Global Illumination with GPU Vertex Connection and Merging

João Norberto Antunes Monteiro

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisor: Prof. João António Madeiras Pereira

Examination Committee
Chairperson: Prof. Mário Jorge Costa Gaspar da Silva
Supervisor: Prof. João António Madeiras Pereira
Member of the Committee: Prof. Adriano Martins Lopes

October 2020
Dedicated to my brother,
who would be proud of my achievements.
Acknowledgments

I would like to thank Instituto Superior Técnico and all my professors for the high quality education I was provided, with a special distinction for my thesis supervisor, João Madeiras Pereira for all the help, support and availability, as well as the valuable feedback since the beginning. My special thanks also to the examination committee as well, who ensured a very smooth and pleasant discussion of this thesis, and used their valuable time to evaluate my work.

I would also like to thank my parents, for the constant support when I told them that I was going to leave my full time job to pursue further education, and when I left Portugal to work abroad.

My best friend João Marques, with whom I shared an apartment for 5 years, that helped me focusing on my goals when times were less easy, but also had lots of fun, like our trips to the beach after work, or our music sessions.

Finally, I would like to thank my girlfriend, Ana Fernandes, for always supporting and believing in me, introducing me to 3D modeling and the world of animation and interactive applications. The journey has been immensely easier with you by my side, especially when not physically together.
Resumo

Ao longo dos anos têm havido desenvolvimentos contínuos no campo dos gráficos gerados por computador. Em 2012 foi apresentado um novo algoritmo de renderização, o vertex connection and merging, que consiste na utilização de diferentes algoritmos já existentes para processar uma cena, de forma a obter as imagens com melhor qualidade geradas entre eles.

A principal desvantagem desta abordagem, que também está presente em cada um dos algoritmos de renderização, encontra-se no tempo que demora a criar um render realista. Uma das abordagens possíveis para reduzir o tempo necessário para processar a informação, está na utilização de uma unidade de processamento gráfico, de forma a tirar partido dos seus múltiplos núcleos de processamento e das suas capacidades de paralelização de instruções.

Os algoritmos de renderização são, por natureza, algoritmos que utilizam a memória de uma forma que não é a mais eficiente, como por exemplo, o uso da cache. Isto deve-se ao facto de caminhos traçados entre a câmara e fontes de luz poderem seguir caminhos bastante distintos, mesmo para píxeis vizinhos, o que causa acessos a zonas distantes na memória. Por outro lado, o processamento de aplicações segundo o modelo de programação CUDA é mais eficiente quando a memória também é acedida de forma mais eficiente.

Esta tese explora a adaptação de uma implementação do vertex connection and merging para ser processada no GPU, tendo sido originalmente desenvolvida para ser processada no CPU, com o intuito de obter imagens geradas por computador de uma forma mais rápida. Diferentes tipos de memória são utilizados para tentar optimizar o desempenho do GPU, e é feita uma análise extensiva do desempenho utilizando as ferramentas de profiling da NVIDIA. No final, esta análise justifica o porquê dos resultados não serem os esperados, sendo as imagens geradas de forma mais rápida na implementação original do algoritmo.

Palavras-chave: CUDA, GPU, VCM, computação gráfica, paralelismo, renderização
Abstract

There have been continuous developments in the field of computer generated graphics. In 2012, a new algorithm was presented, the vertex connection and merging, which consists in using different already existing algorithms to process a scene, to get the generated images with best quality among them.

The main disadvantage of this approach, which is also present on each of the rendering algorithms, consists in the time it takes to create a realistic render. One of the possible approaches to reduce the time needed to process the information is the usage of a graphics processing unit, in order to take advantage of its multiple cores and parallelization capabilities.

The rendering algorithms use, by nature, memory not in the most efficient way, with cache memory as an example. This is due to paths traced between camera and light sources having the possibility to be created in very distinct directions, even for adjacent pixels, which causes distant memory positions to be accessed, resulting in cache misses. On the other hand, CUDA applications are the most efficient when efficient memory access patterns are used.

This thesis explores the conversion of an implementation of the vertex connection and merging algorithm, originally developed to be processed on the CPU, to be processed using NVIDIA GPUs. The goal of this conversion is to have images generated faster by the computer. The different types of memory in the GPU are used to optimize its performance, and a detailed analysis of the performance is done using the profiling tools provided by NVIDIA. Finally, this analysis explains why the obtained results are different from the expected, being the images generated faster using the original algorithm implementation.

Keywords: CUDA, GPU, VCM, computer graphics, parallelism, rendering
Contents

Acknowledgments .......................................................... v
Resumo ........................................................................... vii
Abstract ......................................................................... ix
List of Tables .................................................................... xv
List of Figures ................................................................... xvii
Nomenclature .................................................................... xxi
Glossary ........................................................................... 1

1 Introduction .................................................................... 1
  1.1 Thesis Outline ............................................................ 2

2 Background ...................................................................... 5
  2.1 Illumination ............................................................... 5
    2.1.1 Multiple importance sampling ................................. 5
    2.1.2 Ray tracing .......................................................... 6
    2.1.3 Path tracing .......................................................... 6
    2.1.4 Light tracing ........................................................ 7
    2.1.5 Bidirectional path tracing ....................................... 7
    2.1.6 Photon mapping .................................................... 7
    2.1.7 Progressive photon mapping ................................... 8
  2.2 Vertex connection and merging ..................................... 8
    2.2.1 Light path tracing ................................................ 8
    2.2.2 Camera path tracing .............................................. 9
    2.2.3 Vertex connection ............................................... 9
    2.2.4 Vertex merging ................................................... 9
    2.2.5 Algorithm overview ............................................. 10
    2.2.6 SmallVCM ........................................................ 12
  2.3 Parallel processing using GPU ...................................... 12
    2.3.1 NVIDIA CUDA ................................................ 12
    2.3.2 GPU threads ..................................................... 14
    2.3.3 GPU Memory ................................................... 15
### 4.5.2 Peak signal to noise ratio

4.5.3 Root mean squared error

4.5.4 Structure similarity index metric

5 Results evaluation

5.1 Baseline Solution

5.2 CUDA VCM evaluation

5.2.1 Scene 1

5.2.2 Scene 2

5.2.3 Scene 3

5.2.4 Scene 4

5.2.5 Render time for all scenes per device

5.3 CUDA kernel profiling

5.3.1 Performance overview

5.3.2 Occupancy

5.3.3 Branch statistics

5.3.4 Instruction statistics

5.3.5 Issue efficiency statistics

5.3.6 Memory statistics

5.4 Final considerations

6 Conclusions

6.1 Achievements

6.2 Code structure and hardware usage

6.3 Memory usage

6.4 Profiling

6.5 Future Work

6.6 Render a scene using multiple smaller portions

6.6.1 Shared memory

6.6.2 Hash grid calculation using GPU

6.6.3 Split camera path tracing kernel into smaller ones

6.6.4 Support for non square images

Bibliography

A OptiX framework

A.1 OptiX 6.5 and 7.1 framework comparison

B CUDA VCM source code

B.1 Scene lights in shared memory
# List of Tables

2.1 Streaming processors in a compute capability 8.0 CUDA GPU (Source: NVIDIA 2020 [13]) 13
2.2 Performance Improvements Optimizing $C = A^T A$ Matrix Multiplication (Source: NVIDIA 2017-2 [15]) 16
2.3 Performance comparison between SmallVCM and OptiX VCM implementations (Source: Bico 2016 [19]) 18
2.4 Performance comparison between SmallVCM and Embree implementation (Source: Bico 2016 [19]) 18
3.1 Software and drivers used in development environment 22
3.2 Instructions for different warps sizes 23
3.3 Kernel configurations according to memory and kernels used 24
3.4 Memory requirements for postponing camera connection on each light sub path 27
3.5 Memory requirements for hash grid, for a single iteration 29
4.1 Scenes used for implementation evaluation 31
4.2 Software versions in desktop development environment 32
5.1 Render time for scene 1 40
5.2 Image comparison metrics for scene 1 40
5.3 Render time for scene 2 42
5.4 Image comparison metrics for scene 2 42
5.5 Render time for scene 3 44
5.6 Image comparison metrics for scene 3 44
5.7 Render time for scene 4 45
5.8 Image comparison metrics for scene 4 46
5.9 Execution time for CUDA sample projects 68
List of Figures

2.1 Sampling the light sources, sampling the BSDF and MIS (Source: Veach 1995 [4]) . . . . 6
2.2 Merging of BDPT and PM using an heuristic (highlighted in red and green), and VCM with
MIS (Source: Georgiev et al. 2012 [10]) .............................................................. 8
2.3 Camera and light paths with different lengths, and merging of camera and light vertices
within the acceptance radius (Source: Georgiev et al. 2012 [10]) ................................. 10
2.4 Comparison of the transistor allocation between a CPU and a GPU (Source: NVIDIA 2020
[13]) .................................................................................................................................... 13
2.5 Theoretical performance comparison between a CPU and a CUDA GPU (Source: NVIDIA
2017 [13]) .................................................................................................................................... 14
2.6 Bidimensional grid and block of threads (Source: NVIDIA 2020 [13]) ............................ 15
2.7 Memory layout on a CUDA GPU (Source: NVIDIA 2020 [15]) ........................................ 15
2.8 OptiX 6.5 overview, where the yellow boxes are programmable (Source: NVIDIA 2020 [17]) 17
2.9 OptiX 7.1 overview, where the gray boxes are programmable (Source: NVIDIA 2020 [18]) 17

3.1 CUDA VCM execution overview diagram ................................................................. 20
3.2 Frame buffer with color data for a single iteration ...................................................... 21
3.3 Frame buffer with color data for multiple iterations .................................................... 21
3.4 Color data copy from temporary to main frame buffer ................................................ 29

4.1 Occupancy sample charts ............................................................................................. 34
4.2 Instruction statistics sample charts ................................................................................ 34
4.3 Branch statistics sample charts ................................................................................... 35
4.4 Issue efficiency sample charts .................................................................................... 35
4.5 Memory statistics sample charts .................................................................................. 36

5.1 Scene 1 rendered with 100 iterations, with image 5.1a as reference render .................. 39
5.2 Render artifacts in scene 1 for configurations 2 and 3 .................................................. 39
5.3 Render time for all iterations in scene 1 ........................................................................ 39
5.4 Scene 2 rendered with 100 iterations, with image 5.4a as reference render .................. 41
5.5 Render artifacts in scene 2 for configurations 2 and 3 .................................................. 41
5.6 Render time for all iterations in scene 2 ........................................................................ 41
5.7 Scene 3 rendered with 100 iterations, with image 5.7a as reference render .................. 43
5.8 Render artifacts in scene 3 for configuration 2, with 100 iterations
5.9 Render time for all iterations in scene 3
5.10 Scene 4 rendered with 100 iterations, with image 5.10a as reference render
5.11 Render artifacts in scene 4 for configurations 2 and 3
5.12 Render time for all iterations in scene 4
5.13 Performance overview for scene 1 (top) and scene 2 (bottom) kernels using configuration 1
5.14 Performance overview for scene 1 (top) and scene 2 (bottom) kernels using configuration 3
5.15 Occupancy for kernel used to reset iterations status, in scene 1
5.16 Occupancy for kernel used to reset camera connections, in scene 1
5.17 GPU occupancy for configuration 1 (top) and 3 (bottom)
5.18 Achieved occupancy for light path tracing kernel
5.19 GPU occupancy for kernel that connects light paths to camera
5.20 Achieved occupancy for kernel that connects light paths to camera
5.21 Occupancy for camera path tracing kernel, for scene 1 using kernel configuration 1
5.22 Occupancy for camera path tracing kernel, for scene 2 using kernel configuration 3
5.23 Occupancy for kernel used to get frame buffer data, in scene 1
5.24 Branch statistics for kernel used to reset iterations status, in scene 1
5.25 Branch statistics for kernel used to reset camera connections, in scene 1
5.26 Branch statistics for kernel used to trace light paths, in scene 1 (top) and scene 2 (bottom)
5.27 Branch statistics for kernel used to trace light paths, in scene 1 (top) and scene 2 (bottom),
    using kernel configuration 3
5.28 Branch statistics for kernel used to connect light paths to camera, for scene 1 (top) and
    scene 2 (bottom)
5.29 Branch statistics for camera path tracing kernel, for scene 1 (top) and scene 2 (bottom)
5.30 Branch statistics for kernel used to get frame buffer data, in scene 1
5.31 Instruction statistics for kernel used to reset iterations status, in scene 1
5.32 Instruction statistics for kernel used to reset camera connections, in scene 1
5.33 Instruction statistics for light path tracing kernel, for scene 1 (top) and scene 2 (bottom),
    using kernel configuration 1
5.34 Instruction statistics for light path tracing kernel, for scene 1 (top) and scene 2 (bottom),
    using kernel configuration 3
5.35 Instruction statistics for kernel used to connect light paths to camera, for scene 1 (top)
    and scene 2 (bottom)
5.36 Instruction statistics for kernel used to get frame buffer data, in scene 1
5.37 Instruction statistics for scene 1 (top) and scene 2 (bottom)
5.38 Issue efficiency for kernel used to reset iterations status, in scene 1
5.39 Issue efficiency for kernel used to reset camera connections, in scene 1
5.40 Issue efficiency for light path tracing in scene 1 (top) and scene 2 (bottom), using kernel
    configuration 1
Nomenclature

BDPT  Bidirectional path tracing, page 7
BSDF  Bidirectional scattering distribution function, page 6
CPU   Central Processing Unit, page 1
CUDA  Compute Unified Device, page 2
GPGPU General purpose graphics processing unit, page 12
GPU   Graphics Processing Unit, page 1
IPC   Instructions per clock cycle, page 55
LT    Light tracing, page 7
MIS   Multiple importance sampling, page 5
PM    Photon mapping, page 1
PPM   Progressive photon mapping, page 8
PT    Path tracing, page 1
RT    Ray tracing, page 5
SM    Streaming multiprocessor, page 13
SP    Streaming processor, page 13
VCM   Vertex connection and merging, page 1
Chapter 1

Introduction

Creating physically based renders has been an active subject of research since James Kajiya introduced the light transport equation, also known as rendering equation, in 1986 [1]. Until now, the most efficient way to generate these renders still takes a great amount of processing power, and is not normally done in real time. Physically based rendering takes into account physical equations, in order to simulate accurate lighting, where the law of conservation of energy is taken into account. Complex scenes typically contain multiple light sources and hundreds or thousands of objects, where the light bounces. The indirect light originated at different objects influences the final render result, together with the light that directly reaches the camera. All these properties make complex scenes very computationally intensive to be generated with enough accuracy and with reduced or no noise, and so, different rendering algorithms have appeared over the years. Some focus on simulating light coming from the light sources until the camera is intersected, as is the case of path tracing (PT), while others focus on simulating rays traversing the scene starting from the camera, until a light source is reached, being photon mapping (PM) one of these algorithms. For algorithms that simulate the light behavior, the resulting renders are often noisy if the number of sampled light paths is reduced. This means that additional paths must be sampled to have a better quality render. However, increasing the number of paths also increases linearly the render time. These can be reduced by combining the results of different algorithms, which is exactly what happens in vertex connection and merging (VCM), one of the state-of-the-art rendering algorithms presented in 2012. The simulated light photons, generated by photon mapping, are connected to the camera by using path tracing, and the renders can be processed faster, as less paths are traced, with the advantage of resulting in less noisy renders.

Along with the evolution of rendering algorithms, the processing capabilities of central processing units (CPU) and graphics processing units (GPU) have also been improved through the years, allowing much more data to be processed than before. This does not mean that render times have reduced proportionally, though. As computers allow more work to be done, the requirements from developers and artists also increase. As the number of operations that can be executed per second on a CPU improve at a much slower rate compared to a GPU, in recent years there has been a shift in the devices used to process some applications. Applications that have an inherently parallel architecture and require
a lot of processing power have been developed to be processed using GPUs, as these are specifically designed to process tasks in parallel. Since NVIDIA introduced CUDA in 2006 [2], and the Khronos Group introduced OpenCL in 2008 [3], the development of software to be processed on the GPU was made much easier, allowing developers to take advantage of the parallel processing capabilities of this hardware. CUDA enabled GPUs have hundreds or thousands of cores, depending on the architecture and model being used, and each of these cores executes one thread. The threads running in parallel execute the same functions. As the rendering algorithms consist mainly in repeating the same tasks for each image pixel generated, GPUs are perfectly suitable for processing them. Different types of memory are also part of the CUDA GPU hardware, offering different performance optimizations, but also a more responsible approach to choose the best memory type suited for the tasks to be executed. Slower access memory types are available to all cores, while faster memory types can be accessed within subsets of cores, and are also much more limited in size.

The goal of this thesis is to implement VCM using CUDA, in order to be processed on NVIDIA GPUs and take advantage of their parallelism capabilities. The source code basis for the algorithm implementation is SmallVCM, an application developed in C++ by the algorithm's author. The approach taken is to initially identify the source code portions that are executed multiple times during the rendering, being these the best candidates to be processed by CUDA cores. Light and camera path tracing are executed at least once per pixel, with the possibility to be executed as many times per pixel as the number of segments in a path. These portions of code are then isolated in functions, which are to be run on the GPU. To ensure that the CUDA implementation is generating similar renders, these are regularly compared with the results from SmallVCM. As some of the C++ functions available to the CPU are not available in CUDA libraries, these need to be updated accordingly. Differences in implementation or even in precision may cause differences in the render results, but should not be significant. The performance of the CUDA implementation is then compared with SmallVCM, both in terms of image quality and render time. The CUDA application is also profiled, to give a clear insight on how the algorithm performs in the GPU, and to provide a starting point for future work.

1.1 Thesis Outline

This thesis is organized into 6 chapters. In chapter 2, the relevant rendering algorithms for the implementation of VCM are discussed, followed by a detailed explanation of the state of the art VCM, where the development work of this thesis is focused on. The development is done on top of an implementation done by the algorithm's author, SmallVCM, which is introduced after the algorithm. NVIDIA CUDA and architecture of CUDA capable GPUs are also discussed in this chapter, finalizing with two previous implementations of VCM algorithm, using NVIDIA OptiX and Intel Embree. Chapter 3 focuses on discussing the VCM implementation using CUDA. This chapter starts with an initial proposal, and then continues with the concrete implementation. In the implementation section, different kernel configurations are discussed, and how these can improve the algorithm execution on the GPU. The evaluation methodology is the main subject of chapter 4, where the scenes and hardware used are described,
followed by how CUDA kernels performance and final render results are evaluated. For performance
results, the profiling metrics are introduced, and the image quality metrics are introduced for the image
comparison results. In chapter 5, the results are evaluated according to the methodology described in
the previous chapter. The first part of the evaluation is focused on image quality and rendering time. The
second part focuses on the performance of the GPU on instruction level and memory level, ending with
final considerations about the overall results. Chapter 6 covers the conclusion of the thesis, important
achievements, and future work that can be done to improve the CUDA implementation of VCM. At the
end of the document, there is an appendix with a comparison table between the two current versions
of NVIDIA OptiX, portions of CUDA source code implemented in this thesis, and additional rendering
results data.
Chapter 2

Background

2.1 Illumination

One of the most important elements of an image is illumination. Computer generated scenes have a concept of light, and its physical properties are simulated to achieve a very high degree of realism. Besides the properties of an object material, the light is also responsible for the final colors obtained in the scene. Just like in the real world, the light that illuminates an object may be in direct or indirect line of sight, being it direct or indirect illumination, respectively. In the case where an object is indirectly lit, this light is transmitted by other objects.

Different light transport equation implementations have been developed over the years, having each its own advantages and disadvantages. There is not a single rendering algorithm that can handle all the different aspects of a scene, so multiple algorithms can also be combined. There are also biased and unbiased rendering algorithms, where the unbiased calculate the complete rendering equation. The rendering algorithms that are described next to multiple importance sampling (MIS) section, with exception of the ray tracing (RT) algorithm, are unbiased, providing more physically accurate renders.

2.1.1 Multiple importance sampling

Before describing the rendering algorithms, it is important to note that samples resulting from different algorithms can be combined, in order to get the advantages of them in different locations in a scene. Multiple importance sampling was presented by Veach in 1995 [4], and allows the combination of these different sampling techniques by calculating the weight that each is going to have in the final result. This is done by calculating the variance of each of the techniques used, and giving it a weight in the final combination.

Without using MIS, when rendering a scene where multiple results get combined, there can be a high variance in the samples, and summing the different sampling results worsens the problem, leading to noise in the final render. Figure 2.1 shows the result of combining two rendering techniques using MIS.
2.1.2 Ray tracing

In the ray tracing algorithm, presented by Turner Whitted in 1980 [5], a ray is shot from the camera towards the screen, hitting then the objects in the scene. This was the first algorithm that recursively traced additional rays when the initial ray hit an object that was not diffuse. These new additional rays can have different types, being reflection, refraction or shadow rays. If the object is diffuse, the ray is cast from the intersection point towards the lights in the scene.

However, when additional rays are created, these may take a considerable amount of processing time before reaching any light. In order to avoid processing times that could be too long, each ray has a depth before the path traversal stops. This depth is the total amount of intersections a ray can have, and increasing the depth may also increase the processing time. On the other hand, decreasing the maximum depth can introduce bias in the final render.

Compared with previous algorithms that did not take into account the multiple properties of the objects’ materials, the resulting renders are much more realistic, besides being biased.

2.1.3 Path tracing

The path tracing algorithm, which was presented by Kajiya in 1986 [1], also renders an image by shooting a ray from the camera towards the screen, but the ray bounces from the objects until it reaches a light in the scene. When the ray intersects an object, the bidirectional scattering distribution function (BSDF) is taken into account. This means that the ray that intersects the object follows a direction based on the physical properties of the object, that simulate the surface irregularities of the real world, invisible to the naked eye.

The intersection between the ray and the light can have a very low probability, and just like in the RT algorithm, the depth parameter is also used to stop the ray bounces if no intersection with a light source is found. However, in this case, if the ray does not intersect a light source, it is discarded, resulting in a noisy image.
2.1.4 Light tracing

The light tracing (LT) algorithm, which was presented by Arvo in 1986 [6], uses an approach that is the inverse of the one used in PT. In LT, the ray is cast from the light source, bouncing in the objects according to their BSDF. This algorithm allows the rendering of caustics, but the specular reflections seen by the camera cannot be calculated.

Besides this, another disadvantage of this algorithm consists in the intersection of the rays with the camera, where only a few rays get to it. Due to this, the probability of existing noise in the resulting render is very high, and there is a lot of wasted processing time, as the rays that not appear in the final render get discarded, just like in PT.

2.1.5 Bidirectional path tracing

In 1998, Veach presented a new algorithm called bidirectional path tracing [7] (BDPT). This algorithm uses both PT and LT, and after a predefined number of iterations, the resulting paths of each algorithm are connected. Both the PT and LT paths are initiated at the same time.

These paths can then be connected by using a single ray between them, resulting in a single path, or using multiple rays connecting all the points from both paths, resulting in multiple paths. If there is an occlusion between two points, the resulting path will be discarded.

The main advantage of this approach is that it can handle small aperture cameras and small or hidden lights, which can be hard to reach using PT or LT. The resulting renders are much cleaner regarding noise.

2.1.6 Photon mapping

The photon mapping algorithm was presented in 1996 by Jensen [8], and like BDPT, allows the connection of different types of paths. However, in the case of PM, whenever the rays originated in the light source intersect an object, being these intersection points the photons, their properties are stored on a map.

After the photon tracing, PT is used. When a ray originated in the camera is close enough to a photon intersection point, these are merged. Note that multiple photons can be located in a given radius around the intersection point of the PT ray, being all of these taken into account in the resulting radiation value.

Unlike the previous rendering algorithms, the way to increase the render quality in PM is to increase the number of photons. Having a low number of photons blurs the resulting image.

The main advantage of PM is that light paths can be reused. This is because the photons are stored, which originated in the light sources, and the corresponding paths are already calculated. For complex materials, like glass, where the calculation of caustics is directly related with the amount of photons that traverse the object, this means that once the photons are stored, there is no need to recalculate the same paths again, when merging photons with the camera paths.

On the other hand, this also introduces the main disadvantage of the algorithm. In order to have better results, more photons must be stored in memory, increasing the total amount of memory required.
to render a scene.

### 2.1.7 Progressive photon mapping

Progressive photon mapping (PPM) is an algorithm presented by Hachisuka in 2008 [9], and improves the PM by reducing the amount of memory used. In this algorithm, the first pass is a regular RT pass, identifying all the visible objects from the camera. The subsequent passes are all photon tracing passes.

Whenever a photon is traced, the resulting intersection point is compared with the RT intersections. If the photon is close to one of the RT intersections, its contribution is added to the illumination value, and the properties of the photon intersection are not stored in memory. After each photon tracing pass, an image can be rendered. By adding more photon tracing passes, the image quality is progressively increased, resulting in a better quality render.

### 2.2 Vertex connection and merging

Vertex connection and merging was presented in 2012, by Georgiev [10]. This algorithm uses both BDPT and a reimplementation of PM as a path sampling technique, and allows the merging of both techniques. As the original implementation of the PM algorithm does not increase its render quality by increasing the amount of samples processed, VCM reformulation of PM allows new iterations to increase quality, not being dependent on the number of photons, but on the number of samples.

![Figure 2.2: Merging of BDPT and PM using an heuristic (highlighted in red and green), and VCM with MIS (Source: Georgiev et al. 2012 [10])](image)

To merge BDPT and PM, MIS is used. The render quality is then improved, giving all the benefits of the different algorithms. Small camera apertures and small lights can be rendered easily, while keeping the caustics generated by PM. Consequently, scenes with lots of different materials can now be rendered efficiently, without the noise that was generated with the previously used methods. Figure 2.2 shows the results of BDPT and PM being merged using VCM, eliminating most of the noise.

### 2.2.1 Light path tracing

The first stage of the algorithm is the tracing of light paths. Light paths have origin on each light source, and the resulting hit points, throughput and contribution for both connection and merging are stored. The
stored data is called a light vertex. These contributions are later used to weight the contribution to the final color. Additionally, this stored data ensures that light paths are reused, in a similar way to how PM photons are used.

These paths are divided into sub paths, and are limited in length, which also limits the number of intersections with objects for each path being traced from a light source. The end of each of these sub paths is the light vertex.

When the light vertex is not purely specular and the path length limit is reached, the vertex is connected to the camera, contributing to the final color.

For any given light path, the more sub paths calculated, the smaller the contribution is for subsequent light vertices in the connection and merging stages.

2.2.2 Camera path tracing

The second stage of the algorithm is the tracing of eye or camera paths. These paths are calculated like in BDPT algorithm. Similarly to light paths, each camera path also has a limited length, being also divided into sub paths. The end of each camera sub path is a camera or eye vertex.

Unlike the light vertices, camera vertices are not stored in memory. However, the color contribution for these paths is calculated according to different conditions. If a path does not intersect any geometry, and if there is an environment light, the color contribution for this path is based on this light color. On the other hand, if a path maximum length is reached without any of the camera vertices being close to a light vertex, the color contribution is then sampled from a random light. Finally, if a camera vertex is close enough to a light vertex, both are merged. Otherwise, both are connected.

In PM, the merging is done between the camera path, and all the photons that are within a defined radius. In VCM, each light vertex is assigned to a path index, and the corresponding camera vertex is also defined using the same path index. In other words, for each light path, there is a camera path, being both matched with a path index.

2.2.3 Vertex connection

The vertex connection stage takes place inside the camera path tracing. The path index is used to determine which light vertices should be connected to the camera vertex. This is done by getting the light vertices that match the current camera path. Then, every light vertex previously identified is connected to the current camera vertex, as long as the total length of the path is not greater than the maximum path length. The total length of the path takes into account the current camera path length, and the path length for each light vertex being connected. Finally, the camera vertex is also connected to a random light source. This stage is illustrated in figure 2.3.

2.2.4 Vertex merging

After the vertex connection stage, vertex merging takes place, which is illustrated in figure 2.3. Here, the light vertices are merged with camera vertices. Merging occurs when both vertices are within a specified
radius, the acceptance radius.

![Diagram](image)

**Figure 2.3:** Camera and light paths with different lengths, and merging of camera and light vertices within the acceptance radius (Source: Georgiev et al. 2012 [10])

When a scene is rendered using multiple iterations, if the vertices are merged using the same radius between iterations, the final render may contain errors by being blurred. In order to minimize these errors, the radius is constantly decreased, in subsequent iterations, needing then the vertices to be closer to be merged. Initially the radius has a value $r_1$, and in each iteration, the value is reduced proportionally to a constant $\alpha$, as proposed by the author in equation (2.1).

$$r_i = r_1 \sqrt{\alpha^{-i}}$$  \hfill (2.1)

### 2.2.5 Algorithm overview

As discussed previously, VCM runs in two different main stages, being these the light path sampling and the camera path tracing. During the camera path sampling stage, vertex connection and merging also take place. Finally, after each iteration, the acceptance radius is reduced. The pseudocode for the algorithm is listed in 2.1, based on the algorithm described by the author of VCM [11].

```plaintext
// Renders progressively by averaging the results from N independent iterations
1 function ProgressiveRender(r1, \alpha, N, image)
2 for i = 1 to N do
3     iterationImage = BlackImage()
4     \text{\ } r_i = r_1 \sqrt{\alpha^{-i}}
5     \text{\ } Render(r_i, iterationImage)
6     image = \frac{i-1}{i} image + \frac{1}{i} iterationImage
7 end for
8 end function

// Renders a single image with a given maximum vertex merging radius
9 function Render(r)
10 // Stage 1: Light path sampling
11 for i = 1 to pixelCount do
12     lightVertex = TraceLightRay(SamplePixel(i))
13     while lightVertex is valid do
14         if lightVertex is not specular
15             StoreLightVertex(lightVertex)
16             ConnectToCamera(lightVertex)
17         end if
18 end while
19 end for

/ / Stage 2: Camera path tracing
20 for i = 1 to pixelCount do
21     cameraVertex = TraceCameraRay(SamplePixel(i))
22     while cameraVertex is valid do
23         if cameraVertex is not specular
24             StoreCameraVertex(cameraVertex)
25             ConnectToCamera(cameraVertex)
26         end if
27 end while
28 end for
```

10
During the light path tracing, the first main stage introduced in 2.2.1, light paths are created starting from light sources. When the emitted ray intersects an object in the scene, the light vertex information is stored, and a new ray has a probability of being emitted from the intersection point. This process continues until the path reaches the maximum length, or until an intersection in the scene is not found. When each intersection occurs, the color contribution of the light vertices is also stored in the frame buffer, by connecting these vertices to the camera.

Before tracing the camera paths, which is the second main stage, a range search data structure is created, which is later used to detect the light vertices that are closer to the camera vertices, in order to perform both vertex connection and merging operations. This detection is done using the radius that is defined for the current render iteration.

The camera path tracing stage, introduced in 2.2.2, is then run for every pixel of the image. It starts by emitting a ray from the camera position, and detecting intersections, generating camera vertices. If there is an intersection with a purely emissive material, a light source, the color contribution is added to the frame buffer. All camera vertices are connected to all the light vertices assigned to the current path, connected to a random light, and if there are any light vertices inside the range defined by the
iteration radius, these are also merged with the camera vertex. These contributions are then added to the frame buffer. The connection and merging stages are introduced in 2.2.3 and 2.2.4, respectively. After a camera vertex is created, a new sub path is created based on russian roulette probability, or if the path has not reached the maximum length yet.

2.2.6 SmallVCM

VCM has an experimental implementation, SmallVCM, created by its author [12]. Here, the different techniques used to render an image using the VCM algorithm are implemented and executed solely on the CPU, with multi threading support. When rendering an image, multiple renderers of the VCM algorithm are created and processed in parallel, using OpenMP. The number of renderers used is equal to the number of processors. If the system is using a CPU with a number of virtual cores greater than the number of physical cores, the former is used.

Each renderer has differences in the MIS weight calculation, related with the acceptance radius. After each iteration, the results from each renderer are stored in different buffers and then combined, to get the final render. Each of these renderers is, in fact, rendering individual iterations.

The implementation of VCM follows the approach described in the algorithm pseudocode listed in listing 2.1. Initially, light paths are traced, followed by camera paths. Light and camera vertices are then connected and merged in the camera path tracing stage, using a range search data structure, which is calculated between the light and camera path tracing stages.

SmallVCM has other rendering algorithms implemented, for the purpose of comparing the results of each of these. However, for the scope of this thesis, only the VCM algorithm is considered.

2.3 Parallel processing using GPU

Due to the nature of the GPU hardware, which contains more transistors to process data, a high number of tasks can run in parallel. When dealing with graphics, these are processed using vertex or fragment shaders, among others. Figure 2.4 compares how cores are organized between a CPU and a GPU.

Before the appearance of NVIDIA CUDA in 2006, there was an era where GPUs started being used to process complex science and engineering problems [14]. However, in order to be able to access the hardware resources, developers needed to create very complex shaders and data, in order to process the latter as textures or as 3D objects. This era coincided with the appearance of DirectX 9. When GPUs were being used as general purpose hardware, they were called general purpose GPU (GPGPU).

2.3.1 NVIDIA CUDA

In 2006, NVIDIA introduced a new revolutionary programming model, CUDA [2], which allows the development of software to be processed in the GPU, without having to write custom shaders. NVIDIA introduced a set of extensions in the C language, allowing the developers to write code to be executed in the GPU without having to create the previous complex shaders and data. To make it easier to get into
CUDA development, NVIDIA provides a programming guide [13] and a best practices guide [15], which were used as reference to the CUDA description in this section.

This programming model was introduced in the G80 series, and since this series the GPU come with a number of CUDA cores.

The performance of a CPU, compared to a GPU, is shown in figure 2.5.

### GPU Processors

The hardware present in a CUDA enabled GPU has some important differences when compared to GPUs from previous generations, when CUDA was not available. These GPUs contain processing cores denominated by streaming multiprocessors (SM), which also contain a number of streaming processors (SP). SP are also denominated as CUDA cores.

Depending on the compute capability of each GPU, which is directly related with improvements in the hardware, the number of SP per SM is different. In the latest compute capability, which is currently 8.0, the number of SP is organized according to table 2.1.

To this date, the GPU that offers the highest performance is the Tesla V100S, with 8.2 TFLOPS of...
Double precision and 16.4 TFLOPS of single precision. This GPU has 5120 SP [16].

2.3.2 GPU threads

Despite the number of SP typically increasing with each GPU generation, the SM process threads in sets of 32, called warps. These are especially important when a warp is accessing global memory, which is a slow process, as a new warp starts its execution, reducing the time that the hardware is waiting for data to process.

When a CUDA kernel, which is a function executed in the GPU, is executed, the number of threads needs to be provided besides the kernel arguments. This number of threads corresponds to the number of times that the kernel code is going to be executed, and is a variable number. The number of threads to execute a kernel is also organized in blocks of threads, and grids of blocks. These are defined by using a 3 component vector, making it easier to associate 1, 2 or 3 dimensional data arrays. Figure 2.6 shows the hierarchy of threads in a GPU.

In the latest compute capability, a block of threads can have up to 1024 threads, and all threads in the same block are executed in the same SM. The number of blocks in a grid is also limited, but this limit is much higher than the number of threads per block.

Each thread and block are automatically assigned an ID, being the block ID given by blockIdx.x, blockIdx.y or blockIdx.z, and the thread ID given by threadIdx.x, threadIdx.y or threadIdx.z.

```cpp
1 dim3 blocks(x, y, z); // dimension of grid of blocks
2 dim3 threads(x, y, z); // dimension of blocks of threads
3 sampleKernelCall<<<blocks, threads>>>(parameters);
```

Listing 2.2: Definition of grid and blocks, and execution of a kernel
The definition of the grid and block dimension, and a call to a kernel are listed in 2.2.

### 2.3.3 GPU Memory

Unlike the host system RAM, a GPU has different types of memory available, which can be used when executing a CUDA application, and performance optimizations must be done in order to prevent performance penalties. There are two different groups of memory in a GPU, being one the device memory, where all data is accessible to all SM, and the SM memory itself, only accessible to each SM and corresponding threads. Global, local, constant and texture memory types are located in the device memory (off-chip memory), while the shared memory and registers are located on each SM (on-chip memory). Figure 2.7 illustrates the memory layout inside a CUDA GPU.
Table 2.2: Performance Improvements Optimizing $C = A A^T$ Matrix Multiplication (Source: NVIDIA 2017-2 [15])

The global memory is the amount of RAM available in a GPU which is normally listed in its specifications. This is a slow access memory, available to all SM and to the host system. The data transferred between the host system to the GPU is stored in this memory, and one of the easiest optimizations that should be done consists in reducing the amount of transfers between the host RAM and the GPU global memory. Otherwise, using a GPU can be even more inefficient than using a CPU, if there is a high number of memory transactions between host and GPU.

Local memory is also a slow access memory, that is available on a per thread basis. This memory is used to store automatic variables that would consume too much registers.

Constant memory is a cached memory, but accessing different addresses on this memory by threads from the same warp is done in a serialized way. This means that if all the threads access the same memory address, only one read is performed. This access can be, in this case, as fast as accessing the registers. However, if N threads read N different addresses, N different reads must be performed.

Texture memory is a memory optimized for 2D spatial locality, meaning that threads in the same warp achieve a better performance the closer accessed memory addresses are.

Shared memory is available only to the SM, and all the threads can access it. This is a fast access memory, divided into banks, allowing threads inside a warp to share data without using the much slower global memory. Shared memory can also be used to store data from the global memory if several accesses to the same address in global memory need to be performed. Despite these advantages, if different threads access the same memory bank, these are serialized, creating a bank conflict. Table 2.2 shows how bandwidth in a GPU is impacted when different shared memory access patterns are used.

Registers are also available to the SM, on a thread level. All the registers in a SM are available for each thread. However, if the number of registers needed is greater than the total number of registers available, global memory is used, reducing the performance of the application.

### 2.4 NVIDIA OptiX

NVIDIA created a CUDA general purpose ray tracing framework called OptiX in 2009, which allows the creation of ray tracing based applications. By being programmable and being general purpose, it can be used to implement rendering algorithms, collision detection, physical simulations, among others. The main advantage of using OptiX consists in not having to create all the CUDA code, as there are already fixed function parts of the framework implemented.

Currently, there are two different OptiX APIs maintained by NVIDIA. OptiX 6 is the classical API, and
the newest API, which supports RTX GPUs, is OptiX 7. The overview of both APIs is illustrated in figures 2.8 and 2.9.

![OptiX 6.5 overview](image1)

![OptiX 7.1 overview](image2)

Figure 2.8: OptiX 6.5 overview, where the yellow boxes are programmable (Source: NVIDIA 2020 [17])

Figure 2.9: OptiX 7.1 overview, where the gray boxes are programmable (Source: NVIDIA 2020 [18])

OptiX 7 allows more control of a rendering application compared to OptiX 6, resulting in better performance of the applications. A comparison table with the main differences can be found in appendix A.1.

### 2.4.1 VCM implementation with OptiX

VCM was implemented using OptiX, by Bico in 2016 [19], and its implementation performance was compared with SmallVCM.

Based on the results achieved in table 2.3, the OptiX VCM implementation was consistently faster than SmallVCM, while maintaining the render quality.

According to Bico, the OptiX implementation was not optimized to its fullest, and due to the framework already having fixed functions, there is some code flexibility that is lost by not having access to all the
CUDA capabilities, which can also impact the application performance. On the other hand, it is easier and faster to implement than a CUDA application.

<table>
<thead>
<tr>
<th>Scene</th>
<th>Implementation</th>
<th>Runtime (s)</th>
<th>Iterations/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scene 1</td>
<td>SmallVCM</td>
<td>16.66</td>
<td>2.4009</td>
</tr>
<tr>
<td></td>
<td>OptiX</td>
<td>13.0336</td>
<td>3.37589</td>
</tr>
<tr>
<td>Scene 2</td>
<td>SmallVCM</td>
<td>13.28</td>
<td>3.0120</td>
</tr>
<tr>
<td></td>
<td>OptiX</td>
<td>13.2611</td>
<td>3.6196</td>
</tr>
<tr>
<td>Scene 3</td>
<td>SmallVCM</td>
<td>15.56</td>
<td>1.8638</td>
</tr>
<tr>
<td></td>
<td>OptiX</td>
<td>13.2185</td>
<td>2.7234</td>
</tr>
</tbody>
</table>

Table 2.3: Performance comparison between SmallVCM and OptiX VCM implementations (Source: Bico 2016 [19])

2.5 Intel Embree

Another popular ray tracing framework is Intel’s Embree [20], which consists on a collection of high-performance ray tracing implementations, optimized for Intel CPU and for photo-realistic rendering. This framework is used on the popular industry-level rendering engine Corona. The main advantage of using a CPU implementation is that the performance reduction that occurs when transferring data between host and GPU memories does not occur.

2.5.1 VCM implementation with Intel Embree

Bico implemented, also in 2016 [19], this algorithm using the Embree framework, in order to compare its performance against SmallVCM implementation.

<table>
<thead>
<tr>
<th>Scene</th>
<th>Implementation</th>
<th>Runtime (s)</th>
<th>Iterations/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scene 1</td>
<td>SmallVCM</td>
<td>16.66</td>
<td>2.4009</td>
</tr>
<tr>
<td></td>
<td>Embree</td>
<td>13.1404</td>
<td>3.04404</td>
</tr>
<tr>
<td>Scene 2</td>
<td>SmallVCM</td>
<td>13.28</td>
<td>3.0120</td>
</tr>
<tr>
<td></td>
<td>Embree</td>
<td>13.4155</td>
<td>2.9816</td>
</tr>
<tr>
<td>Scene 3</td>
<td>SmallVCM</td>
<td>15.56</td>
<td>1.8638</td>
</tr>
<tr>
<td></td>
<td>Embree</td>
<td>13.5643</td>
<td>2.1397</td>
</tr>
</tbody>
</table>

Table 2.4: Performance comparison between SmallVCM and Embree implementation (Source: Bico 2016 [19])

Based on the results achieved in table 2.4, the Embree implementation performed with processing times very similar to the OptiX implementation. As the data does not need to be transferred between memory types, and being this framework highly optimized to work with photo-realistic rendering algorithms, the CPU achieved very good rendering times.
Chapter 3

VCM Implementation with CUDA

This chapter describes how CUDA is used to implement the VCM algorithm. It starts with the initial proposal, on how both light and camera path tracing algorithms were initially proposed to be implemented, and then follows the description of the concrete implementation.

3.1 Solution proposal

Due to the inherent parallel implementation that rendering algorithms allow, CUDA is perfectly suitable to handle this kind of applications. Unlike OptiX framework, CUDA gives access to all the low level features and capabilities of the GPU, at the cost of the developer need to manage all the code. The implementation done in the scope of this thesis is based on SmallVCM, introduced in section 2.2.6, which will be modified to be processed by a CUDA GPU.

CUDA VCM implementation focuses on the conversion of the main rendering stages, light path tracing and camera path tracing. Here, the same operations are executed for every path, and multiple times per path, as each can have multiple sub paths.

Due to the way how the GPU processes threads, organized in blocks, the proposed approach is to maximize the amount of paths processed, by assigning each of these to a thread. Depending on the number of threads needed, the number of blocks and threads per block can be adjusted, in order to reduce the idle resources.

To minimize performance penalties due to memory transactions, all the data to be processed is initially sent from the host memory to the GPU memory, and the final render is only retrieved after all the data is processed. Figure 3.1 shows the CUDA VCM execution overview, and a more detailed analysis of this diagram is done in section 3.2.3.

3.1.1 Light path tracing

The first stage of the implementation processes the light paths for every pixel in the image. As each light path has a maximum length, the number of path segments each thread processes for each pixel are the same as this length. For example, if there are $W \times H$ pixels to be processed, with a maximum length for
each path equals to $N$, and all the paths are calculated until they reach the maximum length, the kernel that processes a path is executed $N$ times by $W \times H$ threads.

One downside of having all the threads executing the same code happens when, for example, only a single thread needs to process all the path segments. In this case, all the other threads need to wait for the busy thread before moving to the next operations.

Instead of having a condition inside the light tracing kernel that launches the vertex connection, the information about if a new segment needs to be calculated or not should be stored in a temporary array, besides the information related with each light vertex. This allows the separation of the tracing and the calculation of the color contribution for each light vertex, speeding the process of rendering by reducing the waiting time due to busy threads while others are idle.

### 3.1.2 Camera path tracing

The second stage can also be split like the previous one. Instead of having the conditions that allow the color contribution and vertex connection and merging to be processed, the camera paths are first calculated, storing the information about if the path should continue to be processed or should be stopped, as well as the information of the camera vertex. After all the camera vertices are processed, the merging and connecting can be processed to the vertices that are needed. Once again, separating these processes can increase the performance, by reducing the amount of conditional code inside a kernel.

### 3.1.3 Range search data structure

Before the camera path tracing stage, a range search data structure is created, in order to identify the light vertices that are close to each camera vertex when vertices are being merging. This data structure, identified as hash grid, is calculated using the exclusive prefix sum algorithm, which consists
in adding all elements up to any given element in the input array. This algorithm implementation is trivial for an application running on a CPU, but has some complexity when implemented efficiently for a CUDA application. The calculation of this data structure is part of the proposed solution, and should be performed by the GPU. Unfortunately, its implementation was not successful during the course of this thesis, with only the code being refactored for an easier conversion to CUDA, and the implementation left for future work.

3.1.4 Frame buffer

The frame buffer contains the color data for the iteration being processed. If multiple iterations are to be processed, which a single iteration at a time, a temporary frame buffer array contains the data for the current iteration. This is illustrated in figure 3.2.

\[
\begin{array}{cccc}
(1, 1) & ... & (w, h) & 1 \\
0 & ... & n - 1
\end{array}
\]

Figure 3.2: Frame buffer with color data for a single iteration

In case of processing multiple iterations at once using the GPU, the temporary frame buffer array contains the color data for all the iterations being processed at that given time. At the end of the rendering stage, frame buffer data for all iterations is then combined, resulting in the final image color data. This is illustrated in figure 3.3.

\[
\begin{array}{cccccccccccc}
(1, 1) & ... & (w, h) & 1 & ... & (1, 1) & ... & (w, h) & 1 \\
0 & ... & n - 1 & ... & i \times (n - 1) & ... & i \times n - 1
\end{array}
\]

Figure 3.3: Frame buffer with color data for multiple iterations

If there were iterations processed previously, the already existing frame buffer data is combined with the data from the newly processed iterations.

Each thread updates the frame buffer data for each pixel.

3.2 Solution implementation

In order to implement VCM using CUDA, SmallVCM is used as the code base, and changes are made on top of it, as mentioned previously. The development is done in hardware from older generations,
and due to this, drivers for CUDA, Visual Studio and the debugging tools used are not the most recent ones, for compatibility reasons. The machine used for development is a laptop with a NVIDIA GeForce GTX960M GPU, with 2GB of GDDR5, and a Intel Core i7-6700HQ CPU, with a base clock frequency of 2.6GHz and 16GB of DDR3 RAM. There is also an integrated Intel GPU, which allows the NVIDIA GPU to be used solely for development, including profiling and debugging, without freezing the whole system due to a busy GPU. The drivers and software used are listed in table 3.1.

<table>
<thead>
<tr>
<th>NVIDIA CUDA drivers</th>
<th>NVIDIA NSight Monitor</th>
<th>Visual Studio</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.0.60</td>
<td>5.2.0.16321</td>
<td>2015</td>
</tr>
</tbody>
</table>

Table 3.1: Software and drivers used in development environment

### 3.2.1 CUDA threads

Each iteration processes all pixels of an image. As the number of threads is the same number of pixels, the total number of threads is then $W \times H$ threads. For each pixel, each thread then processes all paths. The paths cannot be processed in parallel, as the values of subsequent paths depend on previous paths. For example, the Multiple Importance Sample (MIS) values from previous paths are divided by the BSDF values of the current path, not being possible to calculate these independently.

A CUDA GPU has the best performance when all threads are busy. As warps can have up to 32 threads active at the same time, the number of threads per block should be a multiple of 32.

Blocks are set to have a maximum of 256 threads. This ensures that there are enough warps ready to be started when others are idle, when data is being read from memory. With this number of threads, there are 8 warps per block. The number of threads per block is calculated according to expressions 3.1 and 3.2.

$$\text{threadsPerIteration} = w \times h$$

$$\text{totalRequiredThreads} = \text{threadsPerIteration} \times \text{iterations};$$

If the number of threads in 3.2 is greater than 256, this is the value that is used instead. For any square image with width and height greater than 16, the number of threads per block is always 256.

Finally, grids are set to have a number of blocks where the number of blocks multiplied by the threads per block match the number of pixels to process. The number of blocks per thread is calculated according to expression 3.3.

$$\text{totalRequiredBlocks} = \text{totalRequiredThreads}/\text{maxThreadsPerBlock};$$

Having a number of threads per block smaller than 32 will create additional warps. This results in more instructions being executed, and, consequently, decreased performance, for a given kernel
with a number $I$ of instructions, considering it does not have any divergence. Table 3.2 compares the instructions required for a kernel where each thread executes 10 instructions, without any divergence.

<table>
<thead>
<tr>
<th>Blocks</th>
<th>Threads per block</th>
<th>Warps</th>
<th>Total instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>256</td>
<td>8</td>
<td>80</td>
</tr>
<tr>
<td>8</td>
<td>32</td>
<td>8</td>
<td>80</td>
</tr>
<tr>
<td>32</td>
<td>8</td>
<td>32</td>
<td>320</td>
</tr>
<tr>
<td>64</td>
<td>4</td>
<td>64</td>
<td>640</td>
</tr>
</tbody>
</table>

Table 3.2: Instructions for different warps sizes

### 3.2.2 CUDA memory

During kernel execution, the objects that are accessed on every frame are copied from global memory to shared memory. Each SM has its own copy of the light objects, in order to reduce memory access collisions caused by threads from different SM. The scene object is also stored in the shared memory.

Having the objects copied to the shared memory is not, however, a guaranteed performance improvement. If an object in the global memory is accessed only once in the whole kernel, there is the additional cost of copying the object from the global memory, storing it, and then accessing it again from shared memory. This will likely result in slightly degraded performance. Besides the limited amount of shared memory available for each SM, the factor mentioned before is equally important for a good memory management and memory access optimization.

The frame buffer is only stored in the global memory, as each pixel that is being processed can have the corresponding color contribution assigned to a completely different pixel in the resulting frame buffer. However, this is the slowest memory in the GPU, and in the worst case, a thread stores a contribution per calculated path.

### 3.2.3 CUDA kernels

Processing intensive tasks, which can also be implemented in parallel, are implemented using different kernels. This means that both vertex connection and vertex merging are fully implemented using separate kernels.

**Kernel configurations**

Before discussing the implementation of the different kernels, it is important to clarify that there are three different kernel configurations, according to type of memory used, and when light vertices are connected to the camera. The main configuration, also identified as configuration 1, uses only global memory, not considering cache transactions, for all memory transactions that take place during the application lifetime. In configuration 2, the light vertices connection to camera is done in a dedicated kernel, after the light path tracing kernel is executed. Finally, configuration 3 adds shared memory to configuration 2, to store some static data that is read often, with lights as an example. Table 3.3 summarizes the
configurations used. Whenever relevant, in the subsequent sections and chapters, the configuration number is used. If no configuration is mentioned, configuration 1 is assumed by default.

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Global memory</th>
<th>Shared memory</th>
<th>Light vertex camera connection deferred</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>2</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>3</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Table 3.3: Kernel configurations according to memory and kernels used

Kernels implemented

In the next paragraph, the kernels are listed and matched with the diagram in figure 3.1 according to their colors. Kernels highlighted with dark green are used by all kernel configurations, and the ones highlighted with blue are used by configurations 2 and 3. The remaining colors are not used outside the diagram, but for reference, light green highlights the vertex connection and merging sub stages that are processed in the camera path tracing kernel, and the code executed on the CPU is highlighted with light orange.

As mentioned previously, the two main kernels that are implemented are GenerateLightPaths and GenerateCameraPaths. As the name implies, these are used to trace light and camera paths, respectively. Two additional kernels are used, ResetIterations and GetFrameBufferData, to set the GPU global memory to a valid state on each iteration, and to get the final frame buffer data, when all iterations are finished, respectively. Two different configurations of the kernels allow postponing the camera connection in the light path tracing kernel and the use of shared memory. For these, a kernel to connect the light vertices to the camera, ConnectLightPathToCamera, and a kernel to reset the information if a light vertex should be connected to the camera, ResetConnections are also used.

Kernel setup

Before running a kernel, all the required arrays are initialized in the host. Data that is required to run the kernels is also initialized and copied at this stage. For example, all the scene data is copied from the host to the GPU. This includes geometry, light and material data. If a kernel that runs multiple iterations at once is initialized, the number of iterations is also passed to the kernel as parameter.

Resources optimization

In order to optimize operations in the GPU, all the memory allocations are done before starting processing any iteration. Data that is static between iterations is also copied at this stage, and kept in the global memory between iterations. Consequently, all this allocated memory is freed when all iterations are finished and frame buffer data is copied to the host memory.
3.2.4 Light path tracing

The light path tracing stage of the algorithm, introduced in section 2.2.1, has an overall implementation very similar to the implementation in SmallVCM. The main portion of SmallVCM code that is changed in the CUDA implementation is listed in listing 3.1.

```
const int pathCount = resX * resY;

// Outer loop, which iterates over every pixel.
for (int pathIdx = 0; pathIdx < pathCount; pathIdx++)
{
    SubPathState lightState;
    GenerateLightSample(lightState);

    for (;; ++lightState.mPathLength)
    {
        (...)
    }
}
```

Listing 3.1: Light path tracing in SmallVCM

Previously, in table 3.3, three different kernel configurations were introduced. Light path tracing kernel is implemented using all these, in order to find the most efficient. One minor difference from the original SmallVCM implementation, is that each pixel has its own random number generator, where the seed is equal to the thread ID. This ensures the generation of different paths by different threads and different iterations.

Configuration 1 - replace outer loop with one thread per pixel

The first configuration consists in just replacing the outer most loop, which iterates over every pixel, in order to calculate the corresponding light paths. The outer loop is listed in 3.1.

The path index is returned using the current block of threads and thread IDs running in the GPU. The resulting code implemented in the kernel is listed in 3.2.

```
int pathIdx = blockIdx.x * blockDim.x + threadIdx.x;
SubPathState lightState;

for (;; ++lightState.mPathLength)
{
    (...)
}
```

Listing 3.2: Light paths with outer loop replaced
**Configuration 2 - postponing camera connection**

Postponing the camera connection operation, available in configuration 2, is part of the proposed solution. In this configuration, light vertices are not connected to the camera until all threads are finished in the light tracing stage.

In order to be able to postpone the camera connection, all the data required must be stored in different arrays, in the global memory. This is to enable passing data between kernels, as the camera connection is processed in a different kernel. The data required is a boolean to indicate if the path is to be connected to the camera, the path state, hit point and BSDF.

In the light path tracing kernel, the temporary data is directly stored in global memory, instead of using the shared memory. As each thread is responsible for storing the data for all the sub paths calculated for each pixel, using shared memory would only result in additional memory allocations and transactions, reducing the performance of the kernel.

The code used to store all temporary objects is listed in 3.3.

```plaintext
1 (...)
2 // Connect to camera, unless BSDF is purely specular
3 if (!bsdf.IsDelta())
4 {
5     if (lightState.mPathLength + 1 >= iterationConfiguration.minPathLength)
6         {
7             connectPath[subPathIndex] = true;
8             pathState[subPathIndex] = lightState;
9             pathHitPoint[subPathIndex] = hitPoint;
10             pathBsdf[subPathIndex] = bsdf;
11         }
12     else
13     {
14         connectPath[subPathIndex] = false;
15     }
16 }
17 (...)
```

Listing 3.3: Light paths with temporary arrays to postpone camera connection

The kernel used to connect the light path with the camera uses a number of threads equal to the total number of sub paths for every pixel. If multiple iterations are being calculated at once, the number of iterations is also included. The total number of sub paths is then given by equation 3.4.

\[
\text{totalSubPaths} = w \times h \times \text{pathLength} \times \text{iterations};
\]  (3.4)

Using this approach results in a significant increase in GPU memory usage. This limits the maximum resolution that a render can have, as well as the iterations that can be processed on each kernel execution.

For reference, the total required memory, in bytes, for a single sub path is 142 bytes. The objects are divided according to table 3.4.
### Table 3.4: Memory requirements for postponing camera connection on each light sub path

<table>
<thead>
<tr>
<th>Component</th>
<th>Data type</th>
<th>Size (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>BSDF</td>
<td>LightBSDF</td>
<td>77</td>
</tr>
<tr>
<td>Connect path</td>
<td>boolean</td>
<td>1</td>
</tr>
<tr>
<td>Hit point</td>
<td>Vec3f</td>
<td>12</td>
</tr>
<tr>
<td>Path state</td>
<td>SubPathState</td>
<td>52</td>
</tr>
</tbody>
</table>

Configuration 3 - use shared memory for frequently accessed data

The third configuration uses shared memory to store the lights and part of the scene data, on top of configuration 2.

The lights are part of the scene, in the original SmallVCM implementation, and in this configuration, the lights are copied to separate arrays in shared memory. However, the geometry data, which is also part of the scene, is not copied to the shared memory.

A separate function is implemented to copy the lights from global to shared memory. This results in a simplification of the kernel code, and in a reusable function for the camera path processing stage. The function source code is listed in appendix B.1.

#### 3.2.5 Camera path tracing

Following the same approach as in light path tracing stage, the camera path tracing stage of the algorithm, introduced in section 2.2.2, has an overall implementation very similar to the implementation in SmallVCM. It is at this stage that vertex connection and merging operations, introduced in sections 2.2.3 and 2.2.4 respectively, take place. This stage is implemented using two different configurations, 1 and 3. A random number generator is also used per thread, to ensure the generation of different paths by different threads. The main portion of camera path tracing code in SmallVCM relevant for this section is listed in 3.4.

```c
1 (...
2 const int pathCount = resX * resY;
3 (...
4 // Outer loop, which iterates over every pixel.
5 for (int pathIdx = 0; pathIdx < pathCount; pathIdx++)
6 {
7     SubPathState cameraState;
8     const Vec2f screenSample = GenerateCameraSample(pathIdx, cameraState);
9     Vec3f color(0);
10    for(;; ++cameraState.mPathLength)
11        {
12            (...
13        }
14    }
15 (...
16 }
17 (...
```

Listing 3.4: Camera paths in SmallVCM
Configuration 1 - replace outer loop with one thread per pixel

Similarly to light paths stage, the first configuration consists in just replacing the outer most loop. This loop iterates over every pixel, in order to calculate the corresponding camera paths. The outer loop is listed in code listing 3.4.

As the loop is the same, the path index is returned using the current block of threads and thread IDs running in the GPU. The resulting code is listed in 3.5.

```cpp
(...) int pathIdx = blockIdx.x * blockDim.x + threadIdx.x;
(...) SubPathState cameraState;
const Vec2f screenSample = cudaGenerateCameraSample(&sharedScene, &rng, pathIndexBounded, cameraState, iterationConfiguration.lightSubPathCount);
Vec3f color(0);
for (;; ++cameraState.mPathLength)
{
   (...
}
```

Listing 3.5: Camera paths with outer loop replaced

Configuration 3 - use shared memory for frequently accessed data

In camera path calculation, for configuration 3, shared memory is again used to store the lights and the scene data. The data stored in shared memory is the same previously mentioned in light path tracing kernel, and light data is copied to shared memory by reusing the same function used in this kernel.

3.2.6 Hash grid

Between the light and camera path tracing stages, the hash grid is created based on the light paths data. The calculation of this data structure is performed by the CPU. Implementing the code similarly to the CPU implementation results in a very slow calculation, slowing down the whole rendering of the scene. Due to this reason, the light vertices data required for calculations is copied from GPU to host memory.

After doing the hash grid calculation, the result is then copied to the GPU, to be used in the vertex merging stage.

The GPU memory required to store the hash grid is allocated before processing any iteration, and there are as many hash grids stored in memory at any given time as the number of iterations being processed. The total amount of memory required for an hash grid, per iteration, is described in table 3.5.

3.2.7 Frame buffer

Color data for the final render is stored in the frame buffer. To improve light and camera path tracing kernel performance, each iteration has its own frame buffer. This ensures that when color data is being
<table>
<thead>
<tr>
<th>Component</th>
<th>Data type</th>
<th>Size (bytes)</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cell ends</td>
<td>int</td>
<td>4</td>
<td>$w \times h$</td>
</tr>
<tr>
<td>Indices</td>
<td>int</td>
<td>4</td>
<td>$w \times h \times \text{verticesPerPath}$</td>
</tr>
<tr>
<td>BBox min</td>
<td>Vec3f</td>
<td>12</td>
<td>1</td>
</tr>
<tr>
<td>BBox max</td>
<td>Vec3f</td>
<td>12</td>
<td>1</td>
</tr>
<tr>
<td>Radius</td>
<td>float</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>Radius squared</td>
<td>float</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>Cell size</td>
<td>float</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>Inverse cell size</td>
<td>float</td>
<td>4</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 3.5: Memory requirements for hash grid, for a single iteration

written, the number of collisions is reduced as much as possible. However, collisions can still occur for a given iteration, as different paths can write color data to the same location.

The temporary color data is copied to the main frame buffer after calculating the contributions for all light and camera vertices, and after vertex connection and merging is performed.

Having the temporary frame buffer arrays increases the memory requirements for each iteration.

Considering that each pixel in the final image is stored in a $\text{Vec3f}$ variable, and the size of it is 12 bytes, the memory required for the temporary frame buffer for each iteration is given by equation 3.5.

$$\text{frameBufferSize} = m\text{Res}X \times m\text{Res}Y \times \text{iterations} \times 12 \tag{3.5}$$

This approach does not eliminate the collisions when writing to the main frame buffer, as paths from different iterations may have color contributions for the same pixels. The copy from temporary to main frame buffer is illustrated in figure 3.4.

![Figure 3.4: Color data copy from temporary to main frame buffer](image)

### 3.2.8 Iteration reset

In order to have an iteration in a clean state, it is enough to reset the iteration frame buffer and the array with the path lengths for the light path tracing stage. All the other data is generated per iteration, without any intermediate memory that requires being reset. The light vertices do not require to be reset, due to only the light vertices that are within the path length being considered. The frame buffer is required to be reset, due to the temporary frame buffer that is used when processing an iteration.
3.2.9 Kernel configurations summary

CUDA VCM can be executed using 3 different configurations, where each configuration changes the types of memory used, and postpones the connection of light path vertices to camera. These configurations are discussed in section 3.2.3, and summarized in table 3.3. The next chapters reference these configurations whenever kernels are being discussed.
Chapter 4

Evaluation methodology

When implementing a new solution or modifying an existing one, the results need to be evaluated, to check if it conforms to the specifications and if the objectives are achieved. In this case, the proposed solution consists in optimizing performance, while keeping the resulting images as close as possible to the original implementation. This means that the verification and validation is done by comparing both hardware performance, and generated images.

4.1 Scenes

In order to compare SmallVCM with CUDA VCM, we need to make sure that the conditions are similar. This means that the scenes to be tested, including the light conditions, must be the same. Otherwise, the render results are different, invalidating a valid comparison. In this case, the scenes are hard coded in the SmallVCM implementation, and the same scenes are used in CUDA VCM. Four different variations of the Cornell Box are used, with different objects and corresponding materials, and illumination types. These are described in 4.1.

<table>
<thead>
<tr>
<th>Scene</th>
<th>Area light</th>
<th>Background light</th>
<th>Directional light</th>
<th>Point light</th>
<th>Mirror</th>
<th>Glass</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scene 1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Large</td>
<td>-</td>
</tr>
<tr>
<td>Scene 2</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>Small</td>
<td>Small</td>
</tr>
<tr>
<td>Scene 3</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>Small</td>
<td>Small</td>
</tr>
<tr>
<td>Scene 4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>Small</td>
<td>Small</td>
</tr>
</tbody>
</table>

Table 4.1: Scenes used for implementation evaluation

The objects in all scenes are spheres, which can be small or large. The materials used are a mirror material, with reflectance equal to 1, and a glass material, which is both reflective and refractive. The walls and ceiling of the box use diffuse materials, and the floor uses a glossy material. All scenes use a single light source.
4.1.1 Reference images

All generated images, including the ones generated by SmallVCM, are compared against reference images for the corresponding scenes. These images are generated using SmallVCM, with a total of 10000 render iterations each.

4.1.2 Iterations per scene

All comparisons are done against scenes rendered with the same amount of samples. Instead of checking how many iterations can be performed within a defined time interval, the number of iterations to be rendered are passed as an argument. This ensures that the image comparison is performed between images that should be equal, or with as less as differences as possible.

All scenes are rendered with the same amount of iterations, and each scene is rendered multiple times, each with a different number of iterations. The target iterations are 1, 5, 10, 20, 50 and 100. All these renders are considered for render time analysis. However, for image comparison purposes, only the renders with 100 iterations are considered.

4.1.3 Light and camera path length

The paths generated in both light and camera path tracing all have the same maximum length, and thus, the same number of sub paths or vertices. The length of these is set to 10, in all scenes.

4.2 Hardware

Final rendering tests are performed on up to date and high end hardware, providing accurate results for the currently available technology. The specifications of the machine where CUDA VCM is executed are a AMD Ryzen 7 3700X 8-Core CPU, with a base clock frequency of 3.6GHz and 16GB of DDR4 RAM, and an NVIDIA GeForce RTX 2080 GPU, with 8GB of GDDR6. Profiling tests are done using the development machine, with a NVIDIA GeForce GTX 960M GPU, as the system is dual GPU, and kernels cannot be profiled otherwise. The drivers and software used are listed in table 4.2.

<table>
<thead>
<tr>
<th>NVIDIA CUDA drivers</th>
<th>NVIDIA NSight Monitor</th>
<th>Visual Studio</th>
</tr>
</thead>
<tbody>
<tr>
<td>11.0.221</td>
<td>2020.1.2.20203</td>
<td>2019</td>
</tr>
</tbody>
</table>

Table 4.2: Software versions in desktop development environment
4.3 Kernel execution on GPU

4.3.1 General performance considerations

When processing the CUDA implementation, the application will use both the CPU and the GPU. The CPU usage is related with all the scene preparation, model and material loading, and the GPU usage is related with the ray tracing algorithms processing and image creation. This means that using the same GPU model but with different CPU, the total processing time can increase or decrease slightly.

4.3.2 Timeout Detection & Recovery feature

When a CUDA application is being executed, it can take a considerable amount of time, depending on the complexity of the code being executed. If the application is running in debug mode, it will also take more time to execute. Longer execution times can cause the display driver to detect the GPU to be in an apparent frozen state, and the Windows restarts the GPU due to the Timeout Detection & Recovery (TDR) feature [21].

This feature can be configured manually using the Windows Registry, or using the Nsight Monitor [22].

It is important to configure TDR correctly, especially when doing performance analysis tests, which can be lengthy, and cause the application to be killed prematurely.

4.3.3 Kernels with different optimizations

During the development of the CUDA application, kernels with different configurations and optimizations were used. Initially, each kernel executed a single iteration, while at a later stage, each kernel processed as much iterations as the hardware allowed. Optimizations on the types of memory that were used also took place, combined or not with the previous configurations. The results will compare these configurations.

4.4 Kernel execution statistics

NVIDIA provides a very useful tool, NVIDIA Nsight monitor [23], that allows debugging and profiling of CUDA applications. NVIDIA Nsight Monitor offers very insightful metrics on GPU performance, by measuring the total amount of used memory, active threads and warps, branch divergence, among others. With this data, it is easier to identify possible performance issues in the kernel execution, instead of using a trial error approach on improving the kernel code, as these metrics show the different ways latency is introduced, and impacting the overall performance. The kernel execution statistics are going to be evaluated for both the slowest and fastest scenes rendered.

This tool works best for multiple GPU configurations, as some features may not be available in a single GPU configuration, in order to avoid the system crashing due to GPU being locked.
4.4.1 Occupancy

The first important metric that is going to be analysed is the GPU occupancy. This metric provides data on how many warps are allocated to each SM, based on the block size, number of registers, and shared memory used. It is noteworthy that kernels with more instructions require a higher register count. If this causes the number of warps per SM to be too small, there may not be enough kernels to hide latency.

As a general goal, the number of active threads at any given time should be the highest possible, while making efficient use of memory. As discussed previously, a higher number of active threads results in better performance, as latency can be hidden by increasing the number of active threads.

![Figure 4.1: Occupancy sample charts](image)

4.4.2 Instruction statistics

The instructions statistics charts contain information on the amount of instructions issued, per clock cycle to a SM, the distribution of instructions per warp, and the percentage of activity for each SM. Having a large number of instructions per SM does not translate directly into better performance. If a kernel contains thousands of instructions, this can introduce latency compared to using multiple kernels, with a smaller number of instructions.

On the other hand, having a very small number of instructions can also introduce latency as the warps may run out of instructions to be issued, before the kernel is finished.

The instructions per clock cycle are proportional to the efficiency of the kernel execution. The greater the value is, the better the performance is as well.

![Figure 4.2: Instruction statistics sample charts](image)
4.4.3 Branch statistics

The divergence is measured by the number of execution paths, in a kernel. In other words, the more conditional statements exist in a kernel, the greater the divergence will be. Code with smaller divergence is also generally more effective.

Nsight Monitor also provides statistics on the number of branches per warp, and the efficiency of the control flow. The latter can be related with threads being terminated earlier than others, warps being executed with less than 32 active threads, or instructions being executed by only a subset of threads.

![Figure 4.3: Branch statistics sample charts](image)

4.4.4 Issue efficiency

Issue efficiency reports if the warps are able to be issued instructions on each cycle. This can be measured by the warp issue efficiency. This chart contains the percentage of warps that are not eligible to execute instructions on a cycle, and the percentage of warps that are eligible to.

If the percentage of warps that are not eligible to execute instructions is too high, the negative impact in performance is also high.

The other important chart in the issue efficiency is the issue stall reasons chart. In this chart the different latency inducing factors are displayed for why the active warps are not able to execute instructions.

For example, if a warp is blocked due to memory being used by the maximum amount of resources, and more resources are not available for additional requests, or if the threads are waiting in a thread synchronization instruction, these are displayed in the chart.

![Figure 4.4: Issue efficiency sample charts](image)
4.4.5 Memory statistics

The memory statistics reports the usage of all different GPU memories, including L1 and L2 caches. The data is reported in different useful ways, including transactions size and count.

These statistics provide very helpful insights on how the memory is being used, how many cache requests are successful, and the amount of cache bank conflicts.

Having the data correctly structured in memory improves cache usage, and allows faster memory transactions.

![Memory statistics sample charts](image)

4.5 Image comparison

Besides the performance comparison, the final render results also must be compared. If we have a faster GPU implementation, but the results are not similar, the data may be being processed differently, and the algorithm not implemented correctly. Some simpler comparison techniques focus only on evaluating the average differences between pixels, not giving any information about the structure of an image, while others perform more complex comparisons, by checking the similarities between the compared images.

4.5.1 Absolute error count

Absolute error count (AE) is a metric that compares the different pixels between two images, by comparing the color values pixel by pixel. The maximum differences is proportional to the pixel count. For two equal images, the result is 0, and for two images where all the pixels are different, the result is equal to the size of the images, in pixels. For example, if the images have both width and height equal to 256, for the latter case, the result is 65536.

4.5.2 Peak signal to noise ratio

Peak signal to noise ratio (PSNR) measures the noise introduced to an image when the image is compressed, in case of image compression, or introduced by the rendering process, which is the scope where it is used here. As the pixels of a rendered image are not processed equally, with some pixels having more samples than others, noise is introduced in these images. The amount of noise is then
compared to the reference image, the signal. Values close to 50 dB, for an 8 bit image, represent an image that is very close to the reference image.

4.5.3 Root mean squared error.

Root mean squared error (RMSE) provides information about the average error when comparing two images. To get the squared errors, the sum of all the squared differences between each pixel is calculated and then divided by the total amount of pixels. After calculating the mean squared error, the square root is calculated, giving the average error between each pixel of the images. The bigger the differences between the images, the bigger the RMSE will be.

4.5.4 Structure similarity index metric.

Another image comparison method, the structure similarity index metric (SSIM), compares the differences in luminance, contrast, and in the image structure. All these components are independent between them, thus a change in one does not affect the others. The result from SSIM is given in a range of [0,1], where the closer it gets to 1, the more similar the images are.
Chapter 5

Results evaluation

5.1 Baseline Solution

The results for the CPU implementation of VCM are taken from SmallVCM. These are used as the baseline to compare both the generated images, and the performance of a GPU implementation against a CPU implementation.

5.2 CUDA VCM evaluation

The GPU implementation of VCM is considered to be better than SmallVCM if the quality is similar or better, and if the processing time is smaller, for the same number of iterations. The image quality metrics are calculated according to the metrics described in section 4.5. As mentioned in chapter 3, the results discussed in this chapter have the corresponding kernel configuration that is used. The configurations can be found in table 3.3. The reference image is also described in section 4.1.1, but as a reminder, the reference image is generated using SmallVCM, with a total of 10000 iterations. In the render results comparison, this is the figure identified as reference.

One very important remark about the results in this section is that the results from the GeForce RTX 2080 are taken from the application running with debug symbols enabled. This introduces a significant performance reduction, due to all performance optimizations on device code being removed [24], and so, the results presented do not match the best case for CUDA VCM running on this hardware. Additional tests were made using NVIDIA sample projects, and the results can be found at the end of this chapter.

5.2.1 Scene 1

This scene consists of a single large sphere, with a mirror material applied to it, and a single area light, in the ceiling. The environment color is black, and is reflected in the front of the sphere. The render results are shown in figure 5.1, with the reference image on the left.
Rendering the scene using configuration 2, where the camera connection is deferred, introduces some visible artifacts in some of the renders, depending on the number of iterations. The same applies to rendering the scene using configuration 3, where shared memory is used to store frequently accessed data. These artifacts are visible in figure 5.2.

Figure 5.2 shows the render time for the scene rendered with all target iterations, using a logarithmic scale. The render time using a linear scale can be found in figure C.1, in appendix C.

Render time for the scene with a target of 100 iterations is listed in table 5.1.
Table 5.1: Render time for scene 1

Table 5.2 lists the differences between the render results for a target of 100 iterations and the reference render, with 10000 samples.

Table 5.2: Image comparison metrics for scene 1

For render time, the performance of both GPUs is significantly worse than the CPU, and the difference is proportional to the number of iterations. Deferring the camera connection and using the shared memory to store frequently accessed data does not have any significant performance impact, being slightly worse when shared memory is used.

The quality of the resulting render for 100 iterations, using configuration 1, is the closest to the reference render. As previously seen in figure 5.2, the renders using kernel configurations 2 and 3 introduce artifacts in the renders, reducing the overall quality according to the metrics results. Figure 5.1d shows some bright pixels in the middle, resulting in the worst results, and figure 5.1e is slightly darker, resulting in a very similar SSIM, but lower PSNR and greater RMSE.

5.2.2 Scene 2

This scene consists of two small spheres, with mirror and glass materials applied to them, and a single environment light. The environment color is light blue, and is reflected in the front of the sphere. The render results are shown in figure 5.4, with the reference image on the left.
Rendering the scene using configurations 2 and 3 introduce some visible artifacts in some of the renders, depending on the number of iterations. These artifacts are visible in figure 5.5.

Figure 5.5: Render artifacts in scene 2 for configurations 2 and 3

Figure 5.6 shows the render time for the scene rendered with all target iterations, using a logarithmic scale. The render time using a linear scale can be found in figure C.2, in appendix C.

Figure 5.6: Render time for all iterations in scene 2

Render time for the scene with a target of 100 iterations is listed in table 5.3.
<table>
<thead>
<tr>
<th>Configuration</th>
<th>Iterations</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SmallVCM</td>
<td>100</td>
<td>15,744</td>
</tr>
<tr>
<td>Configuration 1</td>
<td>100</td>
<td>59,482</td>
</tr>
<tr>
<td>Configuration 2</td>
<td>100</td>
<td>58,175</td>
</tr>
<tr>
<td>Configuration 3</td>
<td>100</td>
<td>58,691</td>
</tr>
</tbody>
</table>

Table 5.3: Render time for scene 2

Table 5.4 lists the differences between the render results for a target of 100 iterations and the reference render, with 10000 samples.

<table>
<thead>
<tr>
<th>Configuration</th>
<th>AE</th>
<th>PSNR (dB)</th>
<th>RMSE (0-1)</th>
<th>SSIM</th>
</tr>
</thead>
<tbody>
<tr>
<td>SmallVCM</td>
<td>257487</td>
<td>33,562</td>
<td>5,351 (0,021)</td>
<td>0,744</td>
</tr>
<tr>
<td>Configuration 1</td>
<td>257626</td>
<td>32,853</td>
<td>5,806 (0,023)</td>
<td>0,740</td>
</tr>
<tr>
<td>Configuration 2</td>
<td>257832</td>
<td>31,749</td>
<td>6,593 (0,026)</td>
<td>0,741</td>
</tr>
<tr>
<td>Configuration 3</td>
<td>262141</td>
<td>16,292</td>
<td>39,078 (0,153)</td>
<td>0,641</td>
</tr>
</tbody>
</table>

Table 5.4: Image comparison metrics for scene 2

Like in scene 1, the render time of both GPUs is significantly worse than the CPU, and the difference is proportional to the number of iterations. Configurations 2 and 3 do not have any significant performance impact.

The quality of the resulting render for 100 iterations, using configuration 1, is the closest to the reference render. The renders using kernel configurations 2 and 3 introduce artifacts in the renders, reducing the overall quality according to the metrics results. When using configuration 3, which uses the GPU shared memory, background light is not read correctly, resulting in even more degraded quality which can be seen in figure 5.4e. This is the image with the worst metrics results. Once again, the render for configuration 2 displays some bright pixels in the middle, as seen in figure 5.4d.

5.2.3 Scene 3

This scene consists of two small spheres, with mirror and glass materials applied to them, and a single directional light. The environment color is black, and is reflected in the front of the sphere. The render results are shown in figure 5.7, with the reference image on the left.
Rendering the scene using configuration 2 introduces some visible artifacts in the render with 100 iterations, visible in 5.8. Contrary to other scenes, only this render contains visible artifacts.

Figure 5.8: Render artifacts in scene 3 for configuration 2, with 100 iterations

Figure 5.9 shows the render time for the scene rendered with all target iterations, using a logarithmic scale. The render time using a linear scale can be found in figure C.3, in appendix C.

Figure 5.9: Render time for all iterations in scene 3

Render time for the scene with a target of 100 iterations is listed in table 5.5.
<table>
<thead>
<tr>
<th>Configuration</th>
<th>Iterations</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SmallVCM</td>
<td>100</td>
<td>19.566</td>
</tr>
<tr>
<td>Configuration 1</td>
<td>100</td>
<td>84.741</td>
</tr>
<tr>
<td>Configuration 2</td>
<td>100</td>
<td>82.256</td>
</tr>
<tr>
<td>Configuration 3</td>
<td>100</td>
<td>82.792</td>
</tr>
</tbody>
</table>

Table 5.5: Render time for scene 3

Table 5.6 lists the differences between the render results for a target of 100 iterations and the reference render, with 10000 samples.

<table>
<thead>
<tr>
<th>Configuration</th>
<th>AE</th>
<th>PSNR (dB)</th>
<th>RMSE (0-1)</th>
<th>SSIM</th>
</tr>
</thead>
<tbody>
<tr>
<td>SmallVCM</td>
<td>254631</td>
<td>33,437</td>
<td>5,429 (0.021)</td>
<td>0.816</td>
</tr>
<tr>
<td>Configuration 1</td>
<td>255913</td>
<td>32,155</td>
<td>6,292 (0.025)</td>
<td>0.809</td>
</tr>
<tr>
<td>Configuration 2</td>
<td>257072</td>
<td>30,547</td>
<td>7,572 (0.030)</td>
<td>0.809</td>
</tr>
<tr>
<td>Configuration 3</td>
<td>256965</td>
<td>30,938</td>
<td>7,239 (0.028)</td>
<td>0.811</td>
</tr>
</tbody>
</table>

Table 5.6: Image comparison metrics for scene 3

The render time for both GPUs is significantly worse than the CPU, and the difference is proportional to the number of iterations, as also happens to scenes 1 and 2. Using configurations 2 and 3, renders are marginally faster, but there are artifacts introduced in some of the final renders.

The quality of the resulting render for 100 iterations using configuration 1 is the closest to the reference render. The kernel configuration 2 introduces the most significant noise in the resulting render, being this the worst quality render. Unlike scene 1 and 2, when the scene is rendered using configuration 3, the final render does not contain any visually identifiable noise.

### 5.2.4 Scene 4

This scene consists of two small spheres, with mirror and glass materials applied to them, and a single point light in the ceiling. The environment color is black, and is reflected in the front of the sphere. The render results are shown in figure 5.10, with the reference image on the left.

Figure 5.10: Scene 4 rendered with 100 iterations, with image 5.10a as reference render
The renders shown in figure 5.11 contain visible artifacts, similar to scene 1 and 2, where the artifacts also depend on the number of iterations. Similarly, these are visible when the scene is rendered using kernel configurations 2 and 3.

(a) Configuration 2 (10 i)  (b) Configuration 2 (100 i)  (c) Configuration 3 (1 i)  (d) Configuration 3 (10 i)

Figure 5.11: Render artifacts in scene 4 for configurations 2 and 3

The render time is shown in figure 5.12, using a logarithmic scale, and the render time using a linear scale can be found in figure C.4, in appendix C.

![Render time graph](image)

Figure 5.12: Render time for all iterations in scene 4

Table 5.7 contains the render time for the scene with a target of 100 iterations.

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Iterations</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SmallVCM</td>
<td>100</td>
<td>24,185</td>
</tr>
<tr>
<td>Configuration 1</td>
<td>100</td>
<td>86,152</td>
</tr>
<tr>
<td>Configuration 2</td>
<td>100</td>
<td>87,282</td>
</tr>
<tr>
<td>Configuration 3</td>
<td>100</td>
<td>87,656</td>
</tr>
</tbody>
</table>

Table 5.7: Render time for scene 4

The results for the quality metrics used to evaluate the rendered scenes are shown in table 5.8. The
scenes are rendered with a total of 100 iterations, and the reference render is rendered with 10000 samples.

<table>
<thead>
<tr>
<th>Configuration</th>
<th>AE</th>
<th>PSNR (dB)</th>
<th>RMSE (0-1)</th>
<th>SSIM</th>
</tr>
</thead>
<tbody>
<tr>
<td>SmallVCM</td>
<td>240149</td>
<td>34,980</td>
<td>4,545 (0.018)</td>
<td>0.836</td>
</tr>
<tr>
<td>Configuration 1</td>
<td>242180</td>
<td>29,010</td>
<td>9,037 (0.035)</td>
<td>0.823</td>
</tr>
<tr>
<td>Configuration 2</td>
<td>243609</td>
<td>26,740</td>
<td>11,736 (0.046)</td>
<td>0.813</td>
</tr>
<tr>
<td>Configuration 3</td>
<td>243307</td>
<td>27,236</td>
<td>11,086 (0.043)</td>
<td>0.819</td>
</tr>
</tbody>
</table>

Table 5.8: Image comparison metrics for scene 4

Following the previous scenes, render time for both GPUs is significantly worse than the CPU, and the difference is proportional to the number of iterations. In this scene, configuration 2, where the camera connection is deferred, and configuration 3, where shared memory to store frequently accessed data is used, render the scene marginally slower, and render artifacts are visible for some target iterations.

Due to the visual artifacts, the render quality for both configurations is the lowest. However, the quality for configuration 1 is also the lowest in this scene, compared to the previous ones, where the PSNR difference to the reference render is almost 6 dB.

5.2.5 Render time for all scenes per device

Additional charts can be found in appendix C.2, where the render time for all scenes is displayed on a chart per device. This enables a quick comparison of the render time for the different scenes discussed in this section, based on the device where the scenes are rendered. These charts use a linear time scale.

5.3 CUDA kernel profiling

During the software development cycle, profiling is used to understand, among other properties, where the application is spending more time during its execution, the amount of memory used, the functions that are called, and how many times. This is very useful to optimize an application, especially in less obvious functions. In order to profile the performance of the GPU, NVIDIA Nsight Monitor is used, and the results provided offer a clear overview on how different kernels perform. In this section, the profiling results for scenes that are rendered both in the lowest and highest execution times are going to be evaluated. In the sections where performance is very similar for different scenes, only one scene is discussed.

5.3.1 Performance overview

Figures 5.13 and 5.14 show an overview of the performance for all executed kernels. It is possible to verify that tracing all camera paths, which also performs the vertex connection and merging operations,
takes approximately 3 times more than tracing the light paths and performing the camera connection for scene 1. For scene 2, it takes approximately 4 times more to trace the camera paths and perform vertex connection and merging.

However, the maximum theoretical GPU occupancy for camera path tracing is half of the maximum theoretical occupancy for light path tracing, and the number of registers used is approximately twice for the camera path tracing. The analysis of registers occupation is done in section 5.3.2.

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Grid Dimensions</th>
<th>Block Dimensions</th>
<th>Start Time (us)</th>
<th>Duration (us)</th>
<th>Occupancy</th>
<th>Registers per Thread</th>
<th>Static Shared Memory per Block (bytes)</th>
<th>Dynamic Shared Memory per Block (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. ResetIterations</td>
<td>(1024, 1, 1)</td>
<td>(256, 1, 1)</td>
<td>2,483,412,394</td>
<td>153,312</td>
<td>100.00%</td>
<td>17</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2. GenerateLightPaths</td>
<td>(1024, 1, 1)</td>
<td>(256, 1, 1)</td>
<td>83,352,335,282</td>
<td>339,996,448</td>
<td>90.00%</td>
<td>56</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3. GenerateCameraPaths</td>
<td>(1024, 1, 1)</td>
<td>(256, 1, 1)</td>
<td>301,825,464,786</td>
<td>2,422,424,544</td>
<td>90.00%</td>
<td>111</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4. GetFrameBufferData</td>
<td>(512, 1, 1)</td>
<td>(512, 1, 1)</td>
<td>304,133,049,346</td>
<td>236,416</td>
<td>100.00%</td>
<td>21</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 5.13: Performance overview for scene 1 (top) and scene 2 (bottom) kernels using configuration 1

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Grid Dimensions</th>
<th>Block Dimensions</th>
<th>Start Time (us)</th>
<th>Duration (us)</th>
<th>Occupancy</th>
<th>Registers per Thread</th>
<th>Static Shared Memory per Block (bytes)</th>
<th>Dynamic Shared Memory per Block (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. ResetIterations</td>
<td>(1024, 1, 1)</td>
<td>(256, 1, 1)</td>
<td>1,470,107,861</td>
<td>183,036</td>
<td>100.00%</td>
<td>17</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2. GenerateLightPaths</td>
<td>(1024, 1, 1)</td>
<td>(256, 1, 1)</td>
<td>34,908,220,121</td>
<td>184,705,958</td>
<td>90.00%</td>
<td>56</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3. GenerateCameraPaths</td>
<td>(1024, 1, 1)</td>
<td>(256, 1, 1)</td>
<td>164,652,705,483</td>
<td>2,025,525,584</td>
<td>90.00%</td>
<td>111</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4. GetFrameBufferData</td>
<td>(512, 1, 1)</td>
<td>(512, 1, 1)</td>
<td>168,511,060,704</td>
<td>235,581</td>
<td>100.00%</td>
<td>21</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 5.14: Performance overview for scene 1 (top) and scene 2 (bottom) kernels using configuration 3

### 5.3.2 Occupancy

The occupancy of a GPU determines how many warps can run per SM, for a given kernel. This is directly related with the number of registers used in the kernel, number of threads per block and amount of shared memory used.

**Reset iterations**

The kernel used to reset iterations, when the number of iterations does not allow all iterations to be processed in a single run, allocates a very small number of variables, and sets data in two different arrays. Due to this, the number of registers used is 17, and the number of warps allowed per SM is 64.
Even though the theoretical occupancy is 100%, the GPU achieves an occupancy between 80% and 85% for both scenes. Figure 5.15 shows the occupancy when rendering scene 1.

![Figure 5.15: Occupancy for kernel used to reset iterations status, in scene 1](image)

**Reset connections**

The kernel used to reset the data for which if a vertex should be connected to the camera, when the number of iterations does not allow all iterations to be processed in a single run, allocates a single variable, and sets data in a single array. This is the kernel with the lowest number of registers used, 8, and the number of warps allowed per SM is 64. The occupancy achieved by the GPU is located between 40% and 45% for both scenes, while the theoretical occupancy is 100%. Figure 5.16 shows the occupancy when rendering scene 1. The occupancy is low due to the very limited number of instructions that warps must perform, requiring a relatively low throughput from the GPU.

![Figure 5.16: Occupancy for kernel used to reset camera connections, in scene 1](image)

**Light path tracing kernel**

The occupancy for light path tracing kernel is shown in figure 5.17. The kernel configuration 3 uses 64 registers per thread, being this value the upper limit to allow 32 warps to be processed per SM. It also uses 424 bytes of shared memory, being 16384 bytes the upper limit for 32 warps per SM. 32 warps per SM account for a theoretical maximum occupancy of 50%. Configuration 1 does not use any shared memory, and uses 56 registers. However, using a different number of threads per block can increase the occupancy, as seen in the varying block size chart, in the top of figure 5.17. Kernel configuration 1 uses less registers, even though it traces the light paths and connect them to the camera, which are
additional instructions. This is due to additional registers being required to allocate shared memory and temporary arrays.

Even though the theoretical occupancy is, at minimum, 50% for both kernel configurations, the achieved occupancy is lower, especially when scene 2 is rendered, as seen in figure 5.18. Considering that the light paths for scene 2 are rendered in approximately half the time, the achieved occupancy can be lower due to the kernel exiting earlier, not generating as much load in GPU as scene 1.

Connect light paths to camera

The kernel used to connect light paths to camera is used in configurations 2 and 3. The theoretical GPU occupancy is 50% when the blocks have 256 threads. With a different number of threads per block, the occupancy can increase slightly. The number of registers used is 53, which accounts for all the registers required for the camera connection operation, and the registers required to access arrays that
contain temporary data. This kernel does not use any shared memory, thus not having any impact in the occupancy. Figure 5.19 supports this data.

![Figure 5.19: GPU occupancy for kernel that connects light paths to camera](image)

Similar to the light path tracing kernel, the achieved occupancy is lower than the theoretical. Again, when scene 2 is rendered, the achieved occupancy is the lowest, as seen in figure 5.20.

![Figure 5.20: Achieved occupancy for kernel that connects light paths to camera](image)

**Camera path tracing**

The kernel used to trace camera paths is the kernel that requires the most registers, with a total of 111 for kernel configurations 1 and 2, and 117 for configuration 3. This register count limits the total number of warps per SM to 16, reducing also the theoretical maximum occupancy to 25%. The shared memory does not have impact on the number of active warps, as there are only 424 bytes used. To keep 16 warps, without changes in the number of registers, the shared memory can be up to 16384 bytes per block.

![Figure 5.21: Occupancy for camera path tracing kernel, for scene 1 using kernel configuration 1](image)
Figures 5.21 and 5.22 show the occupancy for scene 1 using configuration 1, and for scene 2, using configuration 3. Scene 2 achieves slightly more GPU occupancy than scene 1. Even though it is not shown here, the occupancy for each scene, using both kernel configurations, is very similar.

Get frame buffer data kernel

The kernel used to get frame buffer data from the iterations being processed and add it to the global frame buffer data also has a small number of registers allocated, as it allocates a very small number of variables, and gets and sets data in a single array. The number of warps allowed per SM is 64, being the theoretical occupancy 100%. The GPU achieves an occupancy of approximately 87% for both scenes, in both kernel configurations 1 and 3. Figure 5.23 shows this occupancy, and the achieved occupancy when rendering scene 1.

5.3.3 Branch statistics

Branch statistics contains information about the different branches that warps take. Ideally, all warps should take the same branch when executed, in order to have the best efficiency. The more conditional statements executed and more different branches executed by the warps, the less efficient a kernel is.

Reset iterations kernel

In order to reset the iterations data, the warps have only one execution path, as there are no conditional statements. Due to this, the efficiency of the kernel is 100%, as shown in figure 5.24.
Reset connections kernel

Similarly to previous kernel, when the connections are reset in configurations 2 and 3, the warps have only one execution path, as there are no conditional statements. The efficiency of the kernel is then 100%, as shown in figure 5.25.

Light path tracing

The kernel used to trace light paths contains multiple conditional statements. Some of them change how a light path is calculated, while others stop the tracing of a path. The latter can cause some threads to continue tracing different paths, while the remaining threads are idle, waiting for all paths to be calculated. This causes the control flow efficiency to lower for each thread that is not active within a warp. Figure 5.26 shows the branch efficiency for scene 1 and 2 when kernel configuration 1 is used, and figure 5.27 shows the branch efficiency for the same scenes when kernel configuration 3 is used.
The majority of threads within a warp take the same branch, which is shown in both taken and not taken branches. However, there are threads within warps that are taking different branches, as shown in diverged branches. When this happens, the branches within the warp are executed sequentially. If the branches take a considerable part of the execution, even if the percentage is low, the impact is significant. As the control flow is below 30% for all results shown in both figures 5.26 and 5.27, the branch divergence occurring is causing a high number of threads to be idle.
Control flow efficiency for scene 2 is considerably lower than for scene 1, even though this scene is rendered faster. This is due to the number of branches, being executed for scene 2, being 2.5 times less for configuration 1, and 2 times less for configuration 3. Based on this, it can be concluded that rendering scene 2 requires less sub paths to be calculated.

**Connect light paths to camera**

When light paths are connected to the camera, control flow efficiency is significantly low. Considering that tracing light paths without connecting them to the camera has a higher efficiency, this lowers the overall performance, and is the reason why both kernel configurations 1 and 3 have similar rendering time. In this case, the diverged branches also result in a very low control flow efficiency, even though the percentage of diverged branches is low.

![Branch statistics for kernel used to connect light paths to camera, for scene 1 (top) and scene 2 (bottom)](image)

The number of branches required for scene 1 is around 9.3 times the number required for scene 2, reducing the impact of diverged branches. Figure 5.28 shows the charts that support this analysis.

**Camera path tracing kernel**

Camera path tracing is more efficient in scene 2, compared to scene 1, as shown in figure 5.29. Added to this, scene 1 requires 1.7 times the number of branches to be executed, increasing the render time for scene 1. As the control flow is low in both scenes, this means that around 80% of the threads within a warp are idle.
Get frame buffer data

Getting the frame buffer data requires only one execution path, having all warps executing the same operations. This results in an efficiency of 100% for the kernel, as shown in figure 5.30.

5.3.4 Instruction statistics

When a kernel is executed, the warp scheduler is capable of instructing one or two IPC. For the device under test, each SM has 4 warp schedulers [25]. However, due to resources being required to fetch and assign instructions, the maximum achievable IPC is lower than the number of instructions per warp scheduler, multiplied by the number of warps schedulers. In order to have one IPC, the target value must be at least the number of warp schedulers [23], which may not be possible to reach, as it depends on the instructions.
**Reset iterations**

Figure 5.31 shows the IPC for the kernel used to reset iterations. Considering the number of SM for the GPU, all warps have executed 1 instruction for every 9.1 clock cycles. Each warp has an average of 45 instructions, and the average number of warps executed per SM is 16384, for a total of 8192 warps.

![Figure 5.31: Instruction statistics for kernel used to reset iterations status, in scene 1](image)

**Reset connections**

The best performing kernel in terms of IPC is the one that resets light path connection data, with a value of 2.13. This means that for every 2 clock cycles, all warps have executed an instruction. In this kernel, the number of instructions per warp is also very limited, with an average of 9, as shown in 5.32. This kernel is also executed by an average of 16384 warps per SM, for a total of 81920 warps. The number of warps is 10 times the number required for the previous kernel, due to the path length being equal to 10, and each sub path is assigned to one thread.

![Figure 5.32: Instruction statistics for kernel used to reset camera connections, in scene 1](image)

**Light path tracing**

In order to trace light paths, the number of instructions required is, on average, 159537.62 per warp, for a total of 8192 warps as shown in 5.33. Being the IPC 0.64 for scene 1, all warps have executed an instruction after every 6.25 clock cycles. Scene 2 instructions are executed slightly faster, with an IPC of 0.73, and also requires less instructions per warp. The high count of instructions per warp is an indicator that the kernel is probably performing too much instructions.
For kernel configuration 3, the number of instructions required is significantly lower for both scenes, being approximately 0.6 and 0.7 the number of instructions for scene 1 and scene 2, respectively. The number of warps launched is still 8192, as shown in 5.34. The IPC is very similar, being 0.65 for scene 1, and 0.71 for scene 2.

Figure 5.33: Instruction statistics for light path tracing kernel, for scene 1 (top) and scene 2 (bottom), using kernel configuration 1

Figure 5.34: Instruction statistics for light path tracing kernel, for scene 1 (top) and scene 2 (bottom), using kernel configuration 3

57
Connect light paths to camera

Connecting the light paths to camera requires a number of instructions per warp that is 1 order of magnitude lower than the required for tracing the light paths, for scene 1. However, the number of warps launched increases by the same proportion. For scene 2, a warp executes approximately 24 times less instructions. The IPC is higher for this kernel, with values of 0.72 and 0.78, for scene 1 and 2, respectively. The instruction statistics for this kernel are shown in figure 5.35.

Get frame buffer data kernel

Figure 5.36 shows the instruction statistics for the kernel used to add temporary frame buffer data to the global frame buffer. With an IPC of 0.71 for scene 1, similar to scene 2, all warps have executed 1 instruction after approximately 6 clock cycles. As the number of instructions that a warp needs to execute is low, with a value of 121, and the number of warps is 8192, the execution time of the kernel is also low.

Figure 5.36: Instruction statistics for kernel used to get frame buffer data, in scene 1
Camera path tracing kernel

The least performing kernel is the one used to trace camera paths. According to figure 5.37, this kernel has one of the lowest IPC, with 0.45 for scene 1, and 0.44 for scene 2, which combined with the highest number of instructions per warp, 337872.5 for scene 1 and 221552.82 for scene 2, makes this the longest running kernel. After approximately every 9 clock cycles, all warps have executed 1 instruction.

Figure 5.37: Instruction statistics for scene 1 (top) and scene 2 (bottom)

5.3.5 Issue efficiency statistics

Different kernels may have different performance issues, which are traceable in issue efficiency statistics. In this section, the main factors that reduce efficiency are going to be analyzed. For the device under test, a warp is issued an instruction per clock cycle if the number of eligible warps is at least 4[25], as this is the number of warp schedulers for the device, and as there must be at least one warp available per warp scheduler[23]. This is the same value as IPC, discussed in section 5.3.4, and will not be analysed again in this section.

Reset iterations

Figure 5.38 shows the issue efficiency for the kernel used to reset iterations, for scene 1. The efficiency is very similar for both scene 1 and 2. It is possible to observe that only about 10% of the cycles a warp is available to be selected by a warp scheduler. The main reason for this is the fact that data required to execute the instructions is not readily available, coupled with memory dependencies, where all the resources may be unavailable or fully utilized. As both operations performed in this kernel set values in two different arrays in the global memory, the warps need to wait for the memory operations to be completed.
Reset connections

The kernel used to reset temporary data for camera connections is, as seen previously, the kernel with the best performance. This is also verified in the issue efficiency, as shown in 5.39. The kernel is executed with a performance of more than 50%, resulting in 2 warp schedulers issuing an instruction every clock cycle. This kernel sets values in a single array in the global memory, resulting in a large number of requests of the same type. This causes memory to throttle, being the only kernel with memory throttling affecting performance significantly. Having also a very small number of instructions, there are not enough instructions to issue in parallel, to mitigate the effects of memory access, resulting in execution dependencies, which is the main factor reducing the overall performance of this kernel.

Light path tracing

When tracing light paths, the efficiency of warps is between 14% and 17%. This means that, at most, a warp scheduler has one warp that can be selected to issue an instruction only 17% of the total cycles. This happens when scene 2 is being processed, using kernel configuration 1. The main dependency is again memory dependency. As accessed memory locations are not guaranteed to be contiguous, due to light paths having a probability of contributing with color for very distinct locations in the final render, the memory access operations are slow. This happens for both kernel configurations 1 and 3. Instruction fetch is the second major reason for performance being degraded. This means that, between 16% and 23% of the times, the next instruction to be issued to a warp has not been fetched yet. The charts that support these statistics are shown in figures 5.40 and 5.41.
Figure 5.40: Issue efficiency for light path tracing in scene 1 (top) and scene 2 (bottom), using kernel configuration 1

Figure 5.41: Issue efficiency for light path tracing in scene 1 (top) and scene 2 (bottom), using kernel configuration 3

**Connect light paths to camera**

Connecting light paths to camera has slightly better issue efficiency compared to tracing these. The percentage of cycles that a warp scheduler has a warp to assign an instruction is 16.69% and 18.05%, for scene 1 and scene 2, respectively. However, the percentage of issues being stalled due to memory
dependencies is greater than the percentage for the kernel that traces light paths, as shown in figure 5.42, with a value of 55.36% for scene 1 and 60.09% for scene 2.

Figure 5.42: Issue efficiency for light path connection in scene 1 (top) and scene 2 (bottom)

Camera path tracing

Figure 5.43: Issue efficiency for camera path tracing in scene 1 (top) and scene 2 (bottom)

The kernel that traces camera paths has the warp schedulers issuing instructions slightly above 10% for both scenes. For this kernel, the major issue stall reason is memory dependency, probably due to the
poor spatial distribution of memory read and write operations, followed by instruction fetching. This is the kernel where instruction fetching has the highest percentage, as shown in figure 5.43.

Get frame buffer data kernel

In order to add temporary frame buffer data to the global frame buffer, data is read from an array in global memory, and added to another array, also stored in global memory. Due to this, the workload is very memory intensive, which can be identified by the main issue stall reason being memory dependency, with a percentage of 78.24% when rendering scene 1. The warp schedulers have a warp available to be issued a new instruction 15.90% of the time. Figure 5.44 shows this data, and the corresponding charts.

Figure 5.44: Issue efficiency for kernel used to get frame buffer data, in scene 1

5.3.6 Memory statistics

Memory access patterns are very important, as not optimized memory accesses can have a huge impact on a CUDA application. In section 5.3.5, it is clear that memory is a major factor for the overall degraded performance, with memory dependencies contributing between 35% and 80% for the issue stall reasons. In this section, instead of discussing based on kernel, the memory performance is discussed based on the type of memory. The scenes to be used for each type will be the more relevant, instead of choosing a single scene, and only for the global, local, shared and cache types of memory.

Most charts displayed in this section have the average number of transactions required per request. This means that for each memory request, it requires a certain amount of transactions, in order to be fully performed. This value should be as lowest as possible. For requests where objects contain a significant amount of data, it may be required to load or store this data using multiple requests, as it may not fit in a single one, or because data is not aligned according to the access pattern that is most optimized for the type of memory.

In order to get more detail about the specific instructions causing the transaction requests, it is necessary to analyze the generated assembly code. This is not covered in the scope of this thesis.

Global memory

The kernel used to generate light paths for scene 1, using configuration 1, performs 0.73 load L2 transactions, compared to the load requests performed by the kernel. This is shown in figure 5.45. On the
other hand, for each store request, 2,67 cache stores are required. This kernel accesses memory with a very high address divergence, especially when adding the color contribution of the light vertex, as seen in listing 5.1, where x and y can be any point in the image, within the image boundaries.

![Global memory statistics for kernel used to trace light paths, in scene 1](image1)

**Figure 5.45:** Global memory statistics for kernel used to trace light paths, in scene 1

```
CUDA_CALLABLE_MEMBER void Framebuffer::AddColor(const Vec2f& aSample, const Vec3f& aColor, int iteration)
{
    if (aSample.x < 0 || aSample.x >= mResolution.x)
    {
        return;
    }
    if (aSample.y < 0 || aSample.y >= mResolution.y)
    {
        return;
    }
    int x = int(aSample.x);
    int y = int(aSample.y);
    mColor[x + y * mResX] = mColor[x + y * mResX] + aColor;
}
```

**Listing 5.1:** Function used to add light vertex color contribution

Tracing light paths using kernel configuration 3, for the same scene, results in almost half of memory loads, but results in 1.3 times more memory stores. Each store is also more expensive, with an average of 4.11 L2 transactions per request, as shown in figure 5.46.

![Memory statistics for kernel used to trace light paths in scene 1, using configuration 3](image2)

**Figure 5.46:** Memory statistics for kernel used to trace light paths in scene 1, using configuration 3

The code in listing 5.2 is used to store temporary data to connect light vertices to camera.
if (lightState.mPathLength + 1 >= iterationConfiguration.minPathLength) {
    iterationConfiguration.misVmWeightFactor, iterationConfiguration.lightSubPathCount, iteration); 
    connectPath[currentLightPathIndex] = true;
    pathState[currentLightPathIndex] = lightState;
    pathHitPoint[currentLightPathIndex] = hitPoint;
    pathBsdf[currentLightPathIndex] = bsdf;
}
else {
    connectPath[currentLightPathIndex] = false;
}

Listing 5.2: Code used to store temporary data to defer light vertex camera connection

The kernel used to get temporary frame buffer data and add it to the global frame buffer requires 7.6 L2 transactions per load request, and 8 L2 transactions per store request, as shown in figure 5.47. The data in these requests is contained in Vec3f objects, which means, each object contains 3 float variables, with 4 bytes each, which is listed in 5.3. Despite the relatively high number of transactions per store request, the total number of requests per warp is very limited, with 10 load requests, and 3 store requests.

Figure 5.47: Memory statistics for kernel used to add temporary frame buffer data to global frame buffer, in scene 1

Listing 5.3: Code used to get temporary frame buffer data and add it to global data

```cpp
__global__ void GetFrameBufferData(SceneCuda *scene, Framebuffer *frameBuffer, Vec3f * framebufferColorData, Vec3f scaleFactor) {
    int imageSize = scene->mCamera->mResolution.x * scene->mCamera->mResolution.y;
    int pathIndexOffset = blockIdx.x * blockDim.x + threadIdx.x;
    int pathIdx = (pathIndexOffset) % imageSize;
    framebufferColorData[pathIdx] += frameBuffer->GetColorData(pathIdx) * scaleFactor;
}
```

CUDA_CALLABLE_MEMBER Vec3f Framebuffer::GetColorData(int pathIndex) {
    return mColor[pathIndex];
}
Local memory

Figure 5.48 shows the local memory statistics for kernel used to connect light paths to camera. The number of load requests is significantly higher than the number of store requests, mainly because the temporary data needs to be read. The number of L2 transactions per request is 3.57 and 22.79 for load and store, respectively. Contrary to the kernel used in configuration 1, this kernel does not have the requests used to trace light paths averaging the transactions per request to a lower value. The total number of store requests per warp is 204.42, which is around 4 times less the number of store requests for the kernel that traces and connects light paths, with a value of 836.53 store requests per warp.

![Figure 5.48: Local memory statistics for kernel used to connect light paths, in scene 1](image1)

Shared memory

Shared memory has a very limited usage in CUDA VCM, storing only some scene parameters, and all light sources. As seen in figure 5.49, being the shared memory usage very limited, there is still a considerable amount of requests performed by the kernel that generates camera paths, with a number of transactions per request equal to 1 for load and 0.9 for store requests. The number of load requests per warp is 255.23, and the number of store requests is 69.75. During the requests, there are no memory bank conflicts, being the pattern used to access shared memory optimized for bank conflicts. The code used to store lights in shared memory is listed in appendix B.1. The kernel used to generate camera paths also has a very high address divergence for memory requests, as it also uses the function listed in 5.1, but this is only reflected in memory types other than shared.

![Figure 5.49: Shared memory statistics for kernel used to generate camera paths, using kernel configuration 3, when scene 2 is rendered](image2)
Cache memory

The statistics for cache memory are shown in figure 5.50. Both L1 and L2 caches are being used, when there are memory requests. Out of all L1 requests, only 13.99% are successful. This results in 590.02 bank conflicts per warp. Taking and quoting as an example the example given by NVIDIA in the Nsight Monitor User Guide [26], for memory cached in both L1 and L2, if every thread in a warp loads a 4-byte value from sparse locations which miss in L1 cache, each thread will incur one 128-byte L1 transaction and four 32-byte L2 transactions. This will cause the load instruction to reissue 32 times more than if the values would be adjacent and cache-aligned. Due to the bank conflicts, the memory requests may be taking too long, being one reason why the issues are stalled due to memory dependencies, as discussed in section 5.3.5.

![Figure 5.50: Cache memory statistics for kernel used to generate camera paths, in scene 1](image)

5.4 Final considerations

Besides the much higher core count on a CUDA GPU, the code must be implemented using efficient data access patterns. Otherwise, the performance achieved can be significantly lower compared to a CPU, even when the number of threads is theoretically the maximum for the GPU, and faster memory types are used. Profiling tools and respective documentation offered to use together with these GPUs, provide detailed data that can and should be used during the whole development cycle. In the case of rendering algorithms, it is also very difficult to have code optimized to all scene configurations, and render times can be significant different for scenes with similar object shapes, but with different materials or light configurations.

Contrary to what is expected, all render times are significantly worse in the CUDA VCM implementation, when executed in NVIDIA GeForce RTX 2080. This is due to the application running with GPU debug information enabled, by passing flag `-G`. When compiling the application in release mode, without passing this flag, using the most up-to-date CUDA drivers and Visual Studio, CUDA compiler would crash. The root cause why the code could not be compiled was not found. When the GPU debug information is enabled, the application runs significantly slower due to all code optimizations being disabled. However, this does not happen in NVIDIA GeForce GTX960M. When CUDA VCM is executed in this GPU with debug information enabled, render times are at least 5 times longer. To have a better un-
derstanding of the impact of having code optimizations disabled in NVIDIA GeForce RTX 2080, some CUDA sample projects were executed, with both the GPU debug information flag enabled and disabled. The results are described in table 5.9.

<table>
<thead>
<tr>
<th>Project</th>
<th>Time - release (ms)</th>
<th>Time - release with -G flag (ms)</th>
<th>Factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Matrix multiply</td>
<td>0.131</td>
<td>1.555</td>
<td>11.87</td>
</tr>
<tr>
<td>Merge sort</td>
<td>4.312</td>
<td>173.856</td>
<td>40.32</td>
</tr>
<tr>
<td>Scan (large arrays)</td>
<td>0.30</td>
<td>3.890</td>
<td>12.97</td>
</tr>
</tbody>
</table>

Table 5.9: Execution time for CUDA sample projects

According to these results, the execution of CUDA VCM should theoretically be faster in the GeForce RTX 2080 GPU, compared to AMD Ryzen 7 3700X 8-Core CPU, with a base clock frequency of 3.6GHz.
Chapter 6

Conclusions

Fine tuning a CUDA application is a process that requires a deep understanding of how both the software, but most importantly, GPU hardware work. With each new architecture released by NVIDIA, memory access patterns evolve, and new hardware capabilities are added to the GPUs. Data optimization patterns should be used, in order to get the best performance out of the different types of memory available in the GPU, as the number of CUDA cores is not enough to increase performance, as it happens with faster CPUs. That being said, traditional software development techniques cannot be used, where trusting the compiler to optimize the software is often enough. In a CUDA application, the developer is responsible for the most critical optimizations, that may be the difference between a very fast or very slow application. Rendering algorithms have the particularity of the data in memory being accessed with high divergence, coupled with the infinite combinations of materials and objects that can be part of a scene. This high divergence is the result of rays being cast from the camera or from the light sources having a high probability of bouncing in very different directions.

6.1 Achievements

The main goal of this thesis, which consisted in having an implementation of the VCM algorithm to be executed faster on CUDA GPUs, when compared to an implementation to be executed on CPUs, was not achieved. Nonetheless, the overall work is proven to have been a very rich experience about parallel processing using CUDA, and its relation with rendering algorithms, in this case, by implementing VCM. The section 5.2 discusses the traditional image comparison results, with the different renders being compared with reference renders, but it is in section 5.3 that the major contribution of this thesis is provided, focusing on a very detailed profiling of the kernels developed, so that the reasons why the main goal was not achieved are clear, and may be mitigated in future work.
6.2 Code structure and hardware usage

Efficient code in a kernel is directly related with the amount of instructions that need to be executed, and hardware resources used in the GPU, as discussed in sections 5.3.2 and 5.3.4. Kernels must then consist of a good balance between number of instructions present in the code, which translates in number of registers used, number of threads, and memory allocated, both defining the number of warps that can be executed simultaneously in a SM. The kernel used to calculate camera paths, shows that more instructions increases the number of registers used, and reduces the number of warps per SM, as discussed in 5.3.2. Code with a few instructions, as is the case of the kernel used to reset the camera connections, increases the number of warps per SM, and reduces the number of registers. However, it also shows that the number of instructions should not just be reduced as much as possible, as the achieved occupancy is well below the theoretical 100%, thus supporting the good balance between instructions in different kernels.

The results discussed in 5.3.3 also show that an algorithm should be restructured, to minimize conditional code, wherever possible. If these conditions can be removed, the threads are not be stopped prematurely nor are idle waiting for others to finish their branches. In the scope of CUDA VCM, there are multiple conditions that stop light and camera paths being traced, which is then reflected in the overall performance of the GPU.

6.3 Memory usage

The assumption that faster memory means faster performance, and in case of VCM, less time required for renders, is refuted with the profiling of different memory types. Results presented in section 5.3.6 show this, especially for cache memory, where there is a high number of bank conflicts for L1 cache, resulting in additional memory transactions to L2 cache. Once again, the best types of memory to be used should be decided based on how data is stored in memory, and how it is accessed. It may be even better to disable L1 cache completely. For rendering algorithms, it is not an easy task to optimize an application to make good use of cache memory. This thesis focused mostly in the usage of global memory, with some data being stored in shared memory. However, for future work, shared memory can be further explored, as well as constant memory.

6.4 Profiling

Compared to an application that is executed on the CPU, CUDA applications have a much higher number of cores processing data, and small code changes can have a much greater impact on performance. On top of this, as CUDA is a low level API, it allows a great degree of control and tuning, at the cost of the complexity that comes with writing software for specific architectures. The section 5.3 shows that working with tools provided by NVIDIA, like Nsight Monitor, without having to rely on third party providers, ensures that this complexity can be overcome with a careful analysis and understanding of
the data provided, and profiling can be easily done, with highly detailed reports.

6.5 Future Work

In the previous sections, some work that was left out or that can be improved is mentioned. In this section, all these references are summarized, and any future work that is not discussed yet is added, where the current CUDA implementation can be used as a starting point.

6.6 Render a scene using multiple smaller portions

The intrinsic divergence present in rendering algorithms is directly related with the results presented in 5.3.3. This divergence is caused by rays bouncing in different directions, even for the same path, both for camera and light sub paths. Using a combination of rendering smaller portions of a scene, together with storing the data for these portions in adjacent memory positions, can be an efficient method to reduce render times. Having a smaller range in memory where data can be stored promotes better cache usage, thus reducing the memory transactions required. This optimization is left for future work.

6.6.1 Shared memory

Shared memory was implemented only for storing part of the scene properties, and the light sources. Geometry is a very good candidate to be stored in shared memory, as it is used for each sub path, for each iteration. Geometry is static during the whole scene rendering, and is used by all SM. Profiling of the shared memory, in section 5.3.6, shows that where the shared memory is used, it is still very far from reducing the number of warps for each SM.

6.6.2 Hash grid calculation using GPU

The hash grid used to detect vertices that should be merged is implemented in the CPU. This means that there are memory transactions to copy light vertices related data to the system RAM, hash grid is calculated, and the result is then copied to GPU. The hash grid is implemented in the CPU due to the exclusive prefix sum (or scan), that when implemented in the GPU was slower than calculating it in the CPU. However, there is a performance hit due to these copies between memories. NVIDIA provides a sample project [27] that implements this algorithm, as well as an article [28] that discusses it.

6.6.3 Split camera path tracing kernel into smaller ones

The kernel used to trace camera paths is performing too much work, being using a very large number of registers. This causes a low number of warps to be executed by the SM. Splitting the kernel into smaller ones, to the point where register count allows the use of more warps, should improve performance, by having more warps active. This should improve occupancy, reducing the effects of having idle warps.
6.6.4 Support for non square images

The current implementation only allows square images to be render. Due to this limitations, different aspect ratios cannot be rendered. Having the possibility of rendering non square images can be used for both the final rendered image format, or for smaller intermediate portions rendered.
Bibliography


Appendix A

OptiX framework

A.1 OptiX 6.5 and 7.1 framework comparison

<table>
<thead>
<tr>
<th></th>
<th>Microsoft DXR</th>
<th>Vulkan</th>
<th>OptiX 6</th>
<th>OptiX 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operating System</td>
<td>Windows</td>
<td>Windows, Linux</td>
<td>Windows, Linux</td>
<td>Windows, Linux</td>
</tr>
<tr>
<td>RTX shader model</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Custom primitives</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Shader languages</td>
<td>HLSL, DXIL</td>
<td>HLSL, GLSL, SPIR-V</td>
<td>CUDA, OSL, MDL, PTX</td>
<td>CUDA, OSL, MDL, PTX</td>
</tr>
<tr>
<td>Motion blur</td>
<td></td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Multi-level traversal</td>
<td></td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Fully thread safe</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Async</td>
<td>✓</td>
<td>✓</td>
<td>Limited</td>
<td>✓</td>
</tr>
<tr>
<td>Memory management</td>
<td>Explicit</td>
<td>Explicit</td>
<td>Implicit</td>
<td>Explicit</td>
</tr>
<tr>
<td>Multi GPU</td>
<td>Explicit</td>
<td>Explicit</td>
<td>Implicit</td>
<td>Explicit</td>
</tr>
<tr>
<td>NVLink5Scaling</td>
<td>Explicit</td>
<td>Explicit</td>
<td>Implicit</td>
<td>Explicit</td>
</tr>
<tr>
<td>AS Builds</td>
<td>Explicit</td>
<td>Explicit</td>
<td>Implicit</td>
<td>Explicit</td>
</tr>
<tr>
<td>Shader Binding Table (SBT)</td>
<td>Explicit</td>
<td>Explicit</td>
<td>Implicit</td>
<td>Explicit</td>
</tr>
<tr>
<td>GPU Support</td>
<td>Volta, Turing</td>
<td>Volta, Turing</td>
<td>Maxwell, Pascal, Volta, Turing</td>
<td>Maxwell, Pascal, Volta, Turing</td>
</tr>
</tbody>
</table>

Figure A.1: OptiX 6.5 and 7.1 framework comparison (Source: NVIDIA 2020 [29])
Appendix B

CUDA VCM source code

B.1 Scene lights in shared memory

```c
__device__ Lights GetLights(SceneCuda scene)
{
    __shared__ AreaLight sharedAreaLights[MAX_LIGHTS];
    __shared__ BackgroundLight sharedBackgroundLights[MAX_LIGHTS];
    __shared__ DirectionalLight sharedDirectionalLights[MAX_LIGHTS];
    __shared__ PointLight sharedPointLights[MAX_LIGHTS];

    Lights lights;
    lights.areaLights = sharedAreaLights;
    lights.backgroundLights = sharedBackgroundLights;
    lights.directionalLights = sharedDirectionalLights;
    lights.pointLights = sharedPointLights;

    if (threadIdx.x < scene.mLightsAreaCount)
    {
        sharedAreaLights[threadIdx.x] = scene.mLightsArea[threadIdx.x];
    }

    if (threadIdx.x < scene.mLightsBackgroundCount)
    {
        sharedBackgroundLights[threadIdx.x] = scene.mLightsBackground[threadIdx.x];
    }

    if (threadIdx.x < scene.mLightsDirectionalCount)
    {
        sharedDirectionalLights[threadIdx.x] = scene.mLightsDirectional[threadIdx.x];
    }

    if (threadIdx.x < scene.mLightsPointCount)
    {
        sharedPointLights[threadIdx.x] = scene.mLightsPoint[threadIdx.x];
    }

    __syncthreads();

    return lights;
}
```

Listing B.1: Copy lights from global to shared memory
Appendix C

CUDA VCM tests

C.1 Render time - linear scale

![Figure C.1: Scene 1 render time - linear scale](image1)

![Figure C.2: Scene 2 render time - linear scale](image2)
C.2 Render times for different devices
Figure C.6: CUDA VCM - GeForce RTX 2080

Figure C.7: CUDA VCM - GeForce GTX 960M