Utrecht University

Faculty of Science

Game and Media Technology Master Thesis

Thread Divergence Reduction in Path Tracer using Custom Thread Scheduler

First examiner:

Peter Vangorp

Candidate:

Georgios Psomathianos

Second examiner:

Jacco Bikker

July 12, 2024

Abstract

This master thesis' main purpose is to research, implement and experiment improved and innovative methods to reduce thread divergence in the path tracer algorithm. The main research question this work aims to answer is how to make the path tracer algorithm more memory and control flow coherent. Better ray coherency would utilize the SIMT model better, thus, ensuring higher performance of the GPU.

The main suggested method includes the incorporation of thread reordering compute shaders that aim to group threads into the same warps based on the last bounce's spatial information. For the sorting algorithm, a parallel bitonic sort is used and the current state-of-the-art radix sort is suggested as an alternative. The sorting elements include sorting keys that derive from the position and direction of the rays.

Specifically, the sorting keys involve the estimated termination points of the rays from the previous bounce. We start with using a fixed length that derives from the scene's bounds to confirm our hypothesis, whether ray reordering can reduce thread divergence and improve the overall performance. Furthermore, three distinct formulas are examined and compared. These include calculating and caching the average, the moving average, and approximated gaussian distributions of the rays' lengths. Finally, we examined clamping the spatial grid's values based on a fixed distance from the camera's position, as it could potentially yield interesting results.

Contents

1. Introduction

Through the years many advancements have been made in the field of computer graphics for both real-time and offline applications.

Real-time applications such as video-games have high framerate as the main requirement. Specifically, $60fps$ is the minimum standard for players to ensure smooth gameplay without motion sickness. Furthermore, studies have shown that high framerate is associated with higher player performance [1] and quality of experience [2]. Therefore, computationally cheap and fast rendering algorithms can only be used to remain within that small frame time window ($16msec$), with the significant drawback of sacrificing image and lighting quality. The rasterization algorithm has been the dominant method and a lot of research had been conducted to make both the software and hardware efficient in that algorithmic direction. However, as mentioned above it suffers from capturing the physically-based global illumination of the scene.

Offline applications, especially in the film industry, do not have that hard framerate constraint, as high fidelity is at the highest desire. At these applications a different algorithm is used, called path tracer. This algorithm incorporates rays as its main component and accurately approximates light propagation and the underlying light physics perfectly capturing the scene's radiance. This is why film graphics exhibit a higher level of realism and physical fidelity comparaed to their counterparts in video-games. However, it is an expensive algorithm as thousand of samples need to be computed to render a high quality and noise-free image, thus it had been mostly used in offline applications.

The gap in real-time and offline applications had been significant enough in render quality. However, reducing that gap has been more possible now in the last 5 years with the introduction of RTX-cards [3]. NVIDIA managed to redesign its gpu cards to support native hardware ray tracing acceleration. This includes hardware that accelerates ray/triangle intersections and bounded volume hierarchy traversal. Real-time ray tracing has been more possible ever since, as more research has been focused to enable real-time ray tracing in lower framerates and higher resolutions. A lot of video-games have been made over the last half decade, in which their superiority over global illumination is evident[4].

Even with the advancement of the hardware, real-time global illumination is still not as impeccable as in offline and production rendering. Specifically, only a few rays/samples per pixel can be calculated to be able to satisfy the real-time constraint. This obviously either introduces bias (the resulting image has less or more light energy, thus not correct) or variance in a form of spatial and temporal noise.

Many innovations have been made in the last years trying to increase the fidelity of real-time rendering. One approach aims to make the path tracer algorithm more efficient in regards to sampling. State-of-the-art path tracers focus on calculating samples that yield the greatest contribution to the final result. Another approach is reducing the resulting variance by either conventional image processing techniques or deep learning architectures. These methods vary from reusing previous pixels, denoising to resolution upscaling, as will be also discussed in greater detail in section 2. The final approach, which is also the one that will be studied in this thesis is modifying the path tracer algorithm with respect to hardware capabilities. GPUs consist of thousand of thread units enabling massive parallelism. However, threads are organized in threadgroups (waves/warps in AMD/NVIDIA architectures) and execute in lockstep under the SIMT(Single Instruction Multiple Threads) model. This means that threads within the same threadgroup need to execute the same code instructions. If a single thread needs to execute another line of code (e.g. due to an if statement) then the others need to enter to a stall status, waiting for the other one to finish. In the literature, this is known as thread divergence and has a significant performance impact on the modern GPU hardware. Basically, threads remain idle underutilizing the GPU instead of executing useful work. Common techniques include dividing different part of the algorithm into separate kernels instead of an ubershader and thread reordering.

2. Related Work

In this section they will be a short background of the path tracer algorithm and a detailed explanation of the state-of-the-art techniques in the three different approaches that were presented previously.

2.1 Sampling

2.1.1 Background

The global illumination problem consists of solving the rendering equation[5].

$$L_o(\mathbf{x}, \omega_o) = L_e(\mathbf{x}, \omega_o) + \int_\Omega f_r(\mathbf{x}, \omega_i) L_i(\mathbf{x}, \omega_i)(\omega_i \cdot \mathbf{n}) d\omega_i \tag{2.1}$$

The equation 2.1 is a simplified version of the rendering equation of the outgoing radiance L_o of a point x from an outgoing direction ω_o, where L_e is the emission term, f_r the bidirectional distribution function and L_i the incoming radiance [6]. A critical observation is that there is no analytical solution for it. Meanwhile, the term L_i can either be caused due to direct lighting or indirect lighting from light bouncing off surrounding surfaces. In the latter case, recursiveness occurs, making the equation not trivial to solve. Hence, only iterative methods like Monte Carlo estimators [7] that can approximate the result can be used even in offline rendering.

$$L_o(\mathbf{x}, \omega_o) = L_e(\mathbf{x}, \omega_o) + \frac{1}{N} \sum_{i=0}^{N} f_r(\mathbf{x}, \omega_i) L_i(\mathbf{x}, \omega_i)(\omega_i \cdot \mathbf{n}) \tag{2.2}$$

In the equation 2.2 random samples are being drawn using a uniform distribution, hence the division by N. This is a naive approach, since there are samples that offer almost negligible contribution to the equation's estimation, wasting computation time. In practice, there might be only a fraction of the total samples that could greatly contribute to the final image. The importance sampling technique tries to mitigate this by drawing samples from a distribution that better approximates the function at issue.

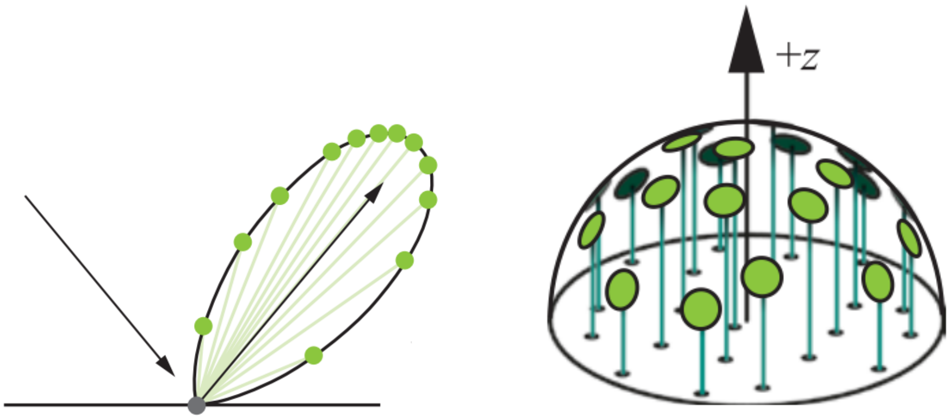
$$L_o(\mathbf{x}, \omega_o) = L_e(\mathbf{x}, \omega_o) + \sum_{i=0}^{N} \frac{f_r(\mathbf{x}, \omega_i) L_i(\mathbf{x}, \omega_i)(\omega_i \cdot \mathbf{n})}{p_i} \tag{2.3}$$

In equation 2.3 p_i is the probability of drawing the sample i from the distribution that was used.

There are many distributions that can be used to approximate the product of $f_r \cdot L_i \cdot (\omega_i \cdot \mathbf{n})$. The inverse method, a widely adopted technique in probability theory, is commonly employed for sampling from a proposed distribution:

1. Compute the CDF $P(x) = \int_0^x p(x')dx'$
2. Compute the inverse $P^{-1}(x)$
3. Sample a uniformly distributed random number $\xi \in [0, 1]$
4. Compute sample $X_i = P^{-1}(\xi)$

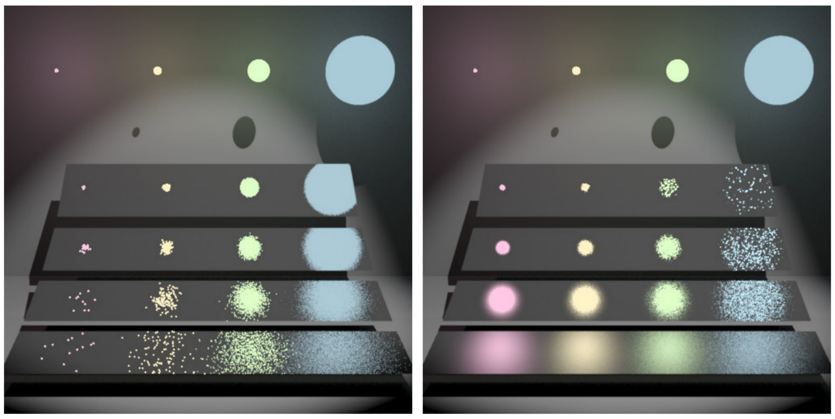
It is trivial to understand that the best distribution would be $P \sim \frac{1}{C} \cdot f_r \cdot L_i \cdot (\omega_i \cdot \mathbf{n})$, where C is the normalization factor(probabilities need to integrate to 1). However, it is not possible to calculate the normalization factor and the CDF, because the specific product and hence the integral is too complex. For example the L_i factor depends on the scene. The most common method is to use a combination of distribution functions. For example, light source sampling for direct lighting (L_i approximation) or brdf sampling for indirect lighting (f_r approximation).



(a) BRDF Distribution **(b)** Cosine-Weighted Distribution

Figure 2.1: Common distributions

There are cases, in which sampling from just one distribution does not greatly result in variance reduction. For example, there cases in which a light source might be very small when projected to a surface, however with high intensity. Sampling from the brdf in that case would yield poor results since hitting such a small light source is statistically unlikely. Similarly, a very smooth surface with a a very thin specular lobe would have high variance, if samples are drawn from e.g. a light source.



(a) Sampling the BSDF **(b)** Sampling the light sources

Figure 2.2: From left to right increasing the light source. From top to bottom increasing the material's roughness

Veach tried to tackle this issue in his thesis[8] by proposing a new method called, multiple importance sampling. The main idea is that multiple samples can be drawn from multiple distributions.

$$F = \sum_{i=1}^{n} \frac{1}{n_i} \sum_{j=1}^{n_i} w_i(X_{i,j}) \frac{f(X_{i,j})}{p_i(X_{i,j})} \tag{2.4}$$

In equation 2.4, n are the number of different distributions or sampling strategies being used. The term $w_i(X_{i,j})$ is a weighting term for taking into consideration the importance of each strategy's sample. Veach concluded that the power heuristic yielded the best results

$$w_i = \frac{q_i^2}{\sum_k q_k^2} \tag{2.5}$$

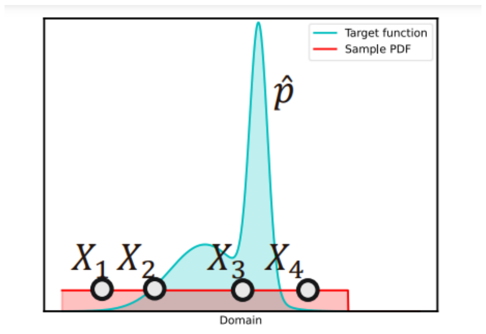
The multiple importance sampling does provide better results compared to single importance sampling. However, it is more computationally heavy, as multiple samples (from different distributions) must be evaluated. In the path tracer usually it is applied during the direct lighting estimation, in which both light and brdf sampling is taking into consideration. In most cases MIS proves to be a substantial variance reduction technique, justifying the computational overhead.

2.1.2 State-Of-The-Art

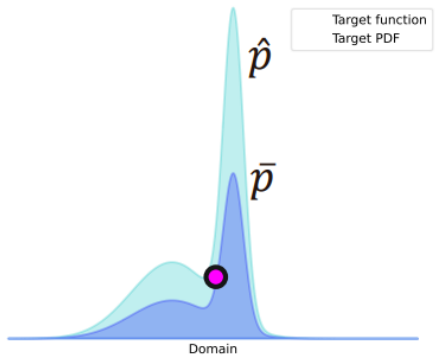
So far the path tracer can yield satisfying results with the modern hardware capabilities and good sampling techniques. However, challenges arise as the number of light sources that must be considered increases. Due to real-time constrains, the feasibility is limited to sampling from a single light source. In cases of a few lights or when lighting is uniform among the various light sources, 1 sample per pixel with the combination of temporal reprojection and denoising, which will be discussed in the next section, can yield decent rendering results. In more complex lighting scenarios, where hundreds of lights exist within the scene, hundreds or more samples are required.

The suggested method called ReSTIR[9] aims to reuse samples across neighboring pixels and frames within the path tracer. Its goal aligns with that of a denoiser, however it does not operate as an image post-process stage, thereby there is not any image blurring and bias introduced, preserving lighting detail. It is claimed that this method can yield 100x to 1000x sample count multiplier, hence enabling the possibility of having hundreds of lights in the scene, while remaining real-time with low variance.

The main concept in ReSTIR is that we can generate a set of samples from a cheap approximation(e.g. uniform) and through the process of reuse and reweighting, estimate a target function. For example, the target function could be the complex product that is impractical to sample from.



(a) Uniform distribution and target function



(b) Approximation of target function

Figure 2.3: Reusing samples drawn from a uniform distribution can be reweighted as if they have been drawn from a target distribution

The significance of ReSTIR stems from the fact that cheap samples can be drawn and by reusing and reweighting them it can be observed as if they were drawn by a distribution function $\overline{p}$ which is normalized and integrates to 1, satisfying the requirements of a valid distribution function. As more samples are being used the probability function approximates the target function $\overset{\prime}{p}$. This is indeed important, because the target function does not have to be a distribution function. In other words, we do not need to know the normalization factor and know in advance the formula of calculating the probability of a sample drawn from that target function. A better approximation of the product $f_r \cdot L_i \cdot (\omega_i \cdot \mathbf{n})$ can be used without having to transform it to a distribution function. Another important observation is that ReSTIR can be combined with MIS(Multiple Importance Sampling), if samples are drawn from various $PDFs$.

So far sample reusage was specified only within the same frame and same pixel. Furthermore, samples can be used from neighboring pixels and previous frames, resulting in a significant number of samples. Basically, this algorithm aggregates many samples from past frames and surrounding pixels giving an even better approximation of the target function, thus yielding significant variance reduction. However, samples from previous frames and neighboring samples might contain invalid samples. Specifically, their path domain might be different and only partially overlap with the sample we want to aggregate with. Samples from different domains need to be tested whether they are eligible for path vertex connection and later a shift mapping domain function need to be applied to them so that they can be used [10]

2.1.3 AI

In ReSTIR, it was explained that samples can be reused to better approximate a target function (perfect importance sampling). However, a sample is being drawn by aggregating previously samples (spatiotemporal reusage) and calculating an unbiased weight value. The true probability and therefore the distribution is not known and can not be calculated, which seemed fine according to the results it produced. A different method called "path guiding" [11] aims to learn a better distribution from which samples can be drawn, as more samples are being used. Specifically, there are decent sampling techniques like brdf, light sampling and next event estimation. However, the radiance term L_i in the rendering equation is not known a-priori, because it is scene dependent. This technique aims to approximate a better distribution function with that term considered from previously drawn samples like in a typical machine learning problem.

The basic concept consists of Gaussian mixture models that can be used to guide the sampling of a new direction when path tracing [12].

$$GMM(\mathbf{s}|\theta) = \sum_{j=1}^{K} \pi_j \mathcal{N}(\mathbf{s}|\mu_j, \Sigma_j) \tag{2.6}$$

New directions can be drawn using the trained Gaussian mixture models that are stored in a spatial cache. Importance and radiance distributions are being trained on the fly using the EM algorithm [13].

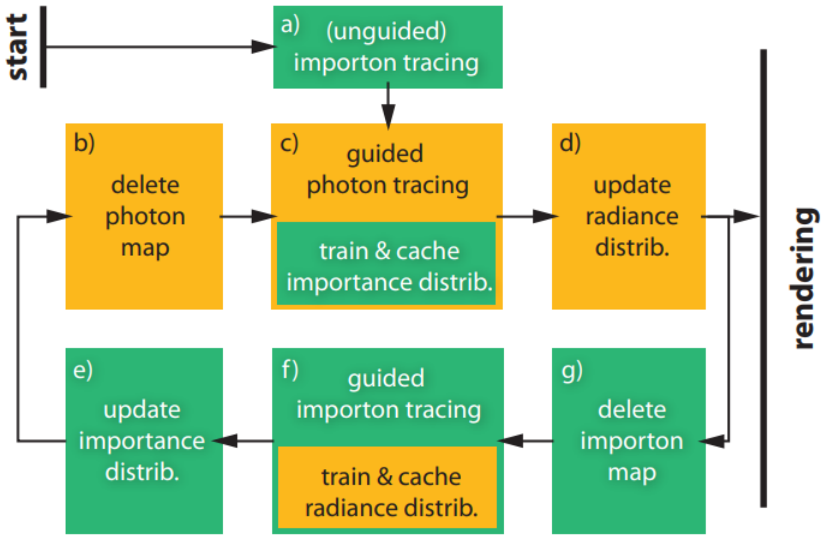

start
a) (unguided) importon tracing
b) delete photon map
c) guided photon tracing
train & cache importance distrib.
d) update radiance distrib.
rendering
e) update importance distrib.
f) guided importon tracing
train & cache radiance distrib.
g) delete importon map

Figure 2.4: The training pipeline of the GMM distributions.

In the paper [14]] which is based on the same method, stores the distributions in framebuffers instead of caching them in spatial data structures. So their algorithm is on a pixel-basis instead of surfaces and points in the scene. Specifically the distributions parameters (e.g. mean value, standard deviation) are stored in a Γ buffer and the radiance estimations are stored as VPLs(Virtual Point Lights) in a Π buffer.

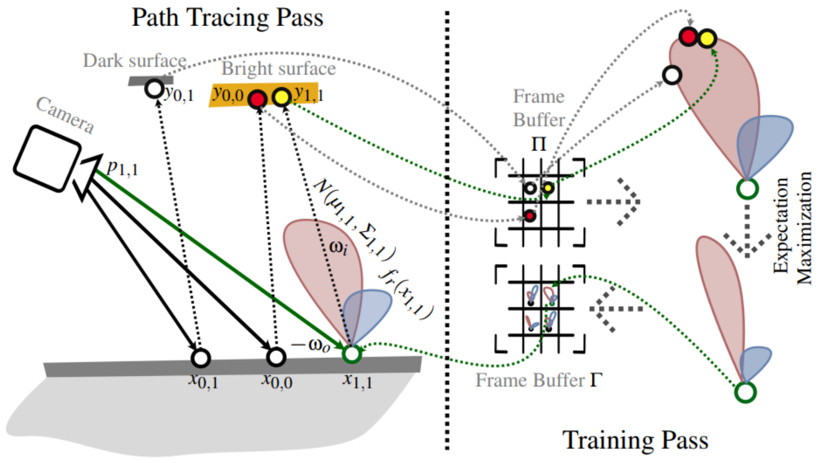


Figure 2.5: Parameters are loaded from the Γ buffer to sample a new direction. The estimated radiance is stored in the Π buffer, which will be used during the training phase to calculate the updated values of the distributions.

2.2 Post-Processing

2.2.1 Denoising

The previous methods focused on making the path tracer more efficient by introducing better sampling methods to reduce the variance. Even with perfect important sampling, post-processing stages are required to temporally reproject and accumulate previous frames and blur the remaining noise.

The Spatio-Temporal Variance Guided Filter (SVGF) [15] has been a popular standard in the industry of real-time ray tracing.

The first stage is temporal reprojection, in which pixels using motion vectors from the G-Buffer are being reprojected in the previous frame to match the same surface. Frames are being accumulated based on the temporal integration function $C'_i = \alpha \cdot C_i + (1 - \alpha) \cdot C'_{i-1}$. The temporal accumulation factor expresses the influence previous frames have on the final accumulated frame. Hence as long as pixels are correctly reprojected and remain valid, more samples are accumulated, thus reducing the variance. Pixels are valid when they refer to the same surface by checking their normal and depth values. If they are not, the accumulation factor becomes zero, instantly dropping the accumulated samples of the pixel.

For spatial denoising, a first stage is required to approximate each pixel's variance. Finally, a multi-pass a-trous wavelet filter is applied to blur any remaining noise. Specifically, a pixel's value is estimated by a weighted sum of its neighboring pixels, which are influenced by edge-stopping functions. The edge-stopping functions reduce the neighbors' contribution based on Gaussian blur kernels that their mean and variance values depend on the absolute difference of normals, depth, luminance and luminance variance that was estimated at the previous stage.

$$w_z = exp(-\frac{|z(p) - z(q|}{\sigma_z|\nabla z(p) \cdot (p - q)| + \epsilon)} \tag{2.7}$$

$$w_n = max(0, n(p) \cdot n(q))^{\sigma_n} \tag{2.8}$$

$$w_l = exp(-\frac{|l_i(p) - l_i(q)|}{\sigma_l \sqrt{g_{3x3}(Var(l_i(p))} + \epsilon}) \tag{2.9}$$

Above we can see the edge-stopping functions. The σ values are user-defined variables that control the influence of each of the function. The variance in the luminance function is being prefiltered in a gaussian 3x3 kernel.

NVIDIA introduced another method to temporally accumulate and denoise, caled ReBLUR [16]. It is 50% faster, more efficient in temporal stability than the SVGF variant, while yielding decent results even at 0.5 ray per pixel.

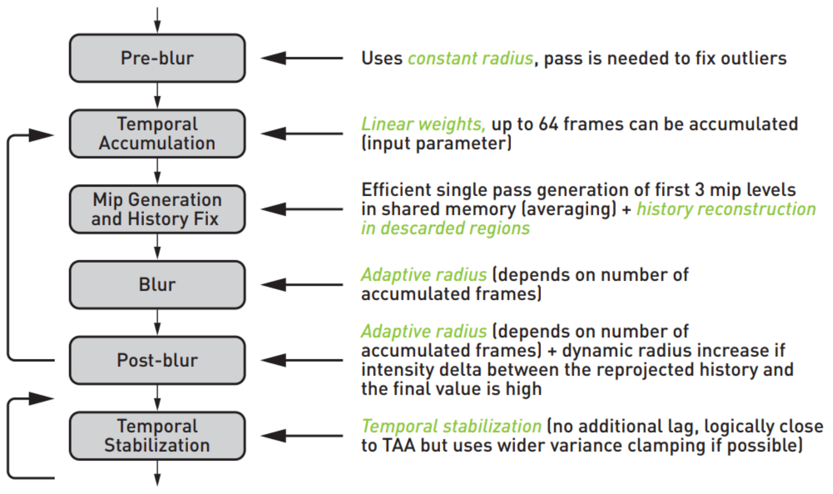

Pre-blur
Temporal Accumulation
Mip Generation and History Fix
Blur
Post-blur
Temporal Stabilization
Uses constant radius, pass is needed to fix outliers
Linear weights, up to 64 frames can be accumulated (input parameter)
Efficient single pass generation of first 3 mip levels in shared memory (averaging) + history reconstruction in descarded regions
Adaptive radius (depends on number of accumulated frames)
Adaptive radius (depends on number of accumulated frames) + dynamic radius increase if intensity delta between the reprojected history and the final value is high
Temporal stabilization (no additional lag, logically close to TAA but uses wider variance clamping if possible)

Figure 2.6: Pipeline overview of REBLUR.

It is worth noting that diffuse and specular lighting is handled differently during the reprojection process. This is due to the fact that surface reprojection works as expected with diffuse lighting, however, on reflections the lighting is not consistent introducing ghosting and lagging artifacts when using only motion vectors. Specifically, the maximum number of accumulated frames is clamped based on the camera movement and the distance between the camera and the surface. Later, virtual-motion based position technique was introducted to reproject to the previous frame, in which the motion of the reflected world is taken into consideration instead. This method works for perfect mirrors, however, in case of rough reflections a scaling factor is calculated that states how close to the surface the reprojected position should be.

$$X_{virtual} = X - V \cdot d_{hit} \cdot f \tag{2.10}$$

where X is the current position, V is the view vector, d_{hit} is the hit distance and f is the scaling factor. The scaling factor is calculated using a formula that takes into account the material roughness, and the dot product between the normal and view vector.

A common problem of denoisers is that they tend to overblur some details. This is more apparent in the SVGF denoiser. Common technique to minimize this is to demodulate albedo (material details) and store just the lighting in the resulting texture buffers. After denoising filtering is applied to the resulting buffers, modulation occurs by just multiplying them with the albedo to retrieve the final result.

2.2.2 Upsampling

A different, though standard, approach is to use an upsampling stage. The cost of the path tracer is analogous to the resolution. In other words, what if we could render in lower resolution to minimize the work load and use a cheaper upsampling stage that brings the path traced image to the desired resolution.

NVIDIA managed to achieve sufficient results in record time with their Deep Learning Super Sampling architecture (DLSS) [17]

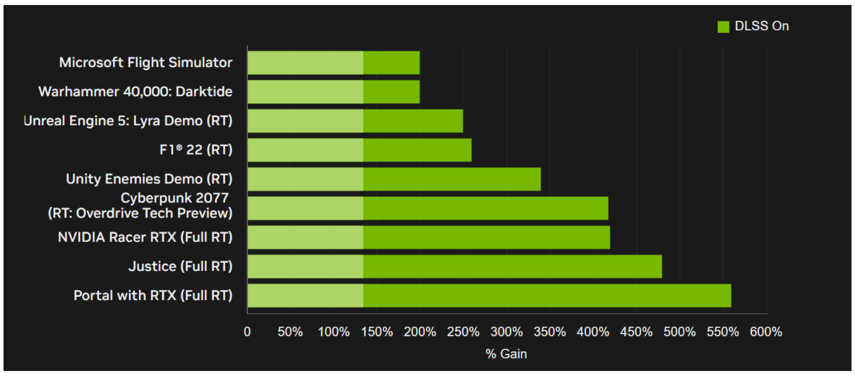


Figure 2.7: DLSS performance gain in popular games for 3840x2160 resolution.

Obviously the picture is not crystal clear and perfect compared to the ray traced image at the same resolution. However, the temporal and spatial artifacts are not that noticeable. The gained performance is definitely worth the reduced image quality.

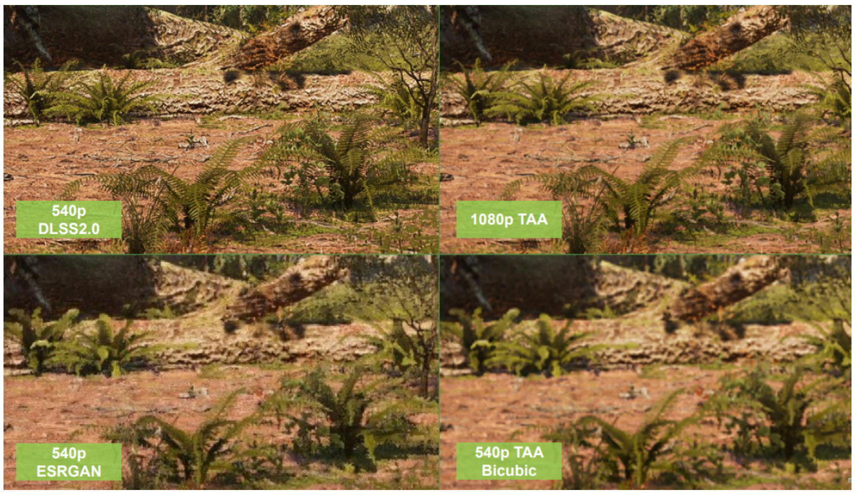


Figure 2.8: DLSS comparison with popular image processing upsampling techniques.

2.3 Thread Divergence

The cost of the path tracer algorithm comes from the tremendous number of rays and samples that are required to achieve an unbiased and low variance result. The above techniques have been suggested to achieve a plausible result just with 1 ray per pixel (or even a half or a quarter). However, a significant impact to performance stems from the inefficiency of the algorithms when they are executed on the graphics cards. On the GPU, the threads are organised in threadgroups and executed in lockstep. Algorithms, shader and kernel programs that follow the SIMD model can achieve a sufficient utilization of the hardware. The path tracer at its nature is not cohesive, meaning that work of threads within the same threadgroup will diverge. When control flow changes within the threadgroup, threads have to stall by being masked out. Basically, they remain idle not producing any useful work. Extensive research has been conducted, trying to innovate methods and techniques that try to redesign the algorithm to achieve a better GPU utilization taking into consideration the hardware capabilities and restrictions.

Aila et al. [18] examined the performance of the trace method, which is responsible for traversing the acceleration structure in order to find which primitive a ray intersects. Any tests and experiments were compared with a hardware simulation in order to calculate an upper bound of the cost and investigate how close to that theoretical bound a practical implementation is. They noticed that only the most coherent part near the root gets accelerated, getting more incoherent rays the deeper the acceleration structure goes.

Packet traversal is a common strategy, in which rays form groups and follow the same path in the tree. Memory accesses are more coherent, however, rays visit nodes that do not actually intersect, thus not doing useful work. The actual performance was not consistent to the simulated counterpart, revealing that there might be limitations due to memory bandwidth. On the other hand, per-ray traversal rays are handled independently and visit exactly the nodes visited. Although the latter is less memory coherent, performs better compared to the former, proving that the cost is not due to memory. Specifically, the if-if trace variant compared to the while-while one is even less memory coherent and performs the best. The reason for that is in the if-if case there are less long-running warps that might cause starvation.

```
if-if trace():
  while ray not terminated
    if node does not contain primitives
      traverse to the next node
    if node contains untested primitives
      perform a ray-primitive intersection test
```

(a) If-if Trace

```
while-while trace():
  while ray not terminated
    while node does not contain primitives
      traverse to the next node
    while node contains untested primitives
      perform a ray-primitive intersection test
```

(b) While-while trace

Figure 2.9: Per-ray traversal

Speculative traversal bypasses the thread scheduler by launching only enough threads in the beginning to fill the GPU once. Work is being fetched from a global pool using atomic operations. This strategy is superior, unless the memory subsystem starts being the bottleneck.

Laine et al.[19] examined the benefits of splitting the whole ray tracing process into smaller kernels instead of a megakernel. Specifically, a megakernel would introduce a lot of divergence to control flow, thus masking out threads and limiting the latency-hiding of the GPUs. Furthermore, a megakernel would require a lot of resource usage (e.g. registers) limiting the number of active threads that can reside simultaneously at the stream processors. Overloading of the caches is, also, another possible scenario. Basically, they split the whole process into 3 smaller stages/kernels; logic, material and ray cast. The logic state is at mostly responsible for everything besides material evaluation. Expensive materials can have their own kernels so that rays can be grouped together and eventually evaluate the same material and thus execute the same instructions. Shadow casting is done in the final stage. The drawback of this is a path state of each ray needs to reside in global memory so kernels can communicate and transfer data. However, according to their results this strategy outweighs the above overhead.

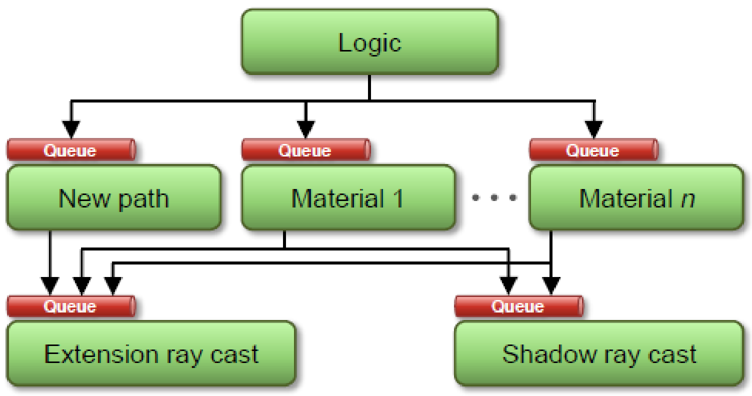


Figure 2.10: Smaller kernels design

Frey et al. [20] investigated branch and termination divergence. The strategy for tackling termination divergence is similar to speculative traversal [18]. Tasks are being fetched from a task pool, whenever a thread finishes with a task a new one is being fetched. There are different types of task pools; local, global and hybrid.

1. Local: Task pools are statically initialized and assigned to each thread block. Since they reside in block memory, task fetching is cheap, however, it does not yield the best performance in cases of high divergence.
2. Global: A global task pool resides in global memory. Each thread has a private task pool from which tasks are fetched. Whenever a private pool becomes empty it fetches a chunk of tasks from the global pool. This strategy offers a very fine-granular of task distribution, leading to

a decent iteration length divergence compensation. However, it can be expensive due to many global memory accesses. Furthermore, tasks within the same warp might be scattered leading to incoherent memory accesses.

3. Hybrid: Combines a local and global task pool. Basically, instead of private pool each warp has a local pool that fills tasks from the global whenever its empty. One thread (smallest id) within a warp is responsible to refill the local pool. This strategy minimizes global memory accesses while preserving locality within the warp.

Meister et al.[21] proposed reordering rays in order to better utilize the SIMD model. Rays can be mapped into 1-D array based on sorting keys. The rays' origin, direction and/or combination with simple bit interleaving can be used as input into hashing functions that calculate those sorting keys. The main concept is that rays with similar origin or direction can have almost identical indices of the array, for instance by differing only in 1 bit (e.g. with Morton code). That way, similar rays reside in the same memory blocks, ensuring data locality. The suggested method concluded that the actual ray data reodering process is the most expensive step. This is due to the incoherent memory accesses in order to create coherent ray buffers. Reducing this overhead could be key to increase the overall performance. Despite that the speedup is noticeable in big and complex scenes.

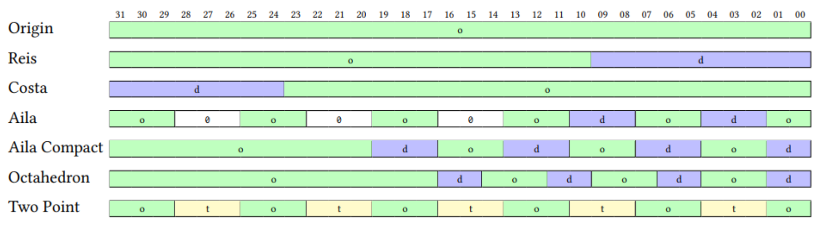
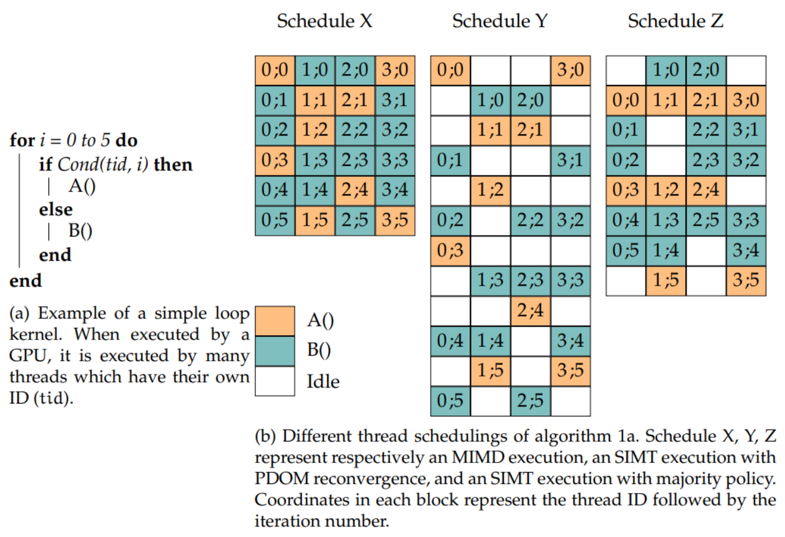


Figure 2.11: Different sorting key computation methods, where *o* is the origin, *d* the direction and *t* the termination point. Mixing different methods is achieved with bit interleaving.

A scheduler can also mitigate divergence in loops according to Blanleuil et al.[22]. Within a loop, there might be change to the control flow due to if statements. Within one iteration some threads will have to execute one code block or the other. A path table can be used to store for each iteration the path a thread has to take due to divergence. A scheduler can use the path table to either delay or advance threads to the next iteration if they execute the warp's current condition.



```
for i = 0 to 5 do
    if Cond(tid, i) then
        A()
    else
        B()
    end
end
```

(a) Example of a simple loop kernel. When executed by a GPU, it is executed by many threads which have their own ID (tid).

(b) Different thread schedulings of algorithm 1a. Schedule X, Y, Z represent respectively an MIMD execution, an SIMT execution with PDOM reconvergence, and an SIMT execution with majority policy. Coordinates in each block represent the thread ID followed by the iteration number.

Figure 2.12: Diagram from the paper [22]

From the figure above, we can see that the conventional scheduler Y stalls threads that have to execute the other condition. The suggested scheduler, on the other hand, advances threads to the next iteration, if they execute the same condition. It is clearly observed that the number of cycles are reduced.

Wald [23] showed that threads from different warps can be compacted to fewer warps. This can improve SIMD efficiency when there are a lot of partially filled warps that occur due to early thread termination. In practice this is done by executing a parallel prefix sum to find the number of active threads. He uses shared memory to store the path data of the active threads as well as a flag of indicating which paths are active. This minimizes the global memory accesses, however, it was reported that too much shared memory usage might drop GPU's occupancy and performance.

Mansson et al.[24] experimented on improving coherency of secondary rays by caching and sorting them into packets using various heuristics. Specifically, packets are sorted using a coherency measure. In other words rays that that will probably follow similar paths form a group in order to take fully advantage of packet traversal. The results show that this system can reduce the calls to the traversal unit and optimize the scene traversal.

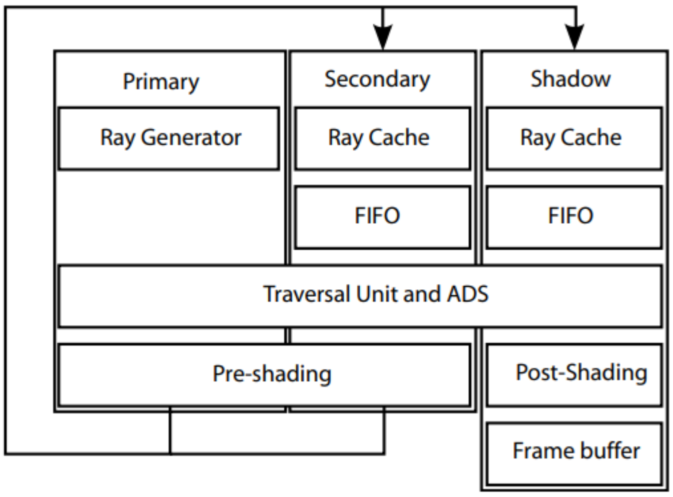
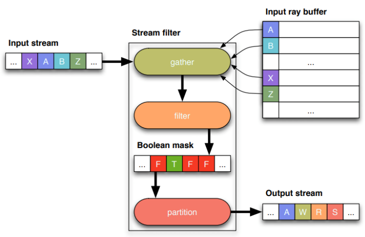
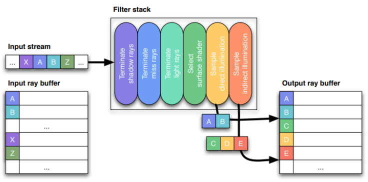


Figure 2.13: Overview of the design [24]: cached rays are being sorted in packets and stored in FIFO queues. Post-shading handles shadow rays by accumulating the final result depending on the shadow tests.

Similarly, Gribble et al. [25] group rays into streams via stream filtering methods. During the stream filtering process rays are being masked out using user-specific conditions. The output stream consists of rays that will follow the same code path, ensuring fully SIMD utilization. Their method can be used not only for scene traversal but for any other process during the ray tracing algorithm such as material evaluation, light sampling and more. Furthermore, filters with different masking out conditions can be aggregated resulting in eliminating inactive rays in a combination of different ray tracing stages.



(a) Stream filter takes as an input a ray stream and outputs a stream which has the inactive and the active rays (based on the masking condition) separated.



(b) Filters can be stacked producing a finer active-inactive ray output stream.

Figure 2.14: Stream filtering in ray tracing.

A new technological advancement on hardware named "shader execution reordering" (SER) by NVIDIA [26] manages to reorder threads in order to increase SIMD utilization. They manage to reduce memory and thread divergence on the fly without a significant overhead, since newer gpu architectures were designed with this feature. This innovative addition to the graphics APIs, compared to older automatic hidden thread reordering and scheduling, gives explicit control to the application and the developer on how thread reordering can occur.

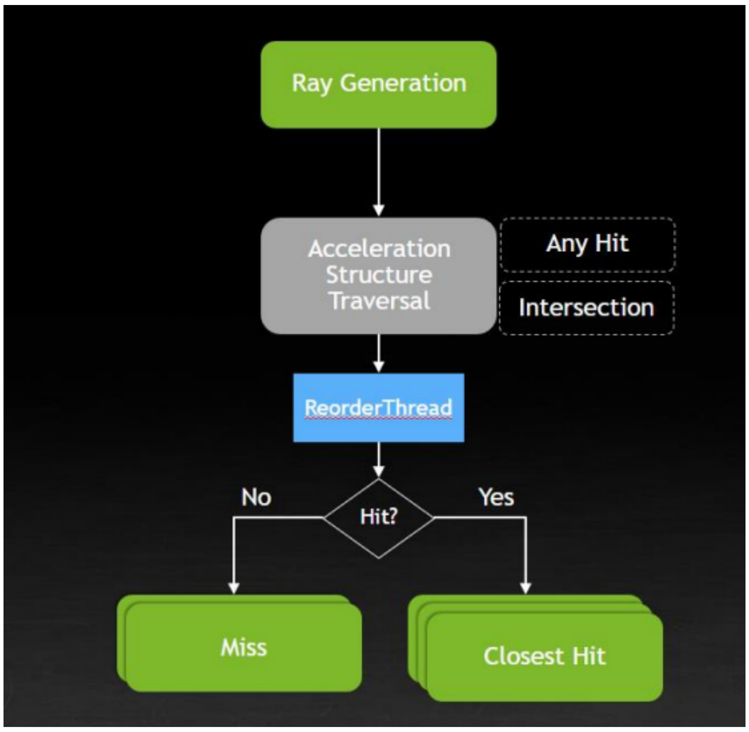


Figure 2.15: SER example of reordering based on the hit location of the primary rays.

3. Method

3.1 Overview

The main focus of this thesis is optimizing the path tracer algorithm. As mentioned in earlier sections, the goal is to make the path tracer more thread and memory coherent. That way, the hardware can be better utilized under the SIMD model and increase the stream processors' occupancy, warp efficiency and overall performance of the algorithm.

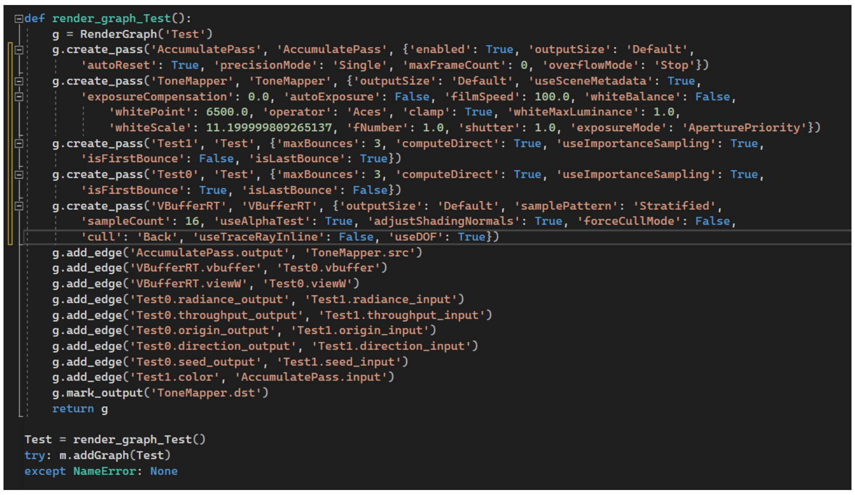
As mentioned in earlier work, ubershaders suffer from low performance due to maximizing the stream processors' register capacity and thread divergence. Hence, the first step includes splitting the path tracer into smaller steps and processes as also proposed by Laine and Karras [19] when dealing with a wavefront path tracer. In this case the focus will be entirely splitting the path tracer into single bounces.

Inspired by Meister et al.'s work [21] the next step is to sort the pixels/threads based on their spatial locality. This consists of including a sorting stage between the bounces in order to form warps and threadgroups that contain threads that will hit surfaces and objects which are in close proximity. Specifically, threads within a warp will traverse through the same nodes of the acceleration structure, thus increasing coherency, minimizing cache misses and thread stalling.

Threads require to calculate and store information that will be used as sorting keys during the sorting phase. The origin and an approximated termination point is a valid combination, since they have yielded promising results according to Meister [21]. In our case only the termination point will be used that will be stored in a spatial grid and updated temporally using the origin as a key. Therefore, the research question is focused around the evaluation of the sufficiency of solely utilizing the termination point as a sorting key in the reduction of thread divergence.

3.2 Software

The implementation is done in NVIDIA's framework Falcor [27] in C++. This framework enables fast experimentation and prototyping of rendering algorithms and techniques as it provides an abstraction of modern and complex graphics APIs such as Vulkan and DX12, hiding entirely the boilerplate code of the host. Furthermore, it includes a convenient style of implementing render passes with the usage of a render graph that combines and aggregates multiple stages of the rendering pipeline. Example passes and scripts that combine multiple passes such as a path tracer with temporal accumulation and the state-of-the-art denoisers are already available within the framework. Precisely, the *Mogwai* tool within *Falcor* consists of the main application that executes rendergraphs and renderpasses. Rendergraphs can be created within the application's graphical user interface or with Python scripting.



```python
def render_graph_Test():
    g = RenderGraph('Test')
    g.create_pass('AccumulatePass', 'AccumulatePass', {'enabled': True, 'outputSize': 'Default',
        'autoReset': True, 'precisionMode': 'Single', 'maxFrameCount': 0, 'overflowMode': 'Stop'})
    g.create_pass('ToneMapper', 'ToneMapper', {'outputSize': 'Default', 'useSceneMetadata': True,
        'exposureCompensation': 0.0, 'autoExposure': False, 'filmSpeed': 100.0, 'whiteBalance': False,
            'whitePoint': 6500.0, 'operator': 'Aces', 'clamp': True, 'whiteMaxLuminance': 1.0,
            'whiteScale': 11.199999809265137, 'fNumber': 1.0, 'shutter': 1.0, 'exposureMode': 'AperturePriority'})
    g.create_pass('Test1', 'Test', {'maxBounces': 3, 'computeDirect': True, 'useImportanceSampling': True,
        'isFirstBounce': False, 'isLastBounce': True})
    g.create_pass('Test0', 'Test', {'maxBounces': 3, 'computeDirect': True, 'useImportanceSampling': True,
        'isFirstBounce': True, 'isLastBounce': False})
    g.create_pass('VBufferRT', 'VBufferRT', {'outputSize': 'Default', 'samplePattern': 'Stratified',
        'sampleCount': 16, 'useAlphaTest': True, 'adjustShadingNormals': True, 'forceCullMode': False,
        'cull': 'Back', 'useTraceRayInline': False, 'useDOF': True})
    g.add_edge('AccumulatePass.output', 'ToneMapper.src')
    g.add_edge('VBufferRT.vbuffer', 'Test0.vbuffer')
    g.add_edge('VBufferRT.viewW', 'Test0.viewW')
    g.add_edge('Test0.radiance_output', 'Test1.radiance_input')
    g.add_edge('Test0.throughput_output', 'Test1.throughput_input')
    g.add_edge('Test0.origin_output', 'Test1.origin_input')
    g.add_edge('Test0.direction_output', 'Test1.direction_input')
    g.add_edge('Test0.seed_output', 'Test1.seed_input')
    g.add_edge('Test1.color', 'AccumulatePass.input')
    g.mark_output('ToneMapper.dst')
    return g

Test = render_graph_Test()
try: m.addGraph(Test)
except NameError: None
```

Figure 3.1: Example of a render graph that executes a 2-bounce path tracer in Python.

For performance evaluation and shader profiling NVIDIA Nsight Graphics [28] is used. It is an industry standard for debugging and optimizing rendering applications. Various performance markers are included providing significant insight on what impacts performance and measures can be taken to write more efficient shaders.

3.3 Path tracer

As mentioned in the above section, Falcor has already a minimal path tracer render graph implemented. Its main components are:

1. A Geometry pass, which is either ray traced or rasterized and stores the required information of the primary rays such as position, normals, and material ids. The rasterized variant is used as it adequately fulfills the objectives of our study.
2. A path tracer, which is multi-bounced and includes simple brdf importance sampling and uniform light sampling for next event estimation.
3. A temporal accumulation stage, which interpolates the current frame and previous frames using the exponential moving average formula as being used in popular TAA algorithms [29]. To maintain simplicity, pixels are not being reprojected and past frames are discarded in case there is camera movement, scene modifications and pipeline state alterations (e.g. maximum number of path length).
4. A tonemapping stage to remap from high to low dynamic range.

While ray tracing through the scene, it is vital to store and preserve the state of the path from each successive bounce to the subsequent one. This requirement is addressed by introducing a user defined custom data structure, also referred to as *payload* in the literature. In order to isolate each bounce as a separate pass it is essential to store and transmit that data structure between the bounces. To achieve that 2D textures are being used to store and load from the payload.

Specifically, the textures used are described below:

1. A RGBA float position texture that stores the xyz position in the first 3 channels and a binary value in the alpha channel that indicates whether the path is terminated or not
2. A RGB float direction texture that stores the sampled direction that

will be used to trace the next hit

3. A RGB float throughput texture that stores the current throughput of the path. The initial value starts with 1 and it is reduced by the pdf weight of the samples and the brdf evaluation.
4. A RGB float radiance texture that basically stores the current final color if that bounce would terminate. It is being calculated as $L_o = thp \cdot L_e + thp \cdot L_i$, where thp is the current throughput L_e the emission if the current surface is emissive and L_i the current sampled light's contribution.
5. A RGBA unsigned integer seed texture that is required to store the state of the random number generator for each pixel.

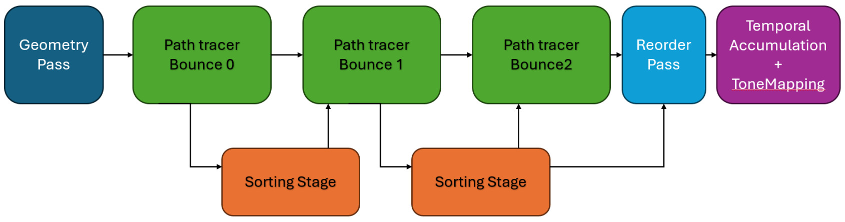


Figure 3.2: Overview of the rendering pipeline.

Apparently, the texture writes and fetches will introduce an overhead, since global memory is significantly slower than local registers. In the upcoming results 4 and discussion 4.4 sections will be deduced whether increasing warp occupancy will exceed that added cost.

3.4 Sorting Keys

Drawing inspiration from Meister's work [21] it is evident that spatial information is a necessity when computing and creating sorting keys. For each pixel, at every bounce, keys are computed and then stored into a global buffer. This buffer will be sorted in ascending order utilizing a parallel sorting algorithm suitable for the gpu.

Although, there are many options for sorting key selection, this thesis primary focus is employing an estimated termination point of the rays as the main source of spatial information. This is the main deviation point from Meister's method, in which both the origin and termination point of the path was incorporated to retrieve the sorting keys. Using only the termination point might be a crucial improvement, since all the bits are used, enabling a finer precision on the spatial grid. Furthermore, the position's information is already incorporated when updating the lengths as explained in section 3.4.2.

The first critical step is converting 3D spatial information into a scalar unsigned integer value. This is effectively achieved by constructing an imaginary 3D grid with a predefined resolution. As a consequence, retrieving a scalar integer key is a straightforward task. This is accomplished by initially identifying the voxel cell to which this point belongs to and later applying Morton code to transform the 3D index to its scalar equivalent.

For paths that are either early terminated or miss and do not hit any geometry the same scalar index is assigned to, which in practice corresponds to the maximum possible value based on the 3D grid's resolution. This is necessary to ensure that missed/terminated paths are grouped into the same warps, since they will no longe generate new paths and trace through the acceleration structure. The performance impact just by grouping the terminated path could already be significant.

Algorithm 1 Find cell indices

```
1: procedure FindCellIndices($point$)
2:     $cellSize \leftarrow gSceneBounds/gResolution$
3:     $cellIndices \leftarrow min3(gResolution - 1, floor(point/cellSize))$
4:     return $cellIndices$
5: end procedure
```

Morton code also known as the Z-order curve is a popular method used in mapping multidimensional data to one dimension [30], while ensuring spatial locality. It works by interleaving the bits of 3D index of a point into 1D array index.

Algorithm 2 Morton Code

```
1: procedure MORTON($x, y, z$)
2:     $morton \leftarrow 0$
3:     for $i \leftarrow 0, 32$ do
4:         $morton \leftarrow morton \mid (x \,\&\, 1 \ll i) \ll 2 \times i \mid (y \,\&\, 1 \ll i) \ll (2 \times i + 1) \mid (z \,\&\, 1 \ll i) \ll (2 \times i + 2)$
5:     end for
6:     return $morton$
7: end procedure
```

The sparsity of the voxels is fundamentally determined by the grid. A finer resolution enhances the precision, giving a profound representation of the points by their corresponding indices. However, due to the curse of dimensionality, the memory requirements increase exponentially. To illustrate this, halving the cells' size yields an 8-fold increase in VRAM. Furthermore, it is crucial to mention that a big grid with a lot of sparse memory access will inevitably yield in lower performance, as cache efficiency drops and therefore the cost of memory fetches increases. Finding the optimal balance between grid resolution and memory usage is key in the context of path tracing algorithms.

3.4.1 Fixed Length

It is apparent that the termination point cannot be known in advance. The main concept is to guess where the rays of the next bounce will intersect in order to sort them based on that information, in hope of achieving coherent ray traversal. Before the actual intersection tests only the origin and the sampled direction are known. However, a termination point can be approximated for a fixed length value using the ray equation.

$$P = O + t_{fixed} \cdot D \tag{3.1}$$

Initially fixed length was used as a proof of concept to assess whether this modification to the path tracer algorithm yields a significant performance improvement. In our tests, the chosen fixed length derives from the half of the maximum axis of the scene's extents. However, Meister et al [21] suggested a fixed length of 0.25 of the scene's extent when using both the origin and termination point.

$$t_{fixed} = 0.5 \cdot max(max(extent.x, extent.y), extent.z) \tag{3.2}$$

3.4.2 Adaptive Length

While employing a fixed termination length could provide a notable speedup, it remains a cheap approximation on where the rays will intersect in the next bounce. A more sophisticated alternative involves dynamically updating ray lengths based on the actual hit positions of the previous bounces and propagate that information along the frames. This is conceptually similar to temporal accumulation. It is required to create a separate data structure that caches the ray lengths and is being updated and accessed between subsequent bounces and reused in the next frames. Analogous to the sorting buffer indexed by the termination points, this new data structure incorporates both ray origins and an additional parameter derived from their directions.

Similar for the termination point the origin is used to calculate the cell indices and then convert to 1D array index. However, we deviate from using Morton code, because our focus is not on preserving proximity between the values. Additionally, due to memory and performance limitations, using a fine-grained grid resolution is not achievable. We leverage also the ray direction to calculate the octant to which the ray belongs. By dividing the 3D grid into eight octants, we can represent efficiently the orientation of the rays within each voxel. Various surfaces within the same voxel cell could potentially spawn rays with different directions traversing through different nodes within the acceleration structure. This is also evident surfaces that have wide reflection lobe due to high roughness values. Consequently, adding the direction information via octants could yield better representation than increasing the spatial grid's resolution.

For each bounce a sorting key needs to be calculated based on the length that will be loaded from the data structure. This process applies to all bounces except for the initial camera rays. The primary rays lacking any past length information, do not require to update the lengths. At the beginning, the length buffer is being initialized with the fixed length value. Different strategies and methods regarding length approximation were developed and experimented. They will be described below along with their performance in the results section.

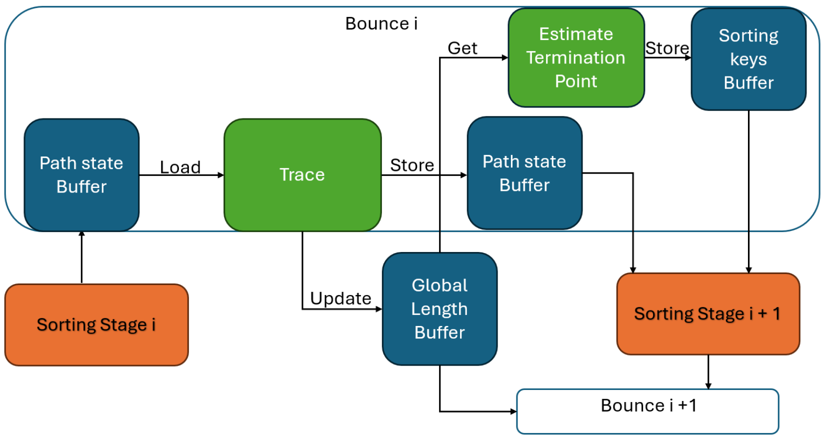


Figure 3.3: Overview of one bounce. Blue boxes involve data structures, orange are compute shaders and green are processes within the path trace stage

3.4.2.1 Average

The primary and most straightforward method involves calculating the average length for each octant per voxel. This calculation is being executed by storing two key pieces of information: the total number of samples and the cumulative length. Therefore, the approximation of a length value, which is used to estimate a termination point, is then simply a matter of performing a division operation. The main advantage of this method lies in its simplicity, as it requires only as single float and one integer per element. However, a notable limitation is its sensitivity to outliers.

3.4.2.2 Gaussian Distribution

As mentioned in section 3.4.2 the spatial grid cannot represent 100% the scene's complexity. Specifically, due the path tracer's stochastic nature, rays that belong to the same voxel and octant would deviate enough to hit entirely different nodes producing a high variance in observed length values.

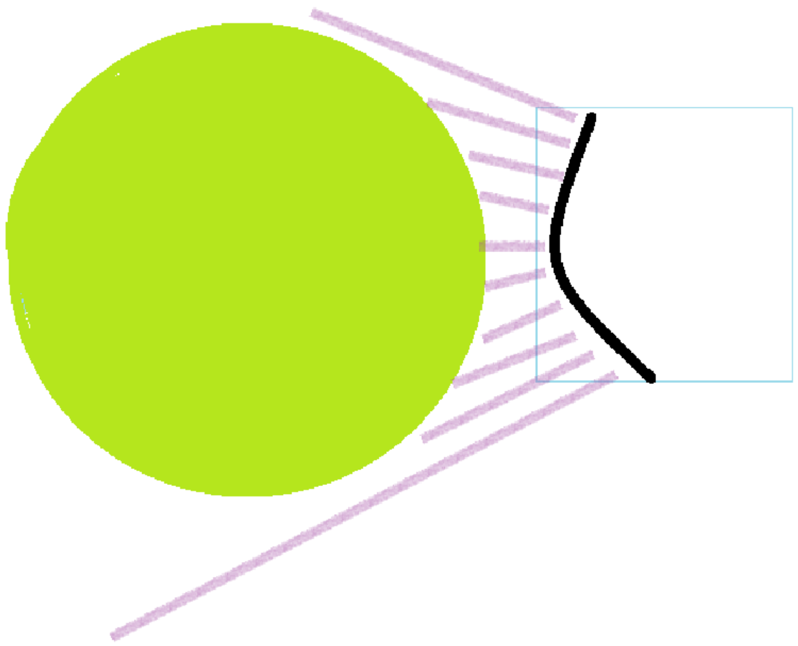


Figure 3.4: Example of outlier rays, where the blue box is a voxel cell, the green circle an object in the scene, the black curve the surface within the voxel and the purple lines the suggested sampled directions.

In image 3.4, we can observe that most rays will hit the sphere, however, they are a few rays that will not hit the sphere and the actual intersection length could be significantly bigger compared to the others. This would potentially produce a inaccurate length approximation by using just the average of the observed length values.

This encourages us to try sampling length values from Gaussian distributions. Storing the number of samples, the sum of all samples and the sum of the squared samples gives us the capability of calculating a Gaussian distribution with an estimated mean and variance. This method has potentially a higher cost, as it requires more expensive calculations compared to using just the average value. Furthermore, one extra float value is required, making the amount of required memory 1.5 times more compared to the previous method. However, extra conditions and parameters can be used to tackle the outlier issue. Specifically, after an adequate number of samples a new sample can be classified as an outlier and thus discarded if the variance exceeds a user-defined threshold. Also, the distribution itself could be reset by discarding all the past samples in case it reaches a user-defined high variance value, making it more sensitive to new data. This, obviously, introduces a layer of hyperparameter tuning and finding the best values to get the optimal performance might not be an easy task due to scene dependency.

3.4.3 Camera Clamping Grid

When calculating the termination point there is no guarantee that the estimated position will not exceed the scene bounds. To ensure that the estimated point will always be within the scene's grid, every estimated position is clamped to the scene bounds and then remapped to positive range, because the indices need to be a non negative value. A notable issue that may arise is bigger grid cell size. This could potentially result in a substantial variance in length values within the same cell. Consequently, parts of the scene that contain complicated geometry would be misrepresented as more different surfaces lie within the same cells producing variable length values as also observed in figure 3.4 . One simple approach is to have different grid resolution among the three dimensions. This means that the biggest axis will have more cells and make the grid finer. In our example scene the bounds in the x and z axis are more than 150, whereas in the y is 50. Increasing the x and z resolution of the grid and decreasing the y one is valid decision.

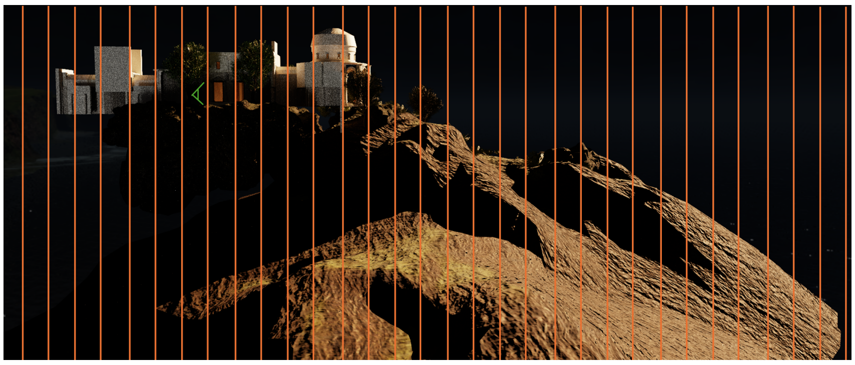


Figure 3.5: Uniform grid along one axis

We took one step further by introducing a camera clamped grid based on the camera's position. Specifically, if the estimated termination points surpass a user-defined distance from the viewer, they are clamped. The camera's position is utilized in this context, as it is preferable to have a detailed grid around the surfaces proximate to the viewer. This is due to the higher likelihood of most rays intersecting these nearby surfaces, since most rays might not escape the area around the camera due to surrounding walls and geometry.

$$minPos = max3(gMinPoint, cameraPos - float3(distance)) \tag{3.3}$$

$$maxPos = min3(gMaxPoint, cameraPos + float3(distance)) \tag{3.4}$$

Where $gMaxPoint$ and $gMinPoint$ are the minimum and maximum points in the scene. Any point is being clamped to $minPos$ and $maxPos$ and cell's size also takes the updated bounds into consideration during the calculation of the cell indices.

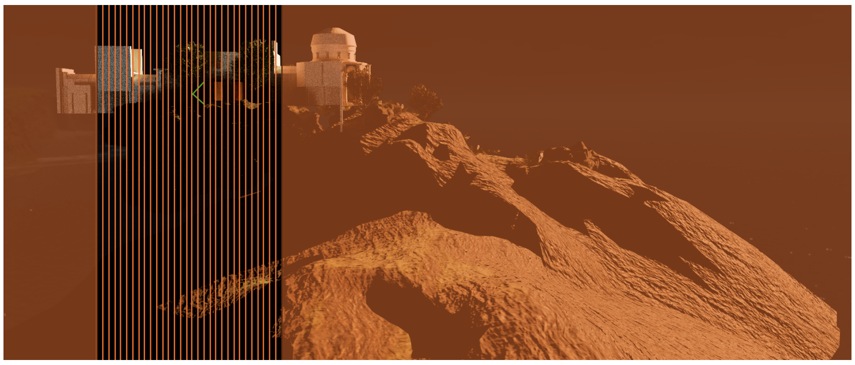


Figure 3.6: Positions' found in the orange areas are clamped and assigned to the indices that either correspond to *minPos* or *maxPos*.

3.5 Sorting Stage

In the previous section the necessary data structures were described as well as the sorting key calculation methods that will populate the buffers to be sorted. The stages between the subsequent bounces include a compute pass that is responsible for sorting the pixels according to the Morton code produced values. Choosing the appropriate sorting algorithm is key as the number of sorting elements is equal to the number of pixels, since the pixels represents the rays in a bounce that need to be sorted based on spatial similarity. Hence, the cost of the sorting algorithm is not insignificant

Researching the state-of-the-art sorting algorithms suitable for the GPU, we came across Adinets'and Merrill's work [31] that utilized a novel approach of a least significant digit radix sort called *OneSweep*. Generally, radix sort processes numbers digit by digit, instead of comparing the actual numbers. The algorithm distributes the numbers into buckets based on each digit's value. By continuously sorting digits from least to most significant it achieves a final sorted order.

The *OneSweep* algorithm consists of 3 different kernel stages:

1. An upfront histogram kernel. This compute pass calculates a global digit histogram for all digit places. Atomic additions and shared memory is used to maximize performance by minimizing register and shared memory overhead. The histograms are private block-wide resulting in less shared memory requirement. Hence, larger histograms are possi-

ble, yielding an increase in radix digit size and a decrease in binning iterations.

2. An exclusive sum kernel. This stage, basically, computes the prefix sum across each of the digit histograms.
3. A chained scan digit-binning kernel. This kernel is a variation of the chained scan with decoupled look-back technique [32]. A grid of thread blocks, where each thread block is one tile of the input. The elements of each tile are evenly distributed across the warps of the thread block. Threads then perform warp-wide key ranking using the warp-level multi-split technique [33]

For the actual implementation of the compute kernels, source code from this repository [34] was used. However, Falcor is using the Slang shading and compiling language instead of pure HLSL. This introduced a lot of errors regarding the intrinsic function that were used in the original implementation. Specifically, after a thorough technical research, it was revealed that some of the wave intrinsic functions are not currently supported by Slang. Furthermore, attempt was made to modify it to make it Slang appropriate, however, undefined behavior was observed.

The sorting algorithm is a fundamental important stage in our pipeline, since it can potentially be a major bottleneck. However, the main research focus of this thesis is the thread reordering in the context of the path tracer algorithm and not the actual sorting algorithm. Consequently, a simple parallel bitonic sort algorithm is used instead.

The basic concept of the bitonic sort algorithm is that it creates ascending and descending motonic series of the input data and then merges them. It is a decent general purpose sorting algorithm, since it is able to sort numbers of various types not just unsigned integers like radix sort. This could potentially enable more versatility when it comes to the type of sorting keys. However, multiple sequential compute passes are required making it not the fastest sorting algorithm in the field [35].

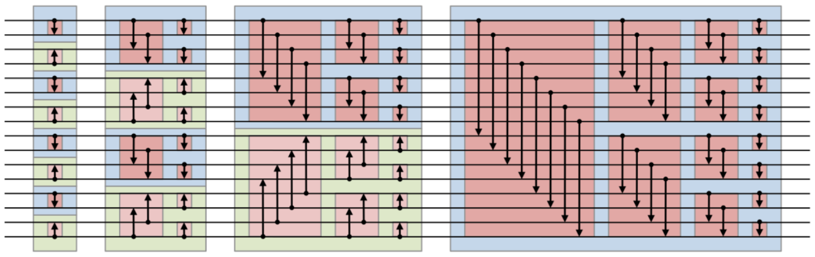


Figure 3.7: Example of a bitonic sorting network of 16 input [36]

When sorting the pixels based on their corresponding sorting keys, it is evident that the pixel spatial proximity is lost. Originally, the simplest approach would be to create an index buffer that stores the indices of the pixels. This method dictates the necessity of updating the index buffer while swaping the sorting elements. When reading from the path state data structures, during the next bounce, the index buffer is used to index the correct memory from which the data will be loaded. The data structures were created and initialized with the original order of the pixels, leading to significant less coherent memory access and poor cache utilization. The path tracer algorithm's performance was tested with that approach and it was observed that it had a negative impact, rendering the whole method significantly slow.

The solution is to swap the path's data buffers as well to preserve locality. This is accomplished by additionally passing these data buffers to the sorting stage and propagating them to the next bounce. Furthermore, it is required to introduce a single additional compute stage just after the final bounce and before any screen-space post process effect. The sole purpose of it, is to restore the order of the pixels back to the original position, otherwise the pixels of the final image would be scattered in entirely different positions, making the final rendering incomprehensible.

4. Results

4.1 Experiment Description

The different methods and heuristics are tested on Unreal Engine's Sun Temple scene [37]. It contains PBR textures, emissive materials and complicated enough meshes for the thesis purpose. The experiments were conducted on a NVIDIA RTX3070 Laptop GPU with 8GB Vram on a resolution of 2048x1024.

The resolution is unusually in the power of two so that bitonic sort can be easily utilized. With some tweaks the bitonic sort could also sort correctly if the number of pixels was not a power of two, however, it was not desired to spend too much time on the sorting algorithm's implementation.

Using the GPU Trace Profiler a sophisticated tool in NVIDIA NSight Graphics, 3 snapshots are taken for each individual method. Specifically, each snapshot involves a specific camera position. Each one of them represents a different scenario depending on the divergence of the rays when path tracing.

The first example shows a narrow alley in which rays will always hit something and will mostly not miss, generating the maximum bounces, because there are no windows or openings. However, it mostly contains walls and floor.

Secondly, a snapshot is taken in the middle of the scene. There are windows, but they are not visible on the camera. The surrounding geometry is more complicated and the rays can spread in a more chaotic fashion. Basically, in that scenario, rays could diverge the most and the path tracer could potentially escape and terminate earlier.

Finally, the camera sees mostly geometry but also the openings. This means that there will be pixels that will only spawn primary rays and will immediately terminate.

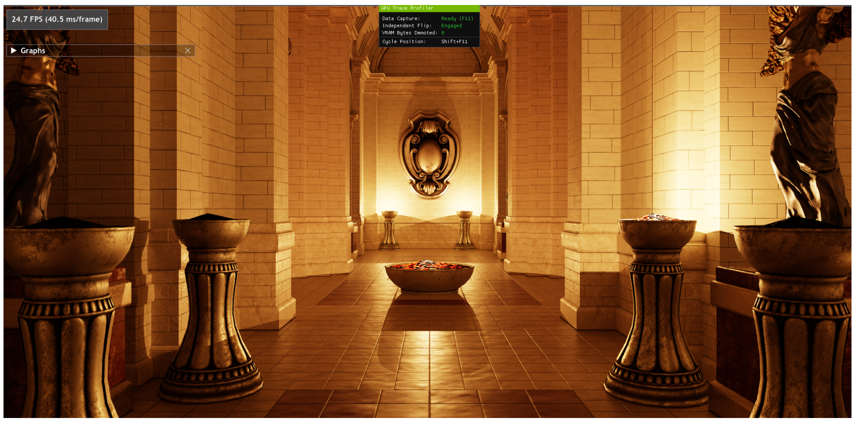

24.7 FPS (40.5 ms/frame)
Graphs
GPU Trace Profiler
Data Capture: Ready (F11)
Independent Flip: Engaged
VRAM Bytes Demoted: 0
Cycle Position: Shift+F11

Figure 4.1: Example with mostly simple geometry and no openings and windows

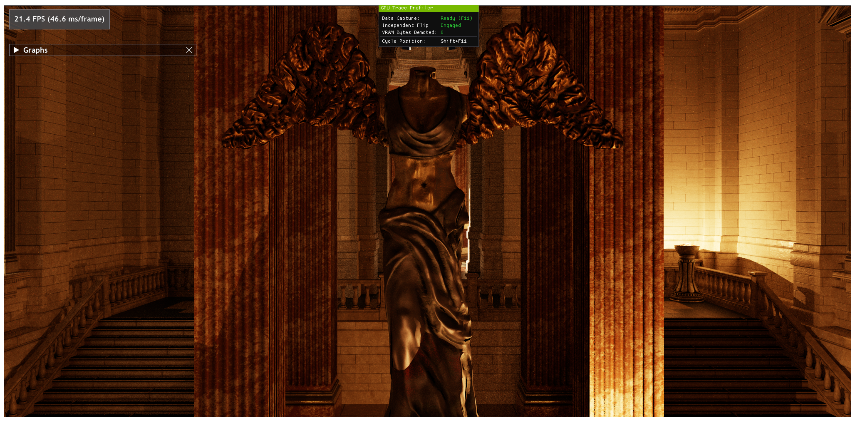

21.4 FPS (46.6 ms/frame)
Graphs
GPU Trace Profiler
Data Capture: Ready (F11)
Independent Flip: Engaged
VRAM Bytes Demoted: 0
Cycle Position: Shift+F11

Figure 4.2: Example with complicated geometry and with windows not directly viewed from the camera

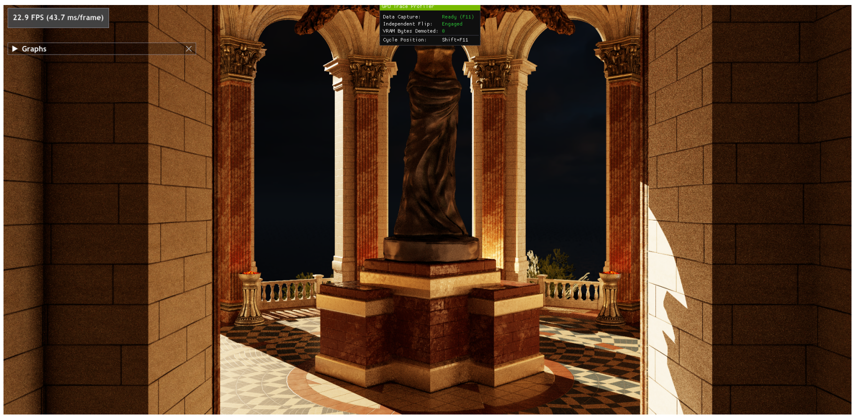

22.9 FPS (43.7 ms/frame)
Graphs
GPU Trace Profiler
Data Capture: Ready (F11)
Independent Flip: Engaged
VRAM Bytes Demoted: 0
Cycle Position: Shift+F11

Figure 4.3: Example in which some primary rays will terminate.

4.2 Ground Truth

Below, the results from Falcor's minimal path tracer with 4 bounces are presented. Later they will be compared with the fixed length reordering solution to evaluate our basic hypothesis.

Metric	Example1	Example2	Example3
Path tracer (msec)	38.10	42.49	42.66
Threads/Warp	9.8	8.8	8.7
L1TEX Hit Rate (%)	63.9	62.8	63.0
L2 Hit Rate (%)	64.4	64.1	60.9
L2 from L1 (%)	64.8	64.4	61.1
L1 Throughput (%)	38.3	34.3	29.2
L2 Throughput (%)	38.9	36.8	33.0
RTCORE Throughput (%)	42.1	38.3	35.8
SM Throughput (%)	31.3	28.6	25.8
VRAM Throughput (%)	63.0	57.4	52.8

Table 4.1: Performance Metrics for the Minimal PathTracer

The duration metric involves only the path tracer stage and not the others like temporal accumulation and tonemapping. The threads per warp are the average active threads per instruction within a warp. L1TEX and L2 hit rate are the percentage of successful cache hits from level 1 and level 2 respectively. L2 from L1TEX metric is the hit rate of data missed in L1TEX and successfully found in L2, in contrast to L2 hit rate, which involves all level 2 accesses regardless of whether they were missed in L1TEX. The throughput metrics state the unit's utilization. For example the RTCORE involves the utilization of the ray tracing nodes, the VRAM the accesses to the global memory and the SM is the stream processors utilization, which on modern hardware 2 warps can reside in a single stream processor. Within the NVIDIA's profiler, there is an exhaustive list of other metrics that are more specific. In the scope of this thesis, the metrics that are considered crucial to the research topic will be presented to keep the results discussion precise and as thorough as necessary.

A naive assumption would be that the third example would ideally cost less, since many pixels will terminate just on their primary rays and will not go through the ray tracing process. However, the above results demonstrate the opposite, proving that thread divergence within warps is indeed a critical issue. Consequently, it also affects other metrics, like a cascade effect. Specifically, poorer cache utilization as well as lower throughput values among various units are noticed. Although, in the first example, all rays will never miss and will trace 4 bounces, the path tracer performs better, because the geometry will bounce off the rays in a similar manner, thus less thread divergence is observed.

Below the table showcases the path tracer performance when it is reordered based on the fixed length solution.

Metric	Example1	Example2	Example3
Path tracer (msec)	25.48	26.20	26.01
Threads/Warp	20.74	19.36	18.42
L1TEX Hit Rate (%)	62.38	65.38	65.12
L2 Hit Rate (%)	67.02	65.50	62.82
L2 from L1 (%)	67.4	65.84	62.88
L1 Throughput (%)	36.42	33.20	28.18
L2 Throughput (%)	50.58	45.62	41.82
RTCORE Throughput (%)	47.48	45.94	42.18
SM Throughput (%)	23.38	24.36	22.08
VRAM Throughput (%)	69.14	59.02	59.22

Table 4.2: Performance Metrics for the Fixed Length Solution

The duration is much lower compared to the original algorithm, while the active threads are significant higher. This confirms our initial hypothesis whether thread reordering can reduce thread divergence and hence increase the path tracer's performance. Furthermore, it is noteworthy to mention that the other metrics are also positively influenced. The L1 hit ratio is bit less in the first example while slighly higher in the other two. The L2 ratio is slightly higher in all examples. The L1TEX throughput is a bit less while the L2 throughput significant higher. Similar higher values to L2 throughput, are observed at RTCORE throughput, which could indicate the better performance of the algorithm. The VRAM throughput is also logically higher, since extra buffers to store the path state are required between the bounces.

It is crucial to mention the cost of the sorting algorithm as well. The bitonic sort of 2048x1024 elements approximately takes from 30 to 35 msec. For every bounce one sorting stage is required so in total this amounts on average to 130 msec. The cost of the reorder pass costs roughly 0.8 msec on average, so it can be ignored. The high cost of the sorting algorithm is already noticeable, making it the main bottleneck. As mentioned in section 3.5, a faster sorting algorithm is suggested called *Onesweep*. Interested readers can study the source code as well as how it compares with other high performance sorting algorithms [38].

4.3 Comparison

The Meister et al's method [21] is also tested on the specific framework. Out of all sorting keys, we used the one that combines the origin and direction of the ray in question, since it achieved the highest performance according to their results. The lengths are cached using the average formula.

For the Gaussian distribution method, the distribution resets when the variance exceeds 20. After some hyperparameter tuning, it was concluded that this threshold achieves the best results.

Below, comparison bar graphs are presented for all the important metrics by taking the average of the three different examples.

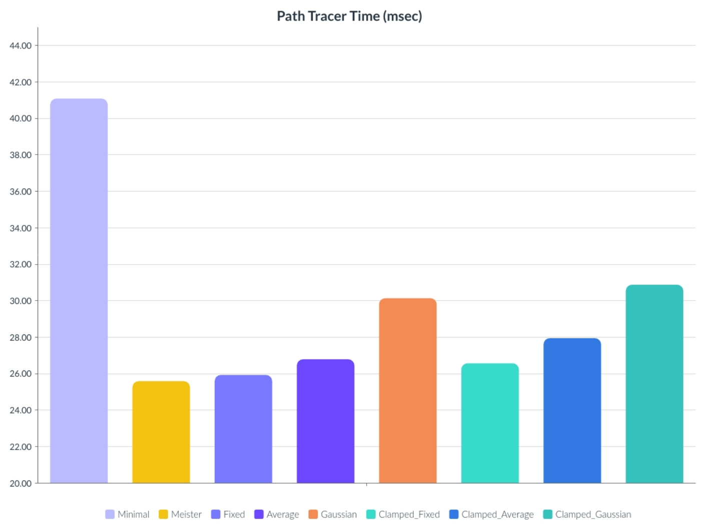


Figure 4.4: Path tracer time comparison bar graph

From the graph 4.4, it is observed that the overall thread reordering technique provides a significant speedup to the path tracing algorithm. The faster path tracer is achieved by the original Meister et al's method [21], which executes the path tracer in just 25.32 msec compared to the ground truth which takes 41.08 msec. The fixed length solution comes next with 25.90 msec. The slowest one is the Gaussian distribution with the camera clamped grid with 30.87 msec.

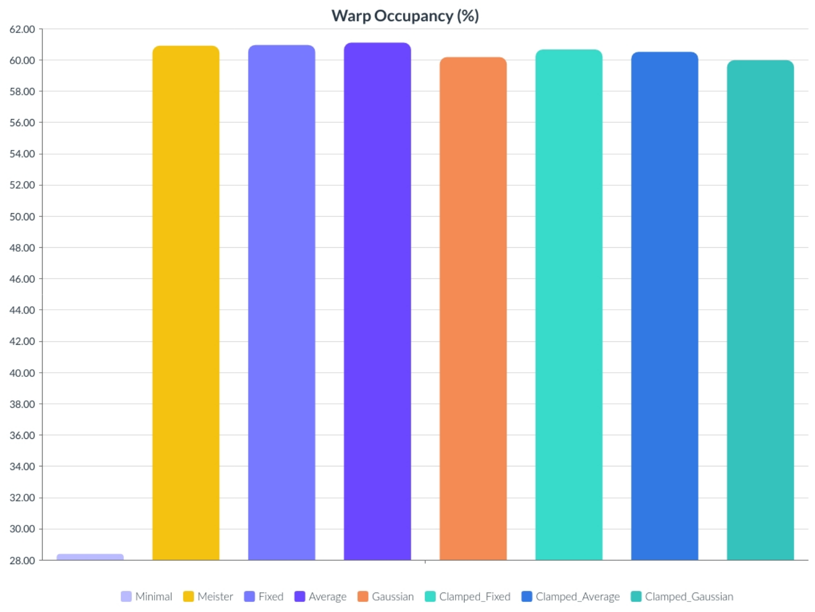


Figure 4.5: Active threads per wap comparison bar graph

The superiority of the thread reordering method is also evident when examining the warp's occupancy. All methods achieve close to 60% or more warp coherence, compared to the ground truth, which achieves half of it, only 28.29%.

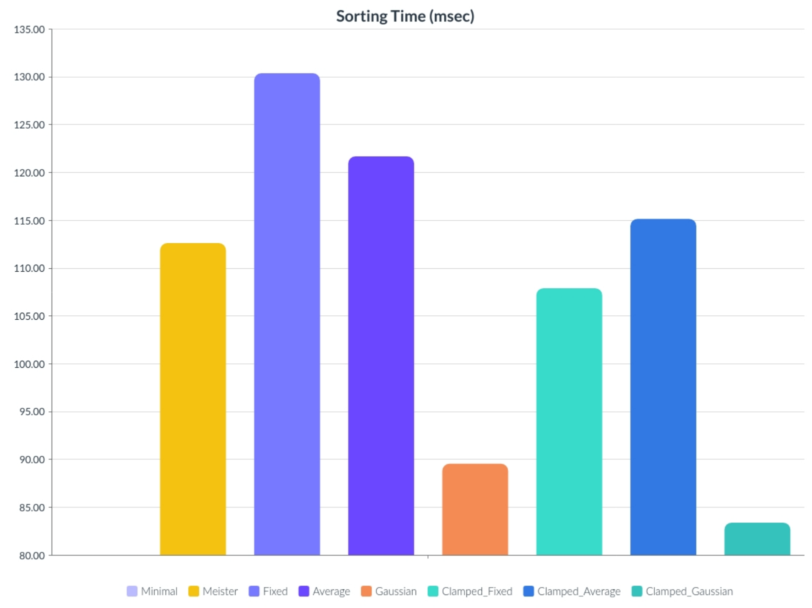


Figure 4.6: Sorting stage duration comparison bar graph

The sorting stage is without doubt the bottleneck for all the implemented methods. However, a significant reduction is achieved when using the Gaussian distribution method on camera clamped grid. Specifically, it only costs 82.95 msec compared to Meister's 112.47 msec and the fixed length solution's 130.32 msec, which is the highest observed from all the other methods.

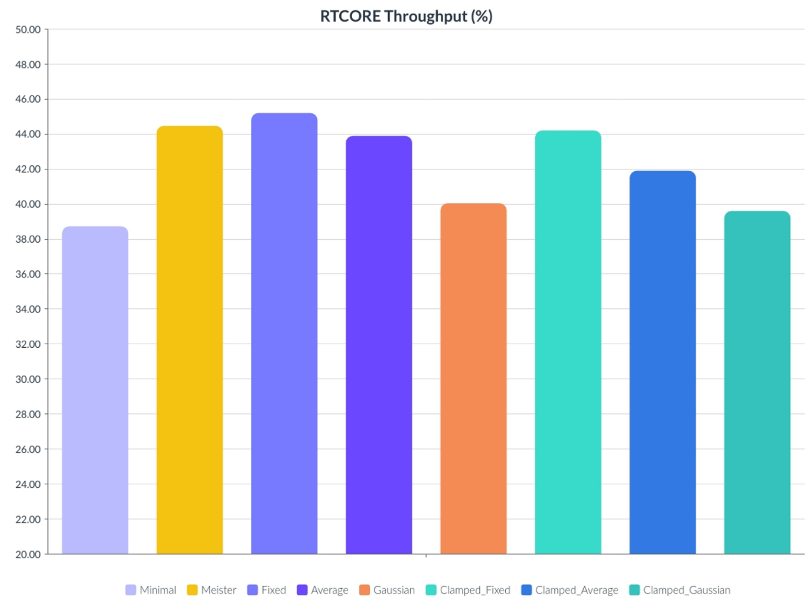


Figure 4.7: RTCORE throughput comparison bar graph

The RTCORE throughput is increased in all methods compared to the ground truth. This means that the RT cores are more efficiently utilized by the thread reordering methods. The fixed length solution achieves the highest throughput of 45.19%. From all ray-reordering methods the Gaussian distribution solution achieves the worse of 40.05%.

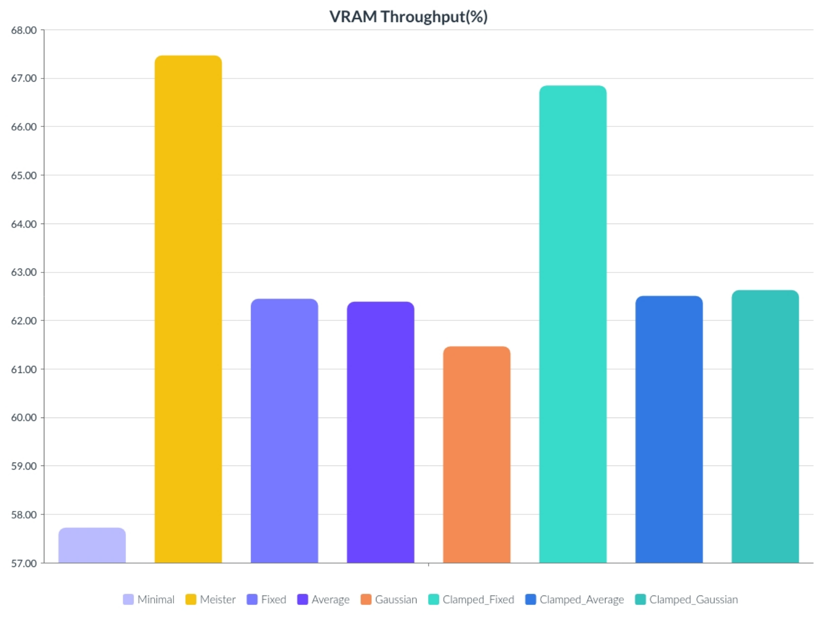


Figure 4.8: VRAM throughput comparison bar graph

All methods exhibit higher VRAM throughput compared to the ground truth. This is explained by the extra buffers that are required to store and load the current bounce's path state. Meister et al's solution and the fixed length method with the camera clamped grid have the highest VRAM throughput. In the graphs lower cache utilization is observed which could indicate the cause of the higher VRAM throughput. Lower cache utilization means that requested data are not found in cache so they need to be fetched from global memory, thus higher VRAM utilization.

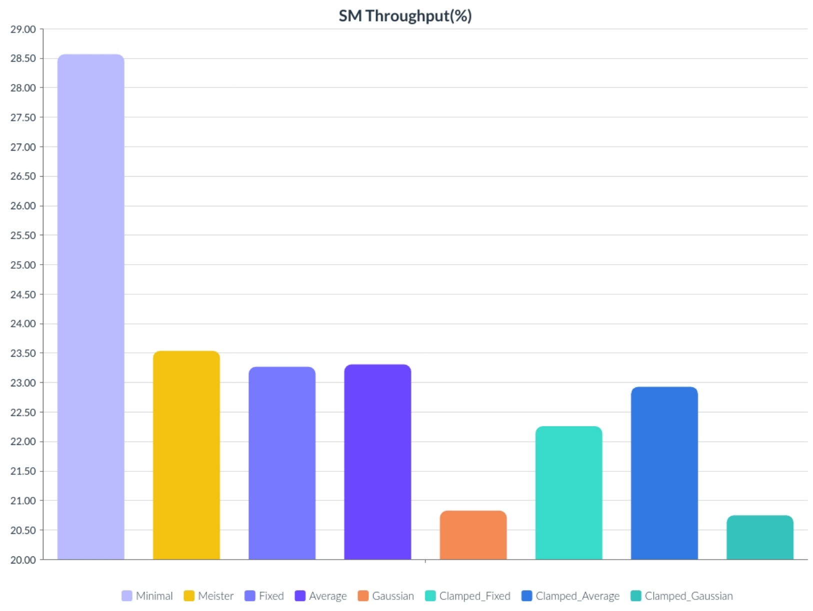


Figure 4.9: SM throughput comparison bar graph

It appears that the original ground truth path tracer achieves higher throughput of the stream multiprocessor. However, this metric by itself cannot provide thorough reason of why the ray-reordering methods achieve less, since this throughput covers the overall utilization of the whole stream multiprocessor.

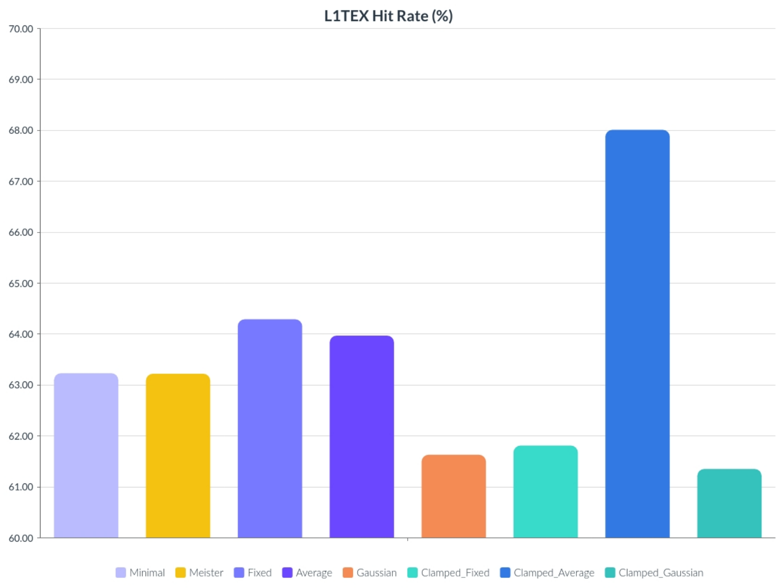


Figure 4.10: L1TEX hit ratio comparison bar graph

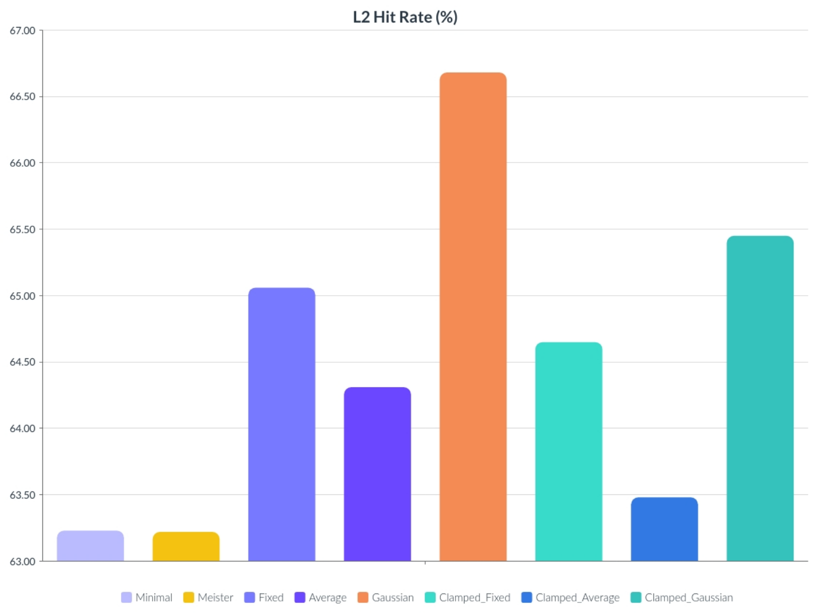


Figure 4.11: L2 hit ratio comparison bar graph

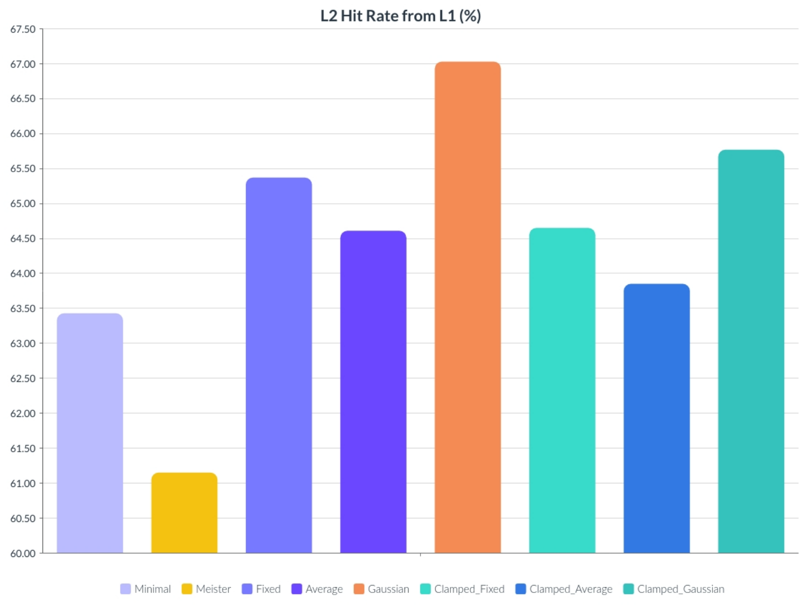


Figure 4.12: L2 from L1 hit ratio comparison bar graph

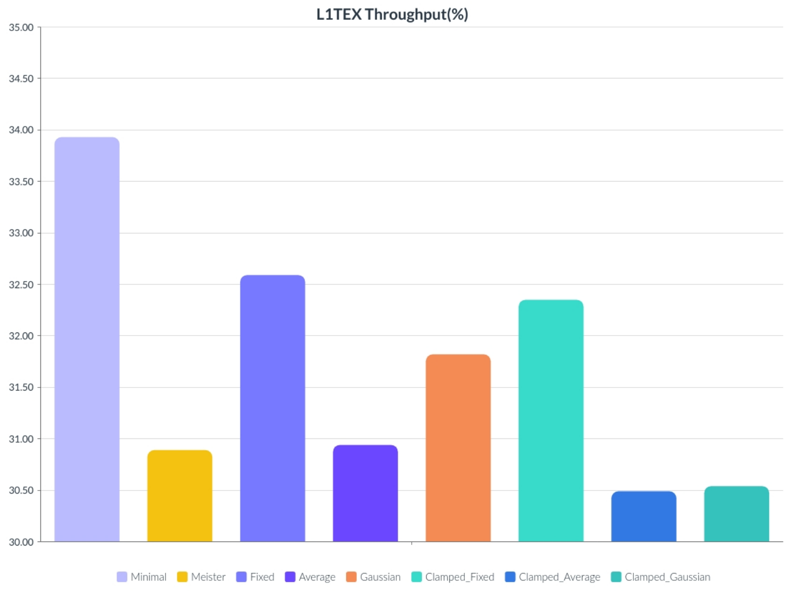


Figure 4.13: L1TEX throughput comparison bar graph

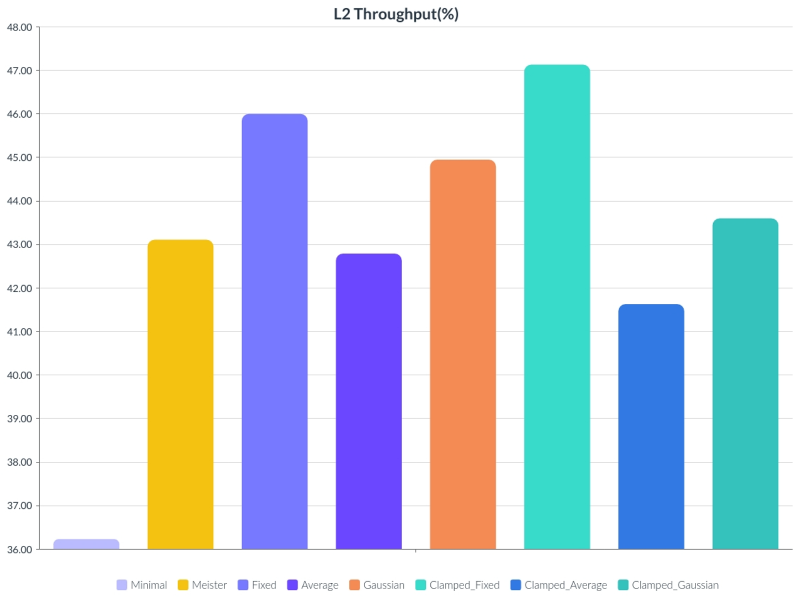


Figure 4.14: L2 throughput comparison bar graph

We do not observe significant difference in the cache metrics among the different methods. Specifically, the average formula on a camera clamped grid achieves the highest L1TEX ratio of 67.94 %, while the corresponding Gaussian method achieves the lowest of 61.38 %, which is even lower than the ground truth. However, the Gaussian method on a standard grid has

66.63 % hit ratio on L2, which is the highest. In the L2 from L1 hit ratio case, Meister et al's method has the lowest value, which is almost 6% less compared to the highest, also achieved by the Gaussian method. The ground truth path tracer has the highest L1TEX throughput and the lowest L2 throughput. This could be explained by the minimal usage of buffers that the other methods require.

4.4 Evaluation

It can be concluded that the occupancy has the most significant role, when considering the path tracer's performance. The duration difference of all the methods when compared to the original uninterrupted path tracer is evident. When comparing the different methods with each other, it is deducted that the RTCORE throughput also influences the path tracer's performance. For example, by examining the graph 4.4 and 4.7, we can clearly see that methods that have higher utilization of the RT cores have also less path tracing time. Specifically, Meister et al's method with the combination of both the position and termination point as well as the fixed length solution has the highest RTCORE throughput and thus path tracer performance.

On the contrary, the cache does not seem to greatly correlate to the path tracer's performance. Despite the superiority of the methods in regards to the path tracer's duration, their cache utilization is lower. Similar observation can be made when looking at the VRAM metric at graph 4.8. High VRAM throughput could lead to less performance, because the stream processor could be stalled due to the slow memory fetches from the VRAM. This can also be confirmed when looking at the SM throughput at graph 4.9. The original ground truth algorithm does not have the extra path state buffers, hence the least VRAM and the highest SM throughput is recorded.

It is crucial to not forget that different methods include more expensive instructions. A trivial example is the Gaussian distribution, which includes extra division, multiplication and a square root to calculate the standard deviation. Also, random sampling of a uniform variable is as well a necessary extra cost. This is also consistent, with other methods that require extra instructions. In other words, there is an apparent overhead for each method that is not trivial to evaluate, as further investigation is required.

Evidently, the sorting stage is the bottleneck of all the ray-reordering methods. However, looking at 4.4 and 4.6, it is clear that the methods that have the lowest path tracer performance, have the highest sorting performance. Specifically, the Gaussian distribution method variant on both a standard and clamped grid have significantly less sorting time. Apparently, there is an optimal balance between better accuracy on the length approximation and sorting stage speed. Judging by the overall execution time of both the path tracer and the bitonic sort's execution, the Gaussian distribution method on a camera clamped grid is superior when compared to the all the other ray-reordering methods.

5. Conclusion

5.1 Discussion

In this thesis, thread reordering was examined exclusively using the termination points as sorting keys. All of the formulas and methods achieved better performance on the path tracer, when compared to Falcor's minimal path tracer, answering the initial research question. However, the fixed length solution yielded the fastest path tracer on average. On the adaptive length formulas more efficient cache utilization and higher RTCore throughput was noticed, despite the slower execution time. All of the methods recorded the same 75% warp utilization, proving the significant role of thread coherency on the path tracer's performance. Further investigation is required in order to determine whether the extra cost is due to the overhead of the extra instructions or due to the inconsistencies of the adaptive termination point lengths.

Furthermore, an adaptive grid based on the camera's position was as well implemented and tested. The results showed slightly worse performance on the path tracer compared to their counterparts.

While the adaptive length solutions achieved higher execution times, they had a positive effect on the sorting stage's performance. In summary, they achieved higher memory and thread coherency in both sorting and path tracer stage. Specifically, calculating length using approximated Gaussian distributions on an adaptive grid, was proved the best method by achieving the highest execution time reduction in the sorting stage. The significance of this reduction stems also from Meister's observation [21] that the sorting kernels are indeed the bottleneck of thread reordering.

5.2 Future Work

For the purpose of this thesis not the fastest parallel sorting method was used, increasing a lot the execution time when incorporating thread reordering. It is crucial to test the fastest algorithm e.g. [31] and investigate whether we notice a similar performance boost using the adaptive length solutions. With that information, we can answer whether the total path tracer pipeline including the sorting stages has a noticeable speedup, when compared to a path tracer without any ray sorting and hence the expensive overhead of the compute kernels.

As the memory requirements of 3D spatial grids grow substantially with the increase of the resolution, adaptive and sparse grids could contribute to a greater spatial representation of the scene. This could potentially lead to a more efficient spatial classification of the bounces and meanwhile reduce the memory requirement and data transfers.

Bibliography

[1] M. Claypool, K. Claypool, and F. Damaa, "The effects of frame rate and resolution on users playing first person shooter games," *Proceedings of SPIE - The International Society for Optical Engineering*, vol. 6071, Jan. 2006. DOI: 10.1117/12.648609.

[2] S. Liu, A. Kuwahara, J. J. Scovell, and M. Claypool, "The effects of frame rate variation on game player quality of experience," in *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*, ser. CHI '23, Hamburg, Germany: Association for Computing Machinery, 2023, ISBN: 9781450394215. DOI: 10.1145/3544548.3580665. [Online]. Available: https://doi.org/10.1145/3544548.3580665.

[3] Jul. 2023. [Online]. Available: https://developer.nvidia.com/rtx/ray-tracing.

[4] J. Archer, *Here are all the confirmed ray tracing and dlss games so far*, Aug. 2023. [Online]. Available: https://www.rockpapershotgun.com/confirmed-ray-tracing-and-dlss-games.

[5] J. T. Kajiya, "The rendering equation," in *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques*, ser. SIGGRAPH '86, New York, NY, USA: Association for Computing Machinery, 1986, pp. 143–150, ISBN: 0897911962. DOI: 10.1145/15922.15902. [Online]. Available: https://doi.org/10.1145/15922.15902.

[6] Aug. 2023. [Online]. Available: https://en.wikipedia.org/wiki/Rendering_equation.

[7] Aug. 2023. [Online]. Available: https://en.wikipedia.org/wiki/Monte_Carlo_method.

[8] E. Veach, "Robust Monte Carlo Methods for Light Transport Simulation," Ph.D. dissertation, Stanford University, 1997.

[9] B. Bitterli, C. Wyman, M. Pharr, P. Shirley, A. Lefohn, and W. Jarosz, "Spatiotemporal reservoir resampling for real-time ray tracing with dynamic direct lighting," *ACM Transactions on Graphics (Proceedings of SIGGRAPH)*, vol. 39, no. 4, Jul. 2020. DOI: 10/gg8xc7.

[10] D. Lin, M. Kettunen, B. Bitterli, J. Pantaleoni, C. Yuksel, and C. Wyman, "Generalized resampled importance sampling: Foundations of restir," *ACM Trans. Graph.*, vol. 41, no. 4, Jul. 2022, ISSN: 0730-0301. DOI: 10.1145/3528223.3530158. [Online]. Available: https://doi.org/10.1145/3528223.3530158.

[11] J. Vorba, J. Hanika, S. Herholz, T. Müller, J. Křivánek, and A. Keller, "Path guiding in production," in *ACM SIGGRAPH 2019 Courses*, ser. SIGGRAPH '19, Los Angeles, California: ACM, 2019, 18:1–18:77,

ISBN: 978-1-4503-6307-5. DOI: 10.1145/3305366.3328091. [Online]. Available: http://doi.acm.org/10.1145/3305366.3328091.

[12] J. Vorba, O. Karlík, M. Šik, T. Ritschel, and J. Křivánek, "On-line learning of parametric mixture models for light transport simulation," *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2014)*, vol. 33, no. 4, Aug. 2014.

[13] P. Liang and D. Klein, "Online EM for unsupervised models," in *Proceedings of Human Language Technologies: The 2009 Annual Conference of the North American Chapter of the Association for Computational Linguistics*, Boulder, Colorado: Association for Computational Linguistics, Jun. 2009, pp. 611–619. [Online]. Available: https://aclanthology.org/N09-1069.

[14] M. Derevyannykh, "Real-time path-guiding based on parametric mixture models," *CoRR*, vol. abs/2112.09728, 2021. arXiv: 2112.09728. [Online]. Available: https://arxiv.org/abs/2112.09728.

[15] C. Schied, A. Kaplanyan, C. Wyman, *et al.*, "Spatiotemporal variance-guided filtering: Real-time reconstruction for path-traced global illumination," in *Proceedings of High Performance Graphics*, ser. HPG '17, Los Angeles, California: Association for Computing Machinery, 2017, ISBN: 9781450351010. DOI: 10.1145/3105762.3105770. [Online]. Available: https://doi.org/10.1145/3105762.3105770.

[16] D. Zhdan, "Reblur: A hierarchical recurrent denoiser," in Aug. 2021, pp. 823–844, ISBN: 978-1-4842-7184-1. DOI: 10.1007/978-1-4842-7185-8_49.

[17] [Online]. Available: https://www.nvidia.com/nl-nl/geforce/technologies/dlss/.

[18] T. Aila and S. Laine, "Understanding the efficiency of ray traversal on gpus," in *Proceedings of the Conference on High Performance Graphics 2009*, ser. HPG '09, New Orleans, Louisiana: Association for Computing Machinery, 2009, pp. 145–149, ISBN: 9781605586038. DOI: 10.1145/1572769.1572792. [Online]. Available: https://doi.org/10.1145/1572769.1572792.

[19] S. Laine, T. Karras, and T. Aila, "Megakernels considered harmful: Wavefront path tracing on gpus," in *High Performance Graphics*, 2013. [Online]. Available: https://api.semanticscholar.org/CorpusID:11807791.

[20] S. Frey, G. Reina, and T. Ertl, "Simt microscheduling: Reducing thread stalling in divergent iterative algorithms," in *2012 20th Euromicro International Conference on Parallel, Distributed and Network-based Processing*, 2012, pp. 399–406. DOI: 10.1109/PDP.2012.62.

[21] D. Meister, J. Boksansky, M. Guthe, and J. Bittner, "On ray reordering techniques for faster gpu ray tracing," in *Symposium on Interactive 3D Graphics and Games*, ser. I3D '20, San Francisco, CA, USA: Association for Computing Machinery, 2020, ISBN: 9781450375894. DOI: 10.1145/3384382.3384534. [Online]. Available: https://doi.org/10.1145/3384382.3384534.

[22] L. Blanleuil and C. Collange, "Scheduling paths leveraging dynamic information in simt architectures," in *COMPAS 2021-Conférence francophone d'informatique en Parallélisme, Architecture et Système*, 2021, pp. 1–6.

[23] I. Wald, "Active thread compaction for gpu path tracing," in *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*, ser. HPG '11, Vancouver, British Columbia, Canada: Association for Computing Machinery, 2011, pp. 51–58, ISBN: 9781450308960. DOI: `10.1145/2018323.2018331`. [Online]. Available: `https://doi.org/10.1145/2018323.2018331`.

[24] E. Mansson, J. Munkberg, and T. Akenine-Moller, "Deep coherent ray tracing," in *2007 IEEE Symposium on Interactive Ray Tracing*, 2007, pp. 79–85. DOI: `10.1109/RT.2007.4342594`.

[25] C. P. Gribble and K. Ramani, "Coherent ray tracing via stream filtering," in *2008 IEEE Symposium on Interactive Ray Tracing*, 2008, pp. 59–66. DOI: `10.1109/RT.2008.4634622`.

[26] *Improve Shader Performance and In-Game Frame Rates with Shader Execution Reordering | NVIDIA Technical Blog — developer.nvidia.com*, `https://developer.nvidia.com/blog/improve-shader-performance-and-in-game-frame-rates-with-shader-execution-reordering/`, [Accessed 27-09-2023].

[27] *Falcor — developer.nvidia.com*, `https://developer.nvidia.com/falcor`, [Accessed 03-10-2023].

[28] *NVIDIA Nsight Graphics — developer.nvidia.com*, `https://developer.nvidia.com/nsight-graphics`, [Accessed 09-10-2023].

[29] L. Yang, S. Liu, and M. Salvi, "A survey of temporal antialiasing techniques," *Computer Graphics Forum*, vol. 39, no. 2, pp. 607–621, Jul. 2020. DOI: `10.1111/cgf.14018`.

[30] J. Baert, A. Lagae, and P. Dutré, "Out-of-core construction of sparse voxel octrees," in *Proceedings of the 5th High-Performance Graphics Conference*, ser. HPG '13, Anaheim, California: ACM, 2013, pp. 27–32, ISBN: 978-1-4503-2135-8. DOI: `10.1145/2492045.2492048`. [Online]. Available: `http://doi.acm.org/10.1145/2492045.2492048`.

[31] A. Adinets and D. Merrill, *Onesweep: A faster least significant digit radix sort for gpus*, 2022. arXiv: `2206.01784 [cs.DC]`.

[32] D. Merrill and M. Garland, "Single-pass parallel prefix scan with decoupled lookback," 2016. [Online]. Available: `https://api.semanticscholar.org/CorpusID:51919482`.

[33] S. Ashkiani, A. Davidson, U. Meyer, and J. D. Owens, "Gpu multisplit," *SIGPLAN Not.*, vol. 51, no. 8, Feb. 2016, ISSN: 0362-1340. DOI: `10.1145/3016078.2851169`. [Online]. Available: `https://doi.org/10.1145/3016078.2851169`.

[34] *GitHub - b0nes164/OneSweep: A simple library-less CUDA implementation of the OneSweep sorting algorithm. — github.com*, `https://github.com/b0nes164/OneSweep?tab=readme-ov-file`, [Accessed 27-05-2024].

[35] *Sorting — linebender.org*, `https://linebender.org/wiki/gpu/sorting/`, [Accessed 22-06-2024].

[36] *Bitonic sorter - Wikipedia — en.wikipedia.org*, `https://en.wikipedia.org/wiki/Bitonic_sorter`, [Accessed 27-05-2024].

[37] E. Games, *Unreal engine sun temple, open research content archive (orca)*, `http://developer.nvidia.com/orca/epic-games-sun-temple`, Oct. 2017. [Online]. Available: `http://developer.nvidia.com/orca/epic-games-sun-temple`.

[38] *GitHub - b0nes164/GPUSorting: OneSweep, implemented in CUDA, D3D12, and Unity style compute shaders. Theoretically portable to all wave/warp/subgroup sizes. — github.com*, `https://github.com/b0nes164/GPUSorting`, [Accessed 24-06-2024].