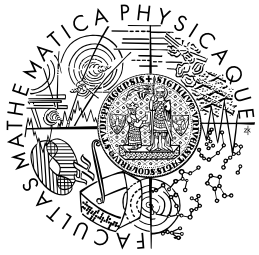


FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Federico Forti

Efficient GPU Path Tracing in Solid Volumetric Media

Department of Software and Computer Science Education

Supervisor of the master thesis:	Mgr. Oskar Elek, Ph.D.
Study programme:	Computer Science
Study branch:	Computer Graphics and Game Development

Prague 2018

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ..... signature of the author

Title: Efficient GPU Path Tracing in Solid Volumetric Media

Author: Federico Forti

Department: Department of Software and Computer Science Education

Supervisor: Mgr. Oskar Elek, Ph.D., Department of Software and Computer Science Education

Abstract: Realistic Image synthesis, usually, requires long computations and the simulation of the light interacting with a virtual scene. One of the most computationally intensive simulation in this area is the visualization of solid participating media. This media can describe many different types of object with the same physical parameters (e.g. marble, air, fire, skin, wax ...). Simulating the light interacting with it requires the computation of many independent photons interactions inside the medium. However, those interactions can be computed in parallel, using the power of modern Graphic Processor Unit, or GPU, computing. This work present an overview over different methodologies, that can affect the performance of this type of simulations on the GPU. Different existing ideas are analyzed, compared and modified with the scope of speeding up the computation respect to the classic CPU implementation.

Keywords: GPU Volumetric Path Tracing, CUDA

I would like to thank Oskar Elek for supervising my thesis and all the people that have released volumetric data which makes this thesis possible. I would also like to thank all the people that have supported me during this time and the professors who showed me how to approach a research project.

Contents

Introduction

Motivation

In the last decade, graphics processor units have emerged as a cheap and powerful data parallel computational platforms. The architectural innovations, in such processors, have transformed the GPU from fixed-function hardware blocks to programmable units for general purpose computations (GPGPU). The gap between floating-point capability of GPU versus CPU is increasing on a year base and other hardware trends encourages to parallelize the existing serial algorithms (Nvidia [2017], McCool et al. [2012]). The evolution of new programming languages, like CUDA- Compute Unified Device Architecture- and OpenCL, gives to the programmer more flexibility in the usage of the GPU processors for general purpose computing. This processing power can be used today to address many of the computational problems, not solvable in adequate time with the previously available resources. The task we are going to focus on is realistic image synthesis of participating media. Most of the light effects, visible in nature, can be described by this type of media. Indeed, those are usually used to describe the property of a generic material. Realistic rendering of those materials requires physical simulations of the light interacting with the media. Those interactions can be thousands or millions based on the type of participating media we are going to simulate. For this reason, simulation of the light can take a long time. Efficiently, using GPU to compute in parallel, those light interactions can improve the time performance to get the final render.

Possible Applications

Realistic image synthesis of solid object interests a wide range of areas. In architecture and product visualization can be used to present a building or a product. In virtual prototyping can be used to predict the appearance of an object before creating it and modifying it in according to that. Realistic Real time rendering is used inside many of the most famous games. Another possible application comes from volume visualization, where realistic rendering plays a central role in 3D shape perception. Most of the available implementation, in this sense, focuses more on performance, rather than perfect light transport simulation. The original idea for this project comes from the 3D printing field. In a recent paper Elek et al. [2017] have used volumetric rendering for accurate prediction of texture colors in 3D printing. This prediction stage is the biggest performance bottleneck of all the pipeline. The reason is that print materials have usually high optical density and simulating the appearance of those materials requires computing thousands of scattering interactions. This work aims to give a comprehensive view on how to optimize a volumetric path tracing for the GPU architecture, taking in consideration the advantages and disadvantages of different design choices. The solutions found can be used inside a completely GPU based renderer or as a stage of longer pipeline for rendering.

Thesis Structure

The thesis has six chapters:

- the first chapter describes the problem that the thesis aims to solve.
- The second chapter shows the work that has already been done on solving this problem.
- The third chapter describe the theoretical background necessary for the methods that the thesis implements.
- The fourth chapter shows the method implemented and the different behavior of each method.
- The fifth chapter describes some implementation details on the solution adopted and the software created.
- Finally, the sixth chapter shows some results obtained with the GPU methods implemented and a comparison with a standard CPU algorithm.

1. Problem Statement

The chapter describes the problem that the thesis aims to solve and describe more in detail the different characteristics of Participating Media.

1.1 Radiative Transport Problem

The problem that the thesis aims to solve is a radiative transport problem and the radiometric quantity, that we are searching for, is the radiance L. This quantity describes how much light arrives from a direction ω on an hypothetical differential area perpendicular to that direction $dA^{\perp}$ Jarosz [2008].

$$L = \frac{d^2\Phi}{d\omega \; dA^{\perp}} = \frac{d^2\Phi}{d\omega \; dA \; \cos\theta} \tag{1.1}$$

Where Φ, expressed in Watt (Joule/sec), is the radiant flux, i.e time rate of flow of radiant energy. When this light energy interact with a participating medium, than it can scatter or be absorbed. In both cases the light energy is attenuated. This attenuation is usually called extinction (see van de Hulst [1981]).

$$\text{Extintion} = \text{scattering} + \text{absorption}$$

In this type of problems a distinction is done between solid objects, which have a well defined boundary, and other media, like gases or liquids. In this work, we are concentrating on the first type of media, considering also cases where the object boundary defines a change on the index of refraction of the medium.

1.2 Participating Media

We can think of a participating media as an agglomerates of small particles. Each of those particles can be described by the following characteristics identified by van de Hulst [1981]:

- C_{sca} (scattering cross section): is the area of the particle which is scattering light;
- C_{abs} (absorption cross section): is the area of the particle that is absorbing light;
- F (scattering diagram of the particle): describes the scattering behavior and it can be used to obtain the more easily manipulable phase function f_p.

Given those characteristics, the final medium is described by the density function $\rho(x)$ of these particles for any point x inside the medium. When a photon arrives to a point x, inside a medium which contains some particles, four types of event can happen:

Absorption: light is absorbed by the particles in that point x, proportionally to their C_{abs}; the amount of light absorbed is defined as the absorption coefficient $\sigma_a(x)[m^{-1}]$.

$$\sigma_a(x) = C_{abs} \cdot \rho(x) \tag{1.2}$$

Out-scattering and In-scattering: light can be out-scattered or in-scattered in the point x , based on the C_{scat} of the particles in that point; the amount of light scattered is defined as the scattering coefficient $\sigma_s(x)[m^{-1}]$.

$$\sigma_s(x) = C_{scat} \cdot \rho(x) \tag{1.3}$$

Emission: finally the medium can emit more energy in the point x; we will call the radiance emitted by the medium in a point x and direction ω, $L_e(x, \omega)$.

There are also some derived quantities important to describe interactions with a participating media. Those are the extinction coefficient σ_t, defined as

$$\sigma_t(x) = \sigma_a(x) + \sigma_s(x) \tag{1.4}$$

and the albedo of the medium α, defined as

$$\alpha = \frac{\sigma_s}{\sigma_t}. \tag{1.5}$$

It is important to notice that it is always possible to derive the first two quantities in equations 1.2 and 1.3 from the last two equations 1.4 and 1.5, and vice versa. By integrating the extinction coefficient along a line segment l, we get the optical thickness of our material τ:

$$\tau(l) = \int_l \sigma_t(x)dx. \tag{1.6}$$

Now, we have all the elements to completely describe the change of radiance inside a medium using the radiative transport equation (RTE), which can be defined with the following equation:

$$(\omega \cdot \nabla)L(x, \omega) = L_e(x, \omega) + \sigma_s(x)L_i(x, \omega) - \sigma_a(x)L(x, \omega) - \sigma_s(x)L(x, \omega). \tag{1.7}$$

where $L_i(x, \omega)$ is the in-scattered radiance at x, through the direction ω, coming by other scattered radiance. The equation 1.7 includes all the light interactions that we have previously described, which are in this order: emission, in-scattering, absorption, out-scattering.

1.3 The Volume Rendering Equation (VRE)

Scenes, containing solid participating media, are usually modeled as a volume $\mathcal{V}$ and a boundary $\partial\mathcal{V}$, where $\mathcal{V} \cap \partial\mathcal{V} = \emptyset$. As shown by Matthias Raab and Keller, the transport can be described differently, based on the position considered on the volume. If the point is on the surface $\delta\mathcal{V}$, we use the classic Rendering Equation

$$L(x, \omega) = L_{e,\delta\mathcal{V}}(x, \omega) + \int_{S^2} f_s(\omega, x, \omega')L(x, \omega')|cos\theta_x|d\sigma(\omega') : \tag{1.8}$$

where S^2 is the set of all directions (in this work we are using only the hemispherical formulation), f_s is the bidirectional scattering distribution function, which describes the scattering behavior at a point x on the surface (in particular is the fraction of incident differential radiation reflected into the direction ω'). Finally, $cos\theta_x$ is the cosine of the angle between direction ω' and the surface normal at x (the reader interested on more informations about this equation can find its complete definition in the work of Kajiya [1986]). On the other hand, if the point is inside the participating media we have to consider the *Volumetric Rendering Equation* (VRE). This equation is obtained integrating the radiative transport equation, that we have described in the previous paragraph, along straight light rays until the next intersection point x_s of the ray with a surface. The equation is:

$$L(x,\omega) = \int_0^s T(x,x_t)\cdot L_v(x_t,\omega)dt + T(x,x_s)\cdot L(x_s,\omega) \tag{1.9}$$

where

$$L_v(x,\omega) = L_{e,\mathcal{V}}(x,\omega) + \sigma_s(x)\int_{S^2} f_p(\omega,x,\omega')L(x,\omega')d\sigma(\omega') \tag{1.10}$$

and $T(x,x_t)$ is the transmittance function defined by the *Beer-Lambert-Bougeuer law* as

$$T(x,x_t) = e^{-\tau(||x-x_s||)} \tag{1.11}$$

with τ the previously described optical thickness of the medium.

A more general equation, which combines both the equations 1.8 and 1.9, is the path integral formulation. In this formulation the space of integration is the union of spaces containing paths with specific length, i.e $\mathcal{P} := \bigcup_{k\in\mathbb{N}} \mathcal{P}_k$, where P_k is the space of paths $\overline{x}$ of length k. The reader interested can find its derivation in the work of Veach [1997].

From a mathematical point of view, the volume rendering equation is a Fredholm integral equation of the 2nd kind. Solving this type of equation analytically is very difficult, even applying simplifications on medium and light transport. Many numerical estimations methods have been studied in the literature. Those methods can be divided in two groups: deterministic methods and stochastic methods. The first ones are based on a discretization of the domain of integration and the solution of large linear systems. However, this type of system is usually affected by some limitations and the recent research has focused more on the second types of methods. In the next chapter, we are going to cover exactly this latter type of methods, giving an overview on some of the most important ones.

1.4 Porting to GPU

GPU-based programs have a number of limitations on when and how memory can be accessed. Computation speed increases at a much faster rate than memory access speeds, which means that, to improve time performance, special care should be taken to maximize bandwidth usage. Simulating light transport, inside participating media, can also be not obvious. For this scope, one of the most used

techniques in the industry of computer graphics is the Monte Carlo path tracing. This is a stochastic method which uses a random process to correctly estimate the light interactions in the medium. The random nature of this method makes the utilization of GPU difficult in this context. GPU is best suited to well predefined tasks with the minimum control decisions made at runtime. Random behavior of different GPU threads can lead to decreased utilization of the device. If a GPU is not utilized at its maximum processing power, the overhead of transferring data and control to it can make the GPU implementation slower than a CPU one.

1.5 Limitations

This work is subject to different restrictions which limits the generality of the results obtained:

- we will not consider possible changes of refraction index inside the medium itself;
- our tested objects are rendered in the vacuum, not considering participating media outside the volume;
- inherited from the scattering theory used, we will consider only independent scattering, meaning that we will consider only participating medium which have well-defined separate particles, and single scattering, which means that the concentration of particles of the medium considered is proportional to the total light intensity scattered (further explanation inside the work of van de Hulst [1981]);
- we will not consider wave effects and only consider ray optics;
- we have tested our implementation only on one GPU architecture and we address only the usage of CUDA enabled GPU.

2. Related Work

In this section, we want to briefly overview some of the well established methods for rendering participating media solving the volumetric rendering equation 1.9 and the modifications proposed to port them on the GPU. After that, we will focus on one of those techniques and cover all the work that has been done on optimizing its implementation on GPU.

2.1 Solving the Volumetric Rendering Equation

We have seen in the previous chapter that our problem can be formulates as the solution of an integral, the volumetric rendering equation in 1.9. There are many possibility to estimate the value of this integral but all the algorithms, that we will treat here, are stochastic and *unbiased*, which means that the expected value of the stochastic estimator is equal to the value of the integral.

$$E[\langle I \rangle] = I$$

We are not going to cover in the next paragraphs the photon-mapping approaches, which are particularly good on rendering some difficult light effects, e.g reflected caustics. A GPU implementation of this type of algorithms can be found in the article by Davidovič et al. [2014]. The article describes GPU implementations of Progressive Photon Mapping, Stochastic Progressive Photon Mapping, Progressive Bidirectional Photon Mapping and Vertex Connection and Merging, which combine photon mapping approach to bidirectional path tracing, using multiple importance sampling. Moreover, a comparison between all the algorithm is presented which shows the advantage of using Path Tracing, for scenes which present no complex lighting.

2.1.1 Path Tracing (PT)

Path tracing (developed by Kajiya [1986]) is one of the most used algorithms in physically-based rendering and it is also the one that we decided to use for our GPU implementation. The reason being that GPU benefits from simple with high arithmetic intensity code. Path tracing is the most simple of all the presented algorithms even using participating media, which only requires distance sampling inside the media to extend the classic surface path tracer. Path tracing can also be coupled with next event estimation, a technique that allows direct connection of the paths with the light sources in the scene, by considering direct illumination from light sources and indirect illumination two separate Monte Carlo processes. This can really improve the converging time of the method in some scenes, for example when the light sources are very small. In this work, we decided to not include the next event estimations technique. In the context of GPU, using this technique means following different execution paths for threads that need to compute shadow rays and the others. Moreover, in the context of solid volumetric media this added contribution will be very small in most of the cases, because the contribution must be also attenuated by the transmittance function. Different authors have proposed efficient implementation of path tracing on GPU in the

context of surface rendering, we are going to cover those methods in the next section after discussing other alternatives.

2.1.2 Bidirectional Path Tracing (BPT)

Bidirectional Path Tracing (developed by Veach [1997]) is a more sophisticated approach combining the advantages of path tracing with the dual method, which starts from the light source and light tracing, using multiple importance sampling for the final Monte Carlo estimator. In this technique, each approach is weighted and summed together and it can yield to unbiased estimations of the integral. The method can be easily generalized for participating media, but special care should be taken for the weights, which can be only approximated. Many authors have ported this algorithm to GPU without considering volumetric media. The first implementation was introduced by Novák et al. [2010], then van Antwerpen [2011] in a successive work improved it. The implementation increment the SIMD efficiency compared to a naive implementation. During a first phase of path generation, active threads are compacted together (we will add more details about this method in the chapter 4). During the connection phase, each GPU thread is used to evaluate every bidirectional connection. The method requires a higher memory consumption in comparison to the Path Tracing implementation. Moreover, in the work it is showed that even if the CPU implementation performs better than the classic Path Tracer, this is not the same for GPU, where the path tracer outperforms BPT in most of the scenes. In a successive work Davidovič et al. [2014] proposed Light Vertex Cache BPT; the key idea of this algorithm is that only a certain number of randomly chosen vertex are connected in the connection phase. This allows to store all the vertex in a single global cache of size given, by the average path length and to simplify the algorithm. The implementation offers a considerable speed up compared to the previous one and shows better performance than Path Tracing in scenes with complex lighting.

2.1.3 Metropolis Light Transport (MLT)

Metropolis Light Transport (MLT)Veach [1997] is a method which leverages Metropolis sampling to sample the path space. This allows to generate paths according to the type of function we are integrating (that, in our case, is the radiance coming from the rendered scene):

$$p(\overline{x}) = \frac{f(\overline{x})}{b}$$

where b is equal to the value that we want to estimate $b = \int_{\mathcal{P}} f(\overline{y} d\sigma(\overline{y})$. The value of the latter can be estimated rendering the scene at low number of samples. One of the samples is then used as first state with a probability equal to $\frac{f}{p}$. The next states are then generated using the tentative transition function $t(\overline{x} \rightarrow \overline{y})$, which proposes a modification of the previous path. The modification is accepted or rejected based on the acceptance function. The method is highly affected by the proposal strategy and has also been explored in the participating media context. The implementation is based on identifying a path by the random number used to create it. The integral over the path space is then transformed to an integral

over the space of those numbers (the primary sample space). New paths in this method are created by perturbing the sample. However, both the algorithms have the risks to continue the computation over and over on the same area of the scene. To avoid that to happen, new random samples have to be generated after random intervals of time. Also this algorithm was addressed by Dietger van Antwerpen [2011]. In its implementation, the algorithm runs many MLT samplers in parallel and mutates the random numbers at the base of the method during the path generation phase. Moreover, the implementation builds on top of the BPT implementation previously described and inherits many of the improvements shown in that case. Also this method requires a high memory consumption respect to the Path Tracer implementation and the improvement achieved, using this technique, depends strongly on the type of scene. Furthermore, also in this case it is shown that, even if the CPU implementation performs better than Path Tracing, this is not true for what concerns GPU.

2.2 Efficient Implementations of Path Tracing on GPU

Path tracing is the core at the base of all the algorithms previously described. Consequently, an efficient implementation of this building block can have a positive effect also on the method previously described. We are not going to cover in this section the work done to reach high GPU performances ray casting large scenes. The reason being that our objective is to focus on a single volume, which requires only the intersection with a simple bounding box. The reader interested can look into the work done by Aila and Laine [2009], which describes the use of Spatial Bounding Volume Hierarchies (SBVH) on GPU. For the same reason we decided to not use any ray-shooting solutions, as NVIDIA's OptiX (more informations in the work of Parker et al. [2010]) or other frameworks, but rather to construct a new GPU solution from the ground up. In the introduction, we have talked about the challenges on porting a CPU algorithm to GPU. To build an efficient version of Path Tracing on GPU, it is essential to address full utilization of the available processing power. However, Path Tracing is a stochastic method, where each path traced can follow a different direction. The termination of some paths, before others, can reduce the full utilization of GPU. Nova'ak et al. Novák et al. [2010] have addressed this problem by regenerating terminated paths. This method uses persistent threadsGupta et al. [2012], which access a global pool of paths when the paths that are computed terminate. The method is improved by van Antwerpen [2011], who considered also compacting the active paths before regenerating new paths for the idle threads (see chapter **??** for more explanation on this). A single kernel version of the described algorithm is given by Wald [2011], who proposed a tiled compaction rather than a global device compaction similar to the method we will show. However, the algorithm showed in the article does not give the expected results due to the hardware limitation in the number of registers available for a single kernel. Laine et al. [2013] propose a different type of path tracing which is best suited for scenes with complex materials requiring a lot of computation and thus leading to thread divergence. The methods use multiple lightweight kernels calls which allow to utilize all the resources present

on the GPU for the specific task. The paths are stored in a large pool of paths, which are sorted in chunks based on the specialized kernel they have to be used on. Davidovič et al. [2014] proposed a single kernel version of path tracing with regeneration, where regeneration is done singularly by each thread. However, in this implementation the regeneration is causing code divergence, which means that while the path is regenerating the other threads in the SIMD group (called warp in CUDA) have to wait before continuing. In this work, a different single kernel regeneration is used that prevents this divergence by regenerating the paths only when all the paths inside the warp have finished. Davidovič et al. [2014] provide also a performance comparison between the available algorithms with the result of having best comparable performances in the regeneration path tracer method with a single kernel and the streaming path tracer with multiple kernels. Frolov and Galaktionov [2016] proposed a different implementation of the path regeneration technique with the objective of lower self cost and avoiding moving ray data on different memory locations. They use tile-based work distribution and regenerate entire tiles instead of single threads only if the number of tiles to regenerate is greater than a certain threshold. Moreover, they apply thread compaction using shared memory for the threads associated to a tile..

3. Background

In this chapter, we will look more in depth at the background theory necessary to discuss this work.

3.1 Volumetric Path Tracing

We have seen in the previous chapter the volumetric rendering equation 1.9. Volumetric path tracing is a technique that tries to estimate this integral. For image synthesis we are usually interested on the radiance (in equation 1.1) which hits a virtual camera pointed toward our scene. In order to do that, it uses the Monte Carlo Integration framework, which we are briefly presenting in the next paragraph.

Monte Carlo Methods

Monte Carlo integration is a general tool for estimating integrals by randomly sampling the domain of integration given a probability distribution function $p(x)$. More formally, this proposition, which we are not going to demonstrate, defines the Monte Carlo estimator.

Proposition 1. Given $f : \mathbb{R}^d \to \mathbb{R}$ a function s.t.

$$I = \int_{\mathcal{Q}} |f(x)|\, dx < \infty$$

where $\mathcal{Q} \subset \mathbb{R}^d$ is a limited set. Let n random d-dimensional independent vectors $X_1, ..., X_n$ with the same Probability Distribution Function (here and after PDF), $p = p_{X_1} = ...p_{X_n}$. We define the random variable:

$$< I >_n := \frac{1}{n} \sum_{i=1}^{n} \frac{f(X_i)}{p(X_i)} \chi_{\mathcal{Q}}(X_i) \tag{3.1}$$

which we call estimator of I. then:

$$E\Big(< I >_n \Big) = I \tag{3.2}$$

for any $n > 0$.

The framework works particularly well for complex high-dimensional integrals and its implementation is very simple, which is perfect for GPU algorithms. Increasing the number of samples the estimator converges to the exact solution with a standard deviation which follows $\sigma = \mathcal{O}(n^{-\frac{1}{2}})$.

Ray integral estimator

Considering our integral in equation 1.9 the Monte Carlo primary estimator for the radiance L in the point x in the direction ω is:

$$< L(x, \omega) >_1 = \frac{T(x, X_1)\, L_v(X_1, \omega)}{p(X_1)} \tag{3.3}$$

if we use uniform sampling of the variable X_1 respect to distance along the ray which start at x and ends at x_s, representing in this case the domain $\mathcal{Q}$ of our integral, we have that

$$p(X) = \frac{1}{||x - x_s||}$$

and the estimator will be

$$< L(x, \omega) >_1 = T(x, X_1)\, L_v(X_1, \omega)\, ||x - x_s|| \tag{3.4}$$

However, this simple estimator can lead to high variance when the medium is particularly dense. To improve the estimator, we can use a different PDF which simplifies the estimator. This technique is called importance sampling and in the next paragraph we will see how to sample based on the transmittance such that, rewriting the transmittance in terms of the distance along the ray $s' = ||x - X_1||$ and considering the transmittance independent from x, we have

$$p(s') = \sigma_t(s')\, T(s') = \sigma_t(s')\, e^{-\int_0^{s'} \sigma_t(p) dp} \tag{3.5}$$

where $\sigma_t(s')$ is a normalization term necessary to transform $T(s')$ in a PDF.

Sampling distances according to the Transmittance

Woodcock tracking, which can be found in algorithm 1, is a technique allowing to sample a point along a ray so that the distribution follows the transmittance function. The algorithm needs the maximum extinction coefficient σ_{max} inside the medium. This coefficient is used to sample the distance according to the Beer-Lambert-Law. After that, the sample is rejected or accepted according to the real extinction coefficient $\sigma_t(x)$ of the medium. If it is rejected, another step is taken according to the same law, otherwise the sampled distance is returned.

We are not demonstrating here the validity of this method to sample the distribution $p(S) = \sigma_t(S)T(S)$. For the interested reader we suggest the original article about the method by Woodcock and T.C. [1965].

Directional integral estimator

Now we can sample from the probability distribution function defined in 3.5. The primary estimator for the integral inside the VRE in equation 1.9 in terms of distance along the ray $s' = ||x - X_1||$ becomes

$$\begin{aligned} < L(x, \omega) >_1 &= \frac{T(x, x + s'\omega)\, L_v(x + s'\omega, \omega)}{p(s')} \\ &= \frac{L_v(x + s'\omega, \omega)}{\sigma_t(x + s'\omega)} \end{aligned} \tag{3.6}$$

However, the term L_v, which can be found in equation 1.10, is constituted by another integral on the set of directions $\mathcal{S}^2$, which means that after sampling a distance we have now to sample also a direction ω'. Also in this case we could just sample according to an uniform distribution, but this will lead most of the times to an high variance estimator. A much better idea is to sample according to

Algorithm 1: Woodcock tracking: technique to sample the distance along a ray traversing an heterogeneous medium according to its transmittance function. This function is used for importance sampling the integral in the Volumetric Rendering Equation (equation 1.9).

function woodcockTracking($\mathbf{x}, \omega, \sigma_{max}, s_{max}$)
Input : x : position on the ray,
ω : direction of the ray,
σ_{max} : maximum extinction coefficient,
s_{max} : maximum distance of the ray,
rand() $\in [0, 1]$ random sample generator
Output: s' : sampled distance along the ray
$s' = 0$;
while $s' \leq s_{max}$ **do**
  $s' += -\log_e(1 - rand())/\sigma_{max}$;
  if *rand()* $< \sigma_t(\mathbf{x} - s'\omega)/\sigma_{max}$ **then**
    break;
  end
end
return s'

some factor of the integrand, which is composed of the phase function $f_p(\omega, x, \omega')$ and the radiance $L(x, \omega')$. The second term is more difficult to use as it is exactly what we are searching for, so in this work we will sample according to the phase function.

Henyey-Greenstein Function

In the first chapter, we have introduced the phase function as a characteristic of the medium we are going to render. In our work we are using as approximation of the Mie Phase functions the Henyey Greenstein function (abbreviated HG function hereinafter), the reader interested on other phase functions can read the comprehensive guide about multiple scattering by d'Eon [2016]. The Mie phase functions describe the scattering behavior of light interacting with perfectly spherical dielectric particles with different diameter size. The phase functions can be characterized by the anisotropy g, which is the first angular moment of the function:

$$g = \int_0^{2\pi} \int_0^{\pi} f_p(\theta', \phi') cos(\theta') sin(\theta') d\theta' d\phi' \tag{3.7}$$

where $g \in [-1, 1]$ and, when it is positive, the light scatters predominantly into forward directions, while if it negative it scatters predominantly into backward directions. The HG function is parametrized by this value and can be written as:

$$f_{HG}(\theta, \phi, g) = \frac{1}{4\pi} \frac{1 - g^2}{(1 + g^2 - 2g\cos\theta)^{3/2}} \tag{3.8}$$

given this function a spherical direction (θ, ϕ) can be sampled inverting analytically the function (more information in the work of Pharr and Humphreys [2004])

as

$$\cos\theta = \frac{1}{|2g|}\left(1 + g^2 - \left(\frac{1-g^2}{1-g+2g\xi_1}\right)^2\right) \tag{3.9}$$

$$\phi = 2\pi\xi_2 \tag{3.10}$$

where $\xi_{1,2} \in [0,1]$ are uniformly distributed random samples.

Volumetric integral estimator

Now we can write the estimator for the term L_v, sampling the direction ω', with the technique described in the previous paragraph. We have that the primary estimator for the L_v term in the volumetric rendering equation 1.9 is

$$\begin{aligned} < L_v(x',\omega) >_1 &= L_{e,\mathcal{V}}(x',\omega') + \sigma_s(x')\frac{f_p(\omega,x,\omega')L(x',\omega')}{p(\omega')} \\ &= L_{e,\mathcal{V}}(x',\omega') + \sigma_s(x')L(x',\omega') \end{aligned} \tag{3.11}$$

and combining all the results obtained, we have the final estimator

$$\begin{aligned} < L(x,\omega) >_1 &= \frac{L_{e,\mathcal{V}}(x',\omega') + \sigma_s(x')L(x',\omega')}{\sigma_t(x')} \\ &= \frac{L_{e,\mathcal{V}}(x',\omega')}{\sigma_t(x')} + \frac{\sigma_s(x')L(x',\omega')}{\sigma_t(x')} \end{aligned} \tag{3.12}$$

where $x' = x + s'\omega$. In the work we are not considering medium which can emit light, we will have $L_{e,\mathcal{V}}(x',\omega') = 0$ and the equation simplify to

$$\begin{aligned} < L(x,\omega) >_1 &= \frac{\sigma_s(x')L(x',\omega')}{\sigma_t(x')} \\ &= \alpha(x')L(x',\omega') \end{aligned} \tag{3.13}$$

where $\alpha(x')$ is the albedo defined in 1.5. This estimator is very simple and perfect for GPU implementation, where the albedo can be easily mapped to an interpolated 3D texture. Using only one sample is usually not enough and, in its general description, the final estimator using n samples is

$$< L(x,\omega) >_n = \frac{1}{n}\sum_{i=1}^{n}\alpha(x')L(x',\omega') \tag{3.14}$$

Volumetric Path Tracer Algorithm

The volumetric path tracer uses exactly this estimator on every pixel of the image sensor. In algorithm 3 a simplified version is presented only considering background light. The algorithm uses also the sampling from a Bsdf (Bidirectional scattering distribution function), that we have not described in the previous paragraphs (the reader can find its description in the work of Pharr and Humphreys [2004]). In this work a physically-based Bsdf is used, called the GGX. Whenever the point is a surface, the GGX is sampled instead of the phase function. We sample the GGX using the distribution of the visible normal. In particular, we are using the sampling strategy described by Heitz [2017] which does not require any look up table to be computed.

Algorithm 2: Recursive estimator of L in the case we are only using background lighting

```
function radiance (x, ω)
Input  : x : position of the camera ray,
         ω : direction of the camera ray,
         intersect(x, ω) → hit, s_max : intersection function for the scene,
         returns weather the scene was hit and the distance s_max
         σ_max: maximum extinction coefficient in the scene
Output: Radiance in the point and direction (x, ω)
throughput = Color(1) ;
while (true) do
    hit, s_max = intersect(x, ω);
    if (hit) then
        s = woodcockTracking(x, ω, σ_max, s_max);
        if (s ≤ s_max) and insideVolume(x) then
            x = x + ω s ;
            ω = samplePhase(ω) ;
            throughput = throughput · albedo(x) ;
        else
            x = x + ω s_max ;
            ω, weight = sampleBsdf(ω) ;
            throughput = throughput · weight ;
        end
    else
        return L_e(x, ω)· throughput ;
    end
end
```

Algorithm 3: Volumetric Path tracing algorithm

```
function volPT
Input  : n iterations for the Monte Carlo estimator
Output: Rendered Image
Image = Image(0) ;
for pixel p in Image do
    for n iterations do
        x, ω = cameraRay(p);
        p += radiance(x, ω) ;
    end
    p = p/n ;
end
```

Russian Roulette

Another technique usually implemented with path tracing is the Russian Roulette. This technique permits to stochastically interrupt a path before reaching a light source. The probability of interrupting can be associated with the amount of light transported by the path. In this way is possible to interrupt with more probability paths which gives smaller contribution to the overall result. This technique is also unbiased if we modify our estimator by dividing for the survival probability of the path. More information about this technique can be found in the work by Pharr and Humphreys [2004]. In the software provided it is possible use the Russian Roulette inside any of the volumetric path tracer showed. However, we didn't focus the research on this particular aspect.

3.2 GPU Architecture

In this section, we want to review some basics of the GPU architecture, that we are going to consider as a data parallel computational platform, and the characteristics of the CUDA platform.

Design Philosophy and Heterogeneous Programming

One of the most used classifications for computer architecture is the Flynn's Taxonomy. In this classification, the architectures are divided into 4 groups:

- Single Instruction Single Data (SISD) : this is the traditional serial architecture.
- Single Instruction Multiple Data (SIMD) : multiple cores execute the same instruction at the same time.
- Multiple Instruction Single Data (MISD) : multiple cores use separate instructions on the same data.
- Multiple Instruction Multiple Data (MIMD) : multiple cores operate on multiple data streams.

Another type of classification can be done based on the memory:

- Multi-node with distributed memory: processors are connected by a network, each having its own local memory.
- Multiprocessor with shared memory: processors are either connected to the same memory or through a low-latency link, e.g. PCI-Express or PCIe.

Finally, we will use other two properties to characterize an architecture:

- throughput (ops/cycle or Tflop/s): rate of operations complete per cycle, usually calculated considering floating point operations and seconds (it can be sometimes confused with latency, cycle or s, which is the waiting time).
- bandwidth (GB/s): rate at which data can be transferred.

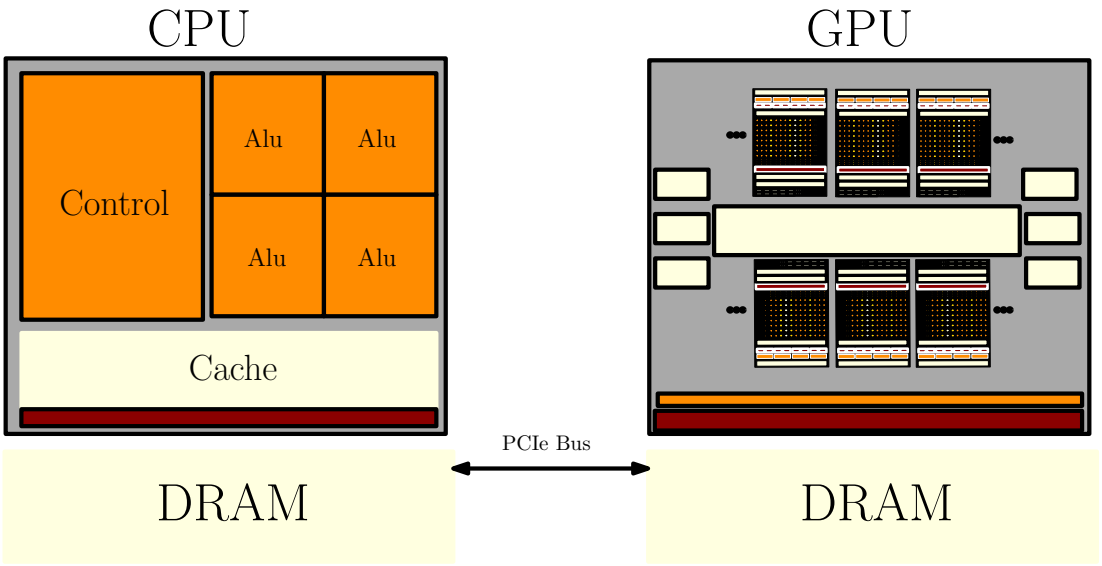


Figure 3.1: CPU and GPU, heterogeneous programming

In this context, GPU is a multiprocessor with shared memory architecture that utilizes SIMD groups, called warps. However, some of the features of the other architectures are also present on NVIDIA GPUs. NVIDIA call their architecture as Single Instruction, Multiple Thread (SIMT). At the base of the different performance between CPU and GPU is the different design philosophy of the architecture. Contrary to CPU, GPU are designed for high throughput and low latency tasks. GPU is many-core architecture which contain a high number of cores and high memory bandwidth. Each core is also very different from a CPU core. CPU cores are heavy-weight and they are designed to handle complex control logic, instead the GPU cores are very light-weight and optimized for data-parallel tasks with simple logic. For this reason, both architectures should be used together in a heterogeneous system for different scopes. Currently GPU cannot be used standalone and execution should be initialized by CPU. Therefore, the CPU is usually called the *host* and the GPU the *device*. In General Purpose GPU programming (GPGPU) execution of code on the device can be divided in different parts, which we will call kernels. The host code can call those kernels during its execution and execute the code of the kernel on the device side. The calls of those device kernels can also be asynchronous, allowing to the host to perform more work while the device is running the kernel.

CUDA - Compute Unified Device Architecture

CUDA is a GPGPU platform introduced by NVIDIA for their GPU on 2006. It is designed for scalable parallelism allowing to any CUDA application to leverage whichever NVIDIA GPU they are running on. This is enabled by the usage of three main abstractions:

- hierarchy of thread groups
- memory hierarchy
- barrier synchronization

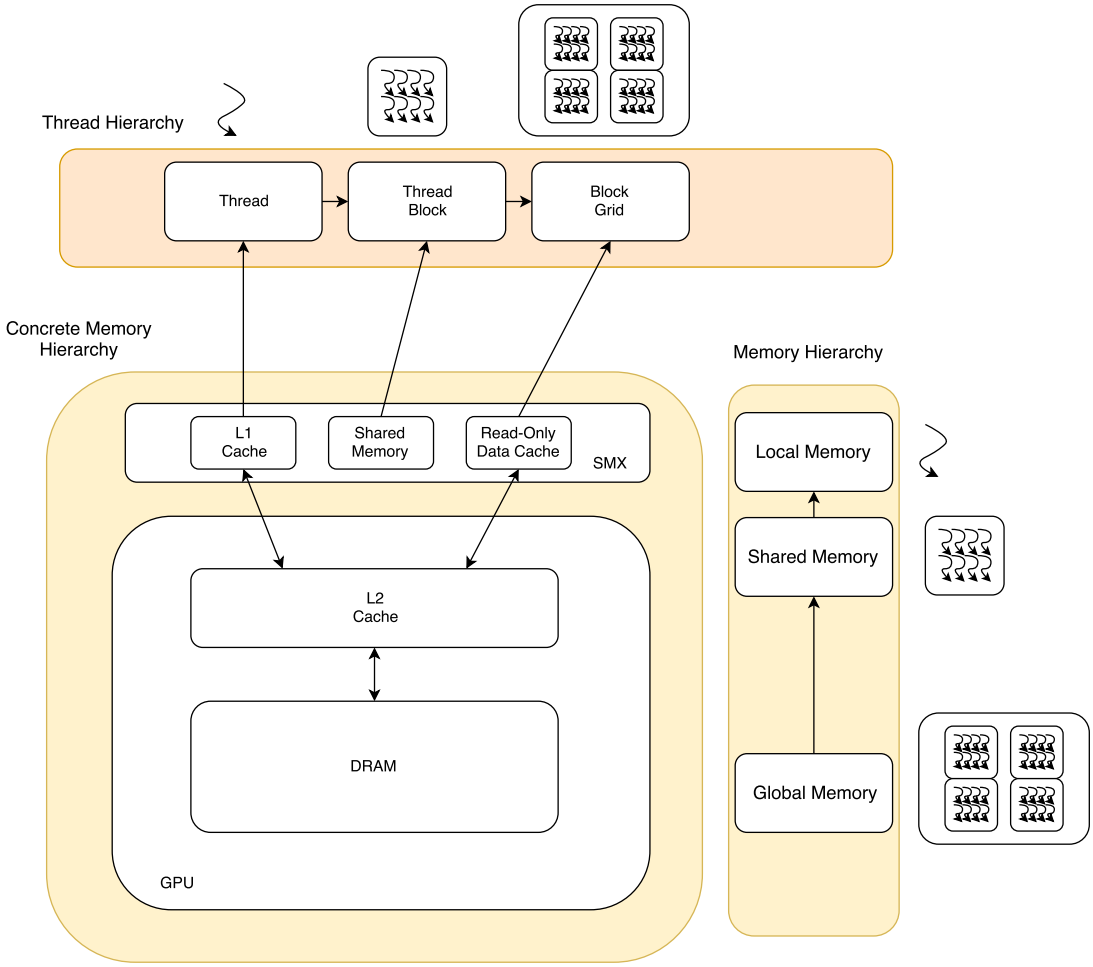


Figure 3.2: Memory and Thread Hierarchy. All threads can also directly access the read-only memory using the constant and texture memory. The concrete Memory hierarchy showed is relative to the Kepler Architecture (compute capability 3.0)

These abstractions are then mapped to concrete implementation on the specific hardware by CUDA. Figure 3.2 shows the thread and memory hierarchy and how the memory hierarchy is mapped to the Kepler GPU architecture. The programmer, to leverage all the parallelism of a GPU, must partition the multi-threaded program into blocks of threads which execute independently and in any order.

In the concrete, implementation of the CUDA model the GPU uses an array of Streaming Multiprocessors (SM or SMX). The multiprocessors schedule threads using SIMD groups of 32 threads called warps. However, threads in a warp can have their own instructions and they are able to branch and execute independently (also if this will not exploit the SIMD capability of the group potentially serializing the program). After the Kernel launch, a multiprocessor can get a thread block to execute. At this point, the warp scheduler partitions it into warps with consecutive thread ID. Program counters and registers of each warps are maintained on the multiprocessor for the entire life of the warp. For this reason, the multiprocessor can switch between different warps with no cost. The warp scheduler can also choose a warp, which active threads are ready to execute

their next instruction, while another warp is waiting (this is called in CUDA latency hiding). The number of block and warps and consecutively threads, that can be processed inside a multiprocessor, depends on the number of registers and shared memory used in the kernel, compared to the resources available on the multiprocessor. We will see what this means for the performance in 4. We want also to point out, in this section, that a CUDA application does not have to be bounded to the data which resides only on the GPU. The unified memory system gives the possibility to have a single address space between CPU and GPU memory (on newer SM architectures this means also on-demand page migration). Moreover, CUDA's zero-copy memory permits pinned memory location on the CPU to be accessed on the GPU. This last feature is used inside our application to give the possibility to load volume 3D textures, which are larger than the Device Global memory. Another feature of CUDA is the possibility to perform atomic operations. Those operations allow concurrent threads to perform read and write operations on data shared in global memory. However, this type of operation should be used carefully because it can potentially serialize the program in case all the threads wants to access the same data. For what it concerns shared memory, the system works differently. To achieve high bandwidth, shared memory is divided into several memory modules of the same size. Those modules are able to serve different read and write requests simultaneously, as long as the address requested resides in different banks. When two or more requested addresses reside on the same bank it is called a bank conflict and the accesses are serialized. However, there is an exception when multiple threads access the same 32-bit world, in that case the word is broadcast to all the threads requesting it (if multiple threads request to write, only one will write on the address, but which one is undefined). The banks are organized such that consecutive 32-bit words map to successive banks allowing to linear access to not cause bank conflict.

Kepler Architecture

In this section we will detail the Kepler architecture (which is categorized by NVIDIA with compute capability 3). As shown in figure 3.3 and 3.4 in this architecture a multiprocessor consist of 192 CUDA cores for arithmetic operations, 64 double-precision units, 32 special function units, 32 load/store units allowing source and destination addresses to be calculated for sixteen threads per clock and 4 warp schedulers. Moreover, two types of cached memory are present: a L2 cache shared by all the processors, used to cache global and local memory accesses, and a L1 cache, used to cache access to local memory (including register spills). The same memory is also used for the shared memory. Each multiprocessor has also a read-only data cache which is used for constant or texture cache. The memory transactions are of the size of 128-byte for memory stored both in L1 and L2 cache and 32-byte if the memory is only cached in L2.

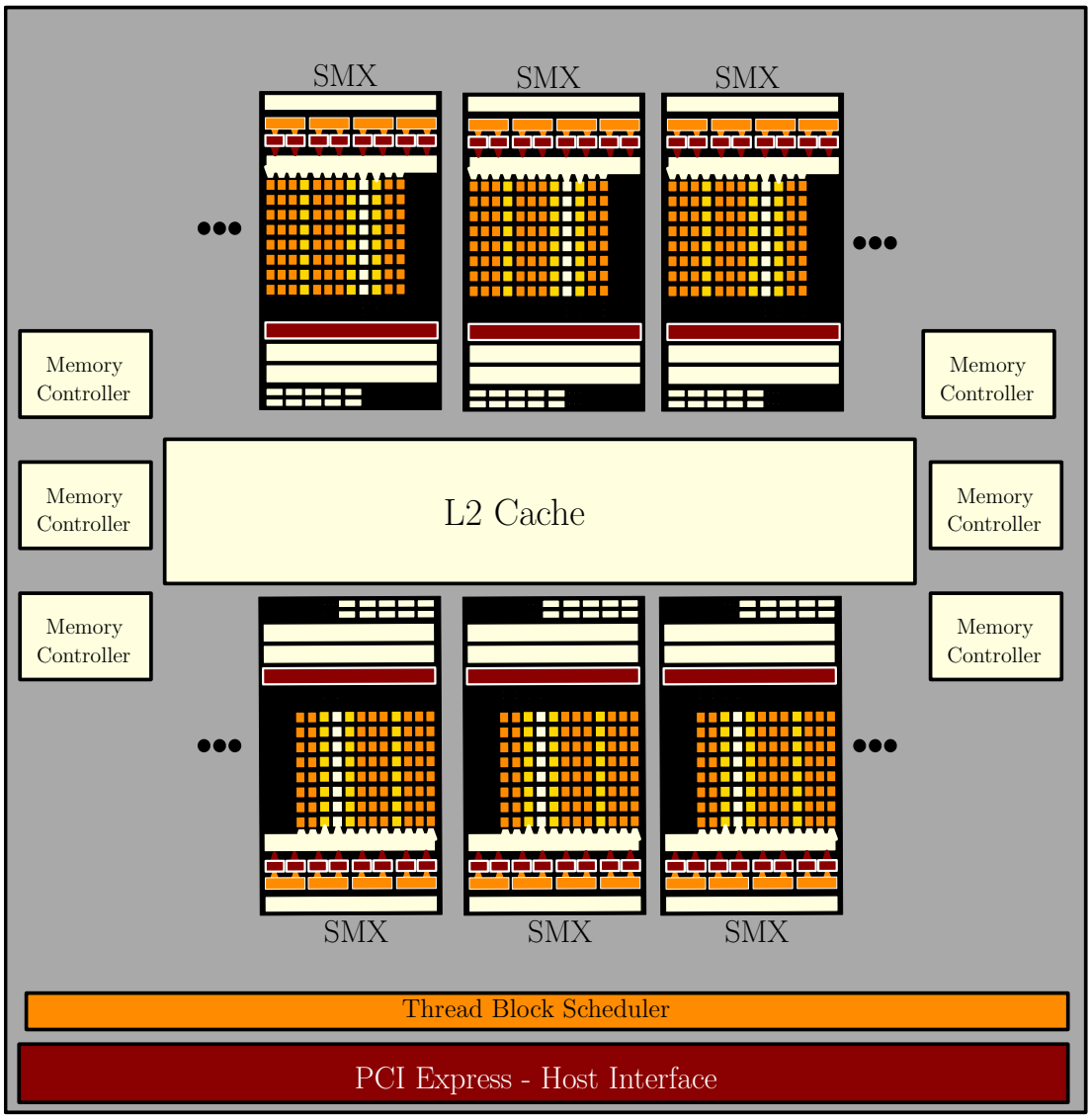


Figure 3.3: Kepler Architecture (compute capability 3)

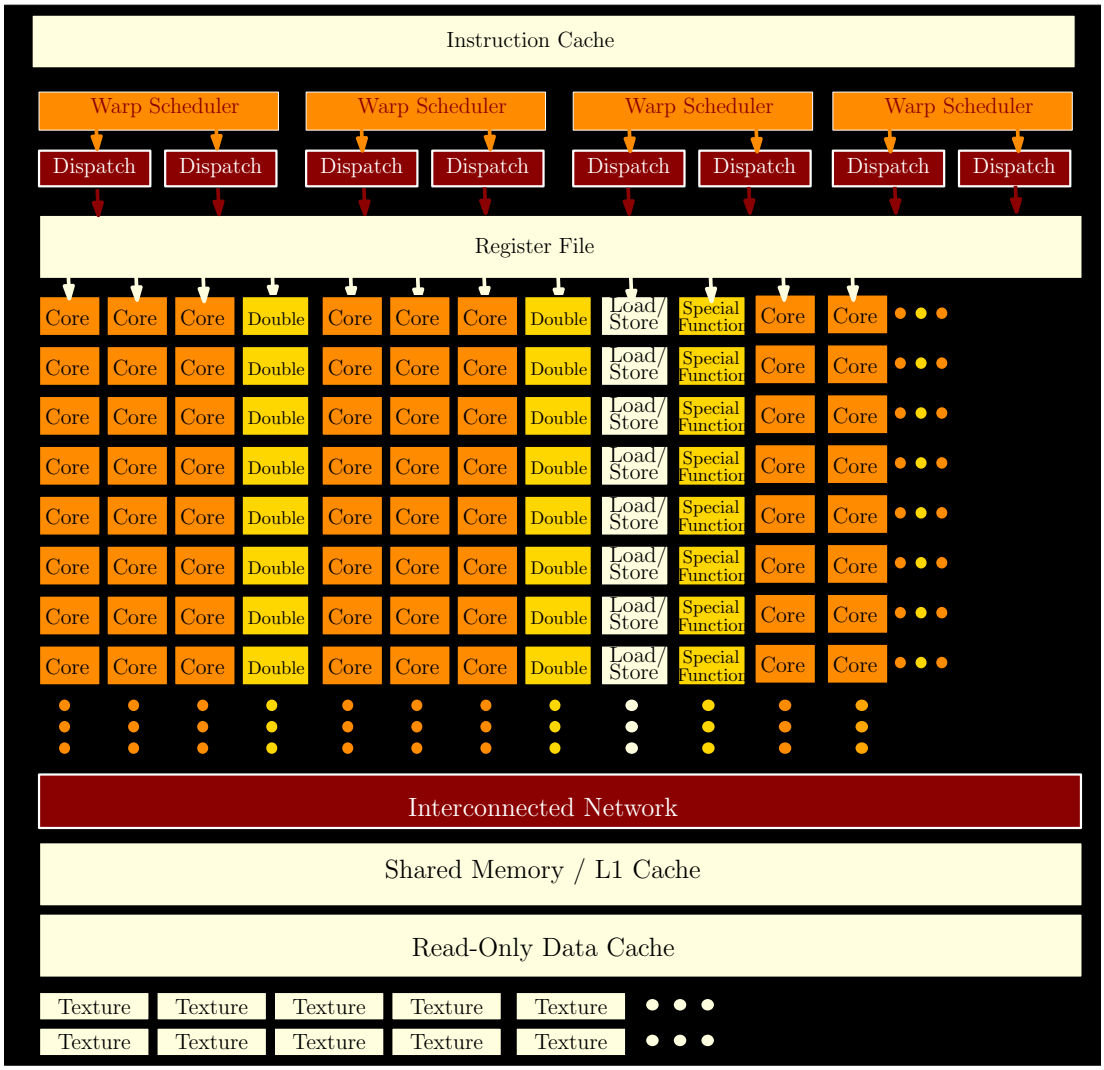


Figure 3.4: Streaming Multiprocessor (Kepler architecture compute capability 3)

4. Optimization Methodology

In this section we are going to present an overview of the method used to optimize the volumetric path tracer. The chapter follows the design decisions that someone should evaluate when approaching this type of problem. Different design paths have different advantages and disadvantages that we are going to cover. In figure 4.1 it is possible to see the exact design decisions tree. At the beginning, we are going to examine how to use GPU and CPU together and this we lead us to a decision on a single versus multi kernel approach. After that, we are going to evaluate how to maximize the utilization of GPU regenerating threads, which become idles during the computation. Finally, we are going to see how to maximize the memory throughput and the data locality compacting together the threads that are still active. In figure 4.2 there are some scenes that we are going to use for the tests: two of them from the application field of 3D printing and the other two from visualization. The volumetric path tracer tests showed in this section are all running without Russian Roulette 3.1.

Testing Hardware

We will use for testing a MacBook Pro with CPU Intel Core i7 quad-core and GPU Nvidia GeForce 650M with the following characteristics:

- Kepler architecture, compute capability 3.0
- Total amount of global memory: 512 MB
- Total number of registers available per block: 65536 (equal to registers per SM)
- Maximum number of threads per block: 1024
- Total amount of shared memory per block: 49152 bytes

more details about the hardware are provided in the Appendix.

4.1 Host Control or Device Control

4.1.1 Single Kernel versus Multi Kernel

The topic of using a single "megakernel" versus multiple lighter kernels has already been widely covered in the literature. A naive single kernel implementation holds all the algorithm and requires all the resources associated with all the different parts of it. Nvidia OptiX (Parker et al. [2010]) utilizes a similar but more sophisticated approach. In this framework, a Just-In-Time compiler combines all the different stages of the path tracer, in the form of PTX (assembly language for NVIDIA GPU), into a single kernel. The kernel is formed as a state machine where, to minimize execution divergence, a scheduler selects a single state for an entire SIMT unit. If a thread is not requiring that state, it remains idle during that iteration. Laine et al. [2013] compared the traditional single kernel approach

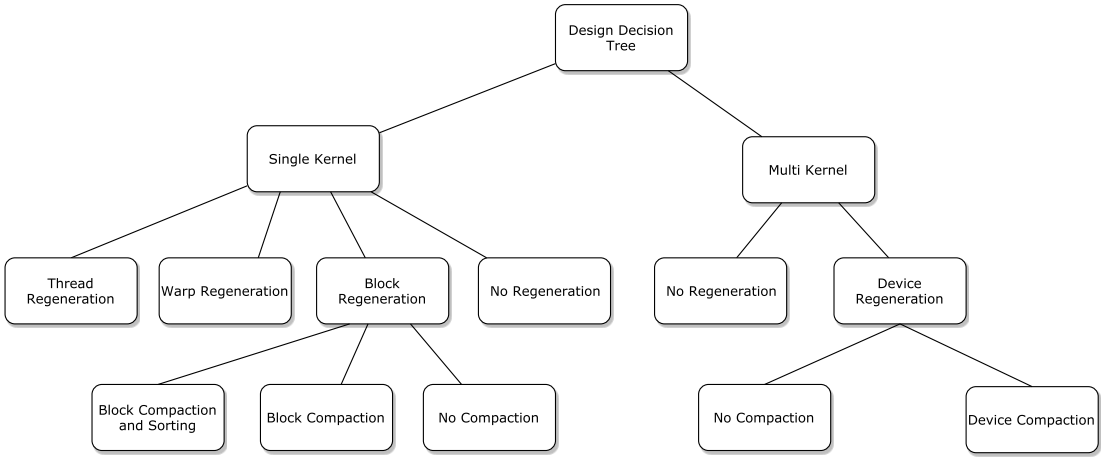


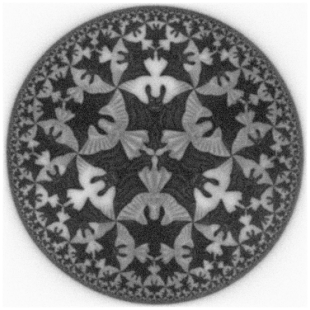
Figure 4.1: Design decision tree. This tree represent the design decisions made to optimize a GPU path tracer. In the first level, the first decision is regarding the use of a single or a multi kernel and more in general, giving more control to the device or to the host. The second level contains all the possibility presented to optimize the utilization of the GPU. Finally, the last level contains the possibility to decrease data access latencies in a volumetric path tracer.

to a wavefront formulation, which uses different kernels for different control flows. This method allows to completely avoid code divergence, i.e. sorting the paths based on their next interaction, dividing them into different bins and launching a kernel only for the paths in the same bin. Moreover, the kernels launched are perfectly specialized for the task and they can utilize the GPU resources completely. However, this work is not containing this possibility. There are different reasons for that:

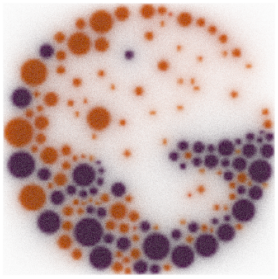
- Dynamic parallelism and global device synchronization: supported only by the newer CUDA GPU (compute capability 3.5 and higher), dynamic parallelism gives the ability to a single CUDA thread of launching a new kernels with its own configuration. This technique reduces the need to transfer execution control and data between host and device (Nvidia [2017]). Moreover, the global device synchronization introduced with CUDA 9.0 (using special cooperative groups) removes the need of launching a new kernel only for synchronizing all the threads running on the device.

- The type of scenes that we are analyzing in this work contains just one object: the volume that we want to render. For this reason there are not many different type of interactions. The only two cases are if the analyzed point is inside the volume or on the boundary (exactly how it is explained in the section 1.3). However, the wavefront formulation is best suited for scenes with many different and complex materials.

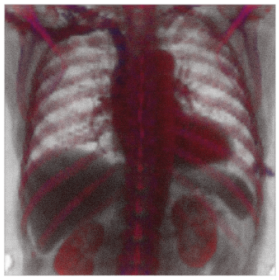
Inspired by results showed by others like Davidovič et al. [2014] and van Antwerpen [2011] the only multi kernel approaches that we are going to analyze are composed by a maximum of 2 kernels. The table 4.1 shows the difference on using a naive approach with a single kernel call or multiple kernel calls. The naiveSK method (naive with a single kernel) consists of a simple volumetric path tracer implementation where each thread computes only one path and all the



(a) ad scene: rendering of a slab with grid resolution (320,320,100). The texture "printed" on the surface has a depth equal to 10 (10% of the total resolution depth) and it is assigned to the albedo volume, the rest of the albedo volume is white. The density volume is always constant and equal to 100.



(b) cgg-logo scene: rendering of a solid unit cube with a grid resolution of (100,100,100). The texture "printed" on the surface has a depth equal to 10 (10% of the total resolution depth) and it is assigned to the albedo volume, the rest of the albedo volume is white. The density volume is always constant and equal to 100.



(c) artifix scene : rendering of the publicly available artifix Data set (Osirix) which have grid resolution of (128, 87, 128). The data has been used to create the density volume, the albedo has been created from the density using a transfer function which maps the density from gray to red to blue based on the density value. The maximum density value is 100.

(d) manix scene: rendering of the publicly available Manix Data set (Osirix) which have grid resolution of (128, 115, 128). The data has been used to create the density volume, the albedo has been created from the density using a transfer function which maps the density from gray to red to blue based on the density value. The maximum density value is 100.

Figure 4.2: Test scenes. top: scenes for 3D printing application, bottom: scenes for data visualization.

Table 4.1: Single Kernel versus Multi Kernel (naive). The speed of the methods is analyzed in terms of millions of traced rays per second (rays/sec). The test has been done using the scene in figure 4.2b rendering a 400x400 image for 100 iterations (number of samples for the Monte Carlo estimation)

Method	rays/sec
naiveSK	3.91
naiveMK	1.19

data is stored on local memory (the algorithm can be found in the work of Davidovič et al. [2014]). The naiveMK method (naive with multiple kernels) consists of a simple volumetric path tracer divided in two kernels. The first kernel is generating the paths and computing the first intersection with the object, while the second kernel extends the active paths computing a new intersection with the scene. It is clear that the single kernel approach wins over the multi kernel one in terms of speed. However, there are also advantages on using multiple kernel calls. From a user perspective, using smaller kernels allows the CPU to take the control of the application and, in case only one GPU is available in the system, it avoids to freeze the user interface for too much time.

4.1.2 Image Tiling

Another possibility for interleaving the control between host and device is using image tiles. That is, the image is divided into equal tiles and the device can work on each of them separately at the same time. This technique allows to lower the memory requirements for running the software, because we need only to store the size of one tile. This also means that the rendering is divided into multiple kernel calls. Each of this kernels have to render a single tile and for this reason is faster. In this perspective, it is important to not use tiles that are too small, because this will decrease the utilization of the hardware and therefore the performance (we will talk more about utilization in the next section). The consequence is that, in order to obtain the maximum performance, the size of the tile should depend on the type of hardware. The tile size is also affecting the specific algorithm in use. Most of the algorithms that we are showing in this work have different memory requirement and different behaviors depending on the tile size. However, from our tests it emerges that the behavior of each algorithm using different tile sizes is comparable with the use of different number of samples. Potentially, a tiled image allows also to decrease the time to transfer the image from the device to the host. In the practical case, however, the size of the image is usually small and the time to transfer it is not comparable with the time for computation. For this reason, also using a second buffer for the image tile (this technique is usually called Double Buffering) and the CUDA asynchronous memory copy to overlap computation and image transfers, the performance improvement is negligible. Table 4.2 shows the results of using different image tiles. From this table it is possible to see that launching multiple kernels on different image tiles is not affecting the performance as far as the number of tiles is not becoming too high. Indeed, the computation remains stable until the number of pixel processed

by the algorithm becomes too small, decreasing the utilization of the device for the kernel call. This result shows also that the real performance bottleneck on dividing the path tracing task is not represented by the kernel launch overhead but from the repetitive loading and storing of the same data.

Table 4.2: Different image tiling settings used for rendering a 1920x1920 image with the scene in figure 4.2b. In the table the number of paths processed per tile is compared to the time to render the all image. Note that, opposed to the naiveSK, the streamingSK kernel launching configuration does not depend on the number of paths processed but rather on the threads available on the GPU. For this reason when the number of paths processed becomes low the naive method improve its performance while the streamingSK decrease in performance.

	tiling setting					
	(1,1)	(2,2)	(4,4)	(8,8)	(32,32)	(64, 64)
n. paths per tile	18432000	4608000	1152000	288000	18000	4500
StreamingSK	61.27	67.98	72.72	76.13	90.66	224.04
regenerationSK	7.99	10.02	9.44	9.48	34.11	98.40
naiveSK	219.47	223.65	221.93	224.59	224.85	123.32

4.1.3 Summary

In this chapter we have seen the first design decision that should be made when approaching this problem: giving the control of the program to the host or to the device. In the first case, the advantage is to have a more responsive UI and, if the kernels are well divided into different tasks, it allows to use efficiently the device resources for the task. In our case, most of the computation concerns the same simple task: the interaction with the medium. For this reason, dividing the computation will not improve the performance. We have also seen another way to divide the computation by rendering separate image tiles. This allows to use the same algorithms on a different hardware, whose memory requirements depend on the image tile size. Moreover, overlapping communication and computation decreases the latency on getting the resulting image. The tile size is affecting also the utilization of the GPU which is upper bounded by the number of paths to compute. Indeed, the number of paths to compute an image tile is equal to

$$\text{paths} = \text{tile.width} \cdot \text{tile.height} \cdot \text{iterations}.$$

While the number of iterations is given by the user, the tile width and height can be specified according to the hardware used, which allows to improve the utilization for the specific hardware. In the next chapter we will see which other factors can decrease the utilization of the GPU and how to prevent it.

4.2 Maximizing Utilization and Hiding Multiprocessor Latency

The objective of maximizing the utilization of the device is achieved when the device is always active until the end of the computation. In a path tracer, the

problem is given by the different lengths of the paths computed. When a path in a thread has terminated, that thread will not be utilized until the termination of all the others. To overcome this inefficiency Novák et al. [2010] have proposed to use path regeneration. This technique consists of using a persistent thread system (more information about persistent thread programming can be found in the work of Gupta et al. [2012]) where new paths, taken from a global pool of paths, are assigned to idle threads (the path is therefore regenerated).

Another problem that can affect the utilization on the system is the number of registers and the shared memory necessary for the kernel. Most of the time those resources are stored in a fast memory (L1 cache in devices with compute capability 3 or higher) that is limited. When the kernel reaches this limit the maximum number of resident blocks in a multiprocessor is decreased. If the kernel requires more registers than available on the multiprocessor, then the compiler will attempt to minimize register usage, or it can be forced to do it, while keeping them stored in the local memory (register spilling). On the other hand, if the kernel requires too much shared memory the only solution is to decrease the number of resident block in a multiprocessor. When the number of threads resident on the same multiprocessor is high then the latency of a inactive warps, which cannot perform their next instruction, can be hidden efficiently. The full utilization in this case is achieved when the warp scheduler can always issue some instruction for some warp during the latency period.

4.2.1 Persistent Thread

The persistent thread style of programming helps a developer to separate the task to compute from the hardware that is running it. The threads are active for the entire duration of a kernel and every thread gets new work from a work queue when it finishes its current task. In our case this means regenerating the terminated paths and tracing new ones. A new path can be created by using an identification number like in algorithm 4. For this reason, the work queue is represented by one global value that is atomically incremented during regeneration. In this paragraph, we analyze the different approach that we may take for regenerations:

- Thread regeneration: a single kernel is launched and new paths are assigned to every thread. When the thread becomes inactive the path is regenerated.
- Warp regeneration: a single kernel is launched and new paths are assigned to every warp. When all the threads in a warp become inactive, all the warp is regenerated incrementing the atomic counter only once per warp by the size of the warp.
- Block regeneration: a single kernel is launched and new paths are assigned to every block. At every path extension, if the path queue is not empty, all the threads in a block are synchronized and all the inactive ones are regenerated.
- Device regeneration: multiple kernel are launched, one kernel is performing regeneration and another one path extension. If the path queue is not empty, after each extension all the inactive threads are regenerated.

Those regeneration techniques require different types of synchronization points: the first one is the classic implementation where if a thread becomes inactive it is immediately regenerated, i.e if only one thread is regenerating the rest of the threads in the warp, we have to wait for it before continuing. The second one is using the warp vote functions (which can be found in the documentation by Nvidia [2017]) to decide weather to regenerate or not the warp. In CUDA 9 and on devices with compute capability higher than 3.0 this can be efficiently done using the warp shuffle functions, which allows to exchange values between threads in the same warp, to get the right path for each thread. The third one requires synchronization of all the threads in a block and it regenerates all the block all together. This latter is similar to what is done by Frolov and Galaktionov [2016]. Finally, the last one is regenerating all the processed paths and requires to synchronize all the devices. We did not include this last one in our study because we have already tested that the multi-kernel approach is not favoring our settings. In the implementation presented the synchronization points are performed after each path extension (unless the path queue is empty). This can be not optimal, as the synchronization point for regeneration should be placed exactly when the performance risks to decrease for not having enough active threads (in the next paragraph we will see what exactly this means). However, this is not only hardware dependent but also scene dependent. For this reason, we did not include this variation in this work. The table in 4.6 shows the results of the different methods on our scenes. We can see that the regenerationSK (regeneration with a single kernel) on a single thread works best on the scenes with constant density (cgg-logo and ad) for which the warps are almost always synchronized. In the other two scenes the warp regeneration is performing better. When the paths start to differentiate, this method allows to maintain the coherence inside a warp by restarting all the warp at once.

Algorithm 4: pseudo-code of a function which regenerate a thread. The algorithm uses a global counter for the number of rays already traced. This counter is used to define the identification number of a ray which is also used to create the ray starting from the camera

```
function regenerate
Input  : thread: thread to regenerate,
         paths_head: global counter for the paths
Output: thread: regenerated thread
path_id = paths_head ++ ;
if path_id > total_paths then
    thread.active = false ;
end
pixel = getPixel(path_id);
thread.path.ray = cameraRay(pixel);
thread.path.throughput = Color(1);
thread.active = true;
return thread;
```

Table 4.3: comparison of different types of regeneration on the scenes presented in figure 4.2. Three regeneration approach are taken in consideration: regenerationSK (thread) doesn't require any type of synchronization and regenerate a thread immediately after it becomes idle. regenerationSK (warp) require all the warp to be idle before regenerating. regenerationSK (block) is synchronizing all the block and regenerating only when all the block is idle. Finally, the naiveSK approach is a simple volumetric path tracer which is not performing any regeneration.

method	rays/sec			
	cgg-logo	ad	artifix	manix
naiveSK	3.88	2.13	8.34	8.45
regenerationSK (thread)	81.62	42.37	11.80	11.42
regenerationSK (warp)	17.13	17.05	12.11	14.82
regenerationSK (block)	3.52	3.21	7.35	7.41

4.2.2 Occupancy

The occupancy of a kernel consists of the number of active warps (SIMD group) on a multiprocessor when launching the kernel. If the occupancy is maximal, then all the warps that can simultaneously reside on a multiprocessor are active and the warp scheduler can hide the latency of the warps which cannot execute their next instruction. However, if the kernel requires too many registers or too much shared memory, it have to limit the number of threads active on a single multiprocessor. If the kernel requires too much shared memory, there is nothing that the compiler could do to increase the occupancy. However, if the problem is the number of registers used, then one possibility is to limit this number using register spilling. In our case, the kernel of our volumetric path tracer requires approximately 64 registers for the compiler, which limits the occupancy to 50%. To achieve the 100% of occupancy we could force the compiler to reduce the number of registers used by our path tracer.

Bounding Register Usage

Bounding the number of registers used by a kernel is not always a good idea. The compiler is usually limiting the number of register used by itself and tries to make the best decision between increasing the occupancy and efficiently storing memory on registers. Sometimes, however, it is possible that this decision is not the one that gives the best performance. For this reason, another possibility is to force a kernel to use a minimum number of blocks per multiprocessor by limiting even more the number of registers used. In the table 4.4 we can see the effect of achieving maximum occupancy using this technique. All the results are worse respect to the version with half the occupancy in table 4.6. By analyzing the process, it is possible to see that also if the number of warps in the multiprocessor is maximum, the latency is higher than before. The time required to get the spilled registers is therefore higher than the latency hiding capability of the device.

Table 4.4: maximize occupancy decreasing registers usage. In those results the number of register used by the kernels is bounded so that the device can achieve maximum occupancy. However, comparing those results with the ones in table 4.6 is possible to see that this method is actually performing much worse than the previous one with only 50% of occupancy

method	rays/sec			
	cgg-logo	ad	artifix	manix
regenerationSK (thread)	39.04	32.81	12.66	11.37
regenerationSK (warp)	4.37	7.02	11.92	10.95

Maximize Register Usage

Increasing the occupancy is not the only way to hide threads latency; another possibility is to leverage the instruction-level parallelism. Indeed, a warp that has subsequent independent instructions can hide the latency of an instruction performing the next one. This technique allows to lower the occupancy needed to obtain the device peak of performance, which can be calculated with Little's law:

$$\text{parallelism} = \text{latency} \cdot \text{throughput}$$

We have used this technique in our volumetric path tracer by assigning more than one path to each thread. This allows to decrease the dependency among subsequent instructions and, therefore, to hide the latency inside a single thread using the instruction-level parallelism. Recent tests Volkov [2015] have demonstrated how the use of more registers per thread, using less threads per multiprocessor, can lead to better performance with smaller occupancy. We have also tested our volumetric path tracing with a smaller occupancy, but without any visible improvements. The conclusion that we may take is that in this case the compiler is doing an optimal choice about the number of registers and threads to use.

4.2.3 Summary

We have seen in this section different application of the persistent thread techniques, which differ on synchronization points. Moreover, we have seen some techniques to achieve better performances at lower occupancy utilizing instruction-level parallelism and more registers per thread. However, the difference in latency between arithmetic operations and memory operations is high and the best way to improve the performance is decreasing this gap. For this reasons, in the next section, we will discuss about localizing memory access inside the kernels to maximize the cache usage and decrease the memory access latency.

4.3 Data Locality and Code Divergence

We have discussed until now all the performance limiting factors regarding the full utilization of the GPU and the logic communication between host and device. In this chapter, we are going to discuss about one of the most important limiting factor in GPGPU: the bandwidth. In the section 3.2 we have seen that access

on every level of GPU memory hierarchy have different bandwidth. Localizing the data access inside a kernel is a key factor for decreasing the number of low bandwidth data transfers. Code divergence is another limiting factor which can drop the performance causing the instruction throughput to decrease. When a warp executing a kernel meets a flow control instruction, it diverges if the threads in the warp follow different flows of execution and if the different instructions are substantial (otherwise predication is used). The hardware maintains a bit vector of active threads and executes the code once for the active and then for the inactive. When all the executions paths are complete the warp re-converge to the original path. All those limitations are present in a Monte Carlo Path Tracer that is based on a random control flow and thus random memory access. It is not straightforward to create a predetermined data access pattern that can be exploited in this case. Dietger et al. van Antwerpen [2011] have proposed a solution based on stream compaction of the active threads at every path extension. In this solution, paths that are still active are compacted together and stored in the global memory of the device. In this section, we will propose a new method that aims to merge the benefits of compacting with the reordering of the rays to exploit warp and data coherence.

4.3.1 Compaction

As we have already discussed in the introduction of the chapter, active thread compaction allows to separate the threads into active and idle. There are two main advantages on using this approach for a MC path tracer:

1. SIMD efficiency increase: the compaction of the active threads allows to have warps fully active or fully inactive.

2. Primary ray coherence is maintained: the regenerated paths usually follow the same control flow path during the first iteration without any code divergence and access the same GPU caches exploiting data locality.

Also in this case we want to analyze the different approaches to compaction:

- Device Compaction (*StreamingMK*): all the active threads inside the device are compacted together. The methods requires two streams of global memory for path data one in input and one in output. The compaction is efficiently performed using an atomic operation which tracks the number of elements written to the output stream. This method requires two kernels, one for regenerating inactive paths and one for extending the paths.

- Tiled Block Compaction (*StreamingSK*): the active threads are compacted only inside a block. The path data is still stored in the global memory, but the compaction of the active threads is performed only on a tile of this data, large as much as the number of threads inside a block. The graphical output of the compaction can be seen in figure 4.4 and pseudo-code in algorithm 5.

In both cases the compaction is performed by the following steps (which are graphically showed in figure 4.3):

1. an exclusive sum on the active labels of the thread allows to find the position of each active thread inside the output stream (which is the same as the input for the tiled block compaction)
2. the number of active threads is updated (the Device compaction requires an atomic operation).
3. the active threads write their data on the output stream.

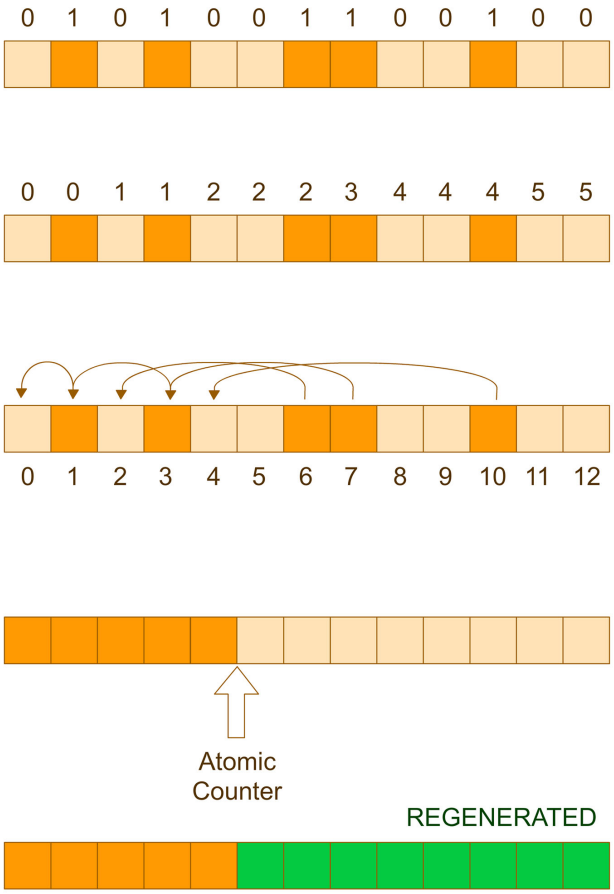


Figure 4.3: steps for compacting a group of threads. In the top active threads are colored in orange and associated with the label 1 inside a row of blocks representing a linear memory of threads. In the second row of threads from the top, an exclusive sum operation is performed on the active labels. In the third row the computed sums are used for identifying the closest empty thread in the memory. The atomic counter is incremented with the last sum computed and the threads which reside after the position indicated by this counter are regenerated

In our tests we have seen a performance boost of the block compaction versus the device compaction. We think that most of the performance gain is given by the use of a single kernel. However, compared to the *RegenerationSK* on a warp the *StreamingVolPTsk* is getting better result, demonstrating the improvement given by the compaction.

Algorithm 5: Kernel which is using block compaction for the active threads. In this algorithm thread is referred to the data which a thread must use for computation. For every path extension the current thread data is reloaded with the function loadThread. After that the total active threads in the block is checked to understand if the thread must regenerate. After regeneration all the block is synchronized (function synchronizeBlock()). The active threads are then extending their paths. After this operations an exclusive sum over all the active labels of the threads is performed. Finally the still active threads are storing their data in the compacted position of the global array of threads.

```
kernel streamingSK
Input  : threads: global memory with the information of each thread,
         scene: scene to render
Output: output: image memory
shared total_active = 0 ;
do
    thread = loadThread(threads) ;
    if thread.id ≥ total_active then
        paths_head = regenerate(thread, paths_head);
    end
    synchronizeBlock();
    if thread.active then
        thread, image = extend(thread, image, scene);
    end
    total_active, compacted_position = exclusiveSum(thread);
    if thread.active then
        threads[compacted_position] = thread;
    end
while total_active > 0 or paths_head < total_paths;
```

Table 4.5: comparison of different types of compaction. The streamingSK method is compacting all the active threads that are in the same block, whereas the streamingMK is compacting all the active threads in all the device. Those two new methods are compared with the regeneration methods which are not using compaction

method	rays/sec			
	cgg-logo	ad	artifix	manix
regenerationSK (thread)	81.62	42.37	11.80	11.42
regenerationSK (warp)	4.37	7.02	11.92	10.95
streamingSK	52.69	37.81	7.42	8.02
streamingMK	6.59	4.70	5.44	5.18

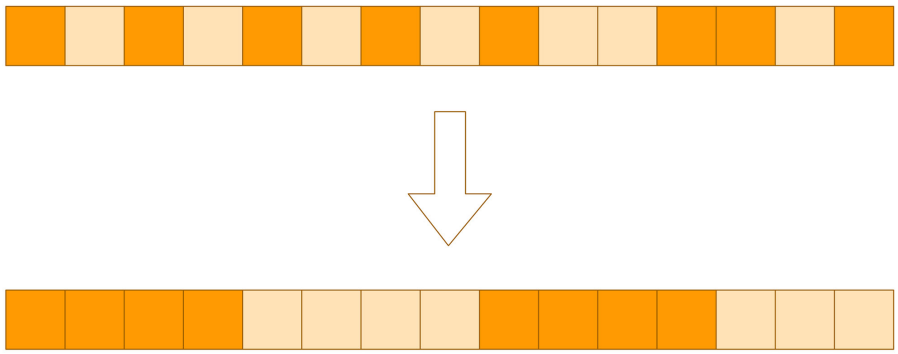


Figure 4.4: Block Compaction. This figure shows how the active threads are compacted using the StreamingSK algorithm. The memory containing the threads is divided in blocks and only inside each block the compaction is performed.

4.3.2 Reordering

In this section we are going to discuss a new method which is based on the previously detailed *StreamingVolPTsk*. We have seen in the previous chapter that compaction is helping on maintaining high SIMD efficiency and primary ray coherence. However, the secondary rays are usually not coherent and they are compacted to the others active rays independently on their position or direction. The method that we propose consists in using the *Morton order*, also called Z-order, to reorder the active rays while compacting. Moon et al. Moon et al. [2010] have already used this technique to achieve a cache-oblivious ray reordering for path tracing; they achieved more than an order of magnitude performance for big models that cannot fit into main memory. Our scope is extending this work to GPU and integrating it with the already discussed streaming compaction.

Morton Order

The Morton order (also called Z-order for its characteristic shape as in figure 4.5) is based on a space filling curve which is a function that maps multidimensional

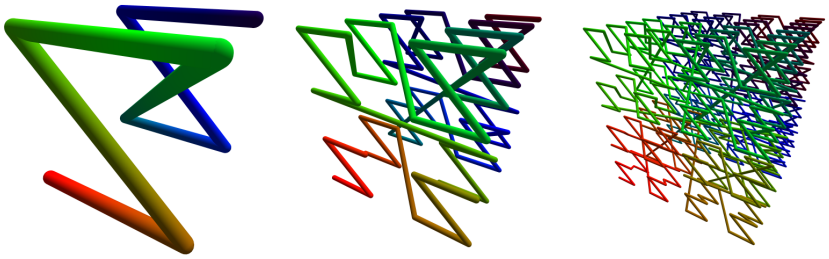


Figure 4.5: Z-Order. from the left to the right we have: 3D z-curve with resolution 2X2X2, 3D z-curve with resolution 4X4X4, 3D z-curve with resolution 8X8X8. It is important to notice how the 3D points are mapped into a linear curve so that points that are near in the 3D space are near also in the curve (the image is taken from the website http://asgerhoedt.dk)

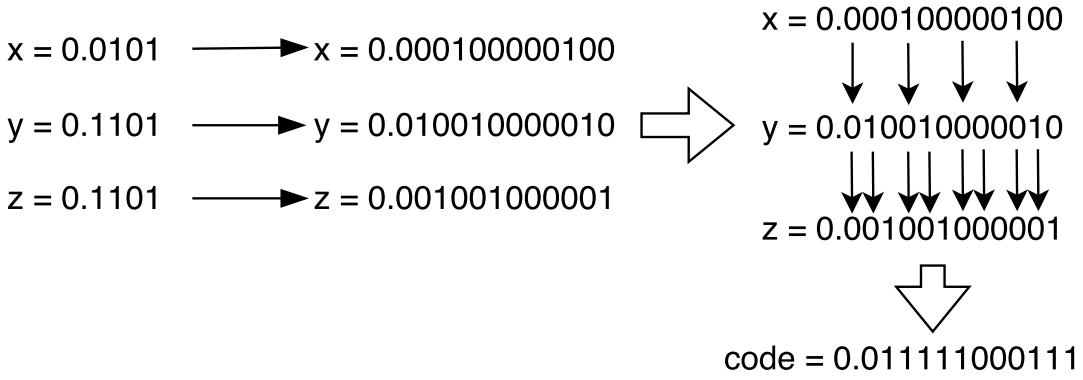


Figure 4.6: Morton code generation: this chart explain the algorithm used to calculate the Morton code (z-value). First the fractional part of the coordinates is taken. Then those coordinates are expanded using 0 values. Finally the coordinates are combined together to create the final code.

data to one dimension, preserving locality of the data points. In our implementation, the z-value of a point is calculated by interleaving bits of the quantized coordinates of the multidimensional data Moon et al. [2010]. The quantized coordinates, which are lying between 0 and 1, are transformed linearly into 10-bit integers. Those integers are then expanded using zeros between the bits of each coordinate, so that the different coordinates can be combined together, interleaving their bits, into a single value. This value represent the z-value of the coordinate (graphical structure of the method can be found in figure 4.6). The resulting ordering correspond of a depth-first traversal of a commonly used 3D data structure called Octree.

Z-Sorting

In our contest, the main source of data is given by the volumetric data that is accessed through the texture memory of the device. Texture accesses in CUDA give best performances in applications where memory access patterns exhibit spatial locality. For this reason reordering rays using the z-order should improve

the access pattern on the texture. The coordinate that have to be used for that is the position of the ray inside the volume. Thus, the bounding box of the volume is used to quantize the ray position. The quantized position are then transformed as we have previously seen into z-values. Finally, we are reordering the rays using the block radix sort, where the keys used for reordering are the z-values of the ray position (as shown in figure 4.7. If the thread is not active the maximum z-value is associated with the ray. This method allows to sort and compact at the same time the threads inside a single block. We have analyzed two possible implementation of the z-sorting:

1. Sorting and compacting before texture access (*StreamingSK* with sorting variant): consists in using the same tiled block compaction already seen before, but this time, instead of using a scan operation to compact the active rays, we are using Morton sorting.

2. Sorting and compacting after texture access (*SortingSK*) : consists in delaying the texture access for the albedo volume of the volumetric path tracer after reordering and compaction of the rays (a pseudo code of the algorithm can be found in 6).

The first option should help to create more coherent rays, which in the next path extension will have more probability to access data that is near in memory. The second option uses a shared array of labels describing if the albedo texture is accessed by the thread during the last extension. This array is then used after reordering and compaction to decide which of the reordered paths need to access the albedo texture. Finally, the texture is accessed by the reordered threads. That is, the last option allows to access the texture with the more coherent way given random access positions.

4.3.3 Summary

In this section we have discussed different techniques to improve data locality and thread divergence. Compaction allows to create coherent primary rays and to improve SIMD efficiency, while reordering of rays tries to improve secondary rays coherency inside the volume. Unfortunately, the results show that the increased data coherence are not affecting the performance of the algorithm. One of the reasons for this behavior could be the too high texture resolution compared to the number of sample points inside the volume, represented by the texture origins. If the ratio of those two values is too big, then the method cannot explore data coherence by reordering the rays. Indeed, in this case, considering the random position of the rays inside the volume it is improbable that their position will be close. Therefore, even if the method is accessing the texture in a coherent order, the texture cache is not big enough to store the values between one texture access and the next one.

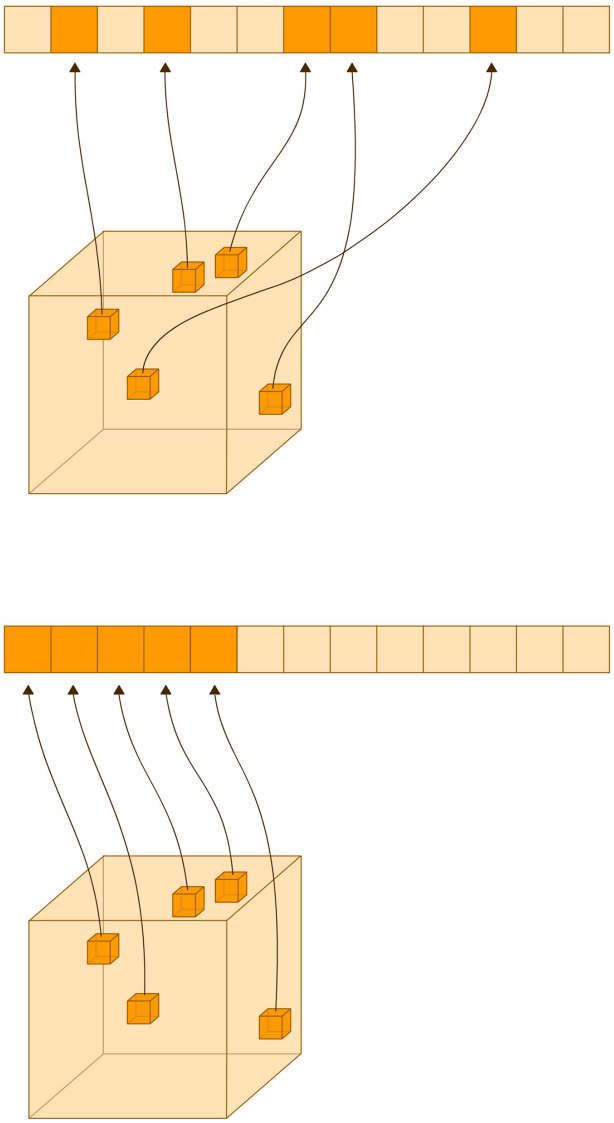


Figure 4.7: z-reordering of the data. In this image is explained the reordering methods used in this work. At the top the points inside the volume (orange blocks inside the cube) that correspond to a thread in the global memory (represented by a row of blocks) are placed independently on their position inside the volume. In the bottom the threads corresponding to the points in the volume are reordered based on the z-value of those points and compacted.

Algorithm 6: kernel which is using z-sorting to reorder rays and it access the texture after the reordering. This algorithm is very similar to the streamingSK algorithm 5. The only difference here is that here reordering instead of compaction is performed and the albedo texture access is delayed after the reordering.

```
kernel sortingSK
shared total_active = 0;
shared tex_access[n_threads] = false;
do
    thread = loadThread(threads);
    if thread.id ≥ total_active then
        paths_head = regenerate(thread, paths_head);
    end
    synchronizeBlock();
    tex_access, thread, image = extend(thread, image, scene, tex_access);
    sorted_position = blockZSorting(thread);
    total_active = sum(thread);
    thread = swapThreads(thread, sorted_position);
    if tex_access[sorted_position] then
        thread = accessTexture(thread);
    end
while total_active > 0 or paths_head < total_paths;
```

Table 4.6: sorting comparison: in this table two different sorting methodologies are adopted. the first one, streamingSK (sorting) is behaving exactly like the streamingSK with the only difference that the sorting with the z-order based on the ray position is used instead of the compaction. The second one, sortingSK, have the only difference of postponing the texture access after the reordering like in algorithm 6. It is clear from the table that the sorting methods are not performing better than the other ones on those scenes.

method	rays/sec			
	cgg-logo	ad	artifix	manix
regenerationSK (thread)	81.62	42.37	11.80	11.42
regenerationSK (warp)	4.37	7.02	11.92	10.95
streamingSK (compaction)	52.69	37.81	7.42	8.02
streamingSK (sorting)	20.41	17.61	6.78	6.74
sortingSK	20.90	17.83	7.02	6.87

5. Implementation Details

In this chapter we are going to cover some details on the implementation of the previously described algorithms. Software design is particularly difficult in CUDA, as to achieve the maximum performance all the branching decision must be made as early as possible. Furthermore, there is no possibility to use virtual functions, abstraction and classic polymorphism. We will see in the next section how we can deal with those constraints and how to create a as generic as possible volumetric path tracer in CUDA. The solution proposed is the base of the implementation provided in the attachment 6.2.

5.1 Generic Programming

To improve code re-usability and flexibility many of the most important CUDA libraries like CUB and thrust, which we are also employing inside the project, use Generic Programming. This technique leverages the power of C++ templates to create a compile-time abstraction layer. That is, all the callback or virtual classes are substituted by a template. This template will represent a data structure which must contain all the functions and members requested. In the case of virtual functions this is done with the use of Functors: data structure which define the operator `()`. In all the implementations of the volumetric path tracing provided this technique is used. The only template which must be defined is the Scene. This template requires to define an intersection structure, called `SceneIsect`, which contains all the information about the intersection with the scene. From the scene should be also possible to access the medium and the BSDF based on the information on the intersection structure. Sampling of the distance with the woodcock tracking 3 and of the albedo is also done using a template Medium. For this reason, the implementation of the kernels are completely general and reusable. Every type of scene can be substituted to the template without requiring any change for the kernel. Moreover, this system allows to create custom device scene and to improve the performance of the method based on the type of scene. In figure 5.1 is shown a part of the software architecture relative to the kernel composition. The right kernel with the right scene is selected at run-time by a `RenderFactory` which creates the renderer based on a configuration object.

5.2 Scene Assembler

The software provides a configuration system that is flexible and easy to extend. All the configurations for all the algorithms are stored inside a single `Config` object, which is used to initialize all the renderer. This configuration object, however, can be populated using different systems. In this moment, the only interface with a user wanting to use the software is represented by the command line. The commands provided by the user are analyzed by a `ConfigParser` object which has the ability to parse and create a `Config` object. The command line is used principally for configurations relative to the algorithm used for rendering. For the configurations relative to the scene, another object is used: the

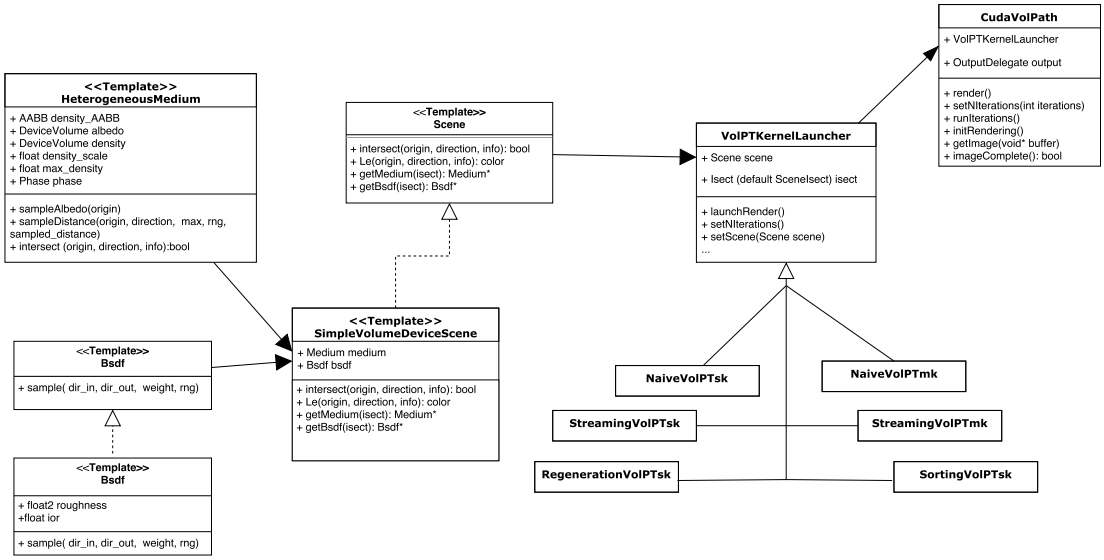


Figure 5.1: kernel launch software architecture. The figure shows the method used to create generic implementation of the volumetric path tracing without affecting the performance. Concretely, different implementation are compiled for each different variant of the algorithm. The template interfaces permits to do that without affecting re-usability of the code.

`SceneBuilder`. The `SceneBuilder` is an interface which allows to load different scene types without affecting the normal work-flow of the renderer. In this moment, there are two implementations of this interface: the `XmlSceneBuilder` and the `MhaSceneBuilder`. The first one allows to load the medium from a Mitsuba Xml scene file, while the second one from a VTK mha volume format. Which one of those builders to use is chosen by the `ConfigParser` object which selects the right one based on the command line argument provided by the user. Finally, the `SceneAssembler` provides an interface between the `SceneBuilder` and the `ConfigParser` and is also responsible for the creation of the `Scene` object, which will be placed inside the final configuration object of the renderer. The figure 5.2 shows a charts of the system just explained.

5.3 Interactive Renderer and Transfer Delegation

There are two main configurations to run the software: testing mode and interactive mode. The testing configuration works only by command line and allows to benchmark one of the algorithms presented in the previous chapter. The user can specify the number of trials and the software will run the algorithm the number of times specified, returning the the mean time, standard deviation and rays/sec for the algorithm on the given scene. Instead, the interactive mode uses the GLFW library, which is an OpenGL multi-platform library, to create a new window where the scene is rendered one iteration for every frame. The interactive renderer works as an additional layer upon the normal renderer. There are different objects which allow to decouple the different elements of the system:

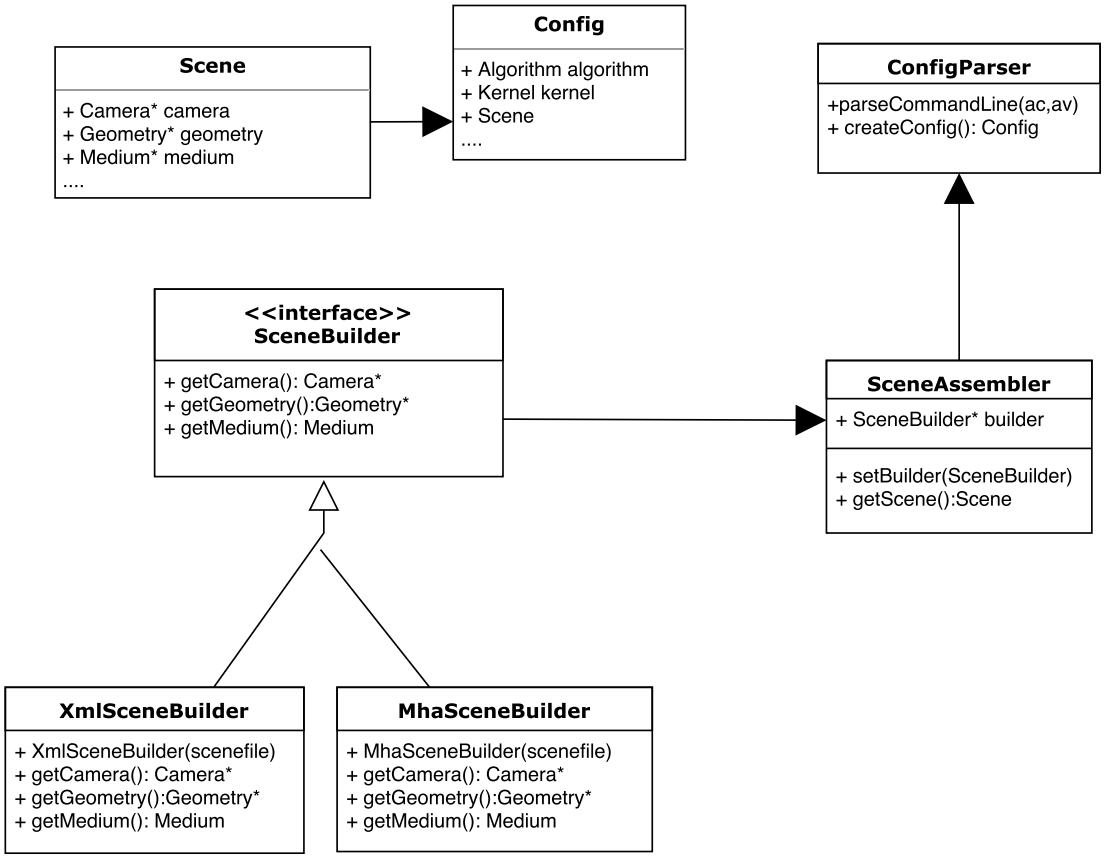


Figure 5.2: Configuration Architecture. In this image is shown the system used to permit the loading of different scene formats inside the software. Moreover, the SceneBuilder interface provide an extension point for the loading of even more formats.

- `GLViewController`: allows to create the main window. By running the OpenGL main loop, it draws the pixel buffer object on the window and controls the `InputController` and the `BufferProcessorDelegate`.
- `InputController`: is an interface which allows to use the GLFW events to modify the scene. An implementation is the `CameraController` which allows to orbit the camera around the center of the object during the rendering.
- `BufferProcessorDelegate`: is an interface which allows to modify the OpenGL pixel buffer object. The only implementation provided is the `CudaInteractiveRenderer` which takes the normal renderer and uses it to render the next frame on the OpenGL pixel buffer object. To make this possible, the object leverages the power of the interoperability between CUDA and OpenGL.

However, there is a difference between the normal renderer and interactive renderer and it depends on the output of the renderer. In one case the output must be transfered in the host memory, while in the other one it should be transfered in another buffer always inside the device memory, so to use the interoperability between CUDA and OpenGL. For this reason, the `CudaVolPath` uses a delegate object to transfer the output buffer from the source to its destination. The interface of this delegate is called `Buffer2DTransferDelegate` and three implementations are provided in the software:

- `HostImageBufferTansferDelegate`: it is the basic method which transfers the buffer from the device memory to the host memory.
- `DeviceImageBufferTansferDelegate`: this transfer delegate transfers the buffer from the device memory to device memory.
- `DeviceTiledImageBufferTansferDelegate`: works exactly like the previous one but it allows to update the image one tile per frame.

Those transfer delegates can take as an argument a Functor, which must be used on the output image before the transfer is complete. In the `CudaVolPath` this transformation allows to scale the rendering output by the number of iterations of the Monte Carlo estimation. Also the transfer delegate is chosen by the `RenderFactory` at the beginning, based on the launching configuration provided by the user. All the system is shown in figure 5.3.

5.4 Zero-Copy Volume

Storing all the volume texture inside the device memory is not always possible. Sometimes the volume is too big and it cannot fit inside the memory available in the hardware. For this reason, the software allows to store the volume data inside the host memory. It is clear that this is not a good idea considering that the bandwidth between CPU and GPU is much lower than the bandwidth with the DRAM. The render will be, in this case, much slower on accessing the volume, while with the use of the texture cache this will happen only if

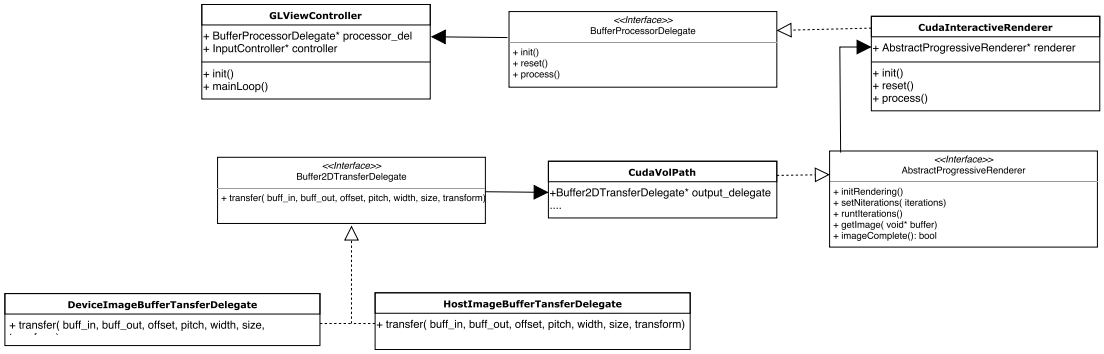


Figure 5.3: Interactive Renderer Architecture. The interactive renderer has been created without compromising the generality of the algorithm. The algorithm created can be used inside a BufferProcessorDelegate which is providing the data necessary to the GLViewController for rendering the image frame by frame.

there is a cache miss. In the newer devices (compute capability 6.x or higher) this can be implemented more efficiently using the managed memory, that has inside a page fault mechanism, for which memory pages can be stored inside the device local memory. However, our target architecture is the Kepler (compute capability 3.0), so the software does not use this technique. Instead, the zero copy memory is used, this technique allows to use a page locked memory on the host directly inside the device. Unfortunately, it is not possible at the moment to create a CudaArray, meaning memory layouts that are optimized for local texture fetching, with this type of memory. Therefore, we create the texture using the linear memory pointer. This has the disadvantage of not exploiting the three-dimensional data locality, inside the texture making useless algorithms, like the previously described sortingSK.

6. Discussion

In this chapter we will discuss about some of the results we have reached. We will compare the render time, achieved with the methods explained in the chapter 4, with a renderer commonly used by the Computer Graphics research community: the Mitsuba renderer by Jakob [2010]. Moreover, we will discuss some correlation between the scenes and the best performing method.

6.1 Results

In table 6.1 we can see the first comparison of the Mitsuba renderer with the scenes presented in figure 4.2. There are multiple algorithms that can be used inside the Mitsuba renderer to render the scene, however the algorithm that works best in this type of scene is the simple volumetric path tracing (called volpath_simple). Contrary to our methods, this algorithm is also using shadow rays. For this reason, the total number of rays traced for this algorithm is calculated by adding the shadow rays and the normal rays traced. The results show that the number of rays traced every second by the Mitsuba volumetric path tracer is much lower than any other algorithm. In the scenes used in this comparison the density is constant and the algorithm that is performing better is the regenerationSK using the single thread regeneration. The reason being that if the density is constant, also the standard deviation over the length of a path that traverses the medium will be small. Thus, the rays are starting and stopping by themselves all together and there is no need for extra care on synchronizing the threads or compacting them. The other conclusion that is possible to take is that the GPU is performing much better than the CPU algorithm also considering the naive single kernel approach.

Table 6.1: comparison with Mitsuba renderer. This table shows the comparison of the algorithm showed in the thesis with a CPU volumetric path tracing implementation provided by Jakob [2010]. The table shows that the CPU algorithm is performing worse in all the case. The two measures provided are the millions of rays traced every second and the total time for rendering a 400x400 image with the scene

method	rays/sec	
	cgg-logo	ad
naiveSK	3.88	2.13
regenerationSK (thread)	81.62	42.37
regenerationSK (warp)	4.37	7.02
streamingSK (compaction)	52.69	37.81
streamingSK (sorting)	20.41	17.61
sortingSK	20.90	17.83
mitsuba (volpath_simple)	0.003	17.83

Let's consider now a scene with varying density but not varying albedo, like the one in figure 6.1. The results relative to this scene are presented in the table

Figure 6.1: smoke scene: heterogeneous volume representing smoke. The file which have grid resolution of (128,128,50) can be found in the website of the Mitsuba renderer Jakob [2010]. The density scale used in this scene is 800.

6.2. The table shows the millions of rays per second and the total time to render the smoke scene with density scale factor 800 which corresponds to the scaling factor of the 0 to 1 density volume. The results are showing that in this case the algorithm that is performing better is the streamingSK. The explanation can be found in the long duration of some rays inside the medium compared to other rays that are ending immediately. In this case, the streaming method is able to group together the long term rays and decrease the divergence of the warps. This results in a better efficiency of the streaming algorithm when the scenes present high varying density. However, the complexity of the density function is not the only factor to take in consideration.

Table 6.2: comparison with Mitsuba renderer on the smoke scene. This table shows the behavior of the algorithms showed in this work in the case the scene present high varying density and high resolution.

method	rays/sec	time (sec)
regenerationSK (thread)	2.52	131
regenerationSK (warp)	7.35	127
streamingSK (compaction)	17.41	53.68
sortingSK	14.64	63.83
mitsuba (volpath_simple)	0.58	1076
mitsuba (volpath)	0.54	1195

The scene in figure 6.2 presents an high varying density but a small texture resolution. In this case the methods that are using compaction are performing worse than the RegenerationSk method with a single thread regeneration as it is showed in the table 6.3. A possible reason for this behavior can be attributed to the ratio between traced paths and grid resolution. That is, in this type of scenes if the number of rays is high enough it is more probable that groups of rays access

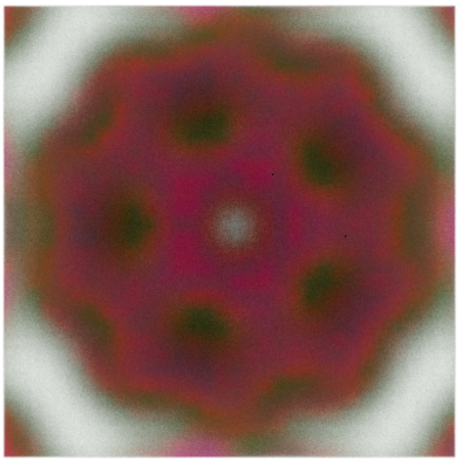


Figure 6.2: bucky heterogeneous volume with grid resolution of (32,32,32). The file is usually used for testing volumetric rendering because it presents an high varying density. In this test the density values are varying between 0 and 40. For the color a static transfer function is applied which maps green to low density values, red to medium density values and blue to high density values. The number of iterations used for the Monte Carlo estimation is 300.

the same texture data. That happen because the texture resolution is much lower respect to the previous cases. If threads that are close together access the same texture values they are behaving similarly to the case of constant density factor and exactly like in that case a simple method, like the regenerationSK, that fully utilize the GPU gives the best performance gain.

Table 6.3: comparison of the GPU rendering algorithms detailed in the chapter 4 on a high varying density scene in figure 6.2 with many density holes and small resolution

method	rays/sec	time (sec)
regenerationSK (thread)	10.96	39.71
streamingSK (compaction)	5.75	52.83
sortingSK	5.69	53.30
naiveSK	4.99	60.86

6.2 Which is the Best?

Looking at the results that we have showed, we can say that the best algorithm to use for GPU volumetric path tracing closely depends on the type of scene we want to render. We have seen that for complex scenes with high varying density and high resolution texture the more sophisticated algorithm which uses compaction and smart regeneration are performing better. On the other hand, if our scene is a simple scene, simple methods which requires less synchronization between threads and less branch divergence are getting better results.

Conclusion

In this work we have implemented and analyzed different methods to optimize a volumetric path tracer. We have seen that, especially on GPU, the type of path tracing algorithm that performs best is tightly coupled with the type of scene we want to render. If the scene is simple, methods that are simple are performing better. If the scene is complex, it is better to use a different strategy and concentrate on utilization and data locality. We have proposed new variant of existing algorithms for regeneration based on the synchronization level we want to achieve. Moreover, we have implemented a new sorting strategy which builds upon the idea of maintaining the ray coherence of the path tracer not only for the primary generated paths but also during scattering. This new method is performing as good as the best streaming algorithm but we believe that, using bigger and different datasets, it can lead to further improvements in time performance.

Future Work

There are many possible future directions to take based on the results showed in this work. The first thing is an optimized usage of out of core textures with the use of CUDA managed memory. Indeed, our work is currently limited by the available memory in the GPU. We have implemented a zero copy texture which however is poor in performance compared to the normal version. The usage of more memory for testing will permit to test bigger scenes where we believe the sorting algorithms can perform better. Moreover, the implementation can be generalized to handle any type of scenes and geometry using the state of the art intersection framework provided by Aila and Laine [2009]. After that the algorithms should be tested on different type of hardware which can lead to more details about the different use of the cache between different GPU architectures. Finally, CUDA is not the only language which permits GPGPU computing and for this reason the implementation should be ported and tested also with other GPGPU programming languages like OpenCL.

Bibliography

Timo Aila and Samuli Laine. Understanding the efficiency of ray traversal on gpus. In *Proceedings of the Conference on High Performance Graphics 2009*, HPG '09, pages 145–149, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-603-8. doi: 10.1145/1572769.1572792. URL `http://doi.acm.org/10.1145/1572769.1572792`.

Tomáš Davidovič, Jaroslav Křivánek, Miloš Hašan, and Philipp Slusallek. Progressive light transport simulation on the gpu: Survey and improvements. *ACM Trans. Graph.*, 33(3):29:1–29:19, June 2014. ISSN 0730-0301. doi: 10.1145/2602144. URL `http://doi.acm.org/10.1145/2602144`.

Eugene d'Eon. *A Hitchhiker's Guide to Multiple Scattering*. 2016.

Oskar Elek, Denis Sumin, Ran Zhang, Tim Weyrich, Karol Myszkowski, Bernd Bickel, Alexander Wilkie, and Jaroslav Křivánek. Scattering-aware texture reproduction for 3d printing. *ACM Trans. Graph.*, 36(6):241:1–241:15, November 2017. ISSN 0730-0301. doi: 10.1145/3130800.3130890. URL `http://doi.acm.org/10.1145/3130800.3130890`.

V. A. Frolov and V. A. Galaktionov. Low overhead path regeneration. *Programming and Computer Software*, 42(6):382–387, Nov 2016. ISSN 1608-3261. doi: 10.1134/S0361768816060025. URL `https://doi.org/10.1134/S0361768816060025`.

K. Gupta, J. A. Stuart, and J. D. Owens. A study of persistent threads style gpu programming for gpgpu workloads. In *2012 Innovative Parallel Computing (InPar)*, pages 1–14, May 2012. doi: 10.1109/InPar.2012.6339596.

Eric Heitz. A Simpler and Exact Sampling Routine for the GGX Distribution of Visible Normals. Research report, Unity Technologies, April 2017. URL `https://hal.archives-ouvertes.fr/hal-01509746`.

Wenzel Jakob. Mitsuba renderer, 2010. http://www.mitsuba-renderer.org.

Wojciech Jarosz. *Efficient Monte Carlo Methods for Light Transport in Scattering Media*. PhD thesis, UC San Diego, September 2008.

James T. Kajiya. The rendering equation. *SIGGRAPH Comput. Graph.*, 20 (4):143–150, August 1986. ISSN 0097-8930. doi: 10.1145/15886.15902. URL `http://doi.acm.org/10.1145/15886.15902`.

Samuli Laine, Tero Karras, and Timo Aila. Megakernels considered harmful: Wavefront path tracing on gpus. In *Proceedings of the 5th High-Performance Graphics Conference*, HPG '13, pages 137–143, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2135-8. doi: 10.1145/2492045.2492060. URL `http://doi.acm.org/10.1145/2492045.2492060`.

Daniel Seibert Matthias Raab and Alexander Keller. Unbiased global illumination with participating media.

Michael McCool, James Reinders, and Arch Robison. *Structured Parallel Programming: Patterns for Efficient Computation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2012. ISBN 9780123914439, 9780124159938.

Bochang Moon, Yongyoung Byun, Tae-Joon Kim, Pio Claudio, Hye-Sun Kim, Yun-Ji Ban, Seung Woo Nam, and Sung-Eui Yoon. Cache-oblivious ray reordering. *ACM Transactions on Graphics*, 29(3):1–10, jun 2010. doi: 10.1145/1805964.1805972.

Jan Novák, Vlastimil Havran, and Carsten Daschbacher. Path regeneration for interactive path tracing. pages 61–64. Eurographics Association, 2010.

Nvidia. cuda-c-programming-guide, 2017. URL `http://docs.nvidia.com/cuda/cuda-c-programming-guide`.

Steven G. Parker, James Bigler, Andreas Dietrich, Heiko Friedrich, Jared Hoberock, David Luebke, David McAllister, Morgan McGuire, Keith Morley, Austin Robison, and Martin Stich. Optix: A general purpose ray tracing engine. *ACM Trans. Graph.*, 29(4):66:1–66:13, July 2010. ISSN 0730-0301. doi: 10.1145/1778765.1778803. URL `http://doi.acm.org/10.1145/1778765.1778803`.

Matt Pharr and Greg Humphreys. *Physically Based Rendering: From Theory to Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004. ISBN 012553180X.

Dietger van Antwerpen. Improving simd efficiency for parallel monte carlo light transport on the gpu. In *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*, HPG '11, pages 41–50, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0896-0. doi: 10.1145/2018323.2018330. URL `http://doi.acm.org/10.1145/2018323.2018330`.

H.C. van de Hulst. *Light scattering by small particles*. Dover Publications Inc., New York, USA, 1981.

Eric Veach. *Robust Monte Carlo Methods for Light Transport Simulation*. PhD thesis, Stanford University, 1997.

Vasily Volkov. Better performance at lower occupancy. 10, 01 2015. URL `http://www.nvidia.com/content/GTC-2010/pdfs/2238_GTC2010.pdf`.

Ingo Wald. Active thread compaction for gpu path tracing. In *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*, HPG '11, pages 51–58, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0896-0. doi: 10.1145/2018323.2018331. URL `http://doi.acm.org/10.1145/2018323.2018331`.

Murphy T. Hemmings P. Woodcock, E. and L. T.C. Techniques used in the gem code for monte carlo neutronics calculations in reactors and other systems of complex geometry. 1965.

List of Figures

List of Tables

List of Abbreviations

Bsdf Bidirectional Scattering Distribution Function
CUDA Compute Unified Device Architecture
GPU Graphical Processing Unit
σ_t extinction coefficient
σ_s scattering coefficient
σ_a absorption coefficient
α albedo
VRE Volumetric Rendering Equation

Attachments

Digital content

implementation/src : source code of the project.

implementation/external : external libraries used inside the project.

implementation/executable : executable of the software for rendering a volume.

implementation/data : example data which can be used by the renderer

implementation/VisualStudio2015_Cuda9.0 : contains the Visual Studio project for compiling the code. It requires Cuda9.0 to be installed.

Thesis : pdf of the thesis

scripts : scripts which permits easily to run the rendering on the different data sets.

Minimum System Requirements

To run the program is important to have a CUDA enabled GPU.

Detailed Hardware Specification

The hardware that we will use for testing is a MacBook Pro with CPU Intel Core i7 quad-core

- 2,3GHz (Turbo Boost until 3,3GHz).
- cache L3 : 6 MB.
- cache L2 per core: 256 KB.
- cache L1 per core: 32 KB.
- ram : 8 GB DDR3 1600 MHz.

and GPU Nvidia GeForce 650M with the following characteristics:

- CUDA Capability Major/Minor version number: 3.0
- Total amount of global memory: 512 MBytes (536870912 bytes)
- (2) Multiprocessors, (192) CUDA Cores/MP: 384 CUDA Cores
- GPU Max Clock rate: 405 MHz (0.41 GHz)
- Memory Clock rate: 2000 Mhz
- Memory Bus Width: 128-bit
- L2 Cache Size: 262144 bytes (number of registers * 4)
- L1 Cache Size: 64 bytes (divided between shared memory and cache)
- Maximum Texture Dimension Size (x,y,z) 1D=(65536), 2D=(65536, 65536), 3D=(4096, 4096, 4096)
- Maximum Layered 1D Texture Size, (num) layers 1D=(16384), 2048 layers
- Maximum Layered 2D Texture Size, (num) layers 2D=(16384, 16384), 2048 layers
- Total amount of constant memory: 65536 bytes
- Total amount of shared memory per block: 49152 bytes
- Total number of registers available per block: 65536 (equal to registers per SM)
- Warp size: 32
- Maximum number of threads per multiprocessor: 2048
- Maximum number of threads per block: 1024
- Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
- Max dimension size of a grid size (x,y,z): (2147483647, 65535, 65535)

- Maximum memory pitch: 2147483647 bytes
- Texture alignment: 512 bytes
- Concurrent copy and kernel execution: Yes with 1 copy engine(s)
- Run time limit on kernels: Yes
- Integrated GPU sharing Host Memory: No
- Support host page-locked memory mapping: Yes
- Alignment requirement for Surfaces: Yes
- Device has ECC (error correction code) support: Disabled
- CUDA Device Driver Mode (TCC or WDDM): WDDM (Windows Display Driver Model)
- Device supports Unified Addressing (UVA): Yes
- Device PCI Domain ID / Bus ID / location ID: 0 / 1 / 0