HASEonGPU—An adaptive, load-balanced MPI/GPU-code for calculating the amplified spontaneous emission in high power laser media

Eckert, C. H. J.; Zenker, E.; Bussmann, M.; Albach, D.;

Originally published:

May 2016

Computer Physics Communications 207(2016), 362-374

DOI: https://doi.org/10.1016/j.cpc.2016.05.019

Perma-Link to Publication Repository of HZDR:

https://www.hzdr.de/publications/Publ-24191

HASEonGPU — An adaptive, load-balanced MPI/GPU-Code for calculating the amplified spontaneous emission in high power laser media

C. Eckert[a,1,*], E. Zenker[a,1], M. Bussmann[a], D. Albach[a]

[a] *Institute of Radiation Physics, Helmholtz-Zentrum Dresden – Rossendorf e. V., Bautzner Landstraße 400, 01328 Dresden, Germany*

Abstract

We present an adaptive Monte Carlo algorithm for computing the amplified spontaneous emission (ASE) flux in laser gain media pumped by pulsed lasers. With the design of high power lasers in mind, which require large size gain media, we have developed the open source code *HASEonGPU* that is capable of utilizing multiple graphic processing units (GPUs). With HASEonGPU, time to solution is reduced to minutes on a medium size GPU cluster of 64 NVIDIA Tesla K20m GPUs and excellent speedup is achieved when scaling to multiple GPUs. Comparison of simulation results to measurements of ASE in Yb^{3+} : YAG ceramics show perfect agreement.

Keywords: Amplified Spontaneous Emission, CUDA, GPU Cluster, Massively Parallel, Monte Carlo Integration, High Power Laser

PROGRAM SUMMARY

Manuscript Title: HASEonGPU — An adaptive, load-balanced MPI/GPU-Code for calculating the amplified spontaneous emission in high power laser media
Authors: C. Eckert, E. Zenker, M. Bussmann, D. Albach
Program Title: HASEonGPU
Journal Reference:
Catalogue identifier:
Licensing provisions: GPLv3
Programming language: C++, Matlab
Computer: GPU cluster or workstation with CUDA-capable GPUs (compute capability ≥ 2.0)
Operating system: Linux
RAM: Several GB, depending on input size and number of GPUs. 4000000000 bytes (4 GB) per GPU is recommended.
Has the code been vectorised or parallelized?: Yes.
Number of processors used: Can utilize 1 CPU core per compatible GPU
Supplementary material:
Keywords: Amplified Spontaneous Emission, CUDA, GPU Cluster, Massively Parallel, Monte Carlo Integration, High Power Laser

Classification: 4.13 Statistical Methods, 6.5 Software including Parallel Algorithms, 15 Laser Physics
External routines/libraries: CUDA, Boost Program Options, OpenMPI

Nature of problem:
The algorithm described by D. Albach in [1,2] uses ray tracing techniques and Monte Carlo integration to calculate Amplified Spontaneous Emission (ASE) with high precision. It requires a high number of sampling points as well as a high number of rays to reach the desired results. Additionally, reflections on the upper and lower surface of the medium increase the workload by an order of magnitude. On traditional CPU-based systems the computation is time-consuming, which limits the number of simulations that can be performed.

Solution method:
HASEonGPU uses a non-uniform distribution of sampling points within the gain medium to focus computation on areas of interest. This is further improved by combining the Monte Carlo integration with importance sampling [3]. To improve execution time further, the algorithm is highly parallelized to run on a GPU and supports adaptive sampling resolutions and random restarts. It can also be executed in a GPU cluster, where linear scaling is achieved by a coarse-granular load balancing that distributes the workload among all GPUs in a master-worker-scheme over MPI.

Restrictions:
Presently, the number of rays used for the Monte Carlo

*Corresponding author; postal address: POB 51 01 19 01314 Dresden Germany; e-mail: carlchristian.eckert@gmx.de; tel: +491799409536

Email addresses: c.eckert@hzdr.de (C. Eckert), e.zenker@hzdr.de (E. Zenker), m.bussmann@hzdr.de (M. Bussmann), d.albach@hzdr.de (D. Albach)

[1] contributed equally

integration of a single sampling point within the gain medium is limited by the available memory on the GPU (about 10^8 rays per GB of GPU memory). Furthermore, when using MPI as a workload distribution mechanism, one of the MPI processes will act as a scheduling master and its GPU can not participate in the computation.

Unusual features:
The software can run on a workstation (threaded) as well as on a large-scale GPU cluster (MPI) that provides the required GPU hardware. The simulation parameters include polychromatic laser pulses as well as surface coatings, cladding, and refractive indices of the gain medium. This allows to also simulate reflections on the upper and lower surface of the medium. If a desired mean square error metric is not met with a set number of rays, the algorithm can automatically increase the number of rays to improve the results.

Additional comments:
The source code also includes a *MATLAB* script that can be used to call HASEonGPU directly from *MATLAB* code to integrate it into existing simulation setups. There are also examples included on how to execute HASEonGPU from the command line as well as an example experiment that uses *MATLAB* and the provided script. More detailed information can be found in the README file.

Running time:
Depending on the number of sampling points, desired sampling resolution for each point, and number of GPUs, the execution time can vary strongly. A typical cylindrical gain medium of 6cm diameter simulated with 4210 non-uniformly distributed sampling points can be simulated with a sufficient precision in 1min on a single NVIDIA Tesla K20m GPU. Running time as well as precision can be further optimized through various parameters.

[1] D. Albach, J.-C. Chanteloup, G. l. Touz e, Influence of ASE on the gain distribution in large size, high gain $Yb^{3+} : YAG$ slabs, Opt. express 17 (5) (2009) 37923801.

[2] D. Albach, Amplified spontaneous emission and thermal management on a high average-power diode-pumped solid-state laser-the Lucia laser system, Ph.D. thesis, Palaiseau, Ecole polytechnique (2010).

[3] E. C. Anderson, Monte Carlo methods and importance sampling, 1999.

1. Introduction

The main principle of every laser (light amplification by stimulated emission of radation) is stimulated emission. In order to permit such stimulated emission, energy has to be stored in a so-called upper laser state (e.g. an excited electronic state). The transistion from the upper to the lower laser state yields the associated photon energy. However, such an excited energy level has a limited lifetime and therefore spontaneous emission does naturally occur. Those spontaneously emitted photons are, if they travel through an excited part of the the laser material, amplified by stimulated emission. This process, called amplified spontaneous emission (ASE), is consequently one of the limiting factors for any laser design, as it ultimately limits the available energy in any laser amplifier medium [1–3].

With the increasing availability of high power lasers, computer aided design of laser gain media, including efficient cooling and the optimization of cladding and mounting, demands multi-physics simulation capabilities. Those simulations link the thermal and optical properties of laser gain media to the amplification process itself.

For high power laser design, energy efficiency is becoming one of the major constraints when aiming to maximize average power at a high peak power. Exact knowledge of the flux of amplified spontaneous emission (ASE) is thus vital, as it drastically reduces the energy stored in the gain medium usable for amplification.

Energy can be dislocated to non-pumped areas, as the ASE photons can freely travel within the gain medium, experiencing repeated amplification along their path. Consequently, a significant part of the stored energy is converted into heat inside the material itself, the surrounding claddings and mounts. ASE becomes an important factor of heat distribution inside and outside of the pumped laser gain medium. With the increasing importance of energy-efficient diode-pumped lasers, this particular topic has come into the focus of interest again [4].

Up until now, coupling models of heat dissipation to models of ASE generation required drastic simplifications, as detailed simulations of ASE are computationally intensive and in most of the applied cases, ASE cannot be treated analytically.

With the advent of massively parallel accelerator hardware such as graphic processing units (GPUs), it has become possible to solve the fundamental ASE gain integral by exploiting the strong scalability of Monte Carlo algorithms. In this paper we focus on the extension of a previously published model [4], its implementation on GPU accelerator hardware and its performance optimization for use on multi-GPU clusters, presenting the open source, multi-GPU ASE simulation code *HASEonGPU* (***H**igh performance **A**mplified **S**pontaneous **E**mission **on GPU***). This algorithm concentrates on quasi-cw pumping, where simula-

tion timesteps are long compared to the time a photon takes to travel the total physical extent of the gain medium.

To our best knowledge *HASEonGPU* is the first massively-parallel code for simulating ASE in laser gain media. In order to encourage the use of the code by high power laser developers we provide a *MATLAB* [5] interface that should allow for easy integration into simulation workflows existing in this scientific community.

HASEonGPU includes all features of the ASE simulation code described in [4, 6] which was written in *C* and is purely serial. All these features are discussed in the following sections that explain the Monte Carlo integration of the ASE gain along a single ray, see Section 2.1, the 3D sampling of the laser gain medium via an anisotropic mesh, see Section 2.2, and the computation of the complete gain via ray tracing, Section 2.3.

HASEonGPU adds to these features by including reflections at the upper and lower surface of the gain medium as detailed in Section 2.4, which increases the computational workload by typically one order of magnitude, and allowing for polychromatic pump spectra of arbitrary form, which, too, increases the computational effort compared to the monochromatic pump used in [4, 6].

With this in mind, major improvements to the existing simulation code [4, 6] are thus the hybrid parallelization for use on GPU clusters, the introduction of sampling techniques to optimize computation of the overall gain and of load balancing as described in Section 3.

From the detailed analysis of time to solution and strong scalability in Section 4.4 we find an overall speedup of about two orders of magnitude comparing the single-core performance of [4, 6] to the single-GPU performance of *HASEonGPU* and linear strong scaling of *HASEonGPU*, reaching time to solution of a few minutes on a medium sized GPU cluster.

2. Ray tracing model for calculating the gain

In order to calculate the impact of ASE on a given point in a laser amplifier medium, it is necessary to compute the density of incoming ASE photons per time interval. This ASE flux (Φ_{ASE}) is given by the integral (2.1). For reasons of simplicity the *gain medium* is assumed to have planar top and bottom surfaces. High resolution spatial sampling of the gain medium results in a large number of *sampling points*, see Fig. 1, for which the amplification of spontaneous emission needs to be determined.

2.1. Monte Carlo integration of the ASE flux

For any given sampling point s_i in the gain medium, Φ_{ASE} can be calculated by solving the integral [6]:

$$\Phi_{ASE}(s_i) = \frac{1}{4\pi} \iint\limits_{V\lambda} \frac{\hat{n}(r)}{\tau_f |\rho(r, s_i)|^2} \; g(\lambda) \; G_{r \to s_i} \; dV d\lambda \tag{2.1}$$

where $\hat{n}(r)$ is the density of excited states at the point of spontaneous emission r, τ_f the life time of the upper laser state, $|\rho(r, s_i)|$ the absolute value of the distance travelled between the position of spontaneous emission r and the point of observation s_i, $g(\lambda)$ the spectral distribution function of the spontaneous emission, $G_{r \to s_i}$ denotes the amplification between positions r and s_i as a line integral along the vector $\overrightarrow{rs_i}$. HASEonGPU solves for Φ_{ASE} by Monte Carlo integration of (2.1) for all sampling points over its volume V and wavelength λ.

For a single wavelength and sampling point s_i, (2.1) can be rewritten as a sum:

$$\Phi_{ASE}(s_i) = \frac{1}{4\pi N \tau_f} \sum_{u=0}^{N-1} \hat{n}(r_{i,u}) \cdot gain(\overrightarrow{r_{i,u}s_i}) \tag{2.2}$$

where N is the number of randomly selected paths $\overrightarrow{r_{i,u}s_i}$ of photons traveling inside the gain medium. The amplification $gain(\overrightarrow{r_{i,u}s_i})$ along the photon path $\overrightarrow{r_{i,u}s_i}$ is computed via Eq. (2.5) as described later.

The method presented here is in many ways similar to ray tracing methods and, thus, we will in the following call $\overrightarrow{r_{i,u}s_i}$ a *ray*. Therefore, our algorithm derives the ASE flux distribution inside the medium by calculating the local ASE flux for each sampling point via repeated computation of the amplification of a multitude of rays all ending in the selected sampling point.

2.2. Sampling of the gain medium using a non-uniform mesh

In order to distribute the sampling points, the medium is treated as a horizontal, two-dimensional plane, extruded to the third dimension. The sampling points in the plane can have non-uniform density, allowing for an increase in spatial resolution in certain areas of interest. These sampling points are

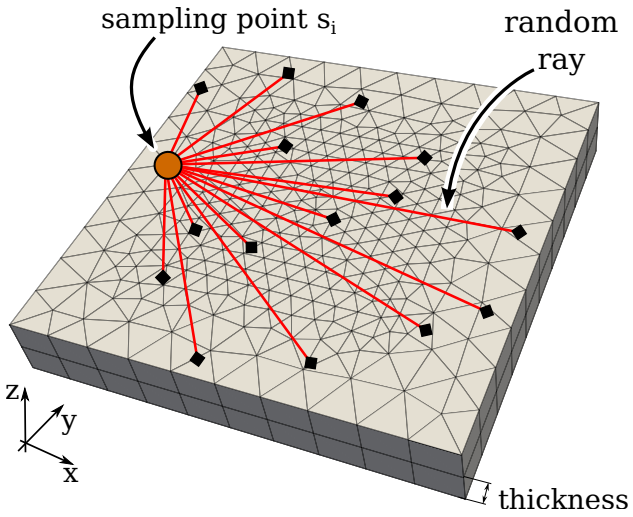

Figure 1: Non-uniform sampling of the active gain medium, forming a mesh of prisms as described in the text. The center of the gain medium is sampled at a higher spatial resolution than the outer regions. Two slices of prisms are shown in z-direction. For each sampling point s_i, a large number of rays starting from random positions $r_{i,u}$ inside the medium points back to the sampling point, each ray representing spontaneously emitted radiation. For each ray, the gain along the path of the ray through the medium is calculated. The amplified spontaneous emission is then deposited at the position of the sampling point.

in turn connected using Delaunay triangulation [7] to form a 2D mesh of triangles.

The extrusion of the triangular mesh in direction of the vertical axis forms a *slice* of right prisms, see also Fig. 2. This slice is then duplicated several times along the z-axis, until the whole medium is divided into prisms, see Fig. 1.

2.3. Ray tracing calculation of the overall gain

In order to calculate the amplification of photons spontaneously emitted at a randomly chosen position $r_{i,u}$ when traversing the medium to a sampling point s_i, rays are traced along their paths through the prisms, see Fig. 3. When the path of a ray intersects the interface between two prisms, it is split into partial rays. Each of these partial rays is localized in the volume of its corresponding prism.

Starting from point $r_{i,u}$ inside a prism, there are five possible intersection points of the ray with the surrounding prism surfaces, one for the horizontal top and bottom surfaces and one for each of the three vertical sides.

Once the intersected plane is determined, with r_x denoting the intersection point, the prism the

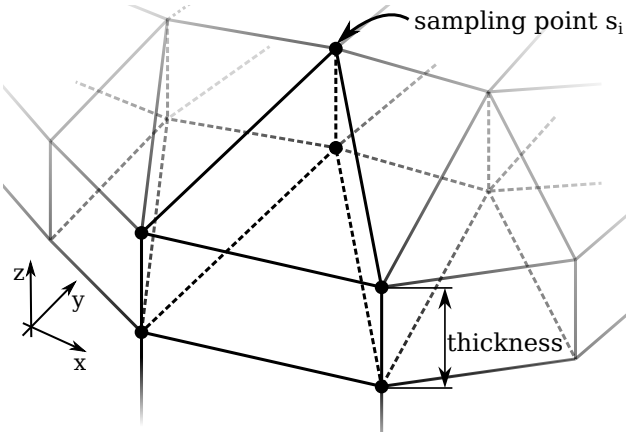

Figure 2: The extruded plane of triangles forms a slice of prisms. The thickness of the slices sets the level of detail in z-direction. Concatenating the slices in the z-directions fills up the volume of the gain medium.

ray will enter next can be determined based on the knowledge about the mesh structure:

vertical side The ray remains in the same slice and the next prism is given by a neighboring triangle. The data structure to determine the neighboring triangle is created during the Delaunay triangulation.

horizontal plane If the ray intersects with either the top or bottom surface of the prism, the next prism will be based on the same triangle as the previous, but will be located in a neighboring slice.

This process is iterated for each prism on the path between $r_{i,u}$ and the selected sampling point s_i. Each intersection point r_x with prism surfaces along the path divides the ray into line segments of a certain length l_x. These segments are used for the gain calculation in the following way: For each prism x, the contribution to the gain is called *partial gain* and calculated as a function of l_x by:

$$partial\_gain(x) = e^{\,g_0 \cdot l_x} \tag{2.3}$$

with

$$g_0 = N_{tot} \cdot (\beta_x(\sigma_e + \sigma_a) - \sigma_a) \tag{2.4}$$

being defined for the quasi three-level laser [4], where β_x is the stimulus in the current prism and σ_e, σ_a are emission and absorption cross-sections for a single wavelength. This wavelength is obtained by equidistant discretization of the polychromatic spectrum followed by the selection of a random wavelength from the discrete spectrum. It is

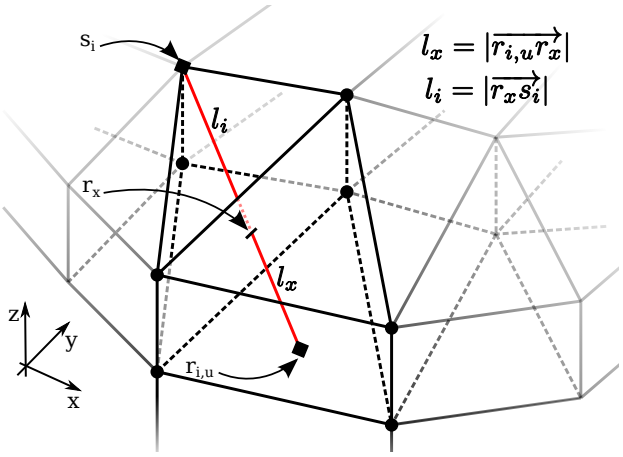

Figure 3: Scheme for tracing the ray $\overrightarrow{r_{i,u}s_i}$ connecting a random point $r_{i,u}$ in the gain medium to a selected sampling point s_i through the prism structure: All prisms intersected are located on the same slice. As the ray intersects the vertical interface of two neighboring prisms at r_x, it is divided into two partial rays of lengths l_x and l_i.

worth noting that the function for the partial gain can be easily extended to include any local characteristic of the gain medium as it is computed for each prism.

The gain over the complete ray is then computed as the product of all partial gains:

$$gain(\overrightarrow{r_{i,u}s_i}) = \frac{\prod^n partial\_gain(n)}{|\overrightarrow{r_{i,u}s_i}|^2} \qquad (2.5)$$

weighed by the square of the total ray length.

2.4. Adding vertical reflections to the model

In order to allow for a more realistic simulation of ASE generation, we include reflections on the outer upper and lower surface of the medium. The ray is split into multiple separate sub-rays, each sub-ray being created by either reflection or transmission at the intersection point, see sub-rays $\overrightarrow{r_{i,u}m_{i,u,1}}$, $\overrightarrow{m_{i,u,1}m_{i,u,2}}$ and $\overrightarrow{m_{i,u,2}s_i}$ in Fig. 4 and Fig. 5. The sum of the energy of both the reflected and transmitted ray is equal to the energy of the incident ray.

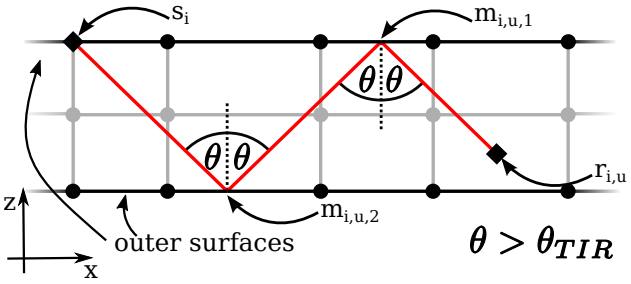

Figure 4: Total internal reflection of a ray consecutively intersecting either the upper or lower suface of the gain medium: The incident angle θ is larger than the angle of total internal reflection θ_{TIR}.

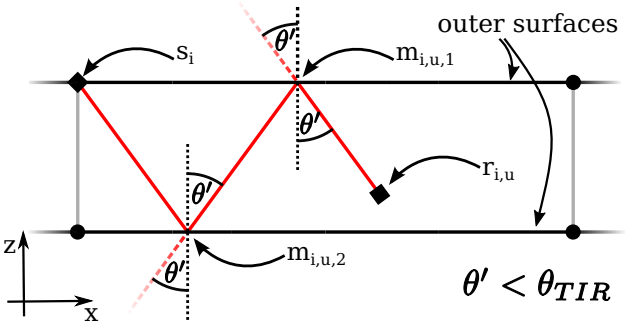

Figure 5: Partial reflection of a ray consecutively intersecting either the upper or lower suface of the gain medium: The incident angle θ' is smaller than θ_{TIR}.

At each reflection point, the angle of total internal reflection θ_{TIR} is calculated to estimate the *initial gain* g_i of a ray that is reflected at position x_i by

$$g_i = \begin{cases} gain(\overrightarrow{x_{i-1}x_i}) & \text{if } \theta \geq \theta_{TIR} \\ gain(\overrightarrow{x_{i-1}x_i}) \cdot \gamma(x_i) & \text{if } \theta < \theta_{TIR} \end{cases}, \qquad (2.6)$$

where $\gamma(x_i)$ is the fraction of the energy of the incident ray reflected as given by the reflectivity of the prism interface and which is non-zero if the incident angle is too small for total internal reflection.

As in the case of the partial gain computation, information on the local surface reflectivity can be easily incorporated into the algorithm.

Each point x_n refers to either a mirror point m_n, a start point $r_{i,u}$ or a sampling point s_i.

The computation of all sub-rays results in a significant overhead over the more simplistic model with just a single ray, which necessitates the use of a high performance algorithm. The current implementation supports locally varying refractive in-

dices as well as varying reflectivities for each Delaunay triangle of the outer surfaces.

Currently, HASEonGPU only supports reflections on the upper and lower surface of the medium. Therefore, our model assumes that the lateral surfaces of the gain medium are coated with an anti-reflective material. This is not a drawback, since a gain medium can be coated with anti-reflective coating on the vertical surface(s), but, disregards the sometimes important issue of transverse lasing.

The ray tracing method for computing the gain presented here can accommodate local, per-prism variations of the gain. Therefore, it can be coupled to a multi-physics model of lasing that provides feedback to the simulation via changing the local reflectivity and partial gain. Such a coupling requires a short time to solution. Consequently, the algorithm depicted above has been implemented in HASEonGPU for multi-GPU clusters as described in the following.

3. Parallelization and sampling methods

In this section, program flow and memory management of HASEonGPU are introduced. Incremental improvement of the computational performance of HASEonGPU will be discussed by first describing the parallelization of the ray tracing algorithm on GPUs and later adding several techniques that improve sampling precision as well as runtime behavior.

All results shown in this section are based on the simulation of a single time step using the same gain medium, see Fig. 1, and stimulus with input properties listed in Table 1.

3.1. HASEonGPU program flow

The basic program flow as depicted in Fig. 6 starts by passing input data from a *MATLAB* [5] script to the CUDA-C host application *CalcPhiASE*. Input data includes the mesh points, local refractive indices and reflectivities as well as information on the absorption and emission spectrum. Input data is then copied from the main memory of the compute node (*host memory*) to the accelerator card's memory (*device memory*) and processed as described in Section 3.2.2. After the computation has succeeded, all data is passed back to the calling *MATLAB* script and the outer loop advances to the next time step. In the following, we will focus on the CUDA-C implementation of *CalcPhiASE* and the corresponding kernel functions.

Points per plane	321
Sampling points	3210
Planes	10
Triangles	600
Prisms	5400
Total height of medium	0.6 cm
Surface area of medium	$4 \times 4\ cm^2$
Medium material	Yb^{3+}:YAG
Yb^{3+} doping	$2\ at.\%$
Spectrum	Monochromatic $\lambda = 1030\ nm$ $\sigma_e = 2.4 \times 10^{-20}\ cm^2$ $\sigma_a = 1.1 \times 10^{-21}\ cm^2$

Table 1: Input parameters for the scaling tests described in the text. The crystal doping gives the amount of Yb^{3+} ions implanted in the YAG crystal and the spectral information includes the center wavelength λ, the spectral cross-sections for photon emission, σ_e, and absorption, σ_a.

3.2. Parallel many-core implementation of the ray tracing algorithm

When moving from a single CPU to a multi-GPU implementation we heavily exploit the Monte Carlo Method's parallel nature, using NVIDIA [8] CUDA-C for the implementation.

Fig. 6 highlights two embarrassingly parallel parts of the algorithm which can be exploited for many-core parallelization. First, for a single sampling point, the calculation of the gain for each ray is completely independent of the other rays. Second, the gain calculation for each sampling point itself is independent of all other sampling points.

3.2.1. HASEonGPU data structures and caching

In the following we will assume basic knowledge of the CUDA threading model and of concepts such as *thread blocks*, *block grids* and *warps* as introduced in [9]. Memory on NVIDIA GPUs in general can be split in three hierarchical levels:

Global memory several Gigabytes, high latency and bandwidth, global access for threads from all thread blocks

Shared memory several kilobytes, medium latency, access by threads of a single thread block

Register memory several 32 bit registers, no latency, exclusive access by a single thread

All HASEonGPU data structures are one-dimensional arrays that are accessible through

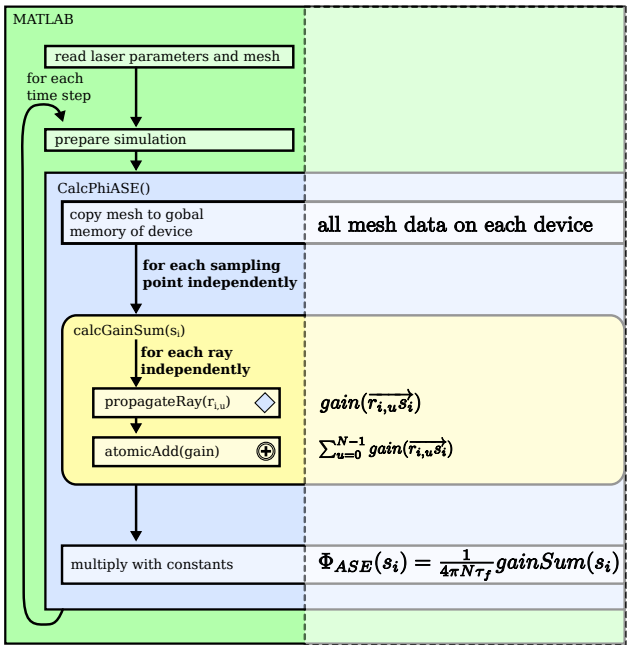

Figure 6: Basic program flow of HASEonGPU: Input data is prepared by a MATLAB script which calls the CUDA-C host application *CalcPhiASE* inside a loop to simulate the development of the ASE flux over multiple time steps. The CUDA-C host application itself calls multiple GPU device functions for each sampling point and returns the results for the whole time step to the calling MATLAB script.

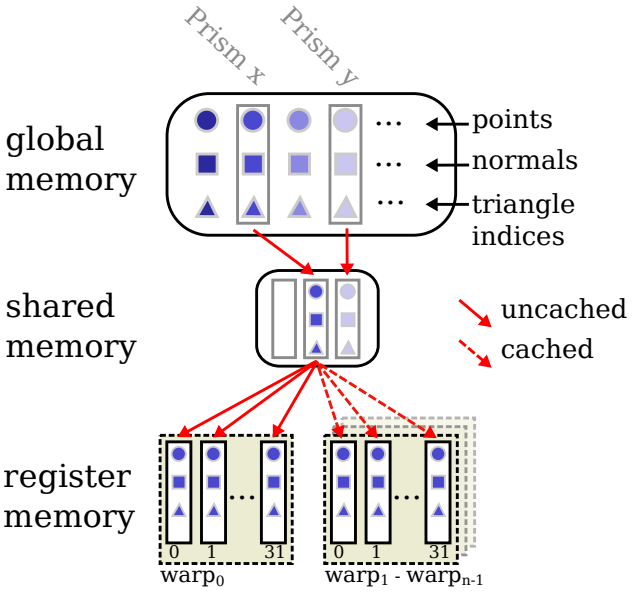

Figure 7: Memory hierarchy for a single CUDA thread block. Prism data consisting of corner points, surface normals and triangle indices are copied only once from global memory to shared memory, which is used as the L1 cache by CUDA. Subsequent reads within a thread block are implicitly cached. Rays with similar paths are grouped together to reuse prism data from the cache allowing for warp execution of threads for these groups of rays.

programmer-friendly interfaces to avoid explicit pointer arithmetic. In particular, the mesh consists of an array of points and an array holding the indices to these points. Three consecutive indices form a triangle. In addition to the triangle corner points, normals and neighbors of all edges are included for use by the intersection and ray tracing algorithms.

Precise Monte Carlo integration requires spawning millions of threads for each sampling point, each thread computing the gain for several rays sequentially. With each ray requiring 4 bytes of memory, the memory consumed for storing all rays by far takes up the largest fraction of global memory, indirectly setting a limit for the sampling resolution by the memory available on a single GPU.

It is not possible to store all mesh data in the (small) register memory. Hence, the mesh is stored in global memory and will be transferred prism by prism into the registers of the threads when needed, see Fig. 7.

The trace of a ray cannot be predicted in advance and, thus, the shared memory as depicted in Fig. 7 acts as an L1 cache, caching prisms from global memory to share them with several warps of threads. Threads for rays starting in the same prism and, henceforth, rays with similar trace through the mesh, are grouped in thread blocks. Thus, warps of threads can reuse cached prism data from their shared memory.

3.2.2. Rays as GPU threads

The tracing of every ray through the mesh structure can be done independently. Fig. 8 demonstrates the parallel processing of rays. For each sampling point s_i, a kernel consisting of a grid of maximum 200 blocks is spawned, with the maximum block number limited by the Mersenne twister implementation [10] used for generating random numbers. Each of these blocks contains 128 threads. This number was determined by the CUDA Occupancy Calculator [11] with the objective to obtain the maximum number of concurrently executed operations for the testing hardware, see Section 4.2.

Depending on the GPU hardware and compute capability, up to B blocks can be executed in par-

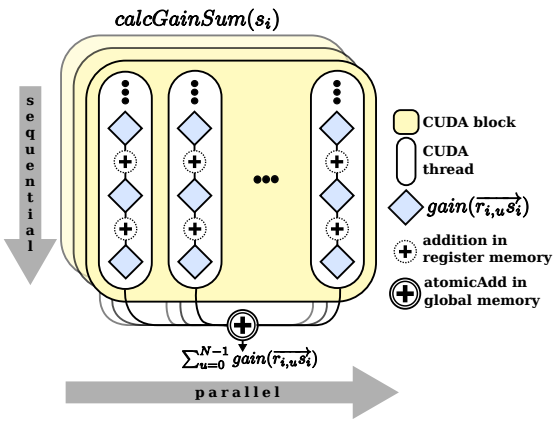

Figure 8: Inside each thread block, threads run in parallel. Each thread executes several ray traces sequentially and accumulates their gain values. The final gain value for each sampling point s_i is obtained by atomic reduction over all blocks in a grid and written to global memory.

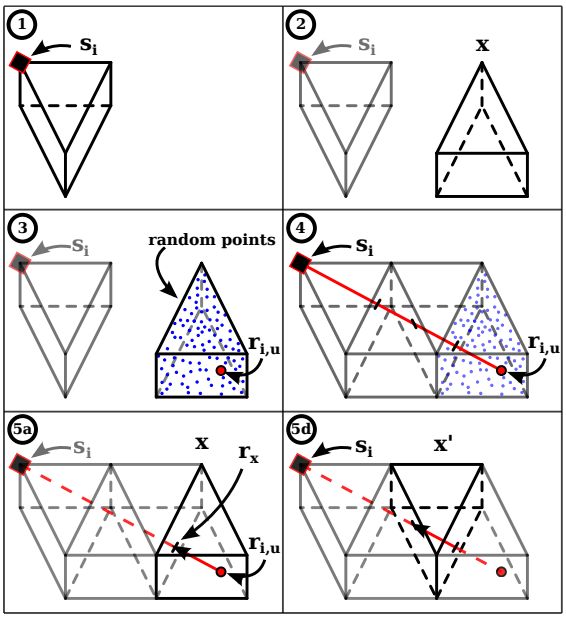

Figure 9: Lifecycle of a single ray within a thread, corresponding to $propagateRay(r_{i,u})$, see also Fig. 6. While the ray propagates through the mesh structure, steps 5a to 5d are repeated for each prism traversed along the path to s_i.

allel on a device. Each thread within a block will continue to request and process new rays until a total of N rays has been simulated on the device. So, instead of relying on a fixed stride, blocks receive their workload through an on-demand mechanism. This scheme allows for efficient inter-thread-block load balancing as described later in the text.

During simulation, a gain value is calculated for each ray by executing the following cycle, see also Fig. 9:

1. Get current sampling point s_i
2. Request a prism x to start ray from
3. Generate random starting point $r_{i,u}$ inside this prism
4. Generate ray $\overrightarrow{r_{i,u}s_i}$
5. Calculate the partial gains for $\overrightarrow{r_{i,u}s_i}$:
   (a) Find intersection point r_x between $\overrightarrow{r_{i,u}s_i}$ and prism x
   (b) Calculate length l_x of partial ray $\overrightarrow{r_{i,u}r_x}$
   (c) Calculate $partial\_gain(x)$ (2.3)
   (d) If the ray did not reach s_i, determine neighboring prism x', set $x := x'$, set r_x as the new start point and repeat the previous steps Item 5a to Item 5d.
6. Calculate $gain(\overrightarrow{r_{i,u}s_i})$ (2.5)
7. Sum gains of all rays for the sampling point s_i via atomic reduction

After a maximum of t iterations, each thread in a block has computed its share of rays pointing towards the sampling point s_i and adds its locally accumulated gain to the gain results of the other threads in a global reduce operation. This reduce operation is implemented as a sequential atomicAdd operation on the GPU (2.2).

Since each distinct ray potentially takes a different path through the gain medium, the execution times between threads can differ substantially. In case of a static mapping from threads to rays, i.e. *strided access*, one thread can end up calculating many rays with long paths. If there are still many of these rays remaining, the thread has to continue working for a long time, while other threads are already finished, causing load imbalances.

In order to improve load balancing between the threads and, thus, the utilization of the GPU multiprocessors, each thread block fetches a workload equal to its number of threads and executes it completely before determining a new workload. Therefore, only threads in the same block are stalled by the gain calculation for the longest ray assigned to the thread block. Meanwhile, threads in other blocks can pick up more work.

3.2.3. Multi-GPU implementation using the Message Passing Interface

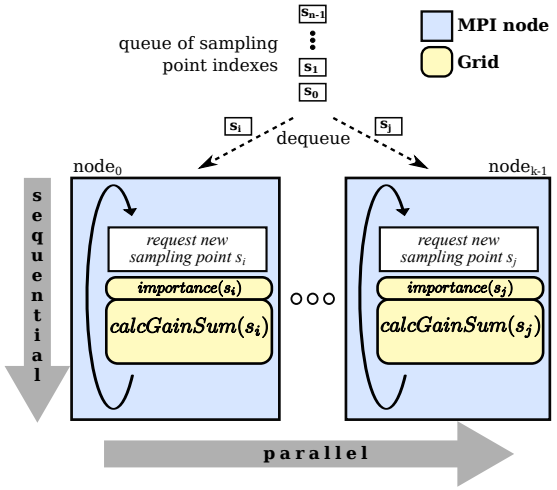

Figure 10: Partitioning of the workload to multiple GPU devices. All compute nodes inside a cluster store the whole mesh in global memory. The head node assigns sampling points to nodes on demand. After a node has finished computation it will request another sampling point.

Each Monte Carlo computation of Equation (2.2), calculates the $\Phi_{ASE}(s_i)$ value for a sampling point s_i independent from the other sampling points. Thus, each $calcGainSum(s_i)$ calculation is executed as a self-contained grid and therefore sampling points can be distributed to several devices.

In order to understand the workload distribution for HASEonGPU in a multi-GPU scenario, consider a message passing interface (MPI) environment [12] with k MPI processes, each handling a single GPU. In this environment, one MPI process is selected as the *master process* responsible for distributing the workload.

The master process holds a queue of all sampling points. Each of the other MPI processes holds all the necessary mesh data. It requests a sampling point from the queue and then begins with the ASE flux calculation. As soon as one process is finished with the calculation it can request new work independently from the other processes, see Fig. 10.

This form of load balancing is important in the case of varying workloads for each sampling point, see Section 3.3.2, which is the most common scenario encountered in real world applications. Although such a so-called master–slave configuration is known to be prone to communication bottlenecks when scaling to large k, the amount of data communicated between master and slave is minute and, thus, conflicts in resource sharing are negligible.

As a result of our load balancing strategy, the speedup is almost linear in the number of GPUs for moderate but realistic numbers of up to 64 nodes (see Section 4.4).

3.3. Techniques for efficient ray sampling

This section describes several techniques implemented to improve the quality as as well as the performance of the simulation algorithm. The latter can simply be expressed as a function of runtime. To assess changes in the precision of the simulation, we utilize the mean squared error (MSE) of the ASE calculation for a given set of rays which serves as a core metric to describe sampling improvements:

$$f(s_i) = \frac{1}{N} \sum_{u=0}^{N-1} gain(\overrightarrow{r_{i,u} s_i}) \tag{3.1}$$

$$f^2(s_i) = \frac{1}{N} \sum_{u=0}^{N-1} gain(\overrightarrow{r_{i,u} s_i})^2 \tag{3.2}$$

$$MSE(s_i) = \sqrt{\frac{f^2(s_i) - f(s_i)^2}{N}} \tag{3.3}$$

This metric can be computed for a single sampling point using only the variation in computed gain values and the number of rays. A lower MSE indicates that the simulation generated a more precise value for the sampling point. Methods to lower the MSE include redistribution of rays to focus on different areas of the medium (see Section 3.3.1) and increasing the number of rays (see Section 3.3.2).

3.3.1. Importance sampling (IS)

Importance sampling is a well known optimization technique when performing Monte Carlo simulations [13]. For each sampling point the IS algorithm decides which part of the gain medium will have the most influence on the calculation of Φ_{ASE}, see 10 and Fig. 11.

Importance sampling distributes N rays to P prisms of the active gain medium, where parts which contribute strongly to the ASE flux will be sampled by more rays, accordingly. This is done by introducing a center point c_x and a triangle surface area T_x for each prism x with $x \in \{0, \ldots, P-1\}$. From the center of each prism, a ray is traced to the gain medium and its preliminary gain used to

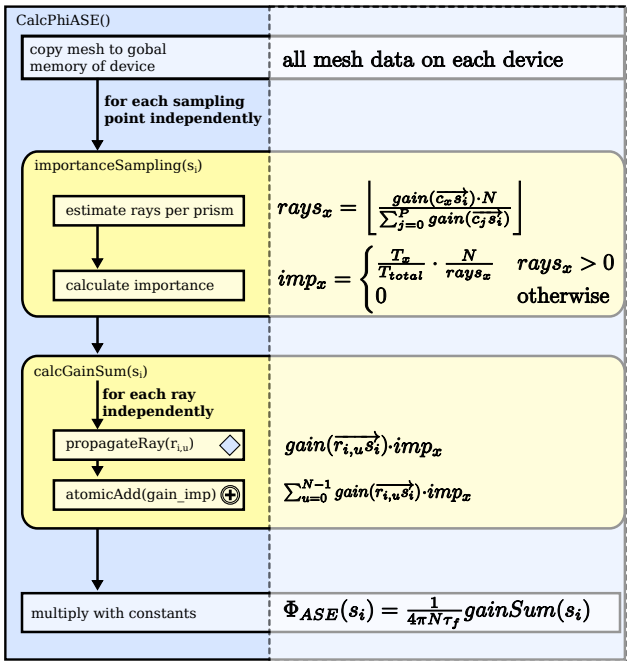

Figure 11: HASEonGPU program flow including importance sampling (IS). Importance sampling influences the distribution of rays in $calcGainSum(s_i)$. Varying numbers of rays per prism are compensated by using imp_x as a weighting factor on the gain calculation.

compute the number of rays to use in the Monte Carlo process.

Rays which start close to the sampling point s_i may produce very high gain values in the later Monte Carlo simulation, which is inevitable for the chosen method due to the form of the gain function (see Equation (2.5)), which shows a steep incline for ray lengths below a certain threshold (see Figure 12). Short rays are not excluded from the calculations but their effect is suppressed by multiplying their gain by a factor k_l. This factor reduces the number of short rays and increases their importance at the same time, thus keeping their contribution to the gain constant while at the same time reducing the probability of strong outliers contributing to the gain when sampling randomly. The threshold for this short distance is denoted by t_l:

$$k_l = |\overrightarrow{r_{i,u}, s_i}| \cdot \alpha \tag{3.4}$$

$$gain\_l(\overrightarrow{r_{i,u,x}s_i}) = \begin{cases} gain(\overrightarrow{r_{i,u}, s_i}) \cdot k_l & |\overrightarrow{r_{i,u}, s_i}| < t_l \\ gain(\overrightarrow{r_{i,u}, s_i}) & \text{otherwise} \end{cases} \tag{3.5}$$

where α is a constant factor. For our experiments,

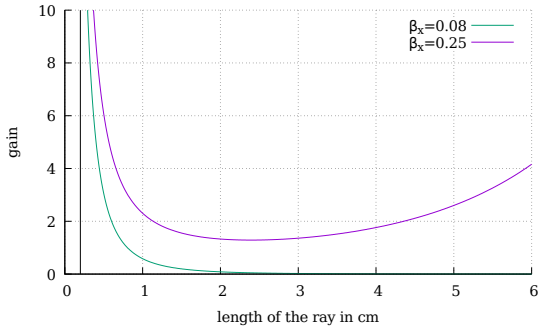

Figure 12: Behavior of the gain function (Equation (2.5)). Very short rays are modeled with extremely high gain values. In order to compensate, the importance sampling uses an additional factor k_l to scale down the gain in these cases. $\beta_x = 0.08$ shows the gain function for an unpumped gain medium, as is the case when starting our simulations, whereas $\beta_x = 0.25$ is the gain curve for a strongly excited medium.

$\alpha = \frac{1}{1000}$ was sufficient.

The number $rays_x$ of rays propagating from prism x to sampling point s_i can be derived from the expected gain for this prism by

$$rays_x = \left\lfloor \frac{gain\_l(\overrightarrow{c_x s_i}) \cdot N}{\sum_{j=0}^{P-1} gain\_l(\overrightarrow{c_j s_i})} \right\rfloor \tag{3.6}$$

In order to compensate for varying values of $rays_x$, the importance imp_x is used as a weighting factor and extends (2.5) to (3.8) where T_{total} is the sum of all triangle surface areas:

$$imp_x = \begin{cases} \frac{T_x}{T_{total}} \cdot \frac{N}{rays_x} & rays_x > 0 \\ 0 & \text{otherwise} \end{cases} \tag{3.7}$$

$$gain\_imp(\overrightarrow{r_{i,u,x}s_i}) = gain(\overrightarrow{r_{i,u}s_i}) \cdot imp_x \tag{3.8}$$

The result of importance sampling for s_i is a distribution of rays, giving the number of rays for each prism. This method greatly enhances the precision of the simulation as it reduces strong peaks in local gain that arise due to overestimating the contribution of prisms with high expected gain when uniformly distributing rays over the medium with constant weighting.

The impact of importance sampling in $\Phi_{ASE}(s_i)$ is seen when calculating the difference

$$\Phi_{ASE\Delta} = |\Phi_{ASE\ IS}(M) - \Phi_{ASE\ US}(N)|$$

between the results obtained by importance sampling (IS) and uniform sampling (US) starting with

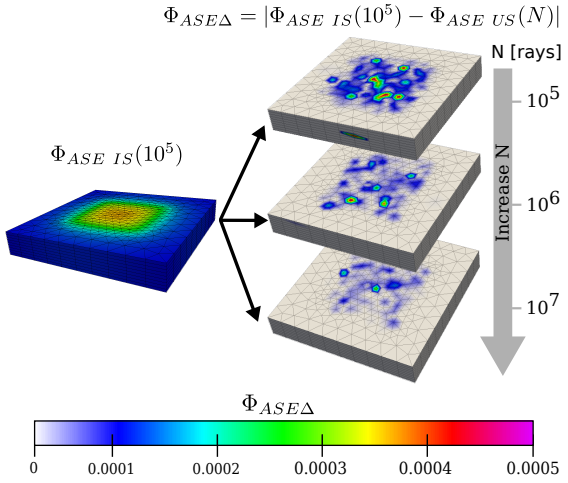

Figure 13: Left: $\Phi_{ASE\ IS}(10^5)$. Right: $\Phi_{ASE\Delta}$. With increasing number N of rays for the simulation with uniform sampling (US), the difference between $\Phi_{ASE\ IS}(10^5)$ and $\Phi_{ASE\ US}(N)$ constantly decreases as the result for US approaches the result obtained by importance sampling with orders of magnitude less rays.

$N, M = 10^5$ rays per sampling point as depicted in Fig. 13. The number N of rays per sampling point was increased twice, each time by one order of magnitude, resulting in sampling with 10^6 and 10^7 rays, respectively.

When increasing N, $\Phi_{ASE\Delta}$ decreases and the result obtained with US approaches the result obtained with importance sampling and fixed number of $M = 10^5$ rays, see also Fig. 14.

We conclude that importance sampling can achieve the same result as obtained by uniform sampling with orders of magnitude fewer rays, thus increasing the efficiency of Monte Carlo integration by reducing variance and simulation runtime.

3.3.2. Adaptive sampling (AS)

Adaptive sampling (AS) utilizes the MSE directly to determine the quality of the result. Since most sampling points s_i exhibit low $MSE(s_i)$, there is no need to sample them with a high number of rays in order to obtain the ASE gain with a sufficient precision.

However, in few cases spurious outliers need to be sampled with a higher resolution to reduce the overall variance of the ASE gain. This can be achieved by an adaptive method which allows for removing strong peaks in the result of the simulation by resampling these points with more rays. The number of rays is increased with each iteration, until the desired MSE threshold is met or a defined maximum number of rays is reached, see Fig. 15.

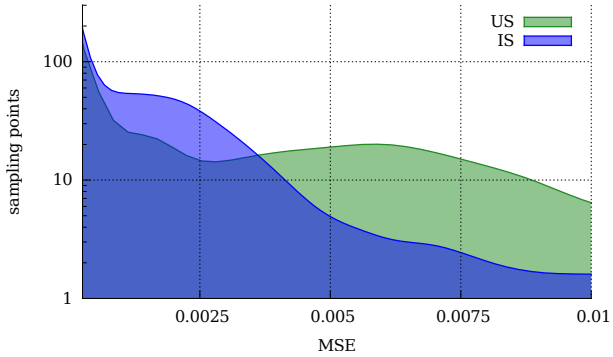

Figure 14: Histogram comparing the mean squared error (MSE) distribution, Equation (3.3), for importance sampling (IS) and uniform sampling (US). Importance sampling shifts the histogram towards lower MSE values for most sampling points while in some cases maintaining high MSE values, thus not reducing the global maximum MSE value.

Note that after each adaptive resampling the importance sampling step needs to be executed again, since the number of rays has been changed, thus changing the overall distribution of rays. Potentially unbalanced workloads, caused by a varying number of rays per sampling point in a multi-GPU scenario are mitigated by the MPI load balancer, see Section 3.2.3.

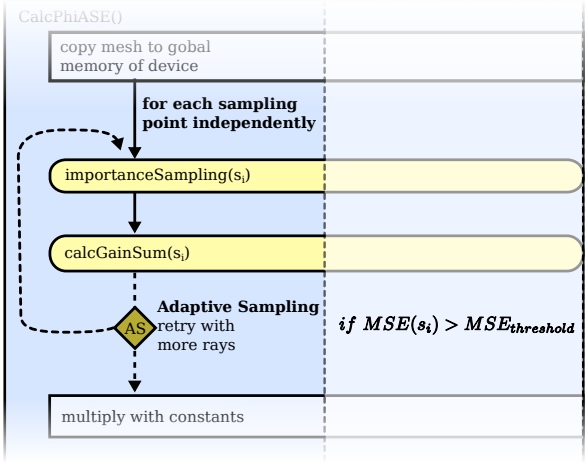

Figure 15: HASEonGPU program flow extended by adaptive sampling (AS). The mean squared error $MSE(s_i)$ of the ASE calculation for a given set of rays is compared to a fixed threshold. If the threshold is exceeded, both $importanceSampling(s_i)$ and $calcGainSum(s_i)$ are restarted using a higher number of rays to increase sampling resolution.

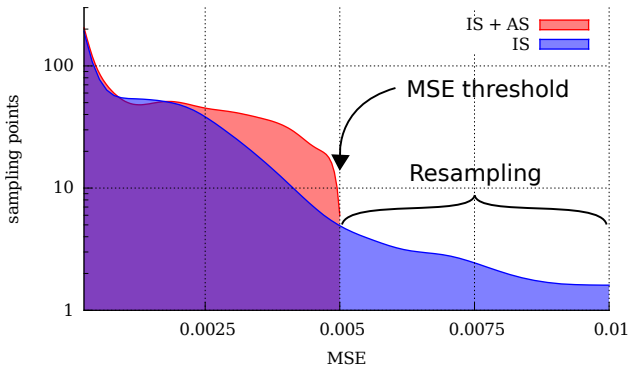

Figure 16: Histogram comparing MSE values for importance sampling (IS) only and importance sampling with adaptive sampling (IS+AS). The addition of adaptive sampling globally removes MSE values higher than a given threshold from the distribution.

For comparison, the impact of importance sampling on the MSE values is shown in Fig. 14. Importance sampling alone already reduces the MSE values for many sampling points significantly, albeit not evenly.

By employing adaptive sampling, the MSE values for all sampling points are reduced to a global, user-defined MSE threshold as seen in Fig. 16 which uses an MSE threshold of 0.005.

All sampling points s_i with $MSE(s_i) > 0.005$ were resampled with more rays per sampling point to lower their MSE value, while the number of rays used for all other sampling points remained unchanged.

Adaptive sampling does not aim to reduce the average MSE over all sampling points, but instead reduces the maximum MSE values of outliers. Thus, AS gives lower errors and a smaller error distribution while maintaining roughly the time to solution.

3.4. Methods summary

It is left to compare the sampling methods US, IS, and AS by runtime and precision. In Fig. 17, the time to reach an MSE value of 0.01 was measured. It is evident that IS+AS performs much better than US or IS alone. We conclude by stating that thread parallelization over rays grouped by proximity allows for efficient caching. Additional load balancing between thread blocks and between MPI processes allows for excellent scaling for all relevant application cases. Time to solution is furthermore reduced drastically by the combination of importance sampling and adaptive sampling which optimizes the distribution of rays over the medium.

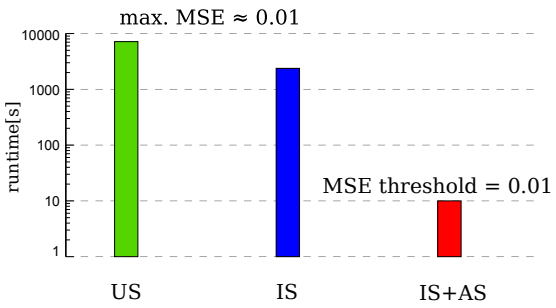

Figure 17: Comparison of runtimes for the presented sampling methods with same MSE value. The addition of AS greatly reduces runtime.

4. Validation of the results obtained with HASEonGPU

4.1. Simulation input data used for validation

In the following, the simulation setup uses a cylindrical gain medium with parallel top and bottom surfaces as depicted in Fig. 18 in order to compare the simulation results to experimental results obtained with a medium of the same form and dimensions. The simulation input as listed in Table 2 was chosen to match the experimental setup.

Points per plane	421
Sampling points	4210
Planes	10
Triangles	812
Prisms	7308
Total height of medium	0.7 cm
Surface area of medium	$\pi \times 3\ cm^2$
Medium material	Yb^{3+}:YAG ceramic
Yb^{3+} doping	2 $at.\%$
Cladding material	Cr^{4+}:YAG
Cr^{4+} doping	0.25 $at.\%$
Spectrum	Polychromatic

Table 2: Input parameters for validating the simulation results in comparison to experiment. Crystal doping is given for both the gain medium and the cladding. The polychromatic input cross-sections are plotted in Figure 19. Note that the overall size of the problem is comparable to that of the test setup given in Table 1.

In order to study the influence of various approximations, we performed simulation using both a monochromatic spectrum peaked at the wavelength of maximum intensity emission and the full polychromatic spectrum. We furthermore studied the impact of reflections on the validity of the simulation results.

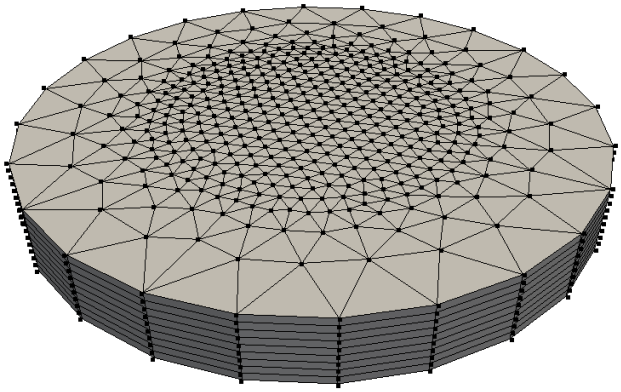

Figure 18: Non-uniform sampling of the circular active gain medium that was used for comparison of HASEonGPU simulation results to experimental data. In z-direction the medium is sampled by 9 slices or 10 planes, respectively.

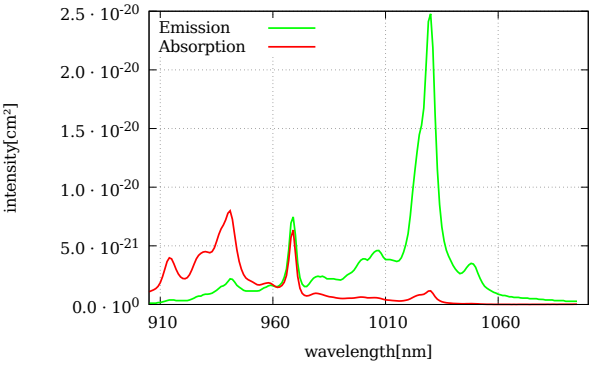

Figure 19: Polychromatic input spectra used in simulations. In the monochromatic setup the emission spectrum was approximated by a monochromatic spectrum of wavelength 1030 nm, marking the position of the maximum emission intensity.

4.2. Technical description of the environment used for validation

The validation simulations were conducted on a GPU cluster consisting of 16 compute nodes connected by InfiniBand over a Mellanox MSX6036F (FDR) IB switch, each equipped with a quad core Intel Xeon CPU E5-2609 CPU (2.40GHz), 64GB RAM and 4 NVIDIA Tesla K20m GPUs. Each GPU contains 5GB GDDR5 RAM and 2496 CUDA cores with a total peak performance of 3520 GFLOP/s. Job submission is handled by TORQUE [14] with Maui [15] as scheduling backend.

4.3. Simulation validation through comparison to experiment

We compare the small signal gain at a wavelength of 1030 nm derived from the simulation and the results obtained in the experiment described in [6]. Note the impact of the reflections as well as the polychromatic approach in our model compared to experimental results.

Fig. 20 shows a variety of simulation configurations and their influence on the gain development over time, compared to the experimentally measured gain. Note that simulation values are discrete and linearly interpolated to guide the eye.

In this particular experiment, monochromatic simulations without reflections (d) show very similar, but smaller, gain values in comparison to the polychromatic simulation with reflections (c).

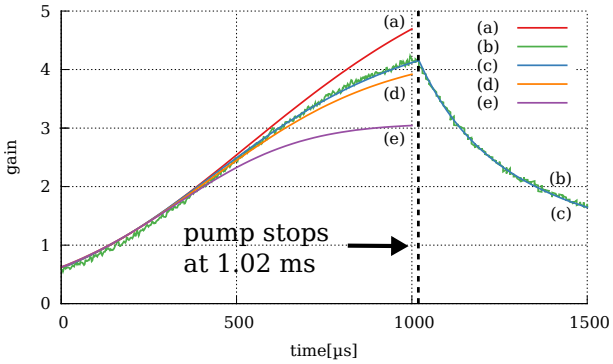

Figure 20: Simulation results compared to experimental measurements: (a) polychromatic no reflections; (b) experimental measurement; (c) polychromatic with reflections; (d) monochromatic no reflections; (e) monochromatic with reflections. Close-to-perfect agreement is found between experimental results and results obtained using a simulation with a polychromatic spectrum and reflections. Simulation results are discrete and interpolated linearly to guide the eye.

Interestingly, results can get worse when more physical details are considered. Adding reflections (e) but keeping the spectrum monochromatic overestimates the ASE impact and consequently underestimates the gain. We mainly attribute this to reflections increasing the average length of raytraces, especially due to total internal reflection. Thus, the energy stored in the gain medium is reduced more severely by ASE than without reflections.

Simulating only a polychromatic spectrum (a) overestimates the gain. As the average emission intensity is lower than the peak value at 1030 nm,

see Fig. 19, the reduced ASE flux leads to an overall higher gain. In comparison, the simulation configuration including both reflections and a polychromatic spectrum (c) matches the measured values (b) within the measurement uncertainty and validates our simulation approach.

For the same simulation input parameters, Fig. 21 visualizes the temporal rise in ASE at five consecutive time steps. Using (2.1) and (2.4), it illustrates $\frac{dn}{dt}\big|_{ASE}$ in the sliced gain medium by plotting the local value of

$$\left.\frac{dn}{dt}\right|_{ASE} = \int_{\lambda} g_0(\lambda) \cdot \Phi_{ASE}(\lambda)\, d\lambda \qquad (4.1)$$

for each sampling point.

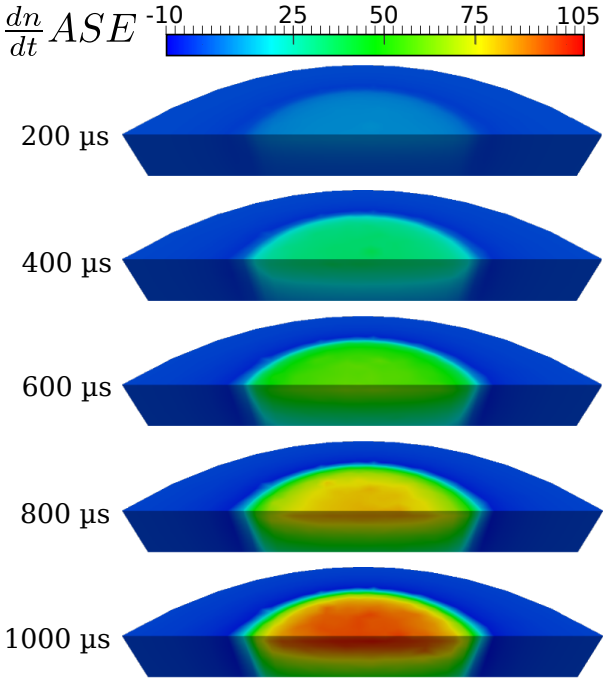

Figure 21: Development of $\frac{dn}{dt}\big|_{ASE}$ in a sliced gain medium for five consecutive timesteps while the medium is pumped. Values in the pumped area are increase until the pump stops.

4.4. Performance benchmarking

In Fig. 22, the runtime of the original single threaded algorithm from [6] is compared to the non-adaptive parallel ASE flux algorithm (IS) with varying numbers of rays to demonstrate scaling for different workloads. All simulations were performed without reflections to ensure comparability. In order to increase the number of GPUs, MPI [12] was

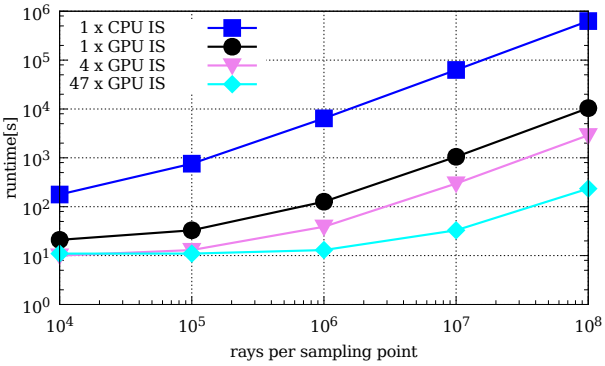

Figure 22: Runtime of the sequential IS algorithm implementation compared to the parallel IS algorithm implementation. Comparing single CPU and single GPU setups, the time to solution is reduced by two orders of magnitude for intensive workloads. Further improvements can be achieved by using more GPUs.

used to distribute the sampling points to all available devices. A sequential overhead for device allocation on the nodes and not fully occupied GPUs are the reasons for the lack of runtime improvement observed for small workloads with a low number of rays.

Adaptive sampling usually does not only eliminate outliers, but also reduces the runtime. By using a predefined MSE-threshold, the precision of the simulation can be adjusted in terms of this threshold rather than simply increasing the number of rays for all the sampling points. Usually, only a small subset of sampling points actually needs to be sampled by a high number of rays and an insignificant increase in runtime can be sufficient to lower the maximum MSE values below the desired threshold, as seen in Fig. 23. Note, that some values in this figure actually display almost the same runtime, albeit much lower MSE values, since the computation always succeeded to stay below the given threshold with very little additional effort. This is indicated by the black rectangle in Fig. 23.

For any adaptive algorithm, simulation times for different sampling points can vary significantly, see Section 3.3.2. Inter-process load balancing via MPI as described in Section 3.2.3 allows to account for this variation in ray number, however it still exhibits a constant initialization overhead of 5 s for sending the initialization data to all MPI processes.

Apart from this, distribution of the computation to multiple devices scales well, as seen in Fig. 24. This result shows that for the same work-

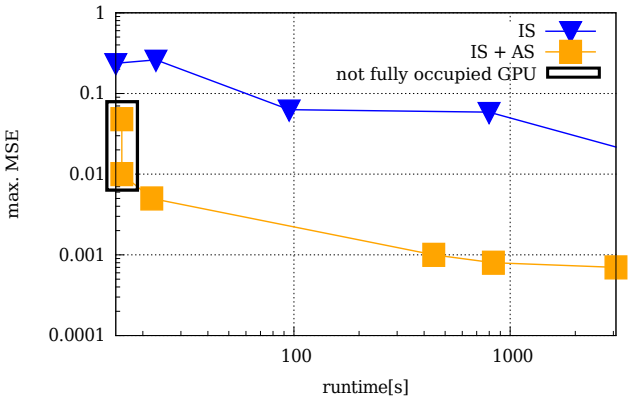

Figure 23: Comparison of maximum MSE values of IS to IS+AS. The same runtime yields much lower maximal MSE values when adding AS.

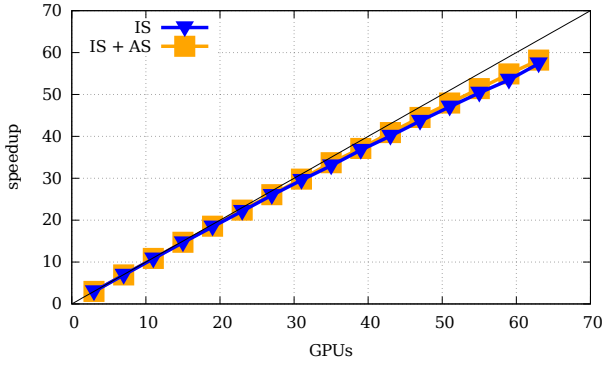

Figure 24: Strong scaling on multiple GPU devices. Benchmarks with up to 64 GPUs in a cluster show that load balancing as implemented in HASEonGPU results in almost linear speedup behavior.

load adding GPU compute processes gives nearly linear speedup with the number of GPUs added.

4.5. Conclusion and outlook

We have presented the open-source, adaptive multi-GPU code HASEonGPU for computing the ASE flux in laser gain media. Inclusion of dynamic load balancing achieves large speedups on the order of two magnitudes of the multi-GPU implementation compared to previously existing CPU implementations. Comparison to experiment shows that HASEonGPU precisely predicts the absolute gain values during and after pumping of the medium.

HASEonGPU implements Monte Carlo integration of the gain using a ray tracing method that can include local information on the gain medium at high spatial resolution using the NVIDIA CUDA extension to C. Dynamic adaptive resolution guarantees to globally adhere to a design precision while at the same time reducing computation time. Including several sampling methods allows for significant reduction of time to solution.

For easy integration into existing workflows, HASEonGPU provides a *MATLAB* interface. By using this interface, experimental details like cladding, surface coating, refractive indexes, polychromatic laser pulses and reflections on the upper and lower surface can be integrated into the simulation. This allows for detailed simulations of application cases and future integration into multi-physics simulation chains relying on an interface widely adopted within the high power laser community.

Future improvements in HASEonGPU will address reflections on the lateral sides of the gain medium. Moreover, it will be helpful to further increase the maximum number of rays, which is currently fixed to 5×10^8 rays per sampling point due to the available GPU memory. This could be done by splitting the simulated rays in groups and calculating these groups iteratively, combining the results afterwards. Such an approach would also allow for higher scaling of HASEonGPU, as currently the granularity of the simulation is limited by the number of sampling points. Regardless of how many GPUs are used for calculation, no further speedup will be obtained as soon as the number of GPUs exceeds the number of sampling points, essentially making an increase in GPUs futile.

However, this limitation does not influence the usability of HASEonGPU strongly, as the several sampling methods implemented strongly reduce the need for high ray number sampling. Thus, HASEonGPU presents an important tool for providing scalable parallel simulations of ASE flux for the upcoming generation of high-power laser systems at high spatial and temporal resolution.

With the growing availability of mid-size high-performance compute clusters using GPUs, this will allow to integrate HASEonGPU into a chain of multi-physics simulations, as the time to solution now enables repetitive simulations with integrated feedback on the local properties of the gain medium and its surroundings. With HASEonGPU being fully open source, adapting the code to more sophisticated applications is straightforward and in many cases only requires a change of the local gain calculation and not a change in the ray tracing algorithm itself.

First validation tests show very good agreement with measurements, promising HASEonGPU to be a tool for real-world applications that can be used by laser designers without a need to deeply delve into GPU programming.

Acknowledgment

We acknowledge support by the CUDA Center of Excellence [16].

[1] G. I. Peters, L. Allen, Amplified spontaneous emission I. The threshold condition, J. of Phys. A: Gen. Phys. 4 (2) (1971) 238–243.
[2] L. Allen, G. I. Peters, Amplified spontaneous emission and external signal amplification in an inverted medium, Phys. Rev. A 8 (4) (1973) 2031–2047.
[3] G. J. Linford, E. R. Peressini, W. R. Sooy, M. L. Spaeth, Very long lasers, Appl. opt. 13 (2) (1974) 379–390.
[4] D. Albach, J.-C. Chanteloup, G. l. Touzé, Influence of ASE on the gain distribution in large size, high gain Yb^{3+}:YAG slabs, Opt. express 17 (5) (2009) 3792–3801.
[5] MATLAB, a numerical computing environment, `http://www.mathworks.de/products/matlab`, accessed: 2014-08-03.
[6] D. Albach, Amplified spontaneous emission and thermal management on a high average-power diode-pumped solid-state laser-the Lucia laser system, Ph.D. thesis, Palaiseau, Ecole polytechnique (2010).
[7] B. N. Delaunay, Sur la sphère vide, Bull. of Acad. of Sci. of the USSR (1934) 793–800.
[8] NVIDIA corporation, `http://nvidia.com`, accessed: 2014-08-03.
[9] CUDA C programming guide, `http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`, accessed: 2014-08-03.
[10] M. Saito, A variant of mersenne twister suitable for graphic processors, `http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/MTGP`, accessed: 2014-12-09 (2010).
[11] CUDA occupancy calculator, `http://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls`, accessed: 2014-07-06.
[12] J. Dongarra, D. Walker, E. Lusk, B. Knighten, M. Snir, A. Geist, S. Otto, R. Hempel, E. Lusk, W. Gropp, et al., MPI - a message-passing interface standard, Int. J. of Supercomp. and High Perform. Comput. 8 (3-4) (1994) 165.
[13] E. C. Anderson, Monte Carlo methods and importance sampling, University lecture notes for statistical genetics, `http://ib.berkeley.edu/labs/slatkin/eriq/classes.htm#GuestLectAnchor` (1999).
[14] A. Computing, TORQUE resource manager, `http://www.adaptivecomputing.com/products/open-source/torque`, accessed: 2014-07-05.
[15] A. Computing, Maui cluster scheduler, `www.adaptivecomputing.com/products/open-source/maui`, accessed: 2014-08-03.
[16] Dresden CUDA Center of Excellence, `https://ccoe-dresden.de`, accessed: 2014-12-11.
[17] E. Zenker, C. Eckert, M. Melzer, F. Liebold, D. Albach, HASEonGPU github repository, `https://github.com/ComputationalRadiationPhysics/HASEonGPU`, accessed: 2014-12-09.

Appendix A. Overview of the HASEonGPU application interface

The application is available as a command-line tool and reads simulation data from plain text files. It was improved over the original application developed by D. Albach [6], adding support for polychromatic spectra, adaptive sampling, multi-GPU computation and reflections.

For easy access, use of the *MATLAB* compatible interface, see Appendix A.1, is highly recommended. Experienced users with the intention to call the application directly should consult the README file, the example code provided or the source code [17] for more details.

Appendix A.1. Installation and Usage

The application was tested and developed under a Linux environment and runs on NVIDIA graphics cards with compute capability 2.0 (Fermi generation) or higher. It can be built by running `make` inside the application directory, provided that `make` (tested with 3.82), `gcc` (tested with 4.6.2) and CUDA (tested with 5.0) are installed. This will also create a *MATLAB* file `calcPhiASE.m` in the application directory.

For running the application from *MATLAB*, 2 steps are necessary to take in advance:

1. Include `calcPhiASE.m` from the application directory into your *MATLAB* path
2. Call the *calcPhiASE* function from the *MATLAB* script:

$$[phiASE,\ MSE,\ nRays] = calcPhiASE(args)$$

Appendix A.1.1. HASEonGPU input arguments

An overview of the most important input arguments is given in the list below:

Mesh information

- Structure (points, triangles, prisms)
- Thickness of a mesh slice
- Number of planes
- Mesh refractive indexes (& surroundings)
- Reflectivities of mesh planes bottom and top

Properties

- Stimuli β in sampling points and prisms

- Crystal Fluorescence τ_f
- Cladding (use of different materials)
- Doping of the active gain medium N_{tot}

Laser information

- Absorption cross section σ_a
- Emission cross section σ_e

Algorithm information

- Maximum number of rays for adaptive sampling
- MSE-threshold for adaptive sampling
- Decide whether to activate reflections

A detailed description of the *MATLAB* compatible interface and the required arguments including units is given in the README file.

Appendix A.1.2. Output

The simulation returns 3 data sets:

phiASE is a matrix of $\Phi_{ASE}(s_i)$

MSE is a matrix of $MSE(s_i)$ reached

nRays is a matrix of number of rays used for the Monte Carlo integration for each sampling point

The return variables are represented as two-dimensional matrices in which columns are slice indexes and rows are point indexes. For example, the matrix for the example gain medium has 321 rows and 10 columns, multiplying to 3210 sampling points. The value for the i-th point and the j-th slice can then be obtained in *MATLAB* by invoking `value = values(i,j)`.

Appendix B. Description of the source code structure

This section describes the structure of the HASEonGPU source code and provides a brief overview from the parsing of the input data up to the return of output data. The following paragraphs show a typical run of the program, the hierarchy of function calls and the files in which they are contained. It is intended as a starting point for understanding the source code in detail. All file paths are related to root folder of the repository [17]. We assume to use simulation data similar to that in the example folders (`example/`). As a graphical reference, see the program flow diagrams Figures 6, 10, 11 and 15.

Appendix B.1. Calling the HASEonGPU code

Starting from a *MATLAB*-based (like `example/matlab_example/laserPumpCladdingExample.m`), the *MATLAB* interface (`src/calcPhiASE.m`) is called through the *MATLAB* function `calcPhiASE()`. The interface then starts the actual HASEonGPU code (`src/main.cc`).

Appendix B.2. Preparing the actual computation

The `main` function first parses the command line parameters as well as the input data for the simulation (`src/parser.cc`, functions `parseCommandLine()`, `parseMesh()`) and stores these parameters in a `Mesh` class (`src/mesh.cc`) and the structs `ExperimentParameters` and `ComputeParameters` according to their meaning for the simulation. Based on the parameters `deviceMode` and `parallelMode`, the correct communication entry point is selected (files `src/calc_phi_ase_mpi.cc`, `src/calc_phi_ase_threaded.cc`, `src/calc_phi_ase_graybat.cc`). For example, the MPI communication corresponds closely to what is described in Section 3.2.3 and Figure 10, where one process becomes the master (`mpiHead()`) and the others are slaves (`mpiCompute()`)that will perform the actual computation (in `src/calc_phi_ase.cu`).

Appendix B.3. Host part of the computation

In `src/calc_phi_ase.cu`, the function `calcPhiAse()` (see Figure 11) creates all datastructures on the GPU that are required for the subsequent kernels, initializes the random generator and enters the loop depicted in Figure 15 which executes multiple GPU kernels in each iteration. The kernels `importanceSamplingPropagation()`, `importanceSamplingDistribution()` and `mapRaysToPrisms()` are located in `src/importance_sampling.cu` `src/map_rays_to_prisms.cu`. The core part of the computation is entered through the GPU kernels `calcSampleGainSum()` or `calcSampleGainSumWithReflection()`.

Appendix B.4. GPU part of the computation

The mentioned kernels are located in `src/calc_sample_gain_sum.cu`, and their parallel flow is depicted in Figure 8. In the case of reflecting rays, the partial rays (Section 2.4, Equation (2.6)) are computed inside

`propagateRayWithReflection()`, which is located in `src/propagate_ray.cu`. Finally, the propagation `propagateRay()` itself is computed with the helper functions from `src/reflection.cu` and `src/geometry.cu`.

Appendix B.5. Gathering of output data

The gain values of each ray are summed inside the `calcSampleGainSumWithReflection` kernel and copied back to the host after the kernel execution. If the MSE value (see Section 3.3) is low enough, the gain for the sample point is returned to the master process and the next sample point requested. After all sample points are processed, the list of gain values as well as the used number of rays and corresponding MSE values are written to the `output` folder through the helper functions found in `src/write_to_file.cc` and `src/write_matlab_output.cc` where it can be read into *MATLAB* again.