSoftware/openvr.

145. MOURS, Patrick. *Accelerating Cycles using NVIDIA RTX*. 2019. Available also from: https://code.blender.org/2019/07/accelerating-cycles-using-nvidia-rtx/.

146. NVIDIA. *NVIDIA Tesla V100 GPU Architecture* [http://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf]. 2017. WP-08608-001_v1.1.

147. IT4INNOVATIONS. *Barbora supercomputer cluster*. 2019. Available also from: https://docs.it4i.cz/barbora/introduction/.

148. NVIDIA. *DGX-A100 SYSTEM User Guide* [https://docs.nvidia.com/dgx/pdf/dgxa100-user-guide.pdf]. 2021.

149. WALT DISNEY ANIMATION STUDIOS. *Moana Island Scene (v1.1)*. 2018. Available also from: https://www.technology.disneyanimation.com/islandscene.

150. BIRN, Jeremy. *3dRender.com: Lighting Challenges*. 2015. Available also from: http://www.3drender.com/challenges/.

151. THREEDSCANS. *Three D Scans*. 2020. Available also from: https://threedscans.com.

152. THE ART INSTITUTE OF CHICAGO. *Discover Art & Artists*. 2020. Available also from: https://www.artic.edu/collection.

153. INSTITUTE, Blender. *Agent 327 — Blender Cloud*. 2020. Available also from: https://cloud.blender.org/p/agent-327.

154. INSTITUTE, Blender. *Spring — Blender Cloud*. 2020. Available also from: https://cloud.blender.org/p/spring/.

155. STRAKOS, P.; JAROS, M.; KARASEK, T.; KOZUBEK, T.; VAVRA, P.; JONSZTA, T. Advanced image processing methods for automatic liver segmentation. In: *COUPLED PROBLEMS 2015 - Proceedings of the 6th International Conference on Coupled Problems in Science and Engineering*. 2015, pp. 125–136. Available also from: www.scopus.com.

156. IT4INNOVATIONS. *Rendering and Vizualization of Scientific Data*. 2021. Available also from: https://www.it4i.cz/en/industry-cooperation/portfolio-of-services/rendering-and-vizualization-of-scientific-data.

157. JAROS, Milan; STRAKOS, Petr. *Repository of CyclesPhi*. 2021. Available also from: https://code.it4i.cz/raas/cyclesphi.

158. JAROS, Milan; STRAKOS, Petr. *Repository of BHeappe*. 2021. Available also from: https://code.it4i.cz/raas/bheappe.

159. AMD. *Introduction to AMD GPU Programming with HIP*. 2019. Available also from: https://www.olcf.ornl.gov/wp-content/uploads/2019/09/AMD_GPU_HIP_training_20190906.pdf.