

Universiteit Leiden

GPU Thread Data Remapping In Monte Carlo Shortwave Ray-tracing

Name:	Ivor Brouwer
Student ID:	s2041685
Date:	August 26, 2026
Specialisation:	Advanced Computing and Systems
1st supervisor:	Ben van Werkhoven
2nd supervisor:	Stijn Heldens

Master's Thesis in Computer Science

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
2333 CA Leiden
The Netherlands

Abstract

Short-wave radiation emitted by the Sun travels through the Earth's atmosphere, where photons may be absorbed or scattered by clouds, gases, aerosols, and the surface. These interactions affect the amount of sunlight reaching the surface and can be modeled using Monte Carlo photon tracing. However, this method is computationally expensive because many photon histories are required to obtain statistically accurate results. GPUs are well suited to photon tracing because many independent photon histories can be processed in parallel. Their efficiency is nevertheless reduced when threads within the same warp encounter different events, such as scattering, surface interactions, or escape through the top of the domain. This causes branch divergence, forcing a warp to execute multiple control-flow paths with only part of its threads performing useful work. This thesis investigates thread-data-remapping as a method for reducing branch divergence in an existing history-based short-wave photon ray-tracer, in which each GPU thread follows an individual photon history. The history-based implementation is first optimized to provide a strong reference, resulting in speed-ups between 2.81× and 4.16× across the evaluated workloads. Several remapping strategies are then developed that reorganize photon states during execution so that threads within a warp process similar events. These strategies use warp-local stacks, block-level stacks, and link-based shared-memory structures. The remapping implementations consistently increased the number of threads performing useful work within each warp, but this generally did not reduce runtime. The best-performing remapping implementation, a warp-local stack, was 9.6% slower than the optimized history-based implementation for a clear-sky workload and 1.4% slower for a moderate-cloud workload, while achieving a 1.9% improvement for a synthetic scattering-intensive workload. These results show that improved warp coherence alone does not guarantee better performance. The benefits of remapping must outweigh the additional costs of photon-state movement, queue management, shared-memory usage, synchronization, and reduced cache capacity. For the evaluated realistic workloads, the optimized history-based implementation therefore remains the preferred approach, although remapping may become beneficial when event paths become more computationally expensive.

Contents

1 Introduction **6**

2 Background **8**
2.1 CUDA architecture 8
2.2 History-based or Event-based solver 10
2.3 The photon ray-tracer 10
2.3.1 Photon state and optical properties 11
2.3.2 Coarse acceleration grid 12
2.3.3 Coarse-cell boundary event 12
2.3.4 Fine-grid evaluation 13
2.3.5 Scattering 14
2.3.6 Null collision 14
2.3.7 Surface interaction 14
2.3.8 Top-of-domain exit and photon reset 15
2.3.9 Control-flow divergence 15

3 Related Work **15**
3.1 GPU resource optimization 15
3.2 History vs Event-based 16
3.3 Thread-data-remapping 16
3.3.1 Memory layout 17
3.3.2 Head-Or-Tail 17
3.3.3 Block-level remapping and vote skipping 17
3.3.4 Link-based remapping 18
3.4 Thesis Positioning 18

4 Methodology **18**
4.1 Base Optimizations 19
4.1.1 Removal of Sobol sequence 19
4.1.2 Better reset locality in warp and work stealing 19
4.1.3 Overall register/complexity reductions 20
4.1.4 Simplified scatter optics 20
4.1.5 Pre-calculated cell absorption and scattering 21
4.1.6 Improved caching 22
4.1.7 Accumulation of atmospheric contributions 22
4.1.8 Float to int simplification and fast approximation 22
4.1.9 Mixed precision 23
4.2 Event Remapping Structure 23
4.3 Implementations 24
4.3.1 Warp-stack 24
4.3.2 Warp-link 25
4.3.3 Block-stack 26
4.4 Overall Design Refinements 27
4.4.1 Mixed Precision 27
4.4.2 Extra Caching 27
4.4.3 Vote Skipping 28

4.5 Stack Design Refinement . . . 28
4.6 Warp-stack Refinements . . . 29
4.6.1 Active Photons in Registers . . . 29
4.6.2 Full-, Drain- and Compaction Phase . . . 29
4.7 Block-stack Refinement . . . 30
4.8 Setup . . . 31
4.9 Evaluated configurations . . . 31
4.9.1 Base . . . 32
4.9.2 Optimized base . . . 32
4.9.3 Warp-stack . . . 32
4.9.4 Block-stack . . . 33
4.9.5 Warp-link . . . 33
4.9.6 Scatter-split, Scattering vs Everything . . . 33
4.10 Parameter tuning . . . 33
4.11 Metrics . . . 34
4.12 Test files . . . 35
4.12.1 Clear sky . . . 35
4.12.2 Moderate clouds . . . 35
4.12.3 Scatter workload . . . 36
4.13 Result validation . . . 37

5 Results **38**
5.1 Clear Sky . . . 38
5.1.1 Runtime and speed-up . . . 38
5.1.2 Divergence and Event Coherence . . . 38
5.1.3 Memory and Cache Behavior . . . 39
5.1.4 Occupancy . . . 40
5.1.5 Instruction count and main stall reasons . . . 40
5.2 Moderate clouds . . . 40
5.2.1 Runtime and speed-up . . . 41
5.2.2 Divergence and Event Coherence . . . 41
5.2.3 Memory and Cache Behavior . . . 41
5.2.4 Occupancy . . . 42
5.2.5 Instruction count and main stall reasons . . . 42
5.3 Scatter workload . . . 43
5.3.1 Runtime and speed-up . . . 43
5.3.2 Divergence and Event Coherence . . . 43
5.3.3 Memory and Cache Behavior . . . 44
5.3.4 Occupancy . . . 44
5.3.5 Instruction count and main stall reasons . . . 45

6 Discussion **45**
6.1 Optimized base . . . 45
6.2 Warp-stack . . . 46
6.3 Warp-link . . . 46
6.4 Block-stack . . . 47
6.5 Scatter-split . . . 47

6.6 Other implementations 48
6.6.1 Block-link 48
6.6.2 Extra caching 48
6.6.3 Shared-Memory Indirection 48
6.6.4 Multiple Event Levels 48
6.6.5 Reset Handling 49
6.6.6 Delayed Scattering 49
6.6.7 Reducing Block-stack synchronization 50
6.6.8 Discarding photons 50

7 Conclusion **50**

1 Introduction

The Sun emits short-wave radiation that travels to Earth as photons and interacts with gases, aerosols, clouds, and the surface. During these interactions, photons can be absorbed or scattered in a different direction. Interactions therefore change how much radiation reaches the atmosphere and surface, producing patterns of shadows and concentrated sunlight. These variations affect atmospheric and surface heating and can influence cloud development. Accurately modeling this behavior can therefore improve predictions of weather, climate change, and solar-energy production.

Radiative transfer modeling describes how radiation moves through the atmosphere while interacting with gases, clouds, aerosols, and the surface. Iwabuchi [1] presents efficient Monte Carlo methods for this problem [2], in which photons are traced through absorption, scattering, reflection, and escape events. Absorption reduces the photon energy, represented in the ray tracer by its weight, while scattering changes its direction. A photon is followed until it loses its weight, reaches the surface, or leaves the atmosphere. In cloudy regions, repeated scattering can substantially increase the length and complexity of these paths.

Explicitly calculating the interactions of every photon emitted by the Sun would be computationally infeasible. Monte Carlo simulation [3] instead represents sunlight using many sampled photon histories and estimates the final result from their average behavior using the law of large numbers. Events such as scattering, absorption, reflection, and escape are sampled probabilistically [1]. This allows three-dimensional radiative transfer to be modeled without simulating every physical photon and provides greater accuracy than earlier lower-dimensional approaches such as column-based or two-stream methods [4]. The estimates contain statistical noise, however, so many photon histories are required to obtain accurate fluxes and heating rates. Increasing the number of samples improves convergence but also increases computational cost.

These simulations are well suited to GPUs because individual photon histories are independent and can therefore be processed concurrently by many GPU threads. Since the simulation follows the path of each photon through the domain, this process is referred to as ray tracing. One such ray tracer was developed by Veerman, Stratum, and Heerwaarden [4] for short-wave radiative transfer. Photons are initialized with a direction determined by the position of the sun and traced through a three-dimensional atmospheric domain. As a photon travels through grid cells, the properties of each cell determine the probability of absorption or scattering. The bottom of the domain represents the surface, while the cells above it represent the atmosphere and may contain clouds. A photon is traced until it loses all of its weight, reaches the surface, or exits through the top of the atmosphere. Although this produces an accurate model, the resulting photon paths are irregular and may differ strongly between photons.

Irregular control flow is inefficient on NVIDIA GPU architectures [5]. GPU threads execute in groups of 32 called warps using the Single-Instruction, Multiple-Thread (SIMT) execution model. When all threads in a warp follow the same control-flow path, they can execute efficiently. When threads take different branches, particularly in nested control flow, branch divergence occurs. The warp must then execute the different paths separately

while disabling threads that do not participate in the current path, leaving part of the warp inactive. Figure 1 illustrates this serialized execution caused by divergence within a warp.

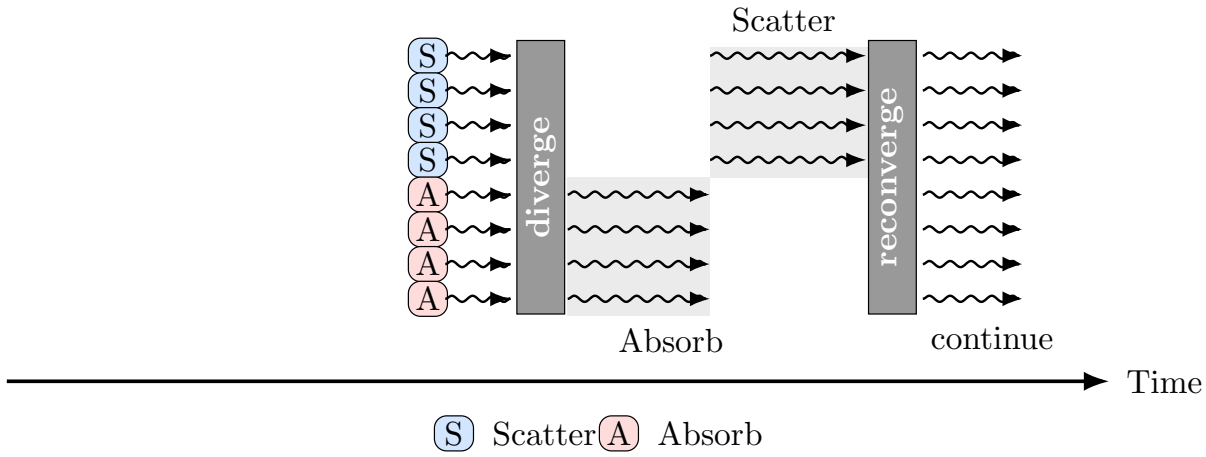


Figure 1: Control-flow divergence serializes different event paths within a warp. Mixed scatter and absorb events first diverge into separate branch bodies, execute the corresponding instructions, reconverge, and then continue with the shared execution path. Based on Source: NVIDIA [6, Fig. 20]

This is a large problem in the above mentioned photon ray-tracer, which could be improved when photons performing the same event are moved to the same warp. This optimization technique where work is moved between threads during execution is called thread-data-remapping [7, 8] and it improves warp coherence as can be seen in Figure 2. This is a great solution as it is possible to have more efficient program execution. The main trade-off is the introduction of extra data movement, memory usage, and synchronization making it a balance between the effects of better warp coherence and increased overhead.

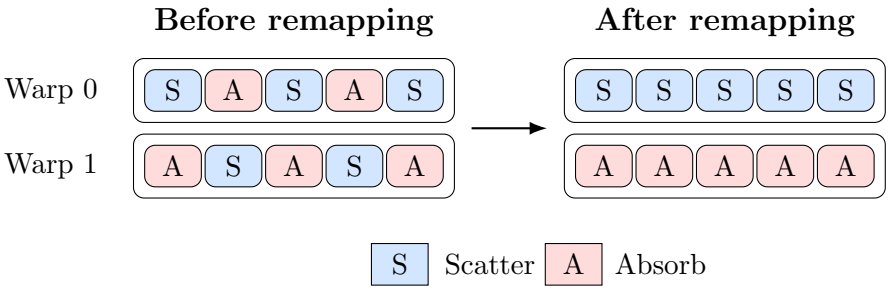


Figure 2: Thread-data-remapping groups photons performing the same event into the same warp. Here, mixed scatter and absorb events are reorganized into more coherent warps.

This study will investigate the improvements and trade-offs associated with implementing thread-data-remapping in the photon ray tracer. This is relevant as it will open up the possibility for faster and larger ray-tracing simulations of short-wave radiative transfer and it will show the impact thread-data-remapping can have on photon based ray-tracers as this has not been explored before. From this the following research questions emerge:

- **RQ1:** How much can the base implementation be sped-up using optimization techniques, such as caching and algorithmic changes?

- **RQ2:** How much speed-up does Thread Data Remapping provide compared to an optimized history-based baseline?
- **RQ3:** What is the cost of remapping in terms of shared-memory usage, executed instructions, register pressure, synchronization, and occupancy?
- **RQ4:** How do different remapping strategies compare and what are the benefits in different event distributions?

The results showed that the optimized history-based implementation achieves substantial performance improvements over the original ray-tracer, with speed-ups between 2.81× and 4.16× across the evaluated workloads. Thread-data remapping consistently improves warp coherence, but the additional cost of moving photon states, managing queues, using shared memory, and synchronization generally prevents this from translating into a lower runtime. The Warp-stack provides the best balance between these costs and benefits: compared with the optimized history-based implementation, it is 9.6% slower for the clear-sky workload and 1.4% slower for the moderate-cloud workload, but becomes 1.9% faster for the synthetic scattering-intensive workload. These results indicate that remapping becomes beneficial only when divergent event paths are sufficiently frequent or computationally expensive to outweigh its associated overhead.

In Section 2 more background is given for important principles, necessary for understanding the research. Section 3 discusses related work, the work that has been done on the topic earlier and their findings, which helped shape the ideas of this thesis. In Section 4 the method of extracting the results is described. Section 5 describes the results, which are interpreted in the Section 6 and Section 7 gives the final conclusion.

2 Background

A deeper explanation for important principles necessary for understanding the research is given in this section.

2.1 CUDA architecture

A CUDA kernel [5] is executed by a grid of threads divided into thread blocks. Each block is assigned to a streaming multiprocessor (SM), where multiple blocks may reside concurrently. Threads within a block are grouped into warps of 32 threads, with the position of a thread within a warp referred to as its lane. Warp schedulers issue instructions from ready warps to the execution units and can hide memory and instruction latency by switching between resident warps. CUDA follows the single-instruction, multiple-thread (SIMT) execution model: each thread maintains its own registers, program state, and memory addresses, while lanes within a warp normally execute the same instruction concurrently. Figure 3 illustrates this hierarchy and its mapping onto GPU hardware. Threads within a block can communicate through block-local shared memory, while all blocks can access GPU-wide global memory.

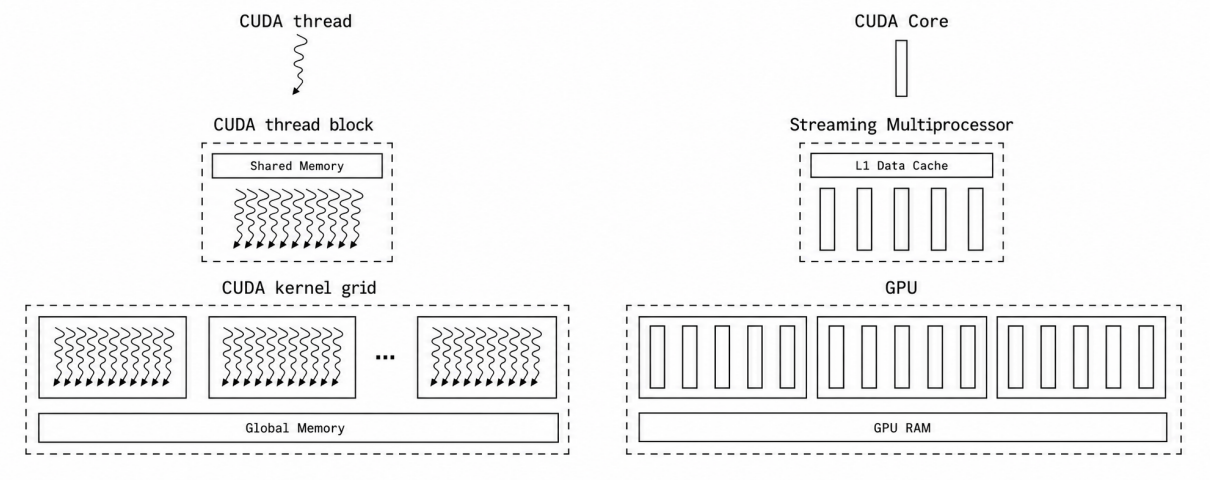


Figure 3: Mapping of the CUDA software hierarchy onto GPU hardware. Threads are grouped into blocks and blocks into a kernel grid. Blocks execute on streaming multiprocessors, using block-local shared memory and GPU-wide global memory. Image source: Modal [9].

As introduced in Figure 1, warp divergence occurs when threads within a warp follow different control-flow paths. These paths must be serviced separately, with only the lanes belonging to the current path enabled, reducing the useful work performed by an issued instruction. Instructions may also be predicated, such that a predicate independently determines whether the instruction affects each lane. A predicated-off thread discards the instruction result but remains part of the warp's active execution mask, whereas an inactive thread is excluded from the mask because it has exited the kernel.

Thread-data-remapping can reduce divergence but introduces additional instructions, register usage, memory operations, and synchronization. An important constraint is occupancy, defined as the ratio between the number of resident warps on an SM and the maximum supported by that SM. Higher occupancy provides more warps with which the scheduler can hide latency, although maximum occupancy does not necessarily provide maximum performance. Occupancy is constrained by the block size, registers per thread, shared memory per block, and architectural limits of the GPU [5]. In particular, increased thread-local state or longer variable lifetimes can increase register usage and reduce the number of blocks and warps that can reside concurrently on an SM.

Memory placement introduces similar trade-offs. Registers provide the lowest-latency storage and hold the thread-local photon state in the ray-tracer. Shared memory is low-latency, explicitly managed block-local storage and is therefore suitable for exchanging photon states during remapping. However, its capacity is limited, so high shared-memory usage per block can reduce occupancy. On many NVIDIA architectures, shared memory and the L1 cache share on-chip capacity, meaning that allocating more shared memory can reduce the available L1 cache and increase accesses to the higher-latency L2 cache. Global memory provides substantially more storage but has much higher latency, and its performance depends on caching, access locality, and whether accesses by neighboring lanes can be combined into coalesced memory transactions. Remapping must therefore balance

reduced divergence against additional instructions, register pressure, shared-memory usage, reduced cache capacity, synchronization, and potentially less efficient global-memory access.

Communication through shared memory also requires synchronization to guarantee that writes by one thread are visible before being read by another. A block-wide barrier waits until all participating threads reach the barrier and makes preceding shared-memory writes visible. All non-exited threads must participate, so early-arriving warps may have to wait for the slowest warp. When communication is limited to lanes within a warp, warp-level synchronization can avoid this block-wide waiting. Communication between blocks is more restricted because blocks do not share shared memory and may execute in any order; it generally requires atomic operations, cooperative grid synchronization, or a separate kernel launch.

These performance factors are strongly interconnected. Additional logic or storage may improve one part of the kernel while increasing register or shared-memory usage and thereby reducing occupancy or cache capacity. Performance also depends on the workload, CUDA compiler, GPU architecture, and configuration parameters such as block and grid dimensions, shared-memory allocation, cache configuration, and algorithm-specific parameters. Since the best value of one parameter can depend on the others, the resulting search space can become large. Kernel tuning therefore evaluates parameter combinations empirically to identify the best-performing configuration for a specific workload and hardware platform.

2.2 History-based or Event-based solver

The main methods of tackling the radiative transfer problem are history-based and event-based solvers [10]. History-based solvers simulate every photon individually and the state of the photon is updated based on the movement of this singular photon. Event-based solvers are implemented by grouping photons based on their next event and are then processed in these groups. An event could be any action a photon could take in the domain, for example absorption or scattering. Both solver methods are possible on GPUs, every thread could simply run a photon until completion or it is possible to have a kernel for every event. When implementing thread-data-remapping its close to an event-based solver as all threads in a warp will process the same event.

2.3 The photon ray-tracer

The baseline program is a history-based short-wave photon solver [4]. Each GPU thread processes one photon history at a time and keeps the current photon state local to the thread. The thread advances the photon through the atmospheric domain until the photon exits through the top of the domain or is terminated after absorption or a surface interaction making the photon lose its weight. A photon that reaches the surface may also be reflected and continue through the domain. After a photon history terminates, the thread initializes a new photon at the top of the domain if additional work remains. The output of the program consists of the direct and diffuse radiative contributions absorbed within the atmosphere, the downward and upward surface fluxes, and the radiation leaving through the top of the domain. Figure 4 shows an example of the surface output.

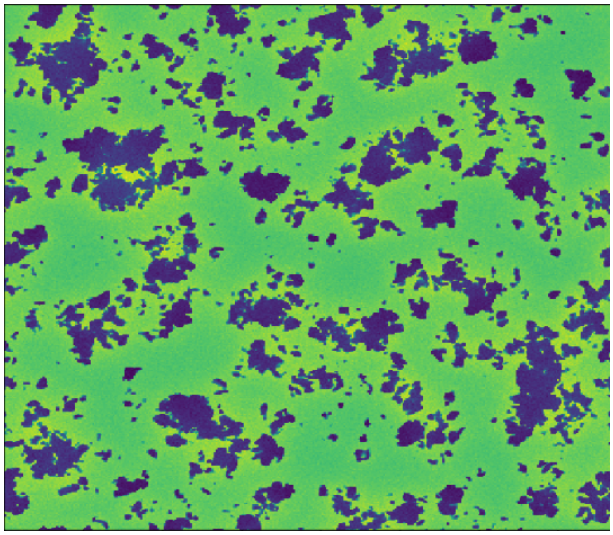


Figure 4: Example surface output of the photon ray-tracer. Lighter regions receive more flux, while darker regions correspond to shadows caused by clouds.

2.3.1 Photon state and optical properties

The atmospheric domain is represented by a three-dimensional fine grid. Each fine-grid cell stores optical properties describing the interaction of radiation with gases, aerosols, and clouds. The state required to continue a photon history includes its position, direction, remaining weight, radiation classification, and several quantities used during grid traversal. The main quantities used throughout this thesis are summarized in Table 1.

The physical extinction coefficient describes the total interaction rate per unit distance. It is the sum of the absorption and scattering coefficients:

$$k_{\text{ext}} = k_{\text{abs}} + k_{\text{sca}}. \tag{1}$$

The total scattering coefficient contains contributions from gas, cloud, and aerosol scattering:

$$k_{\text{sca}} = k_{\text{sca,gas}} + k_{\text{sca,cld}} + k_{\text{sca,aer}}. \tag{2}$$

The single-scattering albedo ssa describes the fraction of physical extinction caused by scattering:

$$\text{ssa} = \frac{k_{\text{sca}}}{k_{\text{ext}}}. \tag{3}$$

A value of ssa close to zero indicates that extinction is dominated by absorption, while a value close to one indicates that it is dominated by scattering. In the original implementation, k_{sca} and ssa are recalculated inside the ray-tracing kernel whenever a fine-grid cell is evaluated. Absorption reduces the photon weight, while scattering changes the photon direction.

Quantity	Meaning
position	Current photon position within the atmospheric domain, represented by three components x, y, and z.
direction	Unit vector describing the current photon direction.
weight (w)	The remaining radiative contribution carried by the photon.
kind	Classification of the photon as direct or diffuse.
tau	Remaining optical depth to the next sampled candidate interaction.
$d_{\max}$	Remaining distance along the current photon direction to the nearest boundary of the current coarse-grid cell.
d_n	Candidate interaction distance calculated from the sampled optical depth and the extinction coefficient of the current coarse-grid cell.
k_{abs}	Absorption coefficient of the current fine-grid cell.
k_{sca}	Total scattering coefficient of the current fine-grid cell.
k_{ext}	Physical extinction coefficient of the current fine-grid cell.
$k_{\text{ext,null}}$	Extinction coefficient stored for the current coarse-grid cell and used for null-collision sampling.
ssa	Single-scattering albedo of the current fine-grid cell, the fraction of physical extinction caused by scattering.
f_{noabs}	Factor calculated during a fine-grid evaluation. The fraction $1 - f_{\text{noabs}}$ is the absorbed radiation.
f_{scat}	Conditional probability that the non-absorbed part of a candidate interaction is a real scattering event rather than a null collision.

Table 1: Notation used to describe the photon state, grid traversal, and optical properties in the photon ray-tracer.

2.3.2 Coarse acceleration grid

Accessing the fine-grid optical properties after every small movement would require many irregular global-memory accesses. The ray-tracer uses a coarser acceleration grid with null-collision sampling [11, 12, 13]. Each coarse-grid cell covers a region containing several fine-grid cells and stores an extinction coefficient $k_{\text{ext,null}}$. $k_{\text{ext,null}}$ is constructed from the maximum physical extinction coefficient within the corresponding fine-grid region. A minimum value is also applied during construction to avoid zero or excessively small extinction coefficients. When a photon enters a coarse-grid cell or changes direction, the ray-tracer calculates $d_{\max}$, the distance to the nearest boundary of that coarse cell. A dimensionless optical depth tau is sampled and converted into a candidate interaction distance:

$$d_n = \frac{\text{tau}}{k_{\text{ext,null}}}. \tag{4}$$

The candidate interaction distance d_n is compared with the remaining distance to the coarse-cell boundary, $d_{\max}$. This comparison determines whether the photon first reaches the boundary or first encounters a candidate interaction within the coarse cell.

2.3.3 Coarse-cell boundary event

If

$$d_n \geq d_{\max}, \tag{5}$$

the photon reaches the coarse-cell boundary before the sampled candidate interaction. The photon is advanced to the boundary, with a small numerical offset used to prevent it from remaining exactly on a grid boundary. The resulting boundary event can be one of three cases:

- the photon crosses into a neighboring coarse-grid cell.
- the photon reaches the surface.
- the photon exits through the top of the atmospheric domain.

For an ordinary coarse-cell crossing, the optical depth consumed while traveling to the boundary is subtracted:

$$\text{tau} \leftarrow \text{tau} - d_{\max} k_{\text{ext,null}}. \tag{6}$$

The remaining optical depth is carried into the next coarse-grid cell. The distance $d_{\max}$ and the coarse-grid extinction coefficient are then recalculated for the new cell.

2.3.4 Fine-grid evaluation

If

$$d_n < d_{\max}, \tag{7}$$

the candidate interaction lies inside the current coarse-grid cell. The photon is advanced by d_n, and the optical properties of the corresponding fine-grid cell are evaluated. Absorption is handled using implicit absorption. Rather than selecting a separate absorption event that immediately terminates the photon, part of the photon weight is added to the current fine-grid cell. In the original implementation, the total scattering coefficient and single-scattering albedo are first calculated from the optical properties of the fine-grid cell. The fraction of photon weight retained after absorption is then calculated as

$$f_{\text{noabs}} = 1 - \frac{k_{\text{abs}}}{k_{\text{ext,null}}}. \tag{8}$$

The absorbed contribution is

$$\Delta w_{\text{abs}} = w(1 - f_{\text{noabs}}), \tag{9}$$

after which the remaining photon weight becomes

$$w \leftarrow w f_{\text{noabs}}. \tag{10}$$

The absorbed contribution is immediately written to either the direct or diffuse atmospheric output according to the current photon kind. The original implementation therefore performs an atomic operation during every fine-grid evaluation. If the updated photon weight falls below a threshold, Russian roulette is applied. The photon survives with probability equal to its current weight. A surviving photon has its weight reset to one,

while a terminated photon is replaced by a newly initialized photon history. This avoids continuously tracing photons whose remaining contribution has become very small while preserving the expected radiative contribution. If the photon remains active, the non-absorbed part of the candidate interaction is classified as either a real scattering event or a null collision. The conditional probability of a real scattering event is calculated as

$$f_{\text{scat}} = \frac{\text{ssa}}{\text{ssa} - 1 + \frac{k_{\text{ext,null}}}{k_{\text{ext}}}}. \tag{11}$$

A uniformly sampled random number is compared with f_{scat}. The candidate interaction is classified as scattering with probability f_{scat} and as a null collision otherwise.

2.3.5 Scattering

During a real scattering event, the photon direction is changed. After the scattering angle and azimuth have been sampled, a local orthogonal basis is constructed around the current photon direction. The sampled direction is transformed into the domain coordinate system and stored as the new photon direction. After scattering, the photon is classified as diffuse. Since its direction has changed, the previously calculated distance to the next coarse-grid boundary is no longer valid. The value of $d_{\max}$ is therefore reset and recalculated during the next traversal iteration.

2.3.6 Null collision

The coarse-grid coefficient $k_{\text{ext,null}}$ is generally larger than the physical extinction coefficient of the local fine-grid cell. The difference introduces candidate interactions that do not correspond to real scattering events. These are referred to as null collisions.
After implicit absorption has been accounted for, a null collision means that the remaining candidate interaction does not change the photon direction or its direct-or-diffuse classification. The photon has nevertheless traveled the distance d_n, so the remaining distance to the current coarse-grid boundary is updated according to

$$d_{\max} \leftarrow d_{\max} - d_n. \tag{12}$$

2.3.7 Surface interaction

When a photon reaches the surface, its incoming weight is accumulated as either direct or diffuse downward surface radiation. The photon weight is then multiplied by the local surface albedo:

$$w \leftarrow w\alpha, \tag{13}$$

where α is the surface albedo. The resulting reflected contribution is accumulated as upward surface radiation. If the reflected weight is above the Russian-roulette threshold, the photon continues with its remaining weight. Otherwise, Russian roulette is applied. The photon is either terminated or allowed to continue with its weight reset to one. A continuing photon is assigned an upward reflected direction and classified as diffuse. If the reflected weight is above the threshold, the photon continues with its remaining weight. Otherwise, the photon history terminates and the thread initializes a new photon if additional work remains.

2.3.8 Top-of-domain exit and photon reset

A photon that leaves through the top of the atmospheric domain contributes its remaining weight to the upward top-of-domain flux. Its history then terminates and the thread initializes a new photon if additional work remains. It replaces a completed photon history with a newly initialized photon and allows the same GPU thread to process several photon histories during one kernel execution.

2.3.9 Control-flow divergence

The acceleration grid reduces the number of detailed fine-grid evaluations by allowing photons to cross coarse cells without repeatedly accessing all local optical properties. However, it also introduces several distinct execution paths into the main tracing loop. A photon may cross a coarse-cell boundary, perform a null collision, scatter, lose weight through absorption, interact with the surface, exit through the top of the domain, or be reset. Different photons encounter these events in different orders and at different frequencies. Threads within the same warp can therefore follow substantially different control-flow paths. The two main paths are coarse-cell advancement and fine-grid evaluation. The resulting warp divergence is the primary motivation for the thread-data-remapping methods evaluated in this thesis.

3 Related Work

In this section an overview follows on prior work done by other researchers relevant to the subject.

3.1 GPU resource optimization

Several studies have investigated GPU implementations of Monte Carlo ray-tracing and have demonstrated that the independence of individual photon histories makes the method suitable for massively parallel execution [14, 15, 16, 17]. These studies show that GPUs can substantially reduce the computational cost of Monte Carlo ray-tracing while retaining the flexibility and physical accuracy of the method. However, the achievable performance depends strongly on how the photon-transport algorithm is mapped to the GPU architecture.

Hijma et al. [18] provides an overview of commonly applied GPU optimization techniques. Several of these techniques are directly relevant to the ray-tracer considered in this thesis. Memory traffic can, for example, be reduced by recomputing inexpensive values instead of storing and loading them. The use of registers and shared memory can reduce repeated accesses to global memory, while changes to the control flow can reduce branch divergence and irregular execution within warps.

These optimizations are applicable to both the baseline history-based implementation and the thread-data-remapping implementations. Both approaches must therefore be optimized independently. Otherwise, an observed performance difference could be caused by general kernel improvements rather than by thread-data-remapping itself. Because

GPU performance depends on interacting parameters such as block size, register usage, shared-memory usage, and compile-time configuration, automatic tuning is used to find suitable configurations. Kernel Tuner [19] is used for this purpose, as it supports the automatic compilation and benchmarking of CUDA kernel variants from Python.

3.2 History vs Event-based

Early research often considered a history-based Monte Carlo solver, as this is approach requires little scheduling and allows the particle state to remain local to the thread. Its main disadvantage on GPUs is that particles within the same warp may encounter different interactions or terminate at different times, resulting in branch divergence and partially inactive warps. Early work by Heimlich, Mol, and Pereira [20] demonstrated that history-based Monte Carlo neutron transport can achieve substantial performance improvements on GPUs by processing many particle histories concurrently compared to a 8-core CPU. Here event-based methods are not yet considered, but a later study by Hamilton, Slattery, and Evans [10] compared history-based and event-based algorithms for multigroup Monte Carlo neutron transport on GPUs. For their evaluated workloads, the fully event-based implementation did not provide the best performance. Instead, their highest performance was obtained with a modified history-based algorithm that periodically removed terminated particles and compacted the remaining active histories. This reduced the number of inactive threads while avoiding the continuous scheduling and data movement of a fully event-based approach.

Ma et al. [21] extended the comparison by evaluating history-based and event-based multigroup transport on both CPUs and GPUs. Their event queues stored particle indices rather than complete particle states, avoiding the repeated movement of the full state when particles were inserted into or removed from queues. Despite these optimizations, the history-based implementation outperformed the event-based implementation in all evaluated cases. Although the event-based version achieved higher memory-bandwidth utilization, this did not result in a lower runtime. The the performance loss was attribute to the additional operations required to reorganize particles, batched atomic operations on the event queues, and reduced particle-data locality.

Both studies done show that improving execution coherence or memory-bandwidth utilization does not by itself guarantee improved performance. Data layout, caching, particle locality, queue operations, and the target GPU architecture all influence the final result.

3.3 Thread-data-remapping

Thread-data-remapping [7, 8] changes the association between logical work items and the GPU threads that process them. Data items that follow the same control-flow path can be reassigned to neighboring threads, thereby increasing execution coherence within a warp. Remapping introduces its own overhead, however, because the implementation must determine the appropriate grouping, exchange references or state between threads, and synchronize the participating threads. Its effectiveness depends on whether the reduction in divergent execution outweighs the cost of reorganizing the work.

3.3.1 Memory layout

Zhang et al. [7] considered reference redirection. Reference redirection leaves the original data in place and changes the indices through which threads access it. This limits the amount of data that must be moved, but can produce irregular and non-coalesced memory accesses. This was considered for multiple different application. A comparable implementation was done for the WARP neutron-transport framework by Bergmann and Vujic [22], which is more closely related to photon transport. Here redirection keeps particle states separately in memory by using a remapping vector. This remapping vector contains references to particle states. The remapping vector is sorted by reaction type after each transport iteration, grouping similar reactions without physically sorting the complete particle states. This lowers the amount of data moved during reorganization, but results in an additional irregular accesses when the particle state is retrieved.

3.3.2 Head-Or-Tail

Lin, Wang, and Liu [8] moved thread-data remapping onto the GPU through the WAGNERR framework. Remapping code is inserted immediately before selected branches, allowing the mapping to be based on runtime branch conditions and to be changed multiple times within a single kernel execution. The Head-or-Tail algorithm targets two-way branches. Each thread places its original reference index into either the head or tail of a shared-memory array according to its branch outcome. Atomic head and tail counters determine the insertion positions, after which the threads synchronize and load their new reference indices from the array. This method is very memory efficient for a two-way problem as it requires only one array of size N for 2 event types, whereas this would originally require an array per event of size N, where N is the problem size. On-GPU remapping may also require thread-local state to be transferred to its new owner. Lin, Wang, and Liu perform such communication through shared memory. Consequently, the amount of private state that must be exchanged affects the cost of each remapping operation. This is particularly relevant to the photon ray-tracer, where the position, direction, weight, optical-depth state, and cached values of a photon may need to remain associated throughout a long history. The original Head-or-Tail algorithm relies on block-level synchronization, which limits its direct use inside nested divergent branches. Lin and Wang [23] addresses this problem through the use of atomic operations and memory fences to provide synchronization among the threads participating in an inner branch. The same work proposes Head-or-Tails 2, which allows a wavefront to continue once the mapping information required by that wavefront has become available, rather than waiting for the complete block-level mapping. This reduces idle waiting and provides more flexible wavefront scheduling. This underlines the benefits of having the least amount of synchronization possible.

3.3.3 Block-level remapping and vote skipping

A related class of work investigates thread compaction in GPU hardware. Fung and Aamodt [24] proposed thread-block compaction, in which threads from several warps are regrouped according to their branch outcome. Combining threads across a block provides a larger pool from which full warps can be formed, but requires warps to synchronize at divergent branches and reconvergence points. They state that Block-wide compaction can

provide meaningful speed-ups when divergence is sufficiently severe, but its synchronization cost can offset or reverse those benefits when warps are already coherent. An implementation therefore needs a way to avoid unnecessary compaction. Li et al. [25] subsequently proposed bypassing compaction for warps that are already coherent and cancelling compaction when it cannot reduce the number of idle lanes. They support the general principle that remapping should be skipped when it provides insufficient improvement in execution coherence. Compaction can also combine the remaining active threads from multiple partially filled warps into fewer, more fully utilized warps. Wald [26] showed that this can substantially reduce the number of warp executions as work diminishes, although the resulting runtime improvement depends on the associated memory and hardware overheads.

3.3.4 Link-based remapping

Cuneo and Bailey [27, 28] developed an asynchronous on-GPU scheduling method for Monte Carlo neutron transport. Particle work is stored in warp-sized work links, with each link containing particles belonging to one event type. Per-warp shared-memory stashes organize empty, partially filled, and full links. Full links provide a warp of event-homogeneous work, while global-memory work links are used when locally generated work exceeds the available shared-memory capacity, this is necessary as in neutron transport the amount of work can grow during runtime. Work is exchanged between local and global storage using lock-free data structures, allowing blocks to progress without inter-block barriers. In the photon ray-tracer the number of simultaneously active photons is bounded, making it possible to investigate link-based remapping without the global-memory spilling required by neutron-transport.

3.4 Thesis Positioning

Previous work has shown that thread-data-remapping can improve execution coherence, but may also introduce substantial overhead through data movement, queue management, synchronization, and reduced memory locality. Most existing studies focus on general GPU applications or Monte Carlo neutron transport. This thesis applies and compares warp-local, block-level, and link-based remapping strategies in a continuous short-wave photon ray-tracer. Their performance is evaluated against an independently optimized history-based implementation to determine whether improved warp coherence outweighs the additional remapping overhead under different workloads.

4 Methodology

First an overview is given of the optimizations done on the earlier mentioned base implementation by Veerman, Stratum, and Heerwaarden. Furthermore the event remapping structure and different remapping implementations are shown and their specific design and implementation differences. Finally it is shown how results are extracted using different workloads, final implementations and result validation.

4.1 Base Optimizations

To ensure a fair comparison between the baseline version and remapped version, the base version is optimized using several techniques. This is done to show the impact of remapping on the program without other factors impacting the performance.

4.1.1 Removal of Sobol sequence

In the base implementation, Sobol sequences [29] are used to initialize the position of photons, this requires extra state variables, grid size variables and logic. Sobol sequences ensure an even distribution over a grid of n-dimensions. On a large enough sample size it will start to look close to a standard uniform distribution on the grid. Initializing the photons uniformly from the start without using Sobol sequencing can free up registers, giving more room for improved occupancy. This also makes it possible to have better spatial locality, in the Sobol version photons were distributed over the full grid. The simpler version only distributes over a section of the full grid based on it's block index. This ensures photons take a slightly more similar path and improves caching of data. In Figure 5 an example of sampling using a Sobol sequence and the simple grid version is shown. One trade-off is that there is less random noise at the start, but this showed no significant impact on the program output.

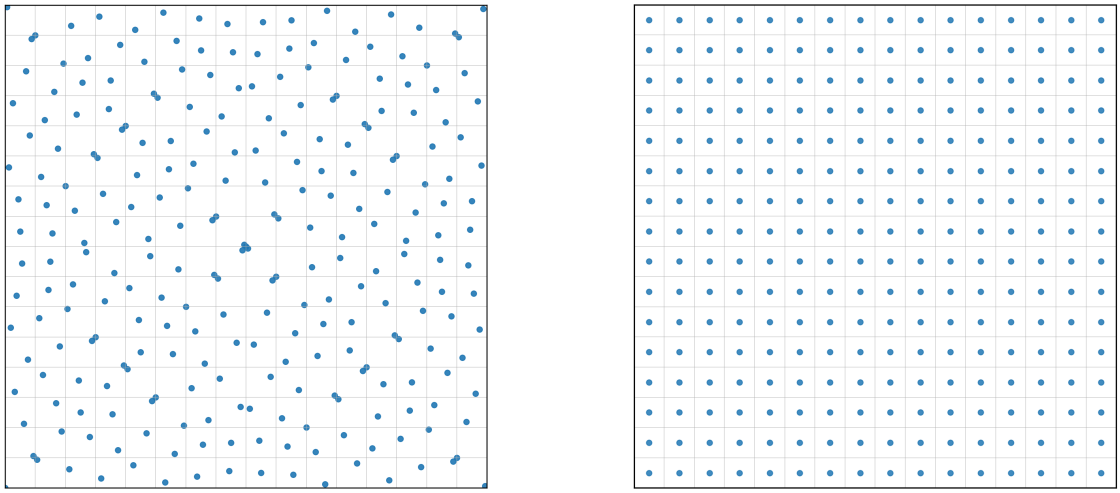


Figure 5: Comparison of 256-point Sobol-sequence initialization over the full grid and 256-point grid initialization.

4.1.2 Better reset locality in warp and work stealing

With the new position initialization it is possible to divide the grid by the thread block index, improving locality and caching. It also makes it simpler to take more work from other threads, in the base implementation each thread would perform a certain number of photons. It improves performance if threads that finish earlier can take work away from threads that have multiple photons with complex ray paths. This resulted in a better distribution of work and ensured thread execution times were closer together. This is implemented using a block-wide counter that tracks the number of photons launched. Whenever a thread initializes a new photon, it increments the counter atomically. Although

atomic operations can be expensive, these updates are distributed sparsely throughout the execution of the block. For example with grid size (64×64), each block processes approximately 256*6*6 photons (photons per cell * sector size). So contention on the counter is limited and the resulting performance overhead is negligible.

4.1.3 Overall register/complexity reductions

It is possible to have additional thread level parallelism but this is limited by multiple factors. One of these factors is register count, and decreasing it can increase occupancy. Some ways to achieve this is to change the way certain calculations are done, or to move or remove certain variables/structs to influence variable lifetimes. This was applied to different parts of the code, first by removing at runtime evaluated variables, as some could be evaluated at compilation time. It was also possible to simplify the cyclic boundary of the domain, in the base version fmod was used which is an expensive operation, but as photons only move one grid-cell at a time implementing the cyclic boundary could be done by doing only a substraction/addition of the domain size. Furthermore the scatter calculation was simplified, in the original version 2 vector structs would be built used for calculating the new direction, keeping more registers occupied. Beyond this, it was also possible to remove some kernel arguments necessary for Sobol sequencing. One example was the total down count, which represented the number of photons shot per grid cell. With Sobol sequencing it was not guaranteed that an exact number of photons would start in every cell as it was quasi-random. Due to this the number of photons per top domain cell had to be counted in global memory, causing extra atomics and more register usage. With the uniform distribution from the start this counter was removed as the number of photons per cell was known from the start.

4.1.4 Simplified scatter optics

The scattering type is sampled from the individual gas, cloud, and aerosol scattering optics. This requires the total scattering coefficient to be reconstructed inside the kernel,

$$k_{\text{sca}} = k_{\text{sca,gas}} + k_{\text{sca,cld}} + k_{\text{sca,aer}} \tag{14}$$

after which the random number is compared against normalized partial sums. As a result three scattering coefficients have to be loaded from global memory. If the sampled interaction is cloud or aerosol scattering, the corresponding asymmetry factor must also be loaded. This then adds up to five loads minimum. This can be improved by moving the probability calculation to an optics pre-processing step as they are all dependent on the same grid position, so for every position they can be calculated once beforehand. Instead of storing only the raw scattering coefficients, two probability thresholds are stored to the optics data,

$$p_{\text{aer}} = \frac{k_{\text{sca,aer}}}{k_{\text{sca}}}, \tag{15}$$

$$p_{\text{non-gas}} = \frac{k_{\text{sca,aer}} + k_{\text{sca,cld}}}{k_{\text{sca}}}. \tag{16}$$

For a uniformly distributed random number $r \in [0, 1)$, the scatter type is then selected using only comparisons:

$$\begin{cases} r \geq p_{\text{non-gas}} & \text{gas scattering,} \\ r < p_{\text{aer}} & \text{aerosol scattering,} \\ \text{otherwise} & \text{cloud scattering.} \end{cases} \tag{17}$$

This reduces the amount of global memory traffic in the scattering logic. In the gas-scattering case, only the non-gas cutoff has to be loaded, reducing the minimum number of global loads from three to one. For cloud and aerosol scattering, the kernel does two global loads, the aer probability and then the asymmetry factor, reducing the maximum number of required global loads from potentially five to three. In a test workload (mentioned in Section 4.12.2), the most common case of scattering is cloud scattering, then gas and lastly aerosol (with 0 occurrences).

4.1.5 Pre-calculated cell absorption and scattering

In the original implementation, the ray-tracing kernel computes both f_{noabs}, the fraction of photon weight retained after implicit absorption, and f_{scat}, the conditional probability that a non-absorbed candidate interaction is a real scattering event. To compute these values, four cell-dependent optical properties have to be loaded from global memory. Since photons access cells in an irregular order, these loads are generally non-coalesced in the ray-tracing kernel. First, the total scattering coefficient for one position is computed as

$$k_{\text{sca,tot}} = k_{\text{sca,gas}} + k_{\text{sca,cld}} + k_{\text{sca,aer}} \tag{18}$$

,
the single-scattering albedo of the cell is then given by

$$\text{ssa} = \frac{k_{\text{sca,tot}}}{k_{\text{ext}}} \tag{19}$$

Using the null-collision extinction coefficient $k_{\text{ext,null}}$, the no-absorption probability is computed as

$$f_{\text{noabs}} = 1 - (1 - \text{ssa}) \frac{k_{\text{ext}}}{k_{\text{ext,null}}} \tag{20}$$

The probability that a non-absorption collision is a real scattering event is

$$f_{\text{scat}} = \frac{\text{ssa}}{\text{ssa} - 1 + \frac{k_{\text{ext,null}}}{k_{\text{ext}}}} \tag{21}$$

Both f_{noabs} and f_{scat} only depend on cell-local optical properties. This makes it possible to pre-compute them in a optics pre-processing kernel. This pre-processing kernel traverses the grid in memory order, so the loads and stores are coalesced. The ray-tracing kernel then only has to load the two precomputed values. This reduces the number of irregular global loads in the coarse-grid path from four raw optical coefficients to two precomputed probabilities. It also removes the repeated arithmetic for $k_{\text{sca,tot}}$, ssa, f_{noabs}, and f_{scat} from the main photon loop.

4.1.6 Improved caching

During traversal of the coarse-grid, multiple consecutive steps can still correspond to the same fine-grid cell. In that case, the optical properties of the fine cell do not have to be reloaded from global memory. Instead, the most recently used fine-grid index is cached in registers. If the new fine-grid index is equal to the cached index, the previously loaded values of f_{noabs} and f_{scat} can be reused. This increases register usage slightly, but reduces repeated global memory loads and therefore lowers the number of global memory stalls.

4.1.7 Accumulation of atmospheric contributions

At every fine-grid cell evaluation, the photon contributes an absorption amount proportional to

$$contribution = w\,(1 - f_{\text{noabs}}) \tag{22}$$

,
this is added to the total atmospheric diffusion or direct light for that cell. In the original implementation, this contribution is written immediately using a global atomic operation. Since atomic global writes are expensive, to optimize this consecutive contributions to the same cell are first accumulated in registers. The kernel stores the last fine-grid index, the last photon type, and a local accumulator. When either the fine-grid index or the photon type changes, the accumulated value is flushed with a single read-modify-write. The photon type has to be included in this condition because direct and diffuse photons are written to separate output arrays. After the flush, the accumulator is reset and the new cell and photon type become the cached state. A final flush is also required after the traversal loop to write the last accumulated contribution. This reduces the number of global atomic writes from multiple in the same fine cell crossing to one per fine-grid cell as long as the photon type is still the same. The trade-off is that there are 3 extra values saved in registers.

4.1.8 Float to int simplification and fast approximation

Using Nsight-compute it is shown that converting the float position to the integer cell value is performed for 10% of the total cycles of the program. Currently the conversion is done by doing a division of the position by the cell size and casting to get an integer position in the domain:

$$\text{integer position} = \text{static_cast}(float/cell_size); \tag{23}$$

.
But it is possible to use the built in functions for calculating int from float (__float2int_rz), and use approximation functions with enough accuracy such as __fdividef instead:

$$\text{integer position} = \text{__float2int_rz}(\text{__fdividef}(float, cell_size)); \tag{24}$$

.
The trade-off is a slight reduction in accuracy of the division, but evaluation showed no significant output difference.

4.1.9 Mixed precision

Mixed precision was introduced as an additional optimization to reduce global memory traffic and improve cache efficiency. In the base implementation, several domain properties are stored in single precision. It is possible to use half precision for $k_{\text{ext,null}}$, f_{scat} and the optical properties without significant impact on the output. Using half precision reduces the storage requirement of these fields by a factor of two. As a result, more optical data fits into the same cache capacity, which can increase cache reuse when neighboring photons access nearby grid cells. It also reduces the amount of data transferred from global memory when these values are loaded. This is beneficial when memory latency and global memory bandwidth can be limiting factors.

4.2 Event Remapping Structure

All remapping implementations follow the same high-level execution pattern. First distinct events are chosen to split the program into different parts. Then depending on the implementation, there are at least enough queues or links created for each event that will hold photons for these events in shared memory. The queue or links will be filled with the maximum possible photons, this is done by classifying the photons and moving a full warp of the work is then moved to the registers of the warp. These photons are then processed and the photon state is either kept in the thread, written to shared memory, or swapped with another photon to again create a full warp of work. This continues until all photons have been processed, at some point partially filled warps will have to be processed as there are no more photons to be reset or evaluated. The structure of the program allows for multiple events and levels of divides, in Table 2 an overview of the levels, the actions of the event and simple complexity metrics can be seen.

Event	Sub-Event	Action	Lines	Global accesses
Coarse-cell advance	Surface hit	Accumulate surface contribution and reduce weight	75	2
	Top-of-domain exit	Accumulate top contribution and reset photon	32	1
	Cell crossing	Continue coarse traversal	12	0
fine-grid evaluation	Null collision	Continue tracing photon	1	0
	Scattering	Update direction and continue as diffuse photon	43	[1, 3]
	Reset	Launch new photon if work remains	27	0

Table 2: Photon events and outcomes used by the remapping implementations. To give some indication of complexity per sub-event, the number of code lines and the global accesses is included.

The most basic form of the remapping has the two main events. The first one being the coarse-cell advance. During the coarse-cell advance, a photon is moved through the acceleration grid using the null-collision extinction coefficient of the current coarse cell. This stage can be divided into three new events. The photon can cross into a neighboring coarse cell, hit the surface, or leave the domain through the top.

The second main event is the fine-grid evaluation. This event occurs when the photon interacts with some property and the local fine-grid optical properties have to be evaluated. The fine-grid evaluation again has three main events. If the sampled interaction is a null collision, the photon state is not physically changed and the photon continues tracing. If the sampled interaction is a real scattering event, the photon direction is updated and the photon changes from direct to diffuse radiation. At the start of the fine-grid evaluation the photons weight is updated, if it is below a certain threshold the photon might be reset.

The first remapping variants use two event categories: coarse-cell advance and fine-grid evaluation. This choice captures the dominant branch split in the kernel while keeping the number of queues small as a queue is necessary for at least each event. A small number of event categories is important because every additional remapped event requires additional queue capacity, shared-memory storage, and control logic. Because of this the initial implementations focus on grouping the two largest computational phases rather than remapping every possible sub-event separately.

4.3 Implementations

Below the main distinct thread-data-remapping versions for this thesis are explained in more detail.

4.3.1 Warp-stack

The warp level stack implementation has fixed-size event queues in shared memory, as illustrated in Figure 6. Each warp owns its own remapping storage, which makes it very simple to implement without any block-wide synchronization. When a thread finishes an event, the resulting photon state is written to the queue corresponding to the next event type. When a warp has finished processing photons the queues are checked for work and the event with the highest count is chosen and its photons are then moved from shared memory to registers. This implementation is fairly simple but the main limitation is that shared memory has to be reserved for each event queue. If there are E event types and each queue stores C capacity and S_{photon} is the number of bytes required to store one photon state. The shared-memory cost grows as

$$S_{\text{warp-stack}} \approx N_{\text{warps}} \cdot E \cdot C \cdot S_{\text{photon}} \tag{25}$$

.
This means that adding extra events can quickly increase shared memory and reduces occupancy or reduce the number of active photons as the capacity per queue will have to be lowered. When a reduction in event capacity happens, more partially filled warps will have to be evaluated as there are not enough active photons to fill a full warp. This is detrimental to performance.

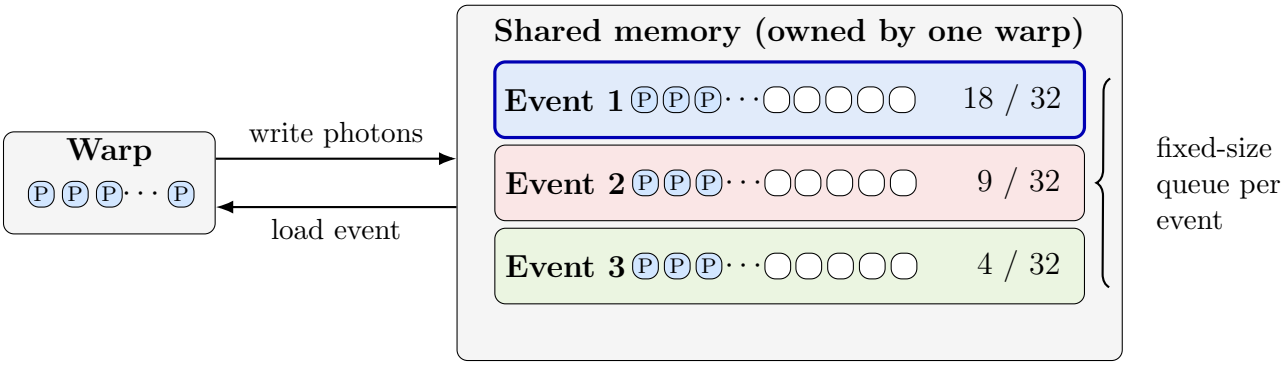


Figure 6: Warp-stack organization. Each warp owns fixed-size event queues in shared memory. Finished photons are written to shared memory according to their next event. When the warp needs more work, it loads photons from the queue with the highest occupancy.

4.3.2 Warp-link

The warp level link-based implementation is based on the link-queue thread-data-remapping approach used for Monte Carlo neutron transport by Cuneo and Bailey [27]. The purpose of this implementation is to test a more flexible remapping structure than a fixed event stack. Instead of storing one fixed queue per event, photon states are stored in shared-memory links. As shown in Figure 7, each link contains storage for one warp of photon states, a counter indicating how many slots are filled, an event type, and a pointer to the next link.

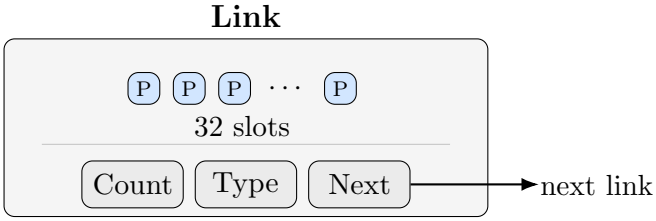


Figure 7: Structure of a link in the warp-link implementation. Each link stores 32 slots of photons together with its occupancy, event type, and a pointer to the next link.

A shared stash, shown in Figure 8, manages three types of links. First, it stores the head of the empty-link list, from which warps can acquire unused links. Second, it stores the head of the full-link list, which contains links that are ready to be processed by a warp. Third, it stores one partially filled link per event type. When a photon produces a given next event, it is inserted into the partial link for that event. Once the link reaches 32 photons, it is moved to the full-link list and becomes available for execution by a warp.

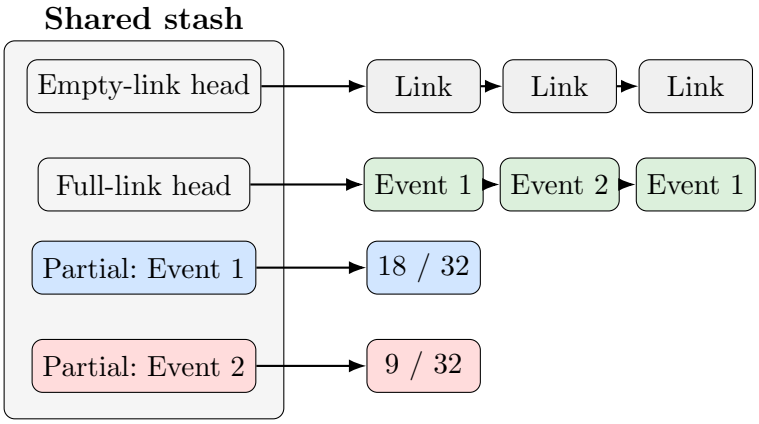


Figure 8: Shared stash and link organization in the warp-link implementation. The stash maintains the heads of the empty- and full-link lists and a partially filled link for each event type.

The advantage of the link-based structure is flexibility. The links can be assigned dynamically to events, so shared memory does not have to be statically divided equally over all event queues. The disadvantage is that the implementation requires more control logic for acquiring links, updating list heads, and moving links between empty, partial, and full states. This can increase instruction count and branch overhead compared to a simpler stack-based implementation. It is easier to add more events with the link based approach, but when no more shared memory is available it is not possible to add additional links. When more events are added but the number of links is not increased, the amount of active photons cannot be increased. This causes performance to deteriorate as the same number of photons will be divided over more links, causing more partially-filled links. When processing partial links more threads are inactive.

4.3.3 Block-stack

The block-level stack implementation extends the stack-based approach from warp level to block level. Instead of each warp maintaining independent queues, all warps in the thread block write photon states to shared block-wide event queues. The advantage is that it allows partial event groups produced by different warps to be combined. For example, if several warps each produce only a few photons for the same event, the block-level stack can merge these into a closer to full warp of work.

The main cost is synchronization. Because multiple warps can read from and write to the same queues, the implementation requires block-level coordination to avoid races between producers and consumers. Synchronization is needed after photons have been written to the queues and before photons are taken back for execution. Compared to the warp level stack, this increases barrier overhead and can leave warps idle while waiting for the rest of the block. This implementation is interesting because it tests if the improved ability to form full event warps outweighs the additional synchronization.

4.4 Overall Design Refinements

This section describes overall refinements that could be applied above mentioned remapping implementations.

4.4.1 Mixed Precision

Shared-memory usage is one of the main limiting factors for the remapping implementations. Each queued photon requires storage for its position, direction and multiple other properties. When full photon states are stored in shared memory, the number of photons that can be stored is directly limited by the size of the photon state. This can reduce occupancy, force smaller queues or link count. Mixed precision is introduced to reduce the storage cost of photon states. Fields that do not require full single-precision accuracy can be stored in half precision, which reduces the number of bytes per photon state. The effect on the byte size can be seen in Table 3, the total reduction in size is around 35%. This allows for larger queues or larger L1 Cache memory. On top of that, half values can be packed together with another half value, which is called half-packing. This reduces the number of total writes done and bytes written to and from shared memory during remapping.

Photon properties	**Purpose**	**Type**	**Initial size**	**New size**
Position (x, y, z)	position	3 Float	12 bytes	12 bytes
Direction (x, y, z)	direction	3 Float	12 bytes	6 bytes
tau	interaction probability	Float	4 bytes	2 bytes
$d_{\max}$	Distance to cell boundary	Float	4 bytes	2 bytes
d_n	Movement in cell	Float	4 bytes	2 bytes
weight(w)	Energy left	Float	4 bytes	2 bytes
$k_{\text{ext_null}}$	extinction of current cell	Float	4 bytes	2 bytes
kind	Direct or diffuse type	int8	1 byte	1 byte
Total			**45 bytes**	**29 bytes**

Table 3: Per-photon shared-memory storage before and after applying mixed precision.

4.4.2 Extra Caching

The baseline implementation benefits from caching several values that are reused across multiple steps. The remapping implementations can also use this caching, but the cost is higher because cached values must travel with the photon state. Before adding mixed precision it was also not possible as the shared memory cap was already reached. Extra caching is only beneficial when the avoided global-memory loads or calculations are more expensive than the additional shared-memory traffic and storage, as it can also reduce queue capacity or occupancy. Opposed to this in the baseline, adding a cached value mainly increases register usage. Table 4 shows the extra byte costs, further reducing sizes greatly impacted output.

Photon properties	Type	Added size
f_{noabs}	Float	4 bytes
f_{scat}	Half	2 bytes
Absorption accumulation in current cell	Float	4 bytes
Last integer position of photon	Int	4 bytes
Coarse grid position of photon	Int	4 bytes
Last kind of photon	Int8	1 bytes
Total		**19 bytes**

Table 4: Per-photon added fields to perform better caching explained in Section 4.1.6 and 4.1.7.

4.4.3 Vote Skipping

Remapping is only useful when threads in a warp produce different next events. If all active threads produce the same event or a large part of the warp is the same work, writing all photon states to shared memory and immediately reading them back adds overhead without reducing divergence. Vote skipping avoids this case by first checking whether all active lanes in a warp have selected the same next event. This can be implemented using warp level vote operations. If the vote shows that all active lanes agree on the next event, the warp continues without remapping. The photon states remain in registers and no shared-memory queue operation is performed. The benefit of vote skipping depends on the workload. In workloads such as clear-sky cases where many photons follow similar paths, vote skipping can remove a large fraction of unnecessary remapping operations. In highly divergent cloudy workloads, the vote will fail more often and the additional vote instruction may provide less benefit.

Majority-event threads	Evaluations	Fraction
32	4,646,359	4.22%
31	10,013,639	9.10%
30	11,595,726	10.54%
29	9,839,892	8.94%
28	7,652,909	6.96%
27–16	66,267,236	60.23%
Total	110,015,761	100.00%

Table 5: Distribution of warp event splits for the moderate-cloud workload(Section 4.12.2). The majority-event thread count indicates the number of threads with the majority event.

Table 5 shows the event split observed in the moderate-cloud workload(Section 4.12.2). In 4.22% of the cases the warp is already fully coherent and requires no remapping. This also motivates extending vote skipping beyond the fully coherent case, since a substantial fraction of warps contain only a small number of threads following the minority event.

4.5 Stack Design Refinement

When only two event types are remapped, separate queues for both events are not strictly necessary. The Head-or-Tail implementation used by Lin, Wang, and Liu [8] can be used to

have a single queue. The idea is to inserting one event type from the head of the queue and the other event type from the tail. The advantage of this design is that it reduces the queue memory by 50% for a two-event split, while having the same amount of active photons. The main disadvantage is that it cannot scale to more events, losing some flexibility. An illustration can be seen in Figure 9.

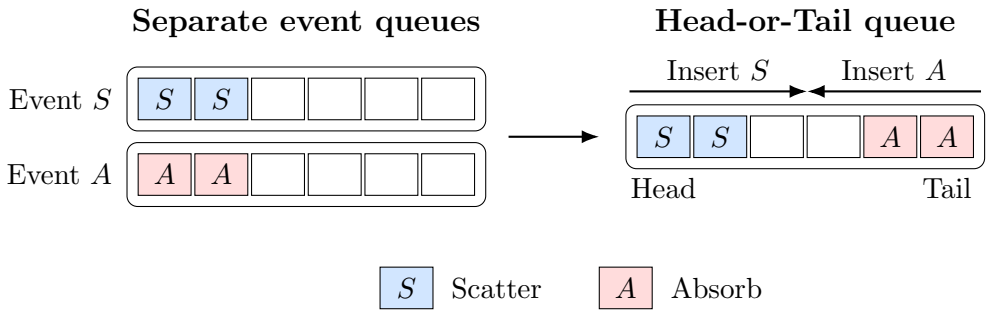


Figure 9: Comparison between separate event queues and the Head-or-Tail queue. Event S is inserted from the head of the shared queue, while event A is inserted from the tail. This allows both event types to share a single queue allocation.

4.6 Warp-stack Refinements

The Warp-stack allows additional optimizations because, unlike the block implementation, it does not require synchronization for each read or write stage.

4.6.1 Active Photons in Registers

A simple remapping implementation writes all photon states to shared memory after each event decision and reloads a full warp belonging to the selected event. While this ensures event grouping, it introduces unnecessary data movement when most lanes already agree on the next event. Instead, the majority event can remain in registers while only minority-event photons are moved to shared memory. The warp determines the majority event among its active photons, after which minority lanes place their photons in the corresponding queue and load a majority-event photon. This reduces shared-memory traffic and keeps more photon states in registers. It also increases the number of active photons by up to one warp (32), as register-resident photons are processed in addition to those stored in the event queues. The trade-off is additional control logic for determining the majority event and swapping photons between registers and queues.

4.6.2 Full-, Drain- and Compaction Phase

Remapping is most efficient when enough photons are available to form complete warps of the same event. As the number of active photons decreases, partially filled warps become unavoidable. Handling these requires additional checks and control logic that can slow the main execution path. To reduce this overhead, execution is divided into three phases.

During the full-warp phase, the implementation assumes enough work is available to fill complete event warps, avoiding much of the logic required for partial queues. The drain

phase starts when a partial warp is loaded for the first time.

The drain phase empties the remaining partially filled queues and processes leftover photons. Whenever a lane becomes empty, work is taken from a queue. The current majority event is evaluated when selecting work to keep the warp as coherent as possible. The drain phase could also finish all photons remaining in registers after the queues are empty. However, this can create a long tail of partially active warps when few photons remain, reducing occupancy and active warp count and causing inefficient execution.

To avoid this, execution transitions to the compaction phase once the shared-memory queues are emptied. The two event queues are reused as a single continuous compaction queue, with a counter for each event. At the start of each compaction iteration, photons held in registers are inserted at either end of the queue according to their event. The warps then synchronize and each warp loads photons from its assigned section of the queue. Warps with work continue processing, while empty warps exit. This repeats until no photons remain. Because a warp could otherwise receive photons from both events, extra queue slots are added to ensure that event groups never overlap within a warp. This requires only minor additional logic and shared-memory capacity. Figure 10 gives a more visual overview of the process.

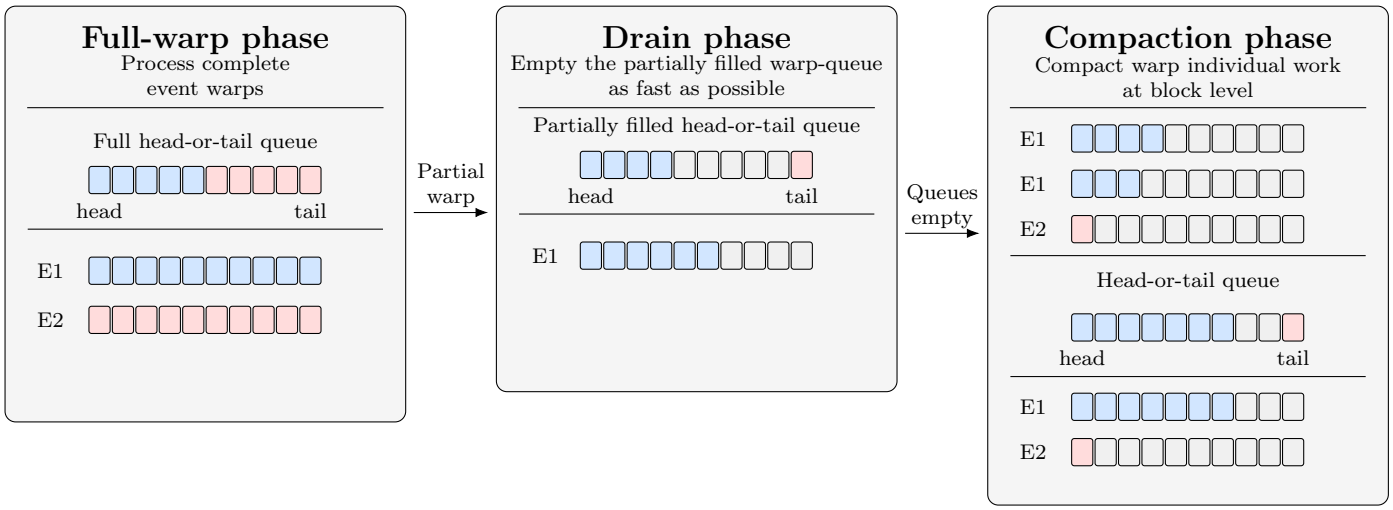


Figure 10: Simplified overview of the full-warp, drain, and compaction phases used by the warp-stack implementation. In the first phase only full warps are processed, until a partial warp is encountered after enough photon resets. The program then moves to the drain-phase where the queue is emptied as fast as possible so the memory can be used for the compaction-phase to compact multiple partial warps into less warps that are more filled with work.

4.7 Block-stack Refinement

For the first implementation all warps would reserve slots using atomics when grabbing work from the queue. When the queue is the exact size of the amount of threads it is faster to let each thread in the block take work from their own index in the queue, this way no atomics are necessary. The trade-off is that vote-skipping is not possible anymore as work

is now assigned based on index, the warp cannot simply skip as it might have to take work from the queue that might otherwise not be processed.

4.8 Setup

The main information of the machine can be found in Table 6. The different programs will first be tuned to this GPU to ensure optimal performance for each kernel. As there are quite a lot of different optimizations and combinations to be made, some final implementations with optimizations are evaluated. The clock frequency of the SMs is limited to 1.45 Ghz to get a more stable runtime performance and create a more fair comparison between program runs.

Component	**Specification**
Operating system	Ubuntu 26.04 LTS
CPU	AMD Ryzen Threadripper PRO 7945WX 12-Cores
System memory	123 GB
GPU	NVIDIA RTX 6000 Ada Generation
GPU memory	49140 MiB
NVIDIA driver	610.43.02
CUDA toolkit / nvcc	13.3
GCC	gcc (Ubuntu 15.2.0-16ubuntu1) 15.2.0

Table 6: Machine setup information.

4.9 Evaluated configurations

The previous sections describe the main remapping structures chosen and the refinements done. Below in Table 7 the final implementations are presented, with the added optimizations. Out of the tested implementations these were the most interesting or best performing, warranting additional investigation, in Section 6 the others will be discussed.

Configuration	Event split	Design choices
Base	None	None
Optimized base	None	Mixed precision Full caching Photon state kept in registers Work stealing
Warp-stack	Coarse/Fine	Mixed precision Full caching Vote skipping Head-or-tail queue Active photons kept in registers Full-warp and drain phases
Warp-link	Coarse/Fine	Mixed precision Limited caching (extinction coefficient) Vote skipping
Block-stack	Coarse/Fine	Mixed precision Full caching Head-or-tail queue
Scatter-split	Everything/Scatter	Mixed precision Full caching Vote skipping Head-or-tail queue Active photons kept in registers Full-warp and drain phases

Table 7: Evaluated configurations and their enabled optimizations and design choices.

4.9.1 Base

The base implementation is the original history-based ray-tracing kernel before the additional baseline optimizations and thread-data-remapping are applied. In this version, each thread processes one photon at a time and keeps the photon state assigned to the same thread until the photon is reset. This version is included to show the total improvement from the original implementation to the final optimized and remapped versions.

4.9.2 Optimized base

The optimized base implementation is the main reference version for evaluating the thread-data-remapping implementations. It includes the baseline optimizations described in Section 4.1. Photon states remain assigned to their original threads.

4.9.3 Warp-stack

The warp-stack is the most promising implementation. It uses the head-or-tail queue optimization for the two main events. It also includes the caching fields used by the optimized base implementation. In addition, the implementation uses vote skipping, a full-warp and drain-phase, and active photons in registers. This version represents the most optimized two-event remapping implementation.

4.9.4 Block-stack

The final block-stack version uses the head-or-tail queue optimization. Since the main remapping split contains two event categories, coarse-cell advance and fine-grid evaluation. This reduces the amount of shared memory required for event storage and leaves more room for cached photon fields. The goal is to evaluate whether block-level grouping can compensate for the additional synchronization required by the centralized queues.

4.9.5 Warp-link

This version evaluates whether the additional flexibility of a link-based remapping structure is beneficial for the photon ray-tracer. The final warp-link version uses mixed precision to reduce the shared-memory footprint of photon states. For caching it only caches the extinction coefficient of the current coarse-cell, as there is not more shared-memory budget. More extensive caching was not included as it reduced the occupancy. Vote skipping is enabled to avoid unnecessary remapping when all active lanes in a warp produce the same next event.

4.9.6 Scatter-split, Scattering vs Everything

A separate warp-stack variant was implemented to test a different event split. This version separates the main program path from scattering work. This is motivated by the observation that scattering accounts for 8% of the measured photon events (shown later in Section 4.12). This means that there is almost always at least 1 scatter event present in a warp. Adding scatter as its own event on top of the two earlier events adds another extra remap step to the scattering, being even more detrimental. This variant tests whether separating only scattering work reduces total amount of stalls in the program, even-though combining null-collision and coarse-grid processing might introduce new stalls.

4.10 Parameter tuning

Kernel Tuner [19] can tune parameters such as block and grid size, as well as program-specific definitions. The tuned kernel grid and block sizes can be found in Table 8. Tuning was done on the test file mentioned in Section 4.12.2.

Implementation	Grid sizes	Block size
Base	(1024)	512
Optimized Base	(32, 32)	1024
Warp-stack	(64,64)	512
Block-stack	(64,64)	512
Warp-link	(32, 32)	512
Scatter-split	(64,64)	512

Table 8: Tuned kernel grid and block sizes.

Five main parameters were tuned in the warp-stack implementation. First, the number of full-phase iterations before remapping can be adjusted, potentially allowing photons to remain in registers longer before being moved to shared memory. Second, the vote-skipping

threshold from Section 4.4.3 determines how many identical work events within a warp are required to skip voting. Third, the drain phase can continue until multiple threads are inactive instead of taking new work as soon as one thread has no active photon. Fourth, the compaction interval determines how many iterations are performed before synchronizing the block and moving all work to the compaction queue. Finally, the shared-memory carve-out can be tuned. On many NVIDIA GPUs, shared memory and L1 cache share on-chip memory, so allocating more shared memory can reduce L1 capacity and increase accesses served by L2 or global memory. More shared memory allows larger queues and more active remapping storage, while more L1 cache can improve the cache hit rate. During evaluation, preferring more shared memory was always beneficial for the remapping implementations. The tuned parameters for the warp-stack implementation are shown in Table 9.

Description	**Value**
Iterations between remapping in the full phase.	1
Minimum number of inactive lanes before refilling during the drain phase.	1
Iterations between block-wide compaction.	4
Threshold used to skip remapping for sufficiently coherent warps.	2

Table 9: Tuned parameters of the warp-stack implementation.

4.11 Metrics

Multiple metrics are used to evaluate the implementations. The primary metric is runtime, averaged over five runs and reported in milliseconds as runtime was relatively stable between runs. The remaining metrics are used to explain runtime differences.

Warp divergence is evaluated using the average number of active and not-predicated-off threads per warp. Active threads are lanes still participating in processing, for example threads that have not yet exited the program. Not-predicated-off threads are lanes that are active that are not predicated-off, so they are doing meaningful work. Higher values therefore generally indicate more useful work per warp.

Memory-system effects are evaluated using L1 and L2 cache hit rates and memory traffic between cache levels/shared memory and the registers. DRAM throughput is also reported, although higher throughput does not necessarily imply better performance, as it may result from increased memory traffic.

Resource usage is evaluated using registers per thread, shared memory per block, achieved and theoretical occupancy, and the average number of active warps per SM. These metrics show how resource requirements affect the number of warps that can execute concurrently.

Finally, instruction count, Warp Cycles Per Instruction (CPI), and the dominant warp stall states are inspected. Warp CPI is the average number of cycles between instructions issued by a warp and consists of the reported stall cycles plus one cycle in which the instruction is selected. Stall states indicate why a warp could not issue its next instruction. The most relevant states encountered in the results are defined in Table 10.

Warp state	Meaning
Long scoreboard	Waiting for a dependency on a global-memory, local-memory or L2 cache operation.
Short scoreboard	Waiting for a dependency with shared-memory or other memory-input/output operations.
Wait	Waiting for the result of an earlier instruction.
Barrier	Waiting for other warps in the block to reach a synchronization barrier.
Branch resolving	Waiting for the target of a branch to be resolved.
No instruction	No instruction is currently available for the warp, for example because of instruction-fetch latency.
Not selected	The warp is ready to issue, but another eligible warp is selected by the scheduler.

Table 10: Definitions of the main warp states used in the results.

4.12 Test files

To evaluate the performance of the different implementations compared to the base there are multiple different workloads to evaluate the impact on different work distributions. The fine-grid and coarse-grid have dimensions $(38 \times 384 \times 129)$, $(48 \times 48 \times 20)$, respectively.

4.12.1 Clear sky

The clear sky workload includes no clouds, causing most photons to travel in the sun direction on the coarse-cell grid, hitting the surface and traveling back up and exiting the domain. This workload has a majority of coarse-cell advance events as can be seen in Table 11. This workload can work as a good indication for the overhead of remapping, as most photons will be taking the same path. There is still slight scattering due to gas and aerosol scattering. Only 245 of the fine-grid events are resets as those only happen when a photon loses all of their energy in that path, but most resets happen on top-of-domain exit.

Event type	Sub-event	Count	Overall share
Event 1: coarse-cell advance	Surface hit	37,564,571	2.71%
	Top-of-domain exit	7,660,593	0.55%
	Cell crossing	992,658,363	71.68%
	Total	**1,037,883,527**	**74.94%**
Event 2: fine-grid evaluation	Null collision	346,178,814	25.00%
	Scattering	859,495	0.06%
	Reset	245	0.00%
	Total	**347,038,554**	**25.06%**
Overall total		**1,384,922,081**	**100.00%**

Table 11: Overall distribution of photon events in the clear-sky workload.

4.12.2 Moderate clouds

The moderate clouds workload includes multiple clouds causing a fine-grid major event distribution. A larger part of photons are being scattered, in Table 12 the full distribution

can be seen. This workload will show the effect of thread divergence in fine-grid evaluations as actual scattering is around 10%, this causes most of the fine-grid processing warps to include 1 or 2 threads with scattering.

Event type	Sub-event	Count	Overall share
Event 1: coarse-cell advance	Surface hit	34,773,746	0.93%
	Top-of-domain exit	9,890,383	0.27%
	Cell crossing	1,078,240,687	28.98%
	Total	**1,122,904,816**	**30.18%**
Event 2: fine-grid evaluation	Null collision	2,307,145,583	62.01%
	Scattering	290,684,660	7.81%
	Reset	122	0.00%
	Total	**2,597,830,365**	**69.82%**
Overall total		**3,720,735,181**	**100.00%**

Table 12: Overall distribution of photon events in the moderate clouds workload.

Figure 11 shows an example of the surface output, the lighter the color the more light is reaching that part of the surface. The majority of photons are still traveling straight to the surface, but the clear dark spots are parts where most light cannot reach due to clouds.

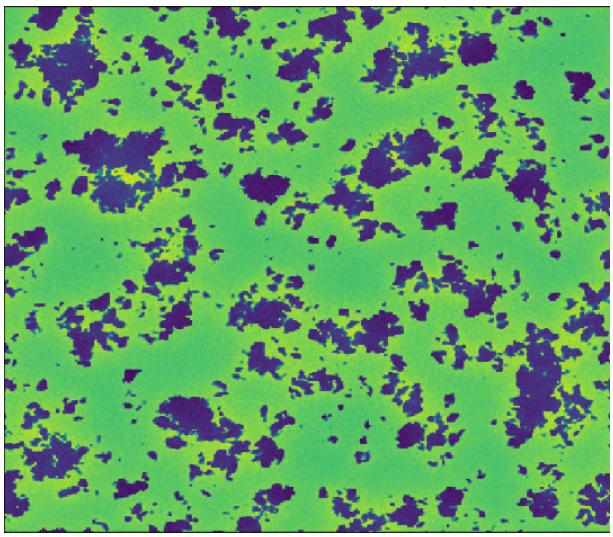


Figure 11: Example output of moderate cloud workload.

4.12.3 Scatter workload

The scatter workload is a synthetic, interaction-heavy test case designed to evaluate the remapping implementations under an extreme amount of fine-grid work. The cloud optical depth and coverage were artificially increased, which makes it an unrealistic weather domain. Instead, it serves as a stress test in which photons have substantially more null collisions and physical scattering events, resulting in longer and more variable photon paths. In Table 13 a full overview is given for the distribution of the events.

Event type	Sub-event	Count	Overall share
Event 1: coarse-cell advance	Surface hit	17,080,598	0.11%
	Top-of-domain exit	24,028,802	0.16%
	Cell crossing	1,426,892,689	9.44%
	Total	**1,468,002,089**	**9.71%**
Event 2: fine-grid evaluation	Null collision	10,180,730,817	67.33%
	Scattering	3,471,301,043	22.96%
	Reset	53	0.00%
	Total	**13,652,031,913**	**90.29%**
Overall total		**15,120,034,002**	**100.00%**

Table 13: Overall distribution of photon events in the high-scattering cloud workload.

The workload is therefore useful for investigating whether thread-data-remapping remains effective when scattering is a more common branch and when fine-grid processing dominates the total execution. In Figure 12, the resulting distribution of light reaching the surface is shown. There are only a couple of concentrated parts of light, most of the surface only has slight diffuse light.

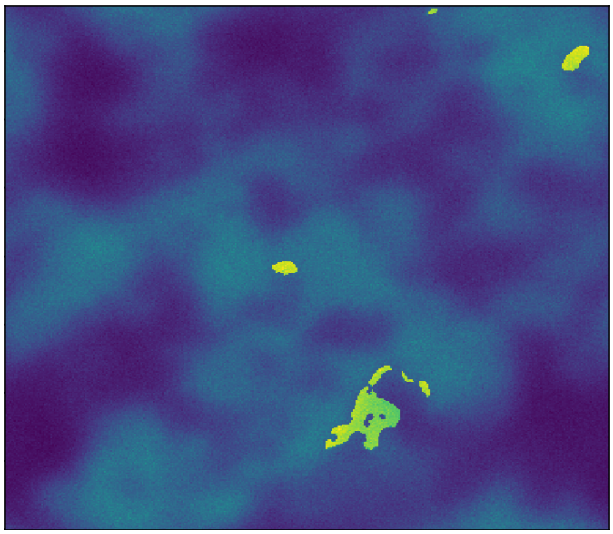


Figure 12: Example surface output of synthetic scatter cloud workload.

4.13 Result validation

Because Monte Carlo simulations contain random noise, algorithm changes affect the output, making correctness difficult to validate using output error alone. Instead, the absolute difference between the base program and each implementation is plotted and inspected for significant deviations. Slight differences in scatter patterns are acceptable, but otherwise should consist of pure random noise. Figure 13 shows such a comparison, where most differences appear as random noise in the surface layer. This procedure can be applied to each output layer.

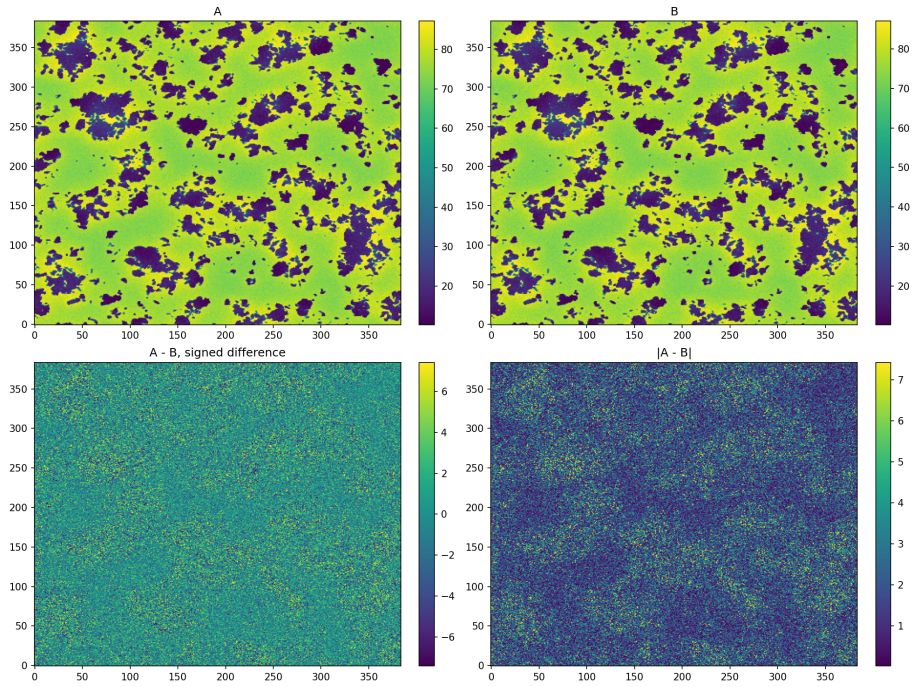


Figure 13: Visual comparison of the surface-layer, showing random and scattering noise in the differences. A is a run of the warp-stack program and B is a run of the original base program.

5 Results

Below are the results for testing the workloads on the different implementations. A short discussion is given for each metric and workload, which will be elaborated on in the Section 6.

5.1 Clear Sky

The clear-sky workload serves as a useful baseline because its dominant event paths are relatively inexpensive. The potential benefit of remapping is therefore limited, making its overhead more visible.

5.1.1 Runtime and speed-up

Figure 14 shows that the optimized base achieves the lowest runtime at 45.03 ms, corresponding to a 4.16× speed-up over the original implementation. The Warp-stack is the closest remapping implementation, but remains 9.6% slower at 49.37 ms. The other remapping variants fall further behind, showing that remapping provides no runtime benefit for the relatively coherent clear-sky workload.

5.1.2 Divergence and Event Coherence

Table 14 shows that thread-data-remapping substantially improves warp coherence in the clear-sky workload. For the Warp-stack, the average number of active threads increases from 12.77 to 17.26 relative to the optimized base, while the number of non-predicated-off

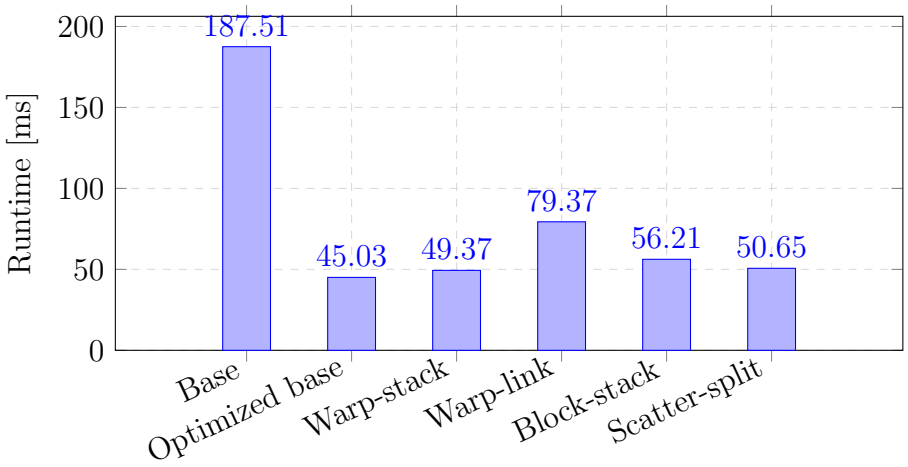


Figure 14: Runtime of the evaluated implementations for the clear-sky workload.

Implementation	Active threads / warp	Not predicated off threads / warp
Base	10.31	9.31
Optimized base	12.77	11.16
Warp-stack	**17.26**	15.11
Block-stack	16.24	14.34
Warp-link	16.70	**15.19**
Scatter-split	14.10	12.16

Table 14: Warp execution statistics for the clear-sky workload.

threads increases from 11.16 to 15.11, both improvements of approximately 35%. Despite this increase in useful lane utilization, the Warp-stack remains slower than the optimized base.

5.1.3 Memory and Cache Behavior

Implementation	Throughput [GB/s]	L1 hit rate [%]	L2 hit rate [%]	L2 to L1 [GB]	L1 to registers [GB]	SMEM R/W [GB]
Base	202.88	37.24	62.50	57.20	0.63	0.00/0.00
Optimized base	107.05	**67.43**	82.98	**11.38**	0.64	0.00/0.00
Warp-stack	93.19	62.95	87.77	20.34	1.09	0.60/0.66
Block-stack	71.89	62.10	**89.50**	21.97	1.11	0.80/1.31
Warp-link	51.35	54.04	86.86	19.07	**0.327**	1.47/2.02
Scatter-split	92.31	63.85	87.59	19.79	1.15	**0.059/0.040**

Table 15: Memory and cache metrics for the clear-sky workload. Shared-memory traffic is reported as read/write traffic.

Table 15 shows that the baseline optimizations substantially improve memory behavior, reducing L2-to-L1 traffic from 57.20 GB to 11.38 GB while increasing both L1 and L2 hit rates. The Warp-stack transfers 20.34 GB from L2 to L1 and additionally introduces shared-memory traffic for photon-state movement. The Warp-link shows the strongest memory overhead, with a lower L1 hit rate and greater shared-memory traffic than the

other warp-local remapping implementation. The consistently lower L1 hit-rate is caused by the smaller total cache due to the shared-memory carveout.

5.1.4 Occupancy

Implementation	Registers	SMEM/block [KB]	Achieved/Theoretical occupancy [%]	Active warps/SM
Base	68	1.02	31.33/33.33	15.03
Optimized base	55	1.02	54.43/66.67	26.12
Warp-stack	56	27.27	42.66/66.67	20.47
Block-stack	**48**	27.65	**63.01/66.67**	**30.24**
Warp-link	63	63.48	25.19/33.33	12.09
Scatter-split	56	27.27	40.71/66.67	19.54

Table 16: Occupancy metrics for the clear-sky workload.

As shown in Table 16, the optimized base increases the number of active warps per SM from 15.03 to 26.12, alongside a reduction in register usage from 68 to 55 registers per thread. The Warp-stack reaches 20.47 active warps per SM, while the Block-stack achieves the highest occupancy at 30.24 active warps per SM. Despite its high occupancy, the Block-stack does not achieve the lowest runtime. Occupancy provides additional opportunities to hide latency, but does not by itself determine performance.

5.1.5 Instruction count and main stall reasons

Implementation	Instructions [Ginst]	Warp CPI	Largest stall	Second-largest stall	Third-largest stall	Other
Base	35.72	16.32	Long scoreboard (10.64)	Wait (2.26)	Branch resolving (0.86)	1.56
Optimized base	**25.31**	10.28	Wait (2.20)	Not selected (1.84)	Long scoreboard (1.59)	3.65
Warp-stack	26.67	8.81	Wait (2.15)	Not selected (1.21)	Long scoreboard (1.05)	3.40
Block-stack	32.21	14.57	Barrier (5.38)	Wait (2.24)	Not selected (1.17)	4.78
Warp-link	34.46	**7.67**	Wait (2.52)	Short scoreboard (1.17)	Branch resolving (1.07)	1.91
Scatter-split	28.67	9.37	Wait (2.15)	Long scoreboard (1.38)	Not selected (1.20)	3.64

Table 17: Executed instruction count and dominant warp stall states for the clear-sky workload. Stall values are reported as average warp cycles per executed instruction.

Table 17 shows that the baseline optimizations reduce both the executed instruction count and the contribution of long-scoreboard stalls. The Warp-stack further reduces Warp CPI relative to the optimized base, but executes approximately 5.4% more instructions. The Warp-link combines an even lower Warp CPI with a substantially higher instruction count, while the Block-stack stands out because barrier stalls dominate its Warp CPI. Reducing stall cycles per instruction is not sufficient, to counter act the additional work.

5.2 Moderate clouds

The moderate-cloud workload represents a realistic distribution of photon events and is used to evaluate whether remapping becomes more effective under increased divergence.

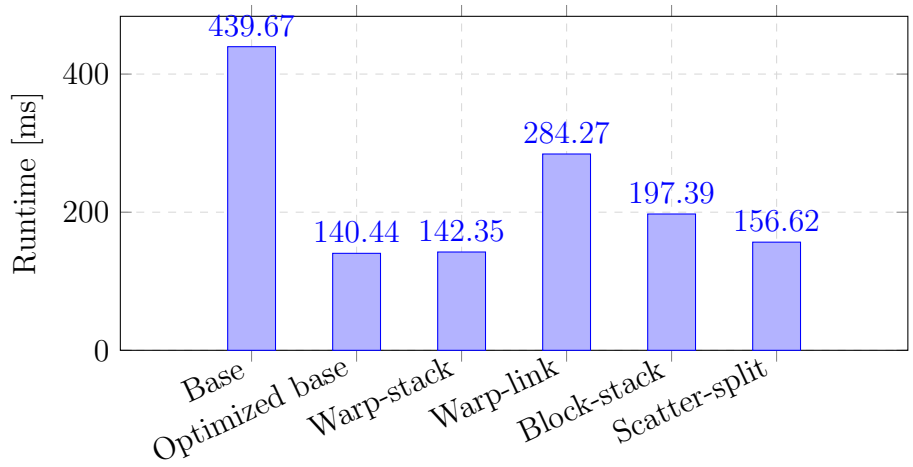


Figure 15: Runtime of the evaluated implementations for the moderate clouds workload.

5.2.1 Runtime and speed-up

Figure 15 shows that the optimized base remains the fastest implementation at 140.44 ms, corresponding to a 3.13× speed-up over the original base. The Warp-stack reaches 142.35 ms and is only 1.4% slower, making the performance difference considerably smaller than in the clear-sky workload. The remaining remapping implementations are clearly slower, with the Scatter-split, Block-stack, and Warp-link respectively 11.5%, 40.6%, and 102.4% slower than the optimized base.

5.2.2 Divergence and Event Coherence

Implementation	Active threads / warp	Not predicated off threads / warp
Base	9.30	8.57
Optimized base	11.70	10.36
Warp-stack	16.91	15.30
Block-stack	**17.64**	**15.97**
Warp-link	16.80	15.51
Scatter-split	13.48	12.12

Table 18: Warp execution statistics for the moderate-cloud workload.

Table 18 shows that the benefit of remapping for warp coherence becomes more noticeable in the moderate-cloud workload. The Block-stack achieves the highest lane utilization, increasing active threads per warp from 11.70 to 17.64 and non-predicated-off threads from 10.36 to 15.97 relative to the optimized base. The Warp-stack achieves similar improvements at 16.91 and 15.30 threads per warp, respectively. Although the remapping implementations execute substantially more coherently, this improvement still does not translate into a lower runtime.

5.2.3 Memory and Cache Behavior

Table 19 shows that the optimized base again requires the least L2-to-L1 traffic, at 37.48 GB. For the Warp-stack this increases to 58.18 GB, approximately 55% more. The Warp-link

Implementation	Throughput [GB/s]	L1 hit rate [%]	L2 hit rate [%]	L2 to L1 [GB]	L1 to registers [GB]	SMEM R/W [GB]
Base	193.08	49.85	80.14	179.46	1.73	0.00/0.00
Optimized base	100.24	**59.05**	88.44	**37.48**	1.43	0.00/0.00
Warp-stack	76.38	51.10	91.76	58.18	1.50	1.32/1.28
Block-stack	56.38	51.27	91.90	57.89	1.51	1.94/2.59
Warp-link	42.99	39.31	**95.35**	99.47	0.77	3.41/4.67
Scatter-split	73.90	49.52	91.32	55.38	1.55	**0.34/0.31**

Table 19: Memory and cache metrics for the moderate-cloud workload. Shared-memory traffic is reported as read/write traffic.

shows the largest memory cost, with 99.47 GB transferred from L2 to L1, a substantially lower L1 hit rate, and the highest shared-memory traffic. The memory results therefore show that the improved warp coherence is accompanied by increased data movement. As work is more divergent, more work has to be moved.

5.2.4 Occupancy

Implementation	Registers	SMEM/block [KB]	Achieved/Theoretical occupancy [%]	Active warps/SM
Base	68	1.02	32.14/33.33	15.43
Optimized base	55	1.02	**62.69/66.67**	**30.09**
Warp-stack	56	27.27	51.81/66.67	24.87
Block-stack	48	27.65	61.11/66.67	29.33
Warp-link	63	63.48	31.13/33.33	14.94
Scatter-split	56	27.27	50.93/66.67	24.45

Table 20: Occupancy metrics for the moderate-cloud workload.

Table 20 shows that the Block-stack retains the highest occupancy among the remapping implementations, reaching 61.11% achieved occupancy and 29.33 active warps per SM. The Warp-stack and Scatter-split reach approximately 51% occupancy and 24 active warps per SM. The Warp-link is the most resource-constrained implementation due to the shared-memory requirement, with 31.13% achieved occupancy and 14.94 active warps per SM.

5.2.5 Instruction count and main stall reasons

Implementation	Instructions [Ginst]	Warp CPI	Largest stall	Second-largest stall	Third-largest stall	Other
Base	124.42	10.29	Long scoreboard (4.65)	Wait (2.27)	Branch resolving (0.91)	1.46
Optimized base	70.96	9.80	Wait (2.18)	Not selected (1.95)	Long scoreboard (1.58)	3.09
Warp-stack	**67.60**	9.65	Wait (2.09)	Long scoreboard (1.98)	Not selected (1.43)	3.15
Block-stack	75.05	14.09	Barrier (5.25)	Wait (2.07)	Not selected (1.26)	4.51
Warp-link	91.83	**7.70**	Wait (2.36)	Long scoreboard (1.14)	Short scoreboard (1.06)	2.14
Scatter-split	73.87	9.51	Wait (2.15)	Long scoreboard (1.65)	Not selected (1.35)	3.36

Table 21: Executed instruction count and dominant warp stall states for the moderate-cloud workload. Stall values are reported as average warp cycles per executed instruction.

Table 21 shows that the Warp-stack executes approximately 4.7% fewer instructions than the optimized base while maintaining a similar Warp CPI. This is an important change from the clear-sky workload, where the Warp-stack executed more instructions, and shows that improved coherence is now sufficient to reduce the total amount of executed work. Nevertheless, its runtime remains 1.4% higher, indicating that the remaining remapping costs are still sufficient to offset this reduction. The Block-stack is dominated by barrier stalls, while the Warp-link combines a low Warp CPI with a substantially higher instruction count.

5.3 Scatter workload

The scatter workload is a synthetically modified case with a higher scattering frequency. It is used to evaluate remapping under more scattering-intensive conditions.

5.3.1 Runtime and speed-up

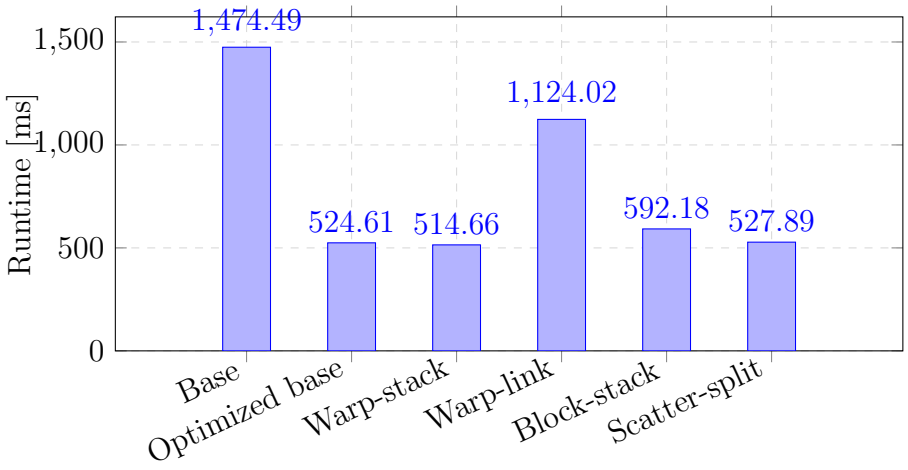


Figure 16: Runtime of the evaluated implementations for the scatter workload.

Figure 16 shows that the Warp-stack achieves the lowest runtime for the synthetic scatter workload at 514.66 ms, outperforming the optimized base by approximately 1.9%. This is the only evaluated workload in which thread-data-remapping produces a runtime improvement over the optimized history-based implementation. The Scatter-split remains close to the optimized base at 527.89 ms, whereas the Warp-link and Block-stack remain substantially slower.

5.3.2 Divergence and Event Coherence

Table 22 shows that remapping again substantially improves warp coherence, with the Block-stack achieving the highest lane utilization at 19.11 active and 17.54 non-predicated-off threads per warp. The Warp-stack increases these metrics from 13.17 and 11.78 for the optimized base to 16.77 and 15.33. Although the Block-stack produces the greatest coherence, the Warp-stack achieves the lowest runtime.

Implementation	Active threads / warp	Not predicated off threads / warp
Base	11.03	10.23
Optimized base	13.17	11.78
Warp-stack	16.77	15.33
Block-stack	**19.11**	**17.54**
Warp-link	17.78	16.56
Scatter-split	15.31	14.07

Table 22: Warp execution statistics for the scatter workload.

5.3.3 Memory and Cache Behavior

Implementation	Throughput [GB/s]	L1 hit rate [%]	L2 hit rate [%]	L2 to L1 [GB]	L1 to registers [GB]	SMEM R/W [GB]
Base	136.55	54.21	85.01	792.62	4.75	0.00/0.00
Optimized base	42.56	**58.20**	93.99	**223.08**	3.56	0.00/0.00
Warp-stack	38.85	49.35	95.59	296.21	3.72	**2.66/2.47**
Block-stack	40.00	47.87	94.60	301.92	3.63	7.09/8.38
Warp-link	22.27	26.44	**97.99**	889.38	**3.15**	13.08/18.22
Scatter-split	40.78	38.43	94.88	269.25	3.29	4.25/3.96

Table 23: Memory and cache metrics for the synthetic scatter workload. Shared-memory traffic is reported as read/write traffic.

Table 23 shows that the optimized base retains the lowest L2-to-L1 traffic at 223.08 GB. The Warp-stack increases this traffic by approximately 33% and additionally performs shared-memory transfers for remapping, yet still achieves the lowest runtime. The Warp-link is the clear outlier, with 889.38 GB of L2-to-L1 traffic, an L1 hit rate of only 26.44%, and substantially more shared-memory traffic than the other implementations. The scatter workload therefore shows that additional memory traffic can be outweighed by the reduction in divergent execution when the workload is sufficiently demanding.

5.3.4 Occupancy

Implementation	Registers	SMEM/block [KB]	Achieved/Theoretical occupancy [%]	Active warps/SM
Base	68	1.02	31.99/33.33	15.36
Optimized base	55	1.02	62.69/66.67	30.09
Warp-stack	56	27.27	50.22/66.67	24.11
Block-stack	48	27.65	**63.72/66.67**	**30.58**
Warp-link	63	63.48	32.08/33.33	15.40
Scatter-split	56	27.27	50.22/66.67	24.11

Table 24: Occupancy metrics for the scatter workload.

Table 24 shows that the Warp-stack achieves the lowest runtime despite reducing the number of active warps per SM from 30.09 for the optimized base to 24.11. The Block-stack reaches slightly higher occupancy than the optimized base but its runtime is substantially slower. The scatter workload therefore provides a clear example that higher occupancy does not necessarily correspond to lower runtime.

5.3.5 Instruction count and main stall reasons

Implementation	Instructions [Ginst]	Warp CPI	Largest stall	Second-largest stall	Third-largest stall	Other
Base	517.27	10.46	Long scoreboard (4.92)	Wait (2.21)	Branch resolving (0.93)	1.40
Optimized base	284.35	10.51	Long scoreboard (3.02)	Wait (2.09)	Not selected (1.87)	2.53
Warp-stack	**261.94**	10.33	Long scoreboard (3.49)	Wait (2.04)	Not selected (1.19)	2.61
Block-stack	286.87	14.01	Barrier (4.66)	Wait (1.95)	Long scoreboard (1.95)	4.45
Warp-link	391.42	10.93	Long scoreboard (4.21)	Wait (2.25)	Short scoreboard (1.48)	1.99
Scatter-split	290.58	**9.20**	Wait (2.05)	Long scoreboard (1.90)	Not selected (1.32)	2.93

Table 25: Executed instruction count and dominant warp stall states for the synthetic scatter workload. Stall values are reported as average warp cycles per executed instruction.

Table 25 shows that the Warp-stack executes approximately 7.9% fewer instructions than the optimized base while maintaining a similar Warp CPI. This is the largest instruction-count reduction observed for the Warp-stack and coincides with the workload in which it becomes faster than the optimized base. The Scatter-split achieves the lowest Warp CPI but executes more instructions than the Warp-stack, while the Block-stack remains dominated by barrier stalls. The results show that the reduction in executed work becomes more substantial as the workload becomes more scattering-intensive.

6 Discussion

Thread-data-remapping consistently improves warp coherence, but that improved coherence does not necessarily result in a lower runtime. The final performance depends on whether the reduction in divergent execution compensates for the additional instructions, shared-memory traffic, queue management, reduced cache capacity, and synchronization introduced by remapping. The individual implementations and their main limitations are discussed below.

6.1 Optimized base

The optimized base shows that a large part of the original runtime was not caused by branch divergence. Reducing memory traffic and repeated calculations by caching more in registers, lowering register usage, improving locality, and balancing photon work between threads already produces a substantial performance improvement. The accompanying reduction in memory stalls and increase in resident warps show that improving the efficiency of the existing history-based execution is more important than immediately reorganizing photon work.

The effect of these optimizations becomes smaller as the workload contains more fine-grid evaluations and scattering. This indicates that the remaining runtime increasingly comes from irregular control flow that cannot be removed through the baseline optimizations alone. The optimized base therefore provides both a strong reference for evaluating remapping and an indication of the workloads in which improved warp coherence may become useful.

6.2 Warp-stack

The Warp-stack provides the best balance between improved coherence and remapping overhead. Its relative performance improves consistently as the workload becomes more divergent, eventually becoming 1.9% faster than the optimized base in the synthetic scatter workload. This trend is also reflected in the executed instruction count: remapping initially introduces additional work in the clear-sky workload, but reduces the amount of executed work once fine-grid processing and scattering become common.

The moderate-cloud workload is particularly interesting because it is close to the break-even point. Here, the Warp-stack already executes fewer instructions than the optimized base and achieves comparable instruction latency, showing that the reduction in divergent execution is sufficient to recover the direct instruction overhead of remapping. It nevertheless remains slightly slower overall. The remaining cost comes primarily from moving photon states through shared memory and from the resource consequences of reserving this storage. In particular, the large shared-memory allocation reduces the capacity available to the L1 cache and lowers the number of resident warps compared with the optimized base.

The Warp-stack limits these costs by keeping remapping local to a warp. It avoids block-wide synchronization, uses a Head-or-Tail queue to share storage between the two main events, and keeps the majority event in registers whenever possible. When testing a version on the moderate cloud workload where remapping is disabled during the full-warp phase, the runtime increased to 150.04 ms over 5 runs. Which further shows that remapping itself provides a measurable benefit, the problem is that much of this benefit is consumed by the surrounding queue, drain, and compaction overhead.

The synthetic scatter workload finally crosses the performance of the optimized base version. Its much larger fraction of fine-grid and scattering work makes divergent execution expensive enough that the reduction in executed work outweighs the additional memory and scheduling costs. This suggests that remapping is most attractive when the separated event paths are sufficiently expensive. However, in any realistic scenario, the small or negative runtime improvement is difficult to justify given the substantial increase in implementation complexity.

6.3 Warp-link

The Warp-link demonstrates that improved coherence is not sufficient when the scheduling structure itself becomes too expensive. Despite producing coherent warps and low Warp CPI values, it performs substantially worse than the Warp-stack. This indicates that its main limitation is not unsuccessful remapping, but the amount of additional work required to manage the links.

Each link must be acquired, assigned an event, moved between empty, partial, and full states, and eventually returned. Such flexibility can be useful when many event types compete dynamically for storage, but the final ray-tracer primarily remaps only two events. For this case, the Head-or-Tail structure provides the required grouping with much less bookkeeping. The link structure also increases pressure on the memory hierarchy. Its shared-memory requirement limits the number of resident warps, while the reduced

L1 capacity results in substantially more traffic through L2 as the workload becomes more demanding. The link-based approach was designed for transport problems in which dynamically growing work makes flexible storage particularly valuable. In this photon ray-tracer, the number of active photons is bounded and shared-memory capacity is already constrained by the size of the photon state. The additional flexibility therefore introduces cost without providing a corresponding advantage.

6.4 Block-stack

The Block-stack explored if increasing the pool of photons available for grouping at the cost of synchronization between warps is worth the cost. Sharing event queues across an entire block does improve coherence, and this advantage becomes larger as the event distribution becomes more irregular. The Block-stack therefore demonstrates that pooling work across warps is effective at forming more complete event groups.

However, the additional coherence does not translate into better runtime because accessing the shared queues requires block-wide coordination. Barrier stalls become the dominant limitation, forcing faster warps to wait for slower warps before processing can continue. The Block-stack can therefore achieve both high occupancy and the highest measured warp coherence while still being slower than the Warp-stack.

This result highlights an important limitation of increasing the scope of remapping. A larger scheduling pool provides more opportunities to construct coherent warps, but also requires stronger synchronization. For the event costs in this ray-tracer, the improvement obtained from block-wide grouping is smaller than the synchronization cost required to achieve it. Warp-local remapping consequently provides a better compromise.

6.5 Scatter-split

The Scatter-split investigates whether remapping should target the most expensive divergent branch rather than the two dominant execution phases. Separating scattering successfully reduces stalls associated with the scattering path, and this becomes increasingly effective as scattering becomes more frequent. However, it does not provide better overall performance than the standard Warp-stack.

The reason is that event selection affects more than the cost of the isolated branch. By grouping scattering separately, the Scatter-split leaves coarse-cell advancement and non-scattering fine-grid evaluation together. Divergence between these two common paths therefore remains, which explains why its overall warp coherence is lower than that of the normal coarse/fine split. At low scattering frequencies there is also too little scattering work to recover the additional remapping cost.

The synthetic scatter workload brings the Scatter-split close to the optimized base, showing that separating an expensive branch becomes more attractive as that branch becomes more frequent. Nevertheless, the standard Warp-stack remains faster because its event split captures a larger fraction of the overall control-flow divergence.

6.6 Other implementations

Some implementations that were tested but not included in the results evaluation are discussed in more detail below.

6.6.1 Block-link

One other implementation tested for the link implementation was at block level. But this version suffers from the extra synchronization checks necessary when leaders in warps attempt to request links. When it fails a warp has to retry to gain another link, again being very expensive. This is due to the usage of compare and swap operations as otherwise multiple warps could request the same link.

6.6.2 Extra caching

When testing the extra caching in the Warp-stack version, it showed that adding all the caching present in the Optimized base version, that the gain was significant but limited by the fact that there is almost 40% more data to write and read. The actual speed-up was about 13%. When removing the actual use of the caching and keeping only the read and writes the program slows down 28%, so most of the caching speed-up is lost to the extra required shared memory read and writes.

6.6.3 Shared-Memory Indirection

The tested remapping implementations store the full photon state inside each queue/link slot. When a photon is selected for execution, all required data can be loaded coalesced from the queue into the thread registers. It is also possible for the queue to only store the index to a slot in the photon-state array somewhere in shared-memory, this way only required fields can be loaded and updated instead of the full photon state. The queue entry can then be reduced to a small integer value, while the full photon state is stored only once. For certain events this allows for them to only update fields of the photon that changed, but loads are now scattered. For the Block-stack version it also provided the possibility for a simple ready state, when the index was added to the queue it would mean that the data in the photon state array was properly updated. Unfortunately, when testing indirection it resulted in even worse performance, attributed to worse memory layout and the current events requiring almost all the photon state. Making the gain of not having to update some fields negligible.

6.6.4 Multiple Event Levels

An important test was to try different event splits as the main event categories contain smaller sub-events. For example, the fine-grid evaluation can result in a null collision or a real scattering event. These sub-events have different computational costs and memory-access patterns. Scattering is especially expensive because it requires sampling a new direction and may load additional optical properties. This creates a second level of divergence inside the fine-grid event. This resulted in testing different splits than only the two main events, but also between selected sub-events. The expectation was an improvement in event coherence, but that it would also increase the number of queues and the amount of remapping work. When testing the increased amount of possible events it showed poor

performance for all versions as it increased the required amount of shared-memory, but another reason is the complexity of the sub-events. Most of the events are not complex enough to warrant a full remapping, scattering is close to being complex enough as could be seen in the results, but for all of the other sub-events the best approach was to process them immediately.

6.6.5 Reset Handling

A photon can be reset through several paths in the ray-tracing loop. For example, a reset can occur when a photon leaves the top of the domain, reaches the surface, or loses its remaining weight. Since these resets occur in different branches, they initially appear to be a good candidate for remapping. Reset work could be grouped into a separate event, reducing divergence between threads that reset and threads that continue tracing. However, reset events differ from the other event types because they do not require the previous photon state to be processed. A reset only needs to initialize a new photon and update the number of photons that have been launched. Because of this, storing reset photons in a queue is often unnecessary. An alternative is to let each thread count how many resets it has to perform and only use warp level aggregation when new photons need to be launched. But in practice, handling resets directly in the current execution path was found to be more efficient than treating reset as a separate event. This is due to reset being such a rare event in the billions of events (resets happen around 37 million times), making it not worth it to try and optimize such a small part of the program. Processing the reset logic immediately avoids additional queue checks, shared-memory writes, and remapping decisions. It also keeps more photons active in registers, which helps hide latency. For this reason, reset handling is kept outside the main remapping queues in the final implementations.

6.6.6 Delayed Scattering

The delayed-scattering implementation extended the optimized Warp-stack version with a dedicated scatter queue. Instead of performing scattering immediately during fine-grid evaluation, scattering photons got stored until enough were available to process a full warp. The aim was to separate the relatively expensive scattering operations, which require additional random sampling, direction updates, and memory accesses, from the faster null-collision path. The ideas was that this separation would prevent null-collision lanes from waiting for scattering lanes, particularly because profiling showed that traversal is affected by long-scoreboard stalls caused by instructions waiting for data from global memory. In practice, however, the additional queue increased shared-memory usage, remapping overhead, and the number of photons reserved by each warp across registers, as the work queue and the scatter queue would be filled with photons. This produced a longer execution tail with partially active warps and increased the imbalance between warps. A warp that reserves several photons with long remaining paths can continue executing significantly longer than other warps, whereas immediate scattering distributes these long-running photons more evenly. In the end, the increased tail and scheduling overhead outweigh the benefit of separating scattering from the null-collision path.

6.6.7 Reducing Block-stack synchronization

Reducing synchronization in the Block-stack implementation could provide a significant performance improvement, as barrier stalls are one of its main bottlenecks. In the Block-stack version all warps synchronize and then take the work from their dedicated regions. This synchronization could be removed if a counter is added for the total number of queued photons and for the number of committed photons for each dedicated queue region. A warp could then begin processing once its commit counter reaches 32, without waiting for the entire block. But the main issue is multiple warps reserving consecutive regions for writing, when the later warp finishes committing first, the next warp cannot safely assume that all preceding slots are ready. To implement this further a per slot ready flag would be necessary, but then each thread would have to keep checking if the slot is ready yet, causing a lot of shared-memory reads. This is already quite a big issue but the problem becomes even more difficult when processing partially filled warps, as a warp cannot reliably tell if it can start processing when it is partially filled or should wait for more photons. The additional bookkeeping needed to resolve these dependencies largely moves the synchronization cost rather than eliminating it.

6.6.8 Discarding photons

As the drain/compaction phase is around 25% of the Warp-stack implementation, that is a large part of the runtime, while it is only the last 10% of photons. It is possible to be faster than the base implementation by simply discarding this left over work. The problem with doing so, comes from a clear shortcoming of the current implementation, as most of the work that will be discarded is at the end of the each blocks cell. This comes from the photon sampling being in order instead of in random order. Re-introducing random order photon sampling could be possible, but to ensure a uniform distribution, Sobol sequencing would have to be considered again, but moving Sobol-states with the photon state through shared memory is too expensive. Just doing random sampling with the already available random number generator in every thread there is very significant added noise to the program, which is most likely above the desired threshold for noise. This added noise is still very prevalent when tracing extra photons to try and run more full warps to compensate for the lost photons.

7 Conclusion

This thesis investigated whether thread-data-remapping can improve the performance of a GPU-based short-wave Monte Carlo photon ray-tracer. Thread-data-remapping reduces branch divergence by reorganizing photon states so that threads within a warp process the same type of event. The central question was whether the resulting improvement in warp coherence is large enough to compensate for the data movement, scheduling, and resource costs introduced by remapping.

The first research question considers how much the original history-based implementation can be improved before introducing remapping. General GPU optimizations result in speed-ups of approximately 4.16×, 3.13×, and 2.81× for the clear-sky, moderate-cloud, and synthetic scatter workloads, respectively. This shows that a substantial part of the

original runtime is caused by avoidable memory traffic, repeated calculations, register pressure, atomic operations, and uneven work distribution rather than branch divergence itself. Establishing this optimized history-based implementation is therefore important for separating the actual effect of thread-data-remapping from more general kernel improvements.

The second research question investigates whether thread-data-remapping can improve upon this optimized baseline. Of the evaluated strategies, the Warp-stack performs best. It remains approximately 9.6% slower than the optimized base for the clear-sky workload and 1.4% slower for the moderate-cloud workload, but becomes approximately 1.9% faster for the synthetic scatter workload. Remapping consistently improves warp coherence and, as divergence increases, eventually reduces the total number of executed instructions. However, improved coherence is not sufficient to guarantee a lower runtime. The synthetic scatter workload is the only case in which the avoided divergent execution becomes large enough to recover the remaining remapping costs.

The third research question concerns the cost of remapping in terms of shared-memory usage, executed instructions, register pressure, synchronization, and occupancy. Shared memory is the largest resource cost: the Warp-stack requires 65.54 KB per block compared with 8.19 KB for the optimized base. This also reduces the number of active warps from approximately 26–30 per SM for the optimized base to approximately 20–25 for the Warp-stack. Register pressure itself increases only slightly, from 55 registers per thread for the optimized base to 56 for the Warp-stack. The executed-instruction cost depends on the workload: compared with the optimized base, the Warp-stack executes 5.4% more instructions in the clear-sky workload, but 4.7% fewer in the moderate-cloud workload and 7.9% fewer in the synthetic scatter workload. Remapping can therefore recover its direct instruction overhead when divergence is sufficiently high. Synchronization is mainly a limitation of block-level remapping, where barrier stalls contribute 5.38, 5.25, and 4.66 warp cycles per instruction for the clear-sky, moderate-cloud, and synthetic scatter workloads, respectively. The cost of remapping is therefore not limited to additional instructions: photon-state storage and movement increase shared-memory pressure, can reduce occupancy and cache capacity, and block-level coordination can introduce substantial synchronization overhead.

The fourth research question compares the different remapping strategies and their benefits under different event distributions. The Warp-stack provides the best overall compromise and becomes increasingly competitive as the workload contains more fine-grid and scattering work. Compared with the optimized base, it is 9.6% slower in the clear-sky workload and 1.4% slower in the moderate-cloud workload, but becomes 1.9% faster in the synthetic scatter workload. The Scatter-split shows a similar dependence on the event distribution: it is 12.5% and 11.5% slower in the clear-sky and moderate-cloud workloads, respectively, but only 0.6% slower in the synthetic scatter workload. This shows that explicitly grouping scattering becomes more useful as scattering becomes more frequent, although the broader coarse/fine split of the Warp-stack remains more effective overall. The Block-stack also benefits from the more irregular synthetic workload, where its improved cross-warp grouping reduces its slowdown to 12.9%, compared with 24.8% in clear sky and 40.6% in moderate clouds. However, its block-wide synchronization prevents the improved coherence from translating into better performance. In contrast, the Warp-link remains

substantially slower in every workload and becomes relatively worse as the workload becomes more demanding, with slowdowns of 76.3%, 102.4%, and 114.3% for the clear-sky, moderate-cloud, and synthetic scatter workloads, respectively. Its additional scheduling flexibility therefore does not compensate for the associated management and memory overhead.

The main conclusion is therefore that reducing warp divergence alone does not guarantee faster Monte Carlo photon tracing. Remapping becomes attractive only when the work avoided through increased coherence is sufficiently expensive to outweigh photon-state movement and scheduling overhead. This also makes scaling to additional event types difficult for the current ray-tracer, as more event categories require additional storage and scheduling, while many of the individual event paths are too inexpensive to recover this cost.

Future work could investigate whether this balance changes on GPU architectures with more on-chip memory or for ray-tracing problems with more computationally expensive event paths, such as long-wave radiative transfer. Reducing the amount of photon state transferred during remapping may further lower the break-even point. A data layout that separates frequently accessed fields from event-specific state could reduce shared-memory traffic without introducing the irregular accesses observed with the tested indirection approach. The final drain and compaction phases are another important target, as they contribute disproportionately to runtime while processing only the remaining fraction of photons. Potentially, cross-warp redistribution during the drain-phase could further reduce reservation overhead from remapping. An implementation utilizing warp specialization could also be interesting to explore.

References

[1] Hironobu Iwabuchi. "Efficient Monte Carlo Methods for Radiative Transfer Modeling". In: *Journal of the Atmospheric Sciences* 63.9 (2006), pp. 2324–2339. DOI: 10.1175/JAS3755.1.

[2] Y. Chen and K. N. Liou. "A Monte Carlo method for 3D thermal infrared radiative transfer". In: *Journal of Quantitative Spectroscopy and Radiative Transfer* 101.1 (2006), pp. 166–178. DOI: 10.1016/j.jqsrt.2005.10.002.

[3] Nicholas Metropolis and Stanislaw Ulam. "The Monte Carlo Method". In: *Journal of the American Statistical Association* 44.247 (1949), pp. 335–341. DOI: 10.1080/01621459.1949.10483310.

[4] M. A. Veerman, B. J. H. van Stratum, and C. C. van Heerwaarden. "A Case Study of Cumulus Convection Over Land in Cloud-Resolving Simulations With a Coupled Ray Tracer". In: *Geophysical Research Letters* 49.23 (2022), e2022GL100808. DOI: https://doi.org/10.1029/2022GL100808. URL: https://agupubs.onlinelibrary.wiley.com/doi/abs/10.1029/2022GL100808.

[5] NVIDIA Corporation. *CUDA Programming Guide*. NVIDIA Corporation. 2026. URL: https://docs.nvidia.com/cuda/cuda-programming-guide/01-introduction/programming-model.html.

[6] NVIDIA Corporation. *NVIDIA Tesla V100 GPU Architecture*. White paper WP-08608-001_v1.1. NVIDIA Corporation, Aug. 2017. URL: https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf.

[7] Eddy Z. Zhang et al. "Streamlining GPU Applications on the Fly: Thread Divergence Elimination through Runtime Thread-Data Remapping". In: *Proceedings of the 24th ACM International Conference on Supercomputing*. ICS '10. Tsukuba, Ibaraki, Japan: Association for Computing Machinery, 2010, pp. 115–126. ISBN: 978-1-4503-0018-6. DOI: 10.1145/1810085.1810104.

[8] Huanxin Lin, Cho-Li Wang, and Hongyuan Liu. "On-GPU Thread-Data Remapping for Branch Divergence Reduction". In: *ACM Transactions on Architecture and Code Optimization* 15.3 (Sept. 2018), 39:1–39:24. DOI: 10.1145/3242089.

[9] Modal. *What Is the CUDA Programming Model?* URL: https://modal.com/gpu-glossary/device-software/cuda-programming-model.

[10] Steven P. Hamilton, Stuart R. Slattery, and Thomas M. Evans. "Multigroup Monte Carlo on GPUs: Comparison of history- and event-based algorithms". In: *Annals of Nuclear Energy* 113 (Mar. 2018), pp. 506–518. ISSN: 0306-4549. DOI: 10.1016/j.anucene.2017.11.032.

[11] Najda Villefranque et al. "A Path-Tracing Monte Carlo Library for 3-D Radiative Transfer in Highly Resolved Cloudy Atmospheres". In: *Journal of Advances in Modeling Earth Systems* 11.8 (2019), pp. 2449–2473. ISSN: 1942-2466. DOI: 10.1029/2018MS001602.

[12] Hironobu Iwabuchi and Rintaro Okamura. "Multispectral Monte Carlo radiative transfer simulation by the maximum cross-section method". In: *Journal of Quantitative Spectroscopy and Radiative Transfer* 193 (May 2017), pp. 40–46. ISSN: 0022-4073. DOI: 10.1016/j.jqsrt.2017.01.025.

[13] M. Galtier et al. “Integral formulation of null-collision Monte Carlo algorithms”. In: *Journal of Quantitative Spectroscopy and Radiative Transfer* 125 (2013), pp. 57–68. ISSN: 0022-4073. DOI: 10.1016/j.jqsrt.2013.04.001.

[14] Yushuo Ren et al. “Research on the advanced GPU-based Monte Carlo neutron transport methods”. In: *Computer Physics Communications* 323 (June 2026), p. 110101. ISSN: 0010-4655. DOI: 10.1016/j.cpc.2026.110101.

[15] Joanna Piper Morgan et al. “The Monte Carlo Computational Summit – October 25 26, 2023 – Notre Dame, Indiana, USA”. In: *Journal of Computational and Theoretical Transport* 53.5 (2024), pp. 361–382. ISSN: 2332-4325. DOI: 10.1080/23324309.2024.2354401. URL: http://dx.doi.org/10.1080/23324309.2024.2354401.

[16] Joanna Piper Morgan et al. “Monte Carlo / Dynamic Code (MC/DC): An accelerated Python package for fully transient neutron transport and rapid methods development”. In: *Journal of Open Source Software* 9.96 (2024), p. 6415. DOI: 10.21105/joss.06415. URL: https://doi.org/10.21105/joss.06415.

[17] Gavin Ridley. “GPU-Oriented Algorithms for Continuous Energy Monte Carlo Neutron Transport”. Doctoral thesis. Cambridge, Massachusetts: Massachusetts Institute of Technology, Sept. 2024. URL: https://hdl.handle.net/1721.1/157192.

[18] Pieter Hijma et al. “Optimization Techniques for GPU Programming”. In: *ACM Computing Surveys* 55.11 (2023), 239:1–239:81. DOI: 10.1145/3570638.

[19] Ben van Werkhoven. “Kernel Tuner: A search-optimizing GPU code auto-tuner”. In: *Future Generation Computer Systems* 90 (Jan. 2019), pp. 347–358. DOI: 10.1016/j.future.2018.08.004.

[20] A. Heimlich, A.C.A. Mol, and C.M.N.A. Pereira. “GPU-based Monte Carlo simulation in neutron transport and finite differences heat equation evaluation”. In: *Progress in Nuclear Energy* 53.2 (2011), pp. 229–239. ISSN: 0149-1970. DOI: https://doi.org/10.1016/j.pnucene.2010.09.011. URL: https://www.sciencedirect.com/science/article/pii/S0149197010001514.

[21] Donghui Ma et al. “Evaluation of Single-Node Performance of Parallel Algorithms for Multigroup Monte Carlo Particle Transport Methods”. In: *Frontiers in Energy Research* 9 (2021), p. 705823. DOI: 10.3389/fenrg.2021.705823.

[22] Ryan M. Bergmann and Jasmina L. Vujic. “Algorithmic choices in WARP – A framework for continuous energy Monte Carlo neutron transport in general 3D geometries on GPUs”. In: *Annals of Nuclear Energy* 77 (2015), pp. 176–193. ISSN: 0306-4549. DOI: 10.1016/j.anucene.2014.10.039. URL: https://www.sciencedirect.com/science/article/abs/pii/S0306454914005787.

[23] Huanxin Lin and Cho-Li Wang. “On-GPU Thread-Data Remapping for Nested Branch Divergence”. In: *Journal of Parallel and Distributed Computing* 139 (May 2020), pp. 75–86. DOI: 10.1016/j.jpdc.2020.02.003.

[24] Wilson W. L. Fung and Tor M. Aamodt. “Thread Block Compaction for Efficient SIMT Control Flow”. In: *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture*. HPCA ’11. IEEE Computer Society, 2011, pp. 25–36. DOI: 10.1109/HPCA.2011.5749714.

[25] Bingchao Li et al. “Bypass-Enabled Thread Compaction for Divergent Control Flow in Graphics Processing Units”. In: *Journal of Shanghai Jiaotong University (Science)* 26.2 (2021), pp. 245–256. DOI: 10.1007/s12204-020-2240-x.

[26] Ingo Wald. “Active Thread Compaction for GPU Path Tracing”. In: *Proceedings of High Performance Graphics*. ACM, 2011, pp. 51–58.

[27] Braxton Cuneo and Mike Bailey. “Divergence Reduction in Monte Carlo Neutron Transport with On-GPU Asynchronous Scheduling”. In: *ACM Transactions on Modeling and Computer Simulation* 34.1 (Jan. 2024), pp. 1–25. DOI: 10.1145/3626957.

[28] Braxton Cuneo et al. “Comparing the Performance of MC/DC’s On-GPU Event-based Processing Methods in Multigroup and Continuous-energy Problems”. In: *Proceedings of the International Conference on Mathematics and Computational Methods Applied to Nuclear Science and Engineering (M&C 2025)*. Denver, CO, USA, 2025, pp. 1944–1953. DOI: 10.13182/MC25-47174. arXiv: 2506.00263 [physics.comp-ph].

[29] I.M Sobol’. “On the distribution of points in a cube and the approximate evaluation of integrals”. In: *USSR Computational Mathematics and Mathematical Physics* 7.4 (Jan. 1967), pp. 86–112. DOI: 10.1016/0041-5553(67)90144-9. URL: https://doi.org/10.1016/0041-5553(67)90144-9.