Dynamic Load Balancing for Real-Time Multiview Path Tracing on Multi-GPU Architectures (preprint)

Erwan Leria[1*], Markku Mäkitalo[1], Julius Ikkala[1], Pekka Jääskeläinen[1]

1. *Tampere University, Finland*

* **Corresponding author,** erwan.leria@tuni.fi

Abstract Stereoscopic rendering and multiview rendering are used for virtual reality and synthetic generation of light fields from three-dimensional scenes. Since rendering multiple views with ray-tracing techniques is computationally expensive, the utilization of multi-processor machines becomes necessary in order to reach real-time frame rates. In this paper, we propose a dynamic load balancing algorithm for real-time multiview path tracing on multi-compute device platforms. The proposed algorithm adapts to heterogeneous hardware combinations and dynamic scenes on the fly. We show that on a heterogeneous dual-GPU platform, our implementation reduces the rendering time on average by about 30–50% compared to uniform workload distribution, depending on the scene and the number of views.

Keywords Virtual Reality, Multiview, Light Field, Heterogeneous Computing

1 Introduction

Virtual Reality (VR) headsets and light field displays are technologies that are becoming more and more used for producing visual immersive experiences [1]. Light fields (LF) originate from the 5D *plenoptic function* [2,3], depicting the flow of light rays in space. Sampling an LF from different viewpoints allows the viewer to see a scene from different angles on an LF display.

An immersive virtual 3D experience requires frame rates of up to 90 frames per second (fps). This corresponds to a frame time of $\sim$ 11 milliseconds (ms), during which an image or a set of images should be rendered and displayed. Moreover, the traditional stereoscopic VR headsets suffer from the vergence–accommodation conflict [4], which can significantly contribute to visual fatigue for VR users. This problem can be mitigated with LF-based displays, such as [5] or [6]. However, multiview rendering

based on images or 3D models is necessary to automatically generate images for such displays.

Real-time generation of multiple views from 3D scenes requires simultaneously rendering multiple images from different viewpoints under a strict time constraint. Photorealistic rendering is required for high-quality immersive results, which is achievable through ray-tracing based techniques [7, 8], such as path tracing [9, 10]. Although modern graphics processing units (GPUs) have hardware acceleration for ray tracing, such techniques remain computationally expensive; producing high-quality and high-resolution path traced content is not currently feasible on a single GPU in real-time. Naturally, path tracing multiple views increases the rendering workload for a fixed number of computing resources.

When path tracing with a single compute device (e.g., GPU, CPU, FPGA, or another type of hardware accelerator), rays are launched in parallel and run on the available hardware threads. The computing capacity of the device becomes overloaded when the number of rays to process exceeds the number of hardware threads. This case outreaches the frame of Gustafson's law [11]: the computing resources are no more able to parallelize the entire workload. This leads to a serial execution scheme, where a set of rays are only executed in a parallel sequence after the previous set of rays has been computed [12]. This serial effect increases the overall rendering time. In addition, each ray may have a different execution time depending on the complexity of the scene. Improving the computing resources with multiple compute devices allows to reduce the hardware limitations of a single device. However, it requires an evenly balanced rendering time for each device and limited data transfers, in order to avoid bandwidth overhead [13] and to reduce the overall rendering time. In a nutshell, both heterogeneous hardware capabilities of multiple compute devices and 3D scenes are sources of dynamism, inducing imbalance.

In this paper, our focus is on the reduction of the rendering time for real-time multiview path tracing within multi-compute device platforms with dynamic camera movement that simulate the motion of the user. The following contributions are made:

- We formalize a model to map path tracing tasks from multiple views to multiple compute devices.
- We present a dynamic load balancing (DLB) algorithm within our model for multiview rendering on such multi-compute device platforms, based on dynamically updated workload ratios of the devices.
- We show that our implementation of the DLB algorithm scales across multiple viewpoints, with a constant performance gain of 30–50% per scene over uniform workload distribution on a hetero-

geneous dual-GPU setup.

2 Related work

2.1 Screen space decomposition for distributed and parallel rendering

In the classification of parallel and distributed rendering [14], the image-parallel (or sort-first) approach exploits the spatial coherency of the screen. It consists of decomposing the screen space of a viewpoint into multiple screen regions, also called screen space tasks that are then assigned to multiple GPUs. They can have different levels of granularity: tiles, rows of pixels, single pixels or rays. The image-parallel approach assumes that the raw data (mesh) of the scene can fit in the memory of each available GPU, unlike the data-parallel (sort-last) approach which is mostly used for volumetric ray-casting. The sort-last approach is based on data space decomposition, which involves the spatial partitioning of the scene data. In the image-parallel approach, depth image compositing is avoided compared to the data-parallel approach. In order to distribute the workload, these screen regions are split across the GPUs.

Most of the existing prior works show task-based ray tracing frameworks that support either the image-parallel or the data-parallel [15] configuration, and sometimes both [16–18]. More advanced frameworks combine both approaches where an image-parallel layer is implemented on top of a data-parallel layer to amortize the data transfer overhead during the compositing stage [19, 20]. These works usually target high-performance visualization on *distributed memory* architectures, such as clusters or networks, which are not in the scope of our work. In multi-GPU architectures processors communicate through CPU's main memory.

The work proposed in [21] introduces a screen space decomposition scheme called *shuffled strips*, which scales well with the number of processors. Thus, it provides a high scaling efficiency compared to regular tiling for homogeneous and heterogeneous computing environments. However, the method is not demonstrated for dynamic load imbalance. Moreover, they consider the notion of workload ratio, based on the performance of the rendering processors to set the number of assigned screen regions.

2.2 Load balancing for ray-tracing techniques

The goal of a load balancing algorithm is to adjust the workload per processor to avoid idling and to get a high scaling efficiency, thus balancing the individual rendering time of each processor. Usually,

the assignment of such tasks to processors beforehand is defined as static load balancing. In contrast, dynamic reassignment of tasks after a given number of frames is called dynamic load balancing. In this paper, we work on dynamic screen space tasks load balancing for real-time path tracing in the context of multiple viewpoints.

The most recent works on real-time load balancing techniques for ray-tracing are often adapted to particular cases only. This is partly due to a lack of formalism in the literature to base future research on.

Xie et al. [22] use a static load balancing at a pixel granularity for real-time cluster path tracing. Each pixel is assigned to a GPU based on a *pixel stride*. They achieve a linear improvement in the rendering time when they add more GPUs. However, they do not take into account heterogeneous hardware capabilities.

In [23, 24], the authors propose a dynamic *cross-segment* load balancing algorithm within the Equalizer framework [18] aimed at rendering on multi-tiled displays. Despite a good dynamic workload distribution, their work is limited to the case where the number of GPUs is greater than the number of displays. In contrast, our proposed method is more general, handling also the case where the number of viewpoints to render is greater than the number of available GPUs or other compute devices.

Biedert et al. [25] propose a framework for multi-tile streaming, where path traced images are sent over a network to multi-tile displays. They design a dynamic *auto-tuned tiling* load balancer, where the size of a tile is changed based on the measured rendering performance of the GPU associated with that tile. They notice that tiles of heterogeneous sizes attest a better scaling efficiency than regular tiling. However, when more GPUs are added, the scaling efficiency becomes worse due to a higher number of smaller tiles. Moreover, they do not provide further evaluations for their method when there are animations or dynamic camera movements.

A dynamic dual-GPU load balancer for super high-resolution light fields is presented in [26]. They achieve an even rendering time on a high-end dual-GPU setup. However, their implementation is limited to split frame rendering (SFR), and it relies on the deprecated Nvidia SLI interface, which also limits the amount of possible GPUs to be used.

2.3 Multiview path tracing

In multiview rendering, spatio-temporal information can be reused between multiple views. For example, in stereoscopic ray-tracing situations, the spatial information can be directly reprojected from one eye to another [27,28], reducing the number of traced rays and the rendering time. In [29], the authors evaluated that using reprojected samples can improve the quality of stereoscopic path tracing applications by a factor of 25 based on the SSIM metric. Although such techniques improve the quality and reduce the computation time, a small amount of information cannot be reused due to e.g. disocclusions, and thus must be entirely path traced or recovered with rasterization techniques like done in [30]. In [31], the authors focus on multiview path tracing with importance sampling where some views share similar light paths [31]. In [32], Fraboni et al. propose to directly exploit the spatial coherence of the light paths for volumetric rendering to conserve density information along the paths for different views.

3 Scaling multiview path tracing

In this section, we present our method to dynamically balance the path tracing workload of multiple views on multi-compute device platforms. Note that while the method is conceptually agnostic of the types of the compute devices, we describe it in terms of GPUs for notational simplicity.

Our method consists of two parts: First we propose to formalize an abstract representation of the rendering workflow. Then from this abstract definition we design our dynamic load balancing (DLB) algorithm.

3.1 Static scheduling for multiview path tracing

Algorithm 1 Uniform workload ratios at initialization

Input: N GPUs

Output: W an array of workload ratios

$i \leftarrow 0$

$W = \{w_0, ..., w_i, ..., w_{N-1}\}$

for $i < N$ **do**

$w_i \leftarrow \frac{1}{N}$

end for

Our dynamic load balancer starts with a static scheduling phase, which serves as an initialization stage before the rendering stage. It describes the multi-GPU pipeline at a high-level. Scheduling means organizing the workflow and choosing how tasks are mapped to the GPUs, which differs from managing the workload balance only.

In the beginning, we make no assumptions about the performance of the GPUs. An alternative to this would be to set an initial static scheduling strategy based on precalculated GPU performance models for ray-tracing techniques. However, this is a nontrivial problem, especially when considering physically based path tracing in heterogeneous computing environments [21]. Note that general performance models do exist for both GPUs [33] and other hardware accelerators in heterogeneous architectures [34], and such models could be used in combination with auto-tuning for ray tracing [35]. While such approaches are likely to produce a better initial estimate than uniform workload initialization, our approach has the benefit of not requiring a separate pre-processing auto-tuning pass. Instead, it adapts to heterogeneous hardware combinations and application-level dynamism on the fly.

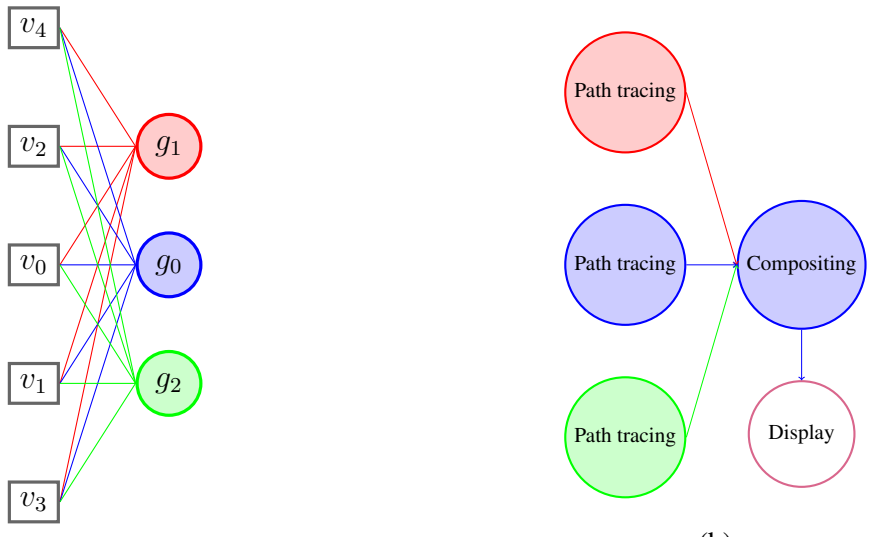
First, we take into account the number of different viewpoints and the number of GPUs. We denote them respectively by M and N, where $M \geq 1$ and $N \geq 2$.

When multiple GPUs work together within the same renderer, they must be assigned rendering tasks that they will process. Once all paths are computed, the screen regions must be gathered to a primary GPU in order to compose the final image. This step is also known as the compositing stage as shown in Figure 1(b).

We want to split the rendering workload uniformly across the N GPUs. In order to avoid additional data structure manipulation, we do not directly manage the tasks, but use the concept of workload ratio as done in [21]. Workload ratios are real numbers contained between 0 and 1, and their sum is equal to 1. We rather assign a quantity of tasks than the tasks themselves.

We assign to each GPU g_i a workload ratio w_i equals to $\frac{1}{N}$ where $0 \leq i < N$, as shown in Algorithm 1. This implies that each GPU will render a ratio w_i of the total workload.

Once the workload ratios are established, an equal number of tasks is mapped to each GPU, as shown in Figure 1(a). Now, for each view, we want to retrieve the amount of tasks to distribute to each GPU according to the associated workload ratio. Let T be a set of rendering tasks as they are defined in Section 2.1. Specifically, T contains the screen regions of each M path traced viewpoints: $T = \{T_0, ..., T_j, ...T_{M-1}\}$



(a) (b)

Figure 1: (a) Mapping graph of uniform workload distribution for $N = 3$ and $M = 5$, and (b) the task graph related to the static scheduling. GPU g_0 is represented in blue, g_1 in red and g_2 in green.

with $0 \leq j < M$, where T_j is a subset of T. T_j contains K screen regions from the viewpoint j, as illustrated in Figure 2. Let job_i be a set of tasks to be processed by a GPU. A GPU g_i is then given an amount of tasks equals to:

$$|job_i| = w_i \cdot |T| \tag{1}$$

The previous statement (Equation 1) is equivalent to:

$$w_i \cdot \sum_{j=0}^{M-1} |T_j| = \sum_{j=0}^{M-1} w_i \cdot |T_j| \tag{2}$$

Since for each GPU g_i, w_i is constant during the execution of the Algorithm 3, Equation 2 shows that each GPU g_i gets a portion of tasks equal to w_i from each view j. Pseudocode for the mapping of tasks from multiple viewpoints to multiple GPUs is presented in Algorithm 2.

3.2 Dynamic load balancing

Previously, we showed that the workload ratios are practical for getting the number of tasks per GPU beforehand. While the program is rendering, we want to balance the rendering time of each GPU in real time. This can be affected for example by the complexity of the scene, user input (such as camera motion), or the hardware.

When the dynamic load balancing algorithm (DLB) is called, we want to estimate the current workload

Algorithm 2 Mapping tasks to GPUs

Input: M viewpoints, N GPUs, W an array of workload ratios, T a set of subsets of tasks

Output: Job an array of tasks to render per GPU

$Job = \{job_0, ..., job_i, ..., job_{N-1}\}$

$j \leftarrow 0$

for $j < M$ **do**

  $i \leftarrow 0$

  for $i < N$ **do**

    $|job_i| \leftarrow w_i \cdot |T_j|$

  end for

end for

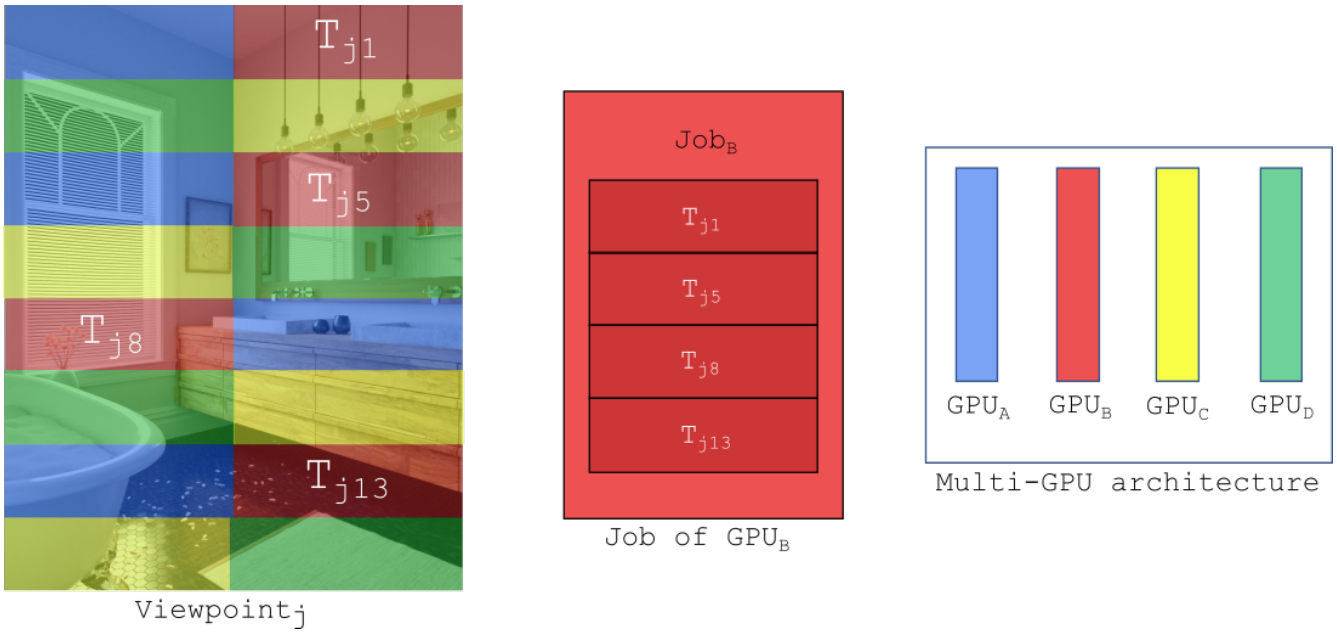


Figure 2: Diagram outlining the different stages of the assignment of screen regions to a GPU. First the screen space of a view to be rendered is decomposed into regions. Based on a distribution scheme, those regions are assigned to a GPU. Then the GPU runs these tasks.

ratio for all the GPUs. Thus, we can adjust the number of pixels for the next frame. We present a dynamic load balancing algorithm based on the information available from the initialization stage and the renderer.

Since we did not make any assumptions on the capabilities of the GPUs during the initialization stage, it is necessary to obtain some information on their performance. Specifically, the algorithm needs to calculate their relative rendering speed from the rendering time and the number of pixels.

Algorithm 3 Dynamic load balancing with workload ratios

Input: M viewpoints, N GPUs, W an array of workload ratios w_i, R an array of rendering speeds r_i

Output: W updated

$i \leftarrow 0$

$r \leftarrow 0$

for $i < N$ **do**

  $r \leftarrow r + r_i$

end for

for $i < N$ **do**

  $w_i \leftarrow \frac{r_i}{r}$

end for

The prerequisite for the rendering application is to provide as input the last rendering time of the GPUs. By knowing the initial workload ratio, we know the number of tasks assigned to the GPUs. Based on these parameters, we can calculate the relative rendering speed per GPU of the last frame for a given number of tasks.

If Δ_i is the rendering time frame of a GPU g_i and $|job_i|$ its number of tasks, then we get the rendering speed r_i as follows:

$$r_i = \frac{|job_i|}{\Delta_i} \tag{3}$$

Equation 3 yields the number of tasks rendered by g_i per unit of time. From the rendering speed r_i, we simply estimate the workload ratio w_i:

$$w_i = \frac{r_i}{r} \tag{4}$$

such as,

$$r = \sum_{i=0}^{N-1} r_i \tag{5}$$

Figure 3: Example visualization (cropped) of shuffled strips screen space decomposition with 2^{19} strips. The colors (red, green, blue and yellow) illustrate which GPU processes which strip.

where r (Equation 5) denotes the sampling frequency of the screen regions. It corresponds to the number of screen regions $|T|$ that can be rendered per unit of time. This DLB procedure is summarized in Algorithm 3.

4 Implementation

Our method is implemented inside our custom C++ Vulkan path tracer implementation. Rays from different viewpoints are traced simultaneously.

To manage the workload ratios in an omniscient way, a high level data structure called *task manager* oversees the scheduling and dynamic load balancing processes. For each GPU, this data structure stores the following information: index, workload ratio, rendering speed, rendering time and number of pixels. These information are first obtained or computed before the application starts to render. They are updated to the task manager from the path tracer at the end of each frame.

We divide the screen space by using the shuffled strips scheme [21], because this method proves to have a high scaling efficiency and can be easily combined with workload ratios. The number of shuffled strips per viewpoint is given by $|T_j| = 2^b$, where b is an integer that should maximize the number of strips inside a viewpoint's screen space. An example shuffled strips distribution is illustrated in Figure 3.

The static scheduling and the dynamic load balancing algorithms are directly implemented as functions belonging to the task manager. In this way, the updated information inside the task manager are directly used during the DLB algorithm.

The renderer evaluates imbalance by comparing the GPUs rendering timings to each other. The first rendering timing (for example from the GPU of index 0) we get from the renderer is considered as the

reference timing r_{ref}. We fix a 10% tolerance threshold above and below the reference timing. The other rendering timings r_i raise the state to *unbalanced* when:

- $r_i < 0.9 * r_{ref}$
- $1.1 * r_{ref} < r_i$

Depending on the state, it may call the DLB algorithm from the task manager.

The renderer avoids excessive dynamic load balancing by checking the imbalance every 5 frames. This way, we sample enough rendering timings to detect any unexpected load. It is done only when the time information returned by the path tracer is valid. When imbalance is not detected, we accumulate the rendering timings for each GPU (locally) and for all of them (globally). When imbalance is detected, we average the rendering timings accumulated over the 5 frames. This gives a local average for each GPU and a global average. When the system is imbalanced, if the rendering time of one GPU is below the global average, then the task manager records its rendering time. However, if its rendering time is higher, then the task manager records 0.75 of the local averaged rendering time of the GPU. This is done in order to limit the effect of rapid performance fluctuations on the DLB algorithm.

The control parameters of our proof-of-concept DLB implementation, specifically the timing tolerance threshold (10%), imbalance checking interval (5 frames), and the fluctuation regulation coefficient (0.75), are set empirically in order to provide stable behaviour in our experiments. However, for more general situations with a wider variety of heterogeneous compute devices and more sources of dynamism, the algorithm could be regulated more robustly by modelling the problem through a PID controller mechanism.

5 Experiments

The experiments are performed either on a heterogeneous dual-GPU machine composed of 1 GeForce RTX 3090 and 1 GeForce RTX 2080Ti, or on a homogeneous dual-GPU machine composed of 2 GeForce RTX 2080Ti. We use three existing scenes (Figure 4) that we customized in order to generate animated camera movements, to modify the materials or to add additional objects.

We conduct three different experiments:

- In Section 6.1 ("DLB vs. uniform workload distribution"), we set a fixed number of 3×3 views

(a) (b) (c)

Figure 4: Scenes used for the experiments: (a) Sponza, (b) San Miguel, and (c) Abandoned warehouse + Chinese dragon. Note: image exposure increased.

for the Sponza scene. We compare the performance on the same scene between 1 GPU, the heterogeneous dual-GPU machine with uniform workload distribution, and the same dual-GPU configuration with our DLB algorithm.

- In Section 6.2 ("Heterogeneous and homogeneous performance"), we use the San Miguel scene with multiple different virtual camera grid sizes: 2×1, 3×3, 5×5 and 9×9 views. We show the overall behaviour of mapping of the tasks to the GPUs with the DLB algorithm, when the number of views increases on a heterogeneous and a homogeneous dual-GPU machine.
- In Section 6.3 ("Spatial coherence"), we consider a homogeneous dual-GPU machine. For this, we use the Abandoned warehouse + Chinese dragon scene with the number of views fixed to 9×9. We exploit the spatial coherence of the views by reusing (reprojecting) the samples in order to decrease the rendering time. This allows to have an overview of the adaptability of our task mapping and our DLB algorithm in a more advanced situation. The reprojection is done from the central viewpoint which is path traced, to the other 80 viewpoints.

For all the experiments, the path tracer is configured with 1 sample per pixel (spp), 8 maximal bounces for paths, 100 frames, and next event estimation enabled. The views are path traced at an HD resolution (1280×720).

6 Results

6.1 Dynamic vs. uniform workload distribution

For the Sponza scene, we run an animation where the virtual camera is first placed on the roof. Progressively, the camera goes down into the courtyard where there are more reflections. We use a 3×3 grid

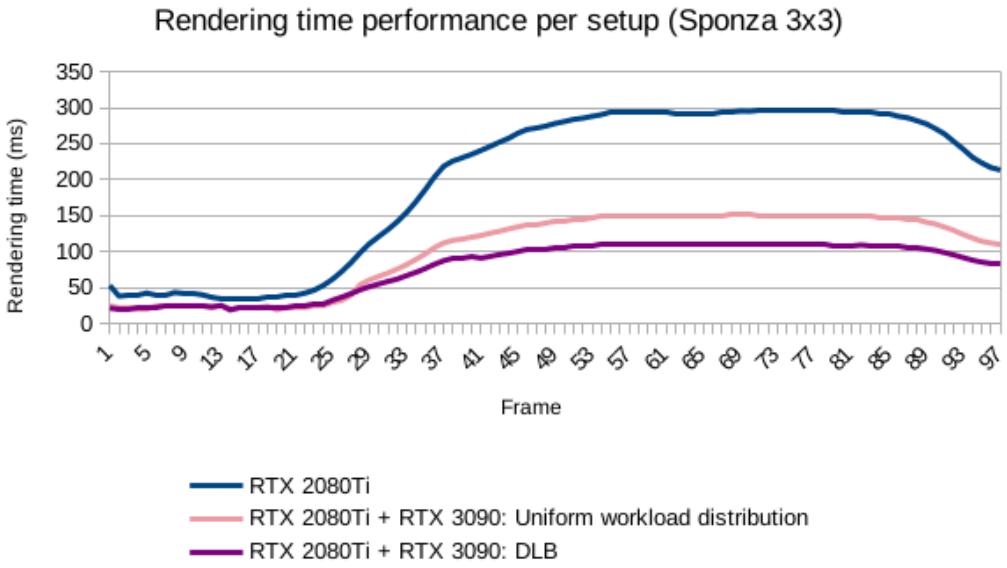


Figure 5: Rendering time over 100 frames for the Sponza scene on one RTX 2080Ti (blue), and on a dual-GPU RTX 2080Ti + RTX 3090 with uniform workload distribution (pink) and with dynamic load balancing (purple).

of viewpoints spaced each by 0.2 meters. The top left viewpoint of the grid is the origin of the virtual camera used to produce the animation.

Figure 5 reports the observed results on the heterogeneous dual-GPU with a 3×3 grid of viewpoints spaced each by 0.2 meters. We see that our DLB algorithm begins to clearly outperform the uniform workload distribution from about frame 30 onward; this is when we enter the courtyard, where the amount of diffuse reflection is more important than on the roof. The average speedup of the DLB algorithm over the uniform one, over 100 frames, is $\times 1.3$. The single-GPU performance is also shown in the figure for reference.

6.2 Heterogeneous and homogeneous performance

The San Miguel scene exhibits more complexity than Sponza, as it contains more intricate geometry. For this scene, the animation starts from a corner of the courtyard in the shadow, and then the camera progressively goes towards a region subject to direct illumination. We evaluate the efficiency and the stability of our imbalance detection policy (Figure 6 and 8). As we can observe for the heterogeneous dual-GPU configuration (Figure 6), during the first frames the dynamic load balancer begins to collect the average timings. Then, once the initial evaluation delay is past, the DLB algorithm decides to balance

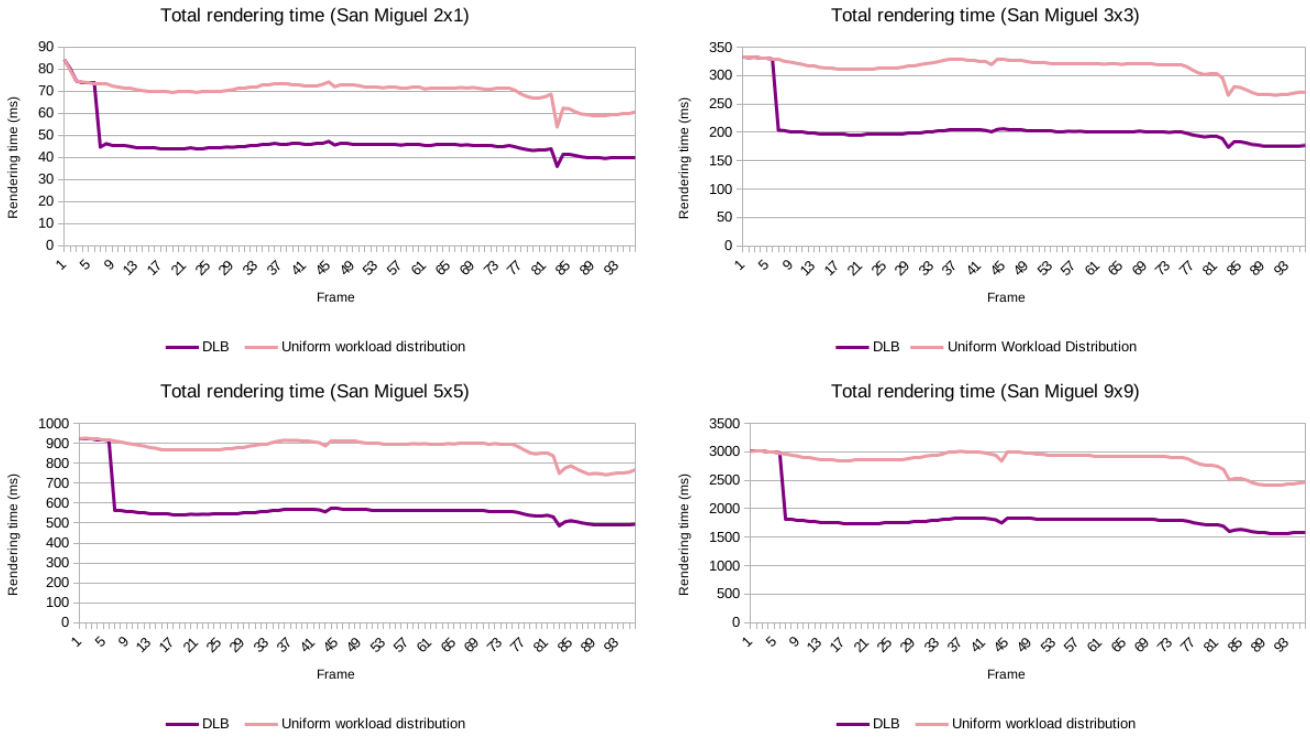


Figure 6: Rendering time on heterogeneous dual-GPU, for various view counts.

the workload (straight vertical purple line). After this, it seems that both GPUs reached their optimal rendering workload, so the dynamic load balancer is not called again, as shown in Figure 7.

On the other hand, for the homogeneous dual-GPU configuration, our imbalance detection policy is enough to handle the case where 2 GPUs have the same hardware capabilities, as shown in Figure 8. In that case, the dynamic load balancer is never called.

6.3 Spatial coherence

Table 1 shows a timing breakdown for the homogeneous dual-GPU setup with spatial reprojection enabled. The scene used is Abandoned warehouse + Chinese dragon scene for a grid of 9×9 viewpoints over 100 frames. The pixels of the source viewpoint are reprojected to the other 80 viewpoints. However, due to differences in the visible areas between viewpoints, some pixels cannot be reprojected, and would have to be filled in by path tracing. For simplicity, we do not consider the filling step, since all framebuffers of the destination viewpoints are rendered on the primary GPU right after the compositing stage.

We can see that the primary GPU RTX 2080Ti (0) spends 31 ms receiving G-buffers from the host.

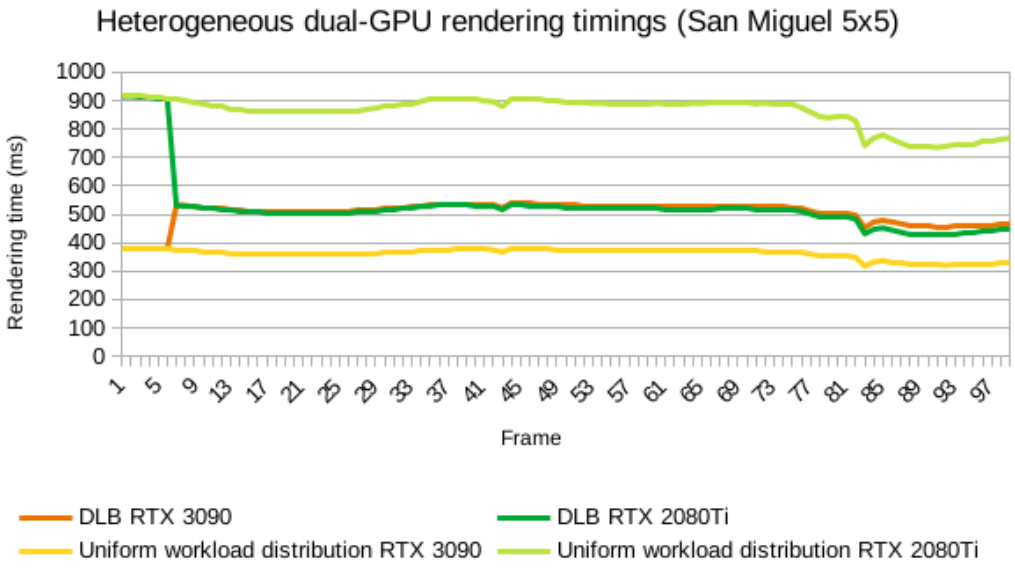


Figure 7: Profile of the rendering timings in the heterogeneous dual-GPU machine for San Miguel with a 5×5 viewpoint grid. Dark green and orange curves depict the rendering time of the two GPUs during the dynamic load balancing process. After 5 frames, they are overlapping each other, which illustrates a near perfect workload balance.

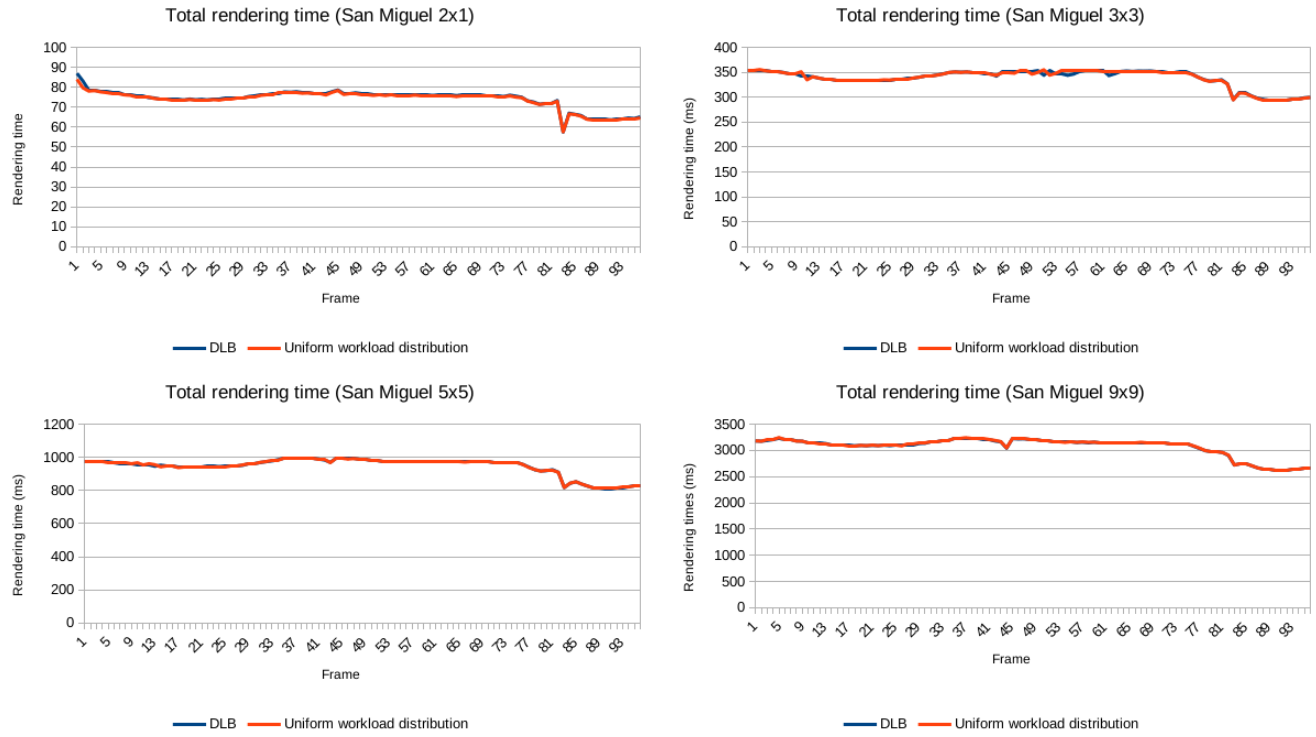


Figure 8: Rendering time on homogeneous dual-GPU, for various view counts.

	Uniform workload distribution		DLB	
	RTX 2080Ti (0)	RTX 2080Ti (1)	RTX 2080Ti (0)	RTX 2080Ti (1)
Path tracing (ms)	11.35	34.19	18.37	24.14
G-buffer transfers from host - for 80 viewpoints - (ms)	31.08	-	31.09	-
Spatial reprojection (ms)	7.41	-	7.40	-

Table 1: Averaged GPU timings (Abandoned warehouse + Chinese dragon, 9×9 viewpoints) on homogeneous dual-GPU **with spatial reprojection.**

The G-buffers are needed for the 80 reprojected viewpoints to determine whether or not reprojection is possible. This large data transfer affects the secondary GPU RTX 2080Ti (1). This leads to stalls in the rendering pipeline. The path tracing stage becomes subject to data access delays when the secondary GPU needs to receive updated scene buffers for path tracing or to send its framebuffer while the G-buffers are being sent to the primary GPU. This is noticeable by comparing the path tracing time of 1 viewpoint with spatial reprojection (Table 1) and without spatial reprojection (Table 2). The spatial reprojection from 1 path traced viewpoint to the other 80 viewpoints takes 7 ms. The secondary GPU is idle for ~ 38 ms on average for each time frame, waiting for the primary GPU to finish the reprojection pass.

Moreover, on the first row of results in Table 1, we see that the DLB algorithm, which is called 7 times, still reduces the time of path tracing compared to the uniform workload distribution. However, it tends to converge with difficulty to an equal path tracing time, since both GPUs have the same hardware capabilities and only data access delays perturb the measured path tracing time.

Overall, the task mapping model reveals its limitation when we try to exploit the spatial coherence between the views. As our spatial reprojection is done based on full path traced views, having a full bipartite graph representation as a task mapping model limits the potential full parallelization for multiview rendering. The frame times still remain high for targeting real-time utilization for displays with more than 2 viewpoints. This leaves room for improvement regarding how tasks are mapped to different GPUs. High frame times are also observed when spatial reprojection is used with a large number of high-resolution viewpoints, because it involves large G-buffer data transfers.

	Uniform workload distribution	
	RTX 2080Ti (0)	RTX 2080Ti (1)
Path tracing (ms)	12.44	11.62

Table 2: Averaged GPU timings (Abandoned warehouse + Chinese Dragon, 1 viewpoint) on homogeneous dual-GPU **without spatial reprojection and without DLB**.

7 Conclusions

We introduced a dynamic load balancing algorithm in our model suitable for multiview path tracing on multi-compute device platforms. The load balancing is done based on compute device workload ratios when imbalance is detected. The screen region decomposition is done by using shuffled strips as in [21].

As expected, the results show that when the GPUs have similar hardware capabilities, our proposed load balancer does not provide notable additional benefits over a static load balancing scheme. However, on a heterogeneous dual-GPU platform, the proposed dynamic load balancer reduces the rendering time on average by about 30–50% compared to uniform workload distribution, depending on the scene and the number of views. Our approach is thus more general than prior scheduling models related to multiview rendering, and is able to adapt to heterogeneous hardware combinations and dynamic camera movements on the fly to target real-time and interactive visualization for next-generation LF displays.

Acknowledgements

This work was partly supported by the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No. 956770 and by the Academy of Finland under Grant 325530.

Declaration of competing interest

We declare that we have no conflict of interest.

References

1 Mohammed S. Elbamby, Cristina Perfecto, Mehdi Bennis, and Klaus Doppler. Toward low-latency and ultra-reliable virtual reality. *IEEE Network*, 32(2), 2018.

2 James R Bergen and Edward H Adelson. The plenoptic function and the elements of early vision. *Computational models of visual processing*, 1, 1991.

3 Ivo Ihrke, John Restrepo, and Lois Mignard-Debise. Principles of light field imaging: Briefly revisiting 25 years of research. *IEEE Signal Processing Magazine*, 33(5), 2016.

4 Gregory Kramida. Resolving the vergence-accommodation conflict in head-mounted displays. *IEEE Transactions on Visualization and Computer Graphics*, 22(7), 2016.

5 Hong Hua. Enabling focus cues in head-mounted displays. In *Imaging and Applied Optics 2017 (3D, AIO, COSI, IS, MATH, pcAOP)*. Optica Publishing Group, 2017.

6 Douglas Lanman and David Luebke. Near-eye light field displays. *ACM Trans. Graph.*, 32(6), nov 2013.

7 Turner Whitted. An improved illumination model for shaded display. *Commun. ACM*, 23(6), jun 1980.

8 Robert L. Cook. Stochastic sampling in computer graphics. *ACM Trans. Graph.*, 5(1), jan 1986.

9 James T. Kajiya. The rendering equation. In *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '86. Association for Computing Machinery, 1986.

10 Matt Pharr, Wenzel Jakob, and Greg Humphreys. *Physically Based Rendering: From Theory to Implementation*. Morgan Kaufmann Publishers Inc., 3rd edition, 2016.

11 John L. Gustafson. Reevaluating Amdahl's law. *Commun. ACM*, 31(5), may 1988.

12 Timo Aila and Samuli Laine. Understanding the efficiency of ray traversal on GPUs. In *Proceedings of the Conference on High Performance Graphics 2009*, HPG '09, 2009.

13 Paul A. Navrátil and William R. Mark. An analysis of ray tracing bandwidth consumption. 2006.

14 Steven Molnar, Michael Cox, David Ellsworth, and Henry Fuchs. A sorting classification of parallel rendering. *Computer Graphics and Applications, IEEE*, 14, 08 1994.

15 Kenneth Moreland, Brian Wylie, and Constantine Pavlakos. Sort-last parallel rendering for viewing extremely large data sets on tile displays. In *Proceedings of the IEEE 2001 Symposium on Parallel and Large-Data Visualization and Graphics*, PVG '01. IEEE Press, 2001.

16 Greg Humphreys, Mike Houston, Ren Ng, Randall Frank, Sean Ahern, Peter D. Kirchner, and James T. Klosowski. Chromium: A stream-processing framework for interactive rendering on clusters. *ACM Trans. Graph.*, 21(3), jul 2002.

17 I Wald, GP Johnson, J Amstutz, C Brownlee, A Knoll, J Jeffers, J Günther, and P Navratil. Ospray - a CPU ray tracing framework for scientific visualization. *IEEE Transactions on Visualization and Computer Graphics*, 23(1), 2017.

18 Stefan Eilemann, Maxim Makhinya, and Renato Pajarola. Equalizer: A scalable parallel rendering framework. *IEEE Transactions on Visualization and Computer Graphics*, 15(3), 2009.

19 Tim Biedert, Kilian Werner, Bernd Hentschel, and Christoph Garth. A Task-Based Parallel Rendering Component For Large-Scale Visualization Applications. In Alexandru Telea and Janine Bennett, editors, *Eurographics Symposium on Parallel Graphics and Visualization*. The Eurographics Association, 2017.

20 Will Usher, Ingo Wald, Jefferson Amstutz, Johannes Günther, Carson Brownlee, and Valerio Pascucci. Scalable Ray Tracing Using the Distributed FrameBuffer. *Computer Graphics Forum*, 2019.

21 Dietger van Antwerpen, Daniel Seibert, and Alexander Keller. *A Simple Load-Balancing Scheme with High Scaling Efficiency*. Apress, 2019.

22 Feng Xie, Petro Mishchuk, and Warren Hunt. Real time cluster path tracing. *CoRR*, abs/2110.08913, 2021.

23 Fatih Erol, Stefan Eilemann, and Renato Pajarola. Cross-Segment Load Balancing in Parallel Rendering. In Torsten Kuhlen, Renato Pajarola, and Kun Zhou, editors, *Eurographics Symposium on Parallel Graphics and Visualization*, 2011.

24 Stefan Eilemann, David Steiner, and Renato Pajarola. Equalizer 2.0–convergence of a parallel rendering framework. *IEEE Transactions on Visualization and Computer Graphics*, 26(2), 2020.

25 Tim Biedert, Peter Messmer, Thomas Fogal, and Christoph Garth. Hardware-Accelerated Multi-Tile Streaming for Realtime Remote Visualization. In Hank Childs and Fernando Cucchietti, editors, *Eurographics Symposium on Parallel Graphics and Visualization*, 2018.

26 Xue Liu, Xinzhu Sang, Xiao Guo, Shujun Xing, Yuanhang Li, Jinhui Yuan, and Binbin Yan. Real-time super high resolution light field rendering with multi-GPU scheduling. In *2021 Asia-Pacific Conference on Communications Technology and Computer Science (ACCTCS)*, 2021.

27 S.J. Adelson and L.F. Hodges. Generating exact ray-traced animation frames by reprojection. *IEEE Computer Graphics and Applications*, 15(3), 1995.

28 Markku J. Mäkitalo, Petrus E. J. Kivi, Matias Koskela, and Pekka Jääskeläinen. Reducing computational complexity of real-time stereoscopic ray tracing with spatiotemporal sample reprojection. In *VISIGRAPP*, 2019.

29 Markku J. Mäkitalo, Petrus E. J. Kivi, and Pekka O. Jääskeläinen. Systematic evaluation of the quality benefits of spatiotemporal sample reprojection in real-time stereoscopic path tracing. *IEEE Access*, 8, 2020.

30 Niko Wißmann, Martin Mišiak, Arnulph Fuhrmann, and Marc Erich Latoschik. Accelerated stereo rendering with hybrid reprojection-based rasterization and adaptive ray-tracing. In *2020 IEEE Conference on Virtual Reality and 3D User Interfaces (VR)*, 2020.

31 Basile Fraboni, Jean-Claude Iehl, Vincent Nivoliers, and Guillaume Bouchard. Adaptive Multi-view Path Tracing. In Tamy Boubekeur and Pradeep Sen, editors, *Eurographics Symposium on Rendering - DL-only and Industry Track*, 2019.

32 Basile Fraboni, Antoine Webanck, Nicolas Bonneel, and Jean-Claude Iehl. Volumetric multi-view rendering. *Computer Graphics Forum*, 41(2), 2022.

33 Souley Madougou, Ana Varbanescu, Cees de Laat, and Rob van Nieuwpoort. The landscape of GPGPU performance modeling tools. *Parallel Computing*, 56, 2016.

34 Unai Lopez-Novoa, Alexander Mendiburu, and Jose Miguel-Alonso. A survey of performance modeling and simulation techniques for accelerator-based computing. *IEEE Transactions on Parallel and Distributed Systems*, 26(1), 2014.

35 Per Ganestam and Michael Doggett. Auto-tuning interactive ray tracing using an analytical GPU architecture model. In *Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units*, 2012.