Utrecht University

Master Thesis

Porting and Maintaining a Developing OptiX Path Tracer for Non-NVIDIA Devices

Supervisors: Dr. ing. Jacco Bikker, Dr. Zerrin Yumak

Satwiko Wirawan Indrawanto

ICA-6201539

Faculty of Science

Department of Informatics and Computing Science

13 August 2019

Contents

1. Introduction

In the past decade, computer research is moving towards Graphics Processing Unit (GPU) computing. Instead of running applications on one high-performance core, GPU computing allows the creation of applications that runs on thousands of smaller cores in a highly parallel manner. However, the cross-platform compatibility of General-Purpose computing on Graphics Processing Unit (GPGPU) is hampered by limitations of programming languages. The GPGPU programming languages are often proprietary and only compatible with specific devices which limits broad usability.

In this case study, we intend to bring LightHouse 2, an interactive GPU path tracer that exclusively runs on NVIDIA devices [1], to non-NVIDIA environments. LightHouse 2 uses the GPGPU programming language NVIDIA CUDA [2] and the *ray tracing* library OptiX [3]. Currently, most GPGPU applications utilizes CUDA. The language's popularity is backed by the fact that NVIDIA held almost three-quarters (74.3%) of the graphics card market share in Q3 of 2018 [4].

Even though CUDA is popular among developers, it is closed-source and only runs on NVIDIA graphics products. This vendor-exclusivity happens due to the highly optimized programming language with the compiler having specific hardware instructions that non-NVIDIA graphics cards cannot execute. For CUDA applications to run on GPUs not manufactured by NVIDIA, such as AMD [5] and Intel [6], *software porting* is required. Software porting is the act of transferring existing software to a new environment [7].

1.1 Objectives

This master thesis focuses on the possibility of porting code written in a device-specific programming language to a language that is compatible across devices from different vendors. We use LightHouse 2's *rendering core* as the basis of our porting project.

The critical step is to determine the *porting cost* by defining a clear efficiency measure and deciding the significant variables. To ease maintenance and avoid feature loss because of differences in environment architecture, deciding a *porting method* using the optimal tools and programming languages is necessary. Reliability experiments are conducted by comparing rendering performance and image quality between the implemented port and the original version.

1.2 Research Questions

We have formulated three software porting relevant research questions that our case study strives to answer.

1. How can we minimize the porting cost of a vendor-specific GPGPU application to a vendor-agnostic platform?
2. How to approach a partial closed-source part of the software in porting?
3. Assuming performance benefits make it worthwhile to keep both implementations: How can we minimize the cost of maintaining the ported code with the original code in sync?

1.3 Content Overview

This literature review is divided into multiple sections. In Section 2, we discuss the preliminaries and dependencies of our case study. Section 3 presents relevant work to software porting. Section 4 describes

our research methodology and experiments. In Section 5, we discuss our implementation and address problems that occur when porting and how we solved them. Section 6 holds the experiment results. We evaluate the port and measure the performance of the application. We conclude our thesis in Section 7 by answering our research questions and discussing opportunities for porting similar projects.

2. Preliminaries

In this section, we provide information on our GPGPU case study application's concepts and software dependencies. We use the application LightHouse 2's rendering core as the basis for our research. The rendering core uses path tracing to generate images and, at a lower level, uses the ray tracing library OptiX. To better understand the scope of our porting research, we explain further about rendering, LightHouse 2, and OptiX.

2.1 Rendering

Rendering is producing an image through rasterization, projecting three-dimensional objects onto a two-dimensional surface. A traditional rasterization engine loops over the visible mesh triangles on the screen (object order algorithm) and projects one triangle to two-dimensional space, one at a time. Unfortunately, conventional rasterization does not account for correct physical light transport, which limits the realism of the produced images.

Ray tracing is a rendering technique that uses object-geometry intersections to accumulate shading information, which allows the creation of photorealistic images. This technique allows the rendering of natural light transport phenomena such as soft shadows, reflections, and refractions [8]. The natural behavior of light is simulated by evaluating the *rendering equation*.

2.1.1 Rendering Equation

The rendering equation was introduced in 1986 by David S. Immel et al. [9] and James T. Kajiya [10]. The equation is an integral that states how light is perceived on a surface point, depending on the incoming light function and the *bidirectional reflectance distribution function* (BRDF) (Equation 1). Evaluating this equation generates an image render.

$$L_o(x, \omega_o) = L_e(x, \omega_o) + \int_{\omega} f_r(x, \omega_i, \omega_o) L_i(x, \omega_i) \cos\theta_i \, d\omega_i \tag{1}$$

- $L_o(x, \omega_o)$ is the total outgoing spectral radiance towards the direction ω_o from position x.
- $L_e(x, \omega_o)$ is the emitted spectral radiance towards the direction ω_o from position x. This term is also known as the direct illumination part.
- $f_r(x, \omega_i, \omega_o)$ is the BRDF of the incoming direction ω_i and outgoing direction ω_o on position x.
- $L_i(x, \omega_i)$ is the incoming spectral radiance from the direction ω_i towards position x.
- $\cos\theta_i$ is the conversion of radiance to irradiance.

The BRDF is a distribution function that defines how an opaque surface reflects incoming light energy [11]. Light reflection and transmission functions are generalized as BxDF, a general distribution function that includes all variants. The BxDF determines how a renderer handles a specific type of material. It is crucial in our research that the port retains the same material behavior.

The rendering equation is a recursive integral, which cannot be evaluated directly. It is typically evaluated using Monte Carlo integration, where random sampling is used to estimate the value of an integral. The

integration only needs to evaluate an integrand $f(x)$ at a random point to estimate the value of the integral $\int_a^b f(x)\,dx$ [12].

$$\int_a^b f(x)\,dx \approx \frac{b-a}{N}\sum_{i=1}^{N} f(X_i) \tag{2}$$

Monte Carlo approximates the expected value of an integral $\int_a^b f(x)\,dx$ as seen in Equation 2. In this equation, X_1 to X_N are members of $[b, a]$. We can use arbitrary N values regardless of the integral dimension. As more samples are taken and N approaches infinity, the result approaches the expected value of f.

$$L_o(x, \omega_o) = \int_{\omega} f_r(x, \omega_i, \omega_o) L_d(x, \omega_i) \cos\theta_i \, d\omega_i \tag{3}$$

$$L_o(x, \omega_o) \approx \frac{2\pi}{N}\sum_{i=1}^{N} f_r(x, \omega_i, \omega_o) L_d(x, \omega_i) \cos\theta_i \tag{4}$$

The recursive rendering integral (Equation 3) thus can be approximated using Monte Carlo (Equation 4). Note that we multiply the equation with 2π because we integrate over the area of a half hemisphere. By averaging many samples on the same pixel using the Monte Carlo approach, we can get an output that is statistically close to the unbiased image. Solving the rendering equation with the approximation approach is introduced with *path tracing*.

2.1.2 Path Tracing

Path tracing is a rendering technique that uses stochastic evaluation of the *rendering equation* to process high-level light transport phenomena (Figure 1).

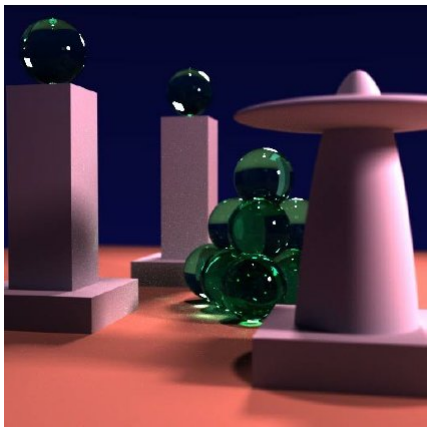


Figure 1. James T. Kaijya's original path traced scene showing soft shadows, reflections, refractions, and color bleeding.

The technique involves casting rays from the observer towards the scene to find ray-object geometry intersections. The path does not start from a light source; to avoid computation of unnecessary paths that would never reach the observer. At a geometry intersection, path tracing bounces the ray to a random outgoing direction while traditional ray tracing immediately ends the path. Path tracer rays terminate when

they hit a light source or reaches the maximum number of bounces. The path provides information to compute the light energy value that is transported to the observer from a light source.

Figure 2. Images generated from path tracing with the difference in quality depending on the samples taken. The image on the left is generated with eight samples per pixel, and the image on the right is generated with 1024 samples per pixel [12].

A single iteration of path tracing generates a noisy image because in general, a single path is not enough to evaluate a pixel. Averaging multiple iterations (by taking more samples) converges the image into a more proper representation of the scene (Figure 2). A path tracer needs an infinite amount of ray bounces to obtain an unbiased image, thus requiring an infinite amount of ray evaluations. The rendering technique uses a stochastic evaluation to approximate the infinite ray paths and solve the rendering equation.

Evaluating many light paths makes path tracing slow and computationally expensive. The number of samples is critical in path tracing, such that reducing ray bounces is generally not an option to trade for performance. Thus, we increase path tracing performance using accelerated structures.

2.1.3 Performance

The general method to improve intersection evaluation performance is by implementing a bounding volume hierarchy (BVH). The BVH is an acceleration structure that reduces the time and complexity for ray-object intersection evaluation. Constructing and traversing BVH in our case study is purely done using NVIDIA OptiX (see Section 2.3). We can replicate this functionality in our port either using an equivalent library or reconstructed using another programming language that supports non-NVIDIA devices.

Another form of optimization is splitting the workload into several processors. Since each ray is not dependent on other rays, there comes an opportunity for optimization using high parallelization devices such as GPUs [13]. Ray tracing on the GPU has been a well-known practice and is capable of being executed on consumer-level hardware. Since we are targeting to port to non-NVIDIA GPUs, we have to port using a GPGPU capable programming language.

2.2 Case Study: LightHouse 2

LightHouse 2 is an interactive GPU path tracer that is built using C++, NVIDIA CUDA, and NVIDIA OptiX. The application is on active development at Utrecht University, and targets commercial use such as online product configurators. These configurators allow users to adjust products with different color and materials, which then render the product within several seconds [14].

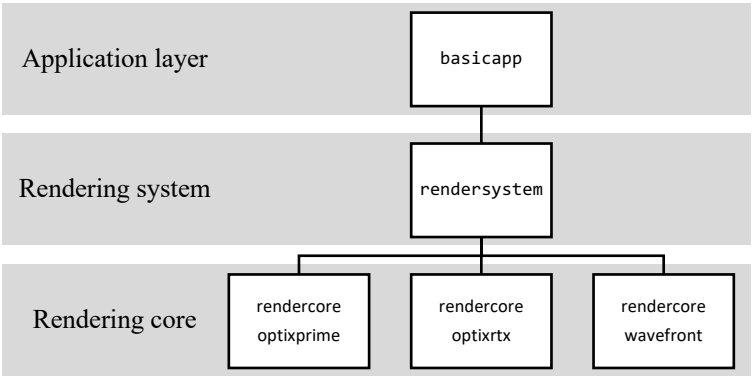


Figure 3. The software hierarchy of LightHouse 2.

LightHouse 2 itself consists of an application layer, a rendering system, and a modular rendering core. These modules are connected in a hierarchical structure (Figure 3), such that their functions are distinct, easily maintained, and having a high degree of extensibility and portability.

2.2.1 Application Layer

The application layer's primary function is to display the render output through a two-dimensional texture buffer. The layer also allows user interaction through a GUI (graphical user interface) for camera movement input and render parameters modification. Two-way communication between the application layer and the render system is established to pass the modified parameters and receive new render frames.

The layer itself is constructed under the 64-bit Windows environment, in such that operating system cross-environment compatibility requires porting. We limit our research scope by only supporting the source application's operating system environment.

2.2.2 Rendering System

The rendering system is an intermediate layer that is responsible for camera control, initializing a rendering core (passing scene data to the core), and receiving rendered pixels from the core. Camera control is a small module that takes input from the application layer and alters the render view based on the given parameters. Initialization is primarily loading a scene (meshes, textures, materials, and lights) into memory for the rendering core. On runtime, the render system triggers the core to render on each time tick.

By having interconnectivity between all the available rendering cores, the rendering system allows LightHouse 2 to swap the rendering cores without any modification on the application layer. In our research, none to small modification in the system may be required when adding our ported core.

2.2.3 Rendering Core

The rendering core is the module that contains all the necessary rendering functions. On initialization, the core prepares scene data from the rendering system by loading it into the core's specific scene format. The core can only communicate with the render system for receiving scene data, camera input, and sending out rendered pixels. This communication approach makes the cores easily swappable with any rendering algorithm.

LightHouse 2's primary rendering core is written in CUDA and uses the ray tracing library OptiX, which only supports NVIDIA devices. Since our project focuses on porting the rendering core to a vendor-agnostic

environment, we must decide on an equivalent solution that substitutes CUDA and OptiX. By solving the platform compatibility problem, one can envision having a fully optimized rendering core for every device available.

2.3 OptiX

OptiX is a high-level, general-purpose ray tracing engine developed by NVIDIA. The engine is a proprietary API and is designed to run only on NVIDIA GPUs and CUDA-enabled devices. It uses the CUDA API for low-level communication with the device. Its functionality consists of:

- Low-level built-in ray-geometry interaction functions.
- A small set of lightweight programmable functions.
- Single-ray programming model.
- Compiler optimization based on the hardware used.
- Node graph architecture for scene management.

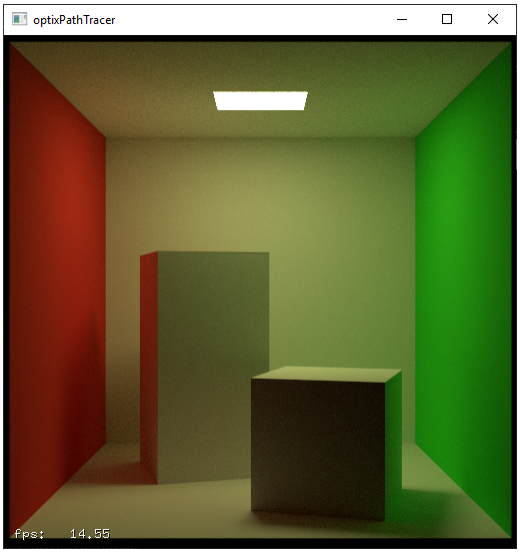


Figure 4. An OptiX path tracer rendering a Cornell Box scene.

OptiX provides a robust ray tracing pipeline with easy-to-use programmable functions, where path tracing can be easily implemented (Figure 4). LightHouse 2, however, does not use OptiX's programmable ray tracing pipeline; it uses OptiX's core operation `rtTrace` directly for locating and responding to a ray-geometry intersection (Figure 5). This practice allows for more user control of the ray generation and shading functions.

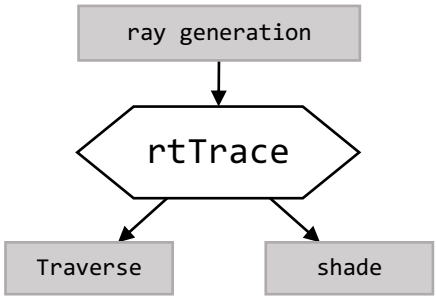


Figure 5. Call graph that shows *rtTrace* function for traversing and shading geometries using generated rays as input.

LightHouse 2's ray generation and shading functions are implemented using CUDA, which can be ported to other programming languages directly. In contrast, `rtTrace` and `Traverse` are black box functions and are not feasible for a direct port. These black box functions mostly cover acceleration structure maintenance and scene traversal.

`Traverse` is a closed-source OptiX function for locating a ray-geometry intersection using OptiX's built-in acceleration structure. A bounding box program first determines primitive bounds for object geometries which allows construction for a CUDA device optimized BVH. Because the BVH construction and traversal is specialized, we do not have to replicate the acceleration structure builder and `Traverse` with the exact behavior. Using other types of acceleration structure and traversal function technically delivers the same image result, although with performance differences. A suitable approach to replicate the functionality is by using open-source ray tracing libraries.

3. Literature Review

Software porting has long been an issue in software engineering. Typically, the software is made to run on a specific device and may not run on another. Even though there exists software that can run on different devices, there has not been any single universal environment that supports everything [7].

With the advancement of GPGPU, hardware development for GPUs is remarkably fast that it becomes impossible for GPGPU applications to support only one device type. Old devices can become obsolete within months. A solution for cross-platform compatibility between devices is required to extend software usability.

3.1 Portability Concepts

Portability is the degree of how much can software be ported. A well-formulated definition of portability, by Poole and Waite [15] is as follows:

> "Portability is a measure of the ease with which a program can be transferred from one environment to another; if the effort required to move the program is much less than that required to implement it initially to move the program is much less than that required to implement it initially, *and* the effort is small in absolute sense, then that program is highly portable."

This portability terminology is the basis of porting guidelines by Andrew S. Tanenbaum in *Guideline for Software Portability* [16]. Tanenbaum indicates that two forms of porting can be distinguished: high-level and low-level porting. High-level is porting the source code, so it compiles on the target environment, while low-level is porting hardware instructions (assembly language), so the program runs directly on the target environment. The machine architecture difference between the host and the target machine defines which porting method to use.

A different problem that is also relevant to our thesis is performance. An optimal application may become suboptimal after porting based on the target system architecture. Tanenbaum states that "optimality is not portable" although portability does not imply inefficiency.

Another portability definition is formulated by James D. Mooney [17].

> "A software unit is portable (exhibits portability) across a class of environments to the degree that the cost to transport and adapt it to a new environment in the class is less than the cost of redevelopment."

Mooney further states that software porting is creating a new executable version of a software in a new environment, based on an existing version [17]. With these definitions, Mooney also established important technical terms that are relevant for software porting [18].

- *Software unit* or *product* is the application, program, or a part of the program that is to be ported.
- *Environment* or *platform* is the designated range of systems that we are interested in porting to or from. The environment may include devices, processors, operating systems, libraries, and even networks. A *class of environments* is a collection of environments within a bounding criterion.

There are also additional quantifying terms that are important to measure portability [19].

- The *degree of portability* is the term that measures a software's quantifiable degree of portability concerning a target environment or class of environments since measuring portability is not using a binary attribute. A software with zero porting cost would be entirely portable; however, it is impossible in practice.
- *Costs and benefits* are parameters associated when porting or redeveloping. There is no absolute formulate to measure value and benefits. It varies for different porting cases.
- *Phases of porting* consists of two steps to port software from its native environment to the target environment: *transportation* and *adaptation*.
    - *Transportation* is the physical movement of the software to the target environment. It includes the software's instructions, data, and associated dependencies.
    - *Adaptation* is the modification needed for the software to operate on the target environment. The software modification can be done using automatic means or manual steps.

In principle, there are three levels of portability; *binary portability*, *source portability*, and *intermediate-level portability*.

- *Binary portability* is directly porting the executable form without any or small adjustments. This level of porting is the ideal type of portability, but it is mostly only possible for very similar environments, emulated target environment, and minimal cases.
- *Source portability* is porting the source language of the application from the host environment to the target environment. The ported source language it then recompiled to create a software version that is executable on the target environment. Source portability is the most common type of porting and is the common goal in portability research.
- *Intermediate-level portability* is a software porting representation case where it is possible to port between the source and binary code. Examples are Microsoft's Common Intermediate Language (CIL) and Java bytecode.

Mooney also stated that porting is not always desired. Some software has critical parameters that prohibit porting; when porting extends the release date, porting makes performance or efficiency drop that is not tolerable, the software is made exclusive to a specific environment, or intellectual property rights protect the software. Re-development is desirable in these cases.

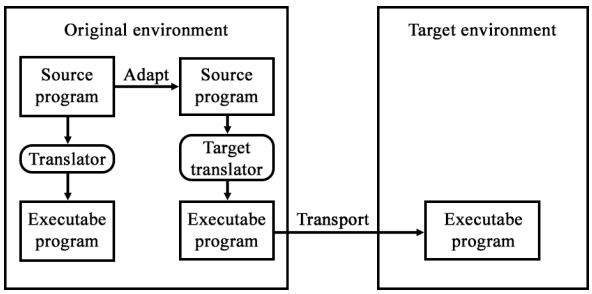


Figure 6. Porting a program with source portability for limited target environment [18].

In this thesis, our *software unit* is LightHouse 2. Our main target *class of environments* is different types of GPUs. Our desired method is porting and compiling on the host environment and be able to transport the executable across different environments (Figure 6).

3.2 Porting Cost

As a metric, porting cost is useful to rationalize design decision and trade-offs in limited porting duration. As stated before, the porting cost does not have a definitive formula. Researchers have attempted to narrow variables that affect porting cost.

In 1993, James D. Mooney formalized *degree of portability* definition [7]. This definition is limited to the source application's lines of code and work hours, omitting human experience and knowledge.

$$DP(su) = 1 - \frac{C_{port}(su, e_2)}{C_{rdev}(req, e_2)} \tag{5}$$

The function compares porting cost C_{port} and redevelopment cost C_{rdev} for a target environment e_2 with a required specification req. The host environment e_1 is only used for the software initial developing cost in which is trivial for our case. A perfect degree of portability (DP) of a software unit (su) will have $\mathrm{DP}(su) = 0$, caused by the porting cost C_{port} having a value of one (Equation 5). Based on this function, portability is cost effective if and only if $\mathrm{DP}(su) > 0$.

$$C_{rdev}(req, e_2) = C_{rdes}(req) + C_{rcod}(req, e_2) + C_{rtd}(req, e_2) + C_{rdoc}(req, e_2) \tag{6}$$

Redeveloping cost (Equation 6) is composed of *redesigning cost* (C_{rdes}), *recoding cost* (C_{rcod}), *retest and debug cost* (C_{rtd}), and *redocumenting cost* (C_{rdoc}). Redesigning does not need a target environment because it uses the same design.

$$C_{port}(su, e_2) = C_{mod}(su, e_2) + C_{ptd}(req, e_2) + C_{pdoc}(req, e_2) \tag{7}$$

Porting cost (Equation 7) is composed of the manual *modification cost* (C_{mod}), *test and debug cost* (C_{ptd}), and *documentation cost* (C_{pdoc}). To achieve the best result, we want to have an equation where the cost of modification is far less than the test, debug, and documentation costs combined ($C_{mod} << C_{ptd} + C_{pdoc}$).

In 1997, Mitsuari Hakuta and Masato Ohminami determined that porting cost is an accumulation of the *portable impediment index*, *human factors*, and *environmental factors* [20]. Human factors were not included in Mooney's equation, such that the formula by Hakuta et. Al. to calculate porting cost is significantly different.

The *portable impediment index* consists of four points; the difference in processor architecture, operating system disparity, the difference in language processor, and hardware-dependent functions. The index mostly addresses the factors between the source and target environments that determine if a software modification is necessary for porting.

Human factors are mainly tied to knowledge and experience of the software structure, the environments, software porting, the tools, and the programming language.

Lastly, the *environmental factors* are about the availability of development tools, test units, and system environments. The following regression equation is the formula to calculate porting cost Y in the form of working hours.

$$Y = C \cdot \prod_{i=1}^{n} 10^{\beta_i \alpha_i} \cdot X^{\beta_s} \quad (8)$$

In Equation 8, X is the program source code size; C, β_i, and β_s are constants from the regression analysis. The number of factors used is three, which makes $n = 3$. Lastly, α_i is the value of factor set i, where α_1 is the portability impediment index, α_2 is the human factor, and α_3 is the environmental factors.

Including human factors will give a more realistic approach for calculating porting cost, but there lies the problem of measuring human knowledge and experience without any ground truth. We will use Mooney's porting cost formula because it is more general and more straightforward to adapt in this case study.

3.3 GPGPU Development Technologies

There are several programming languages to write applications that run on the GPU. These languages, however, are mostly case-specific, vendor-specific, and proprietary. Popular languages include CUDA, OpenCL, DirectCompute, and OpenGL. We need to determine which language is the right candidate for porting a GPGPU application.

Early development of GPGPU APIs started with DirectX and OpenGL. These APIs use the traditional rendering pipeline of GPUs by doing computations in the pixel shader. To generate code, DirectX uses High-Level Shading Language (HLSL) while OpenGL uses OpenGL Shading Language (GLSL). These early GPGPU APIs are known as Compute Shade (CS). To support CS, Microsoft launched DirectCompute API for DirectX and NVIDIA released Cg.

To overcome shading language limitations, NVIDIA released CUDA in 2006. CUDA made GPGPU more versatile as it introduces many features and development flexibility. It is however limited to NVIDIA hardware. To overcome GPGPU portability, OpenCL was introduced in 2009. Apple originally developed OpenCL and is later maintained by the Khronos Group.

Later in 2015, AMD introduced GPUOpen, a GPGPU developing suite aiming for open-source and portability [21]. One of the GPUOpen tools that are relevant for cross-platform porting is Radeon Open Compute Platform (ROCm) [22]. ROCm is a collection of open-source APIs that are relevant for GPGPU development mainly targeting AMD GPUs. ROCm includes a tool called HIP that enables common C++ code to be compiled cross-platform to NVIDIA or AMD GPUs. Although still in early development, HIP provides the highest portability with the opportunity to develop once and compile to many.

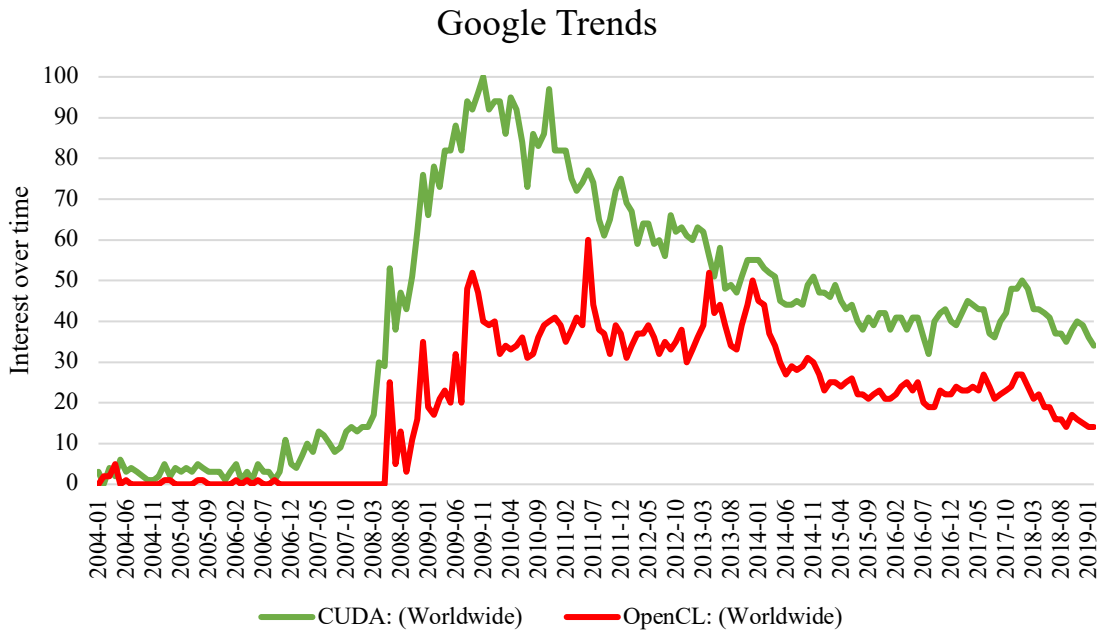


Figure 7. Comparison between CUDA and OpenCL search interest over time using Google [23]. Numbers on the Y-axis represent search interest relative to the highest point on the chart for the given region and time. A value of 100 is the peak popularity for the term. A value of 50 means that the term is half as popular. A score of 0 means there was not enough data for this term.

From these handfuls of programming languages, we are interested in CUDA and OpenCL based on popularity and features. Even with particular vendor limitation, CUDA is a very popular framework to develop GPGPU (Figure 7). Many attempts were made to port CUDA to other platforms automatically, but most are not perfect and still requires manual adaptation.

3.4 NVIDIA CUDA

NVIDIA CUDA is a C/C++ language extension that enables developers to create applications that run exclusively on CUDA-enabled GPUs. This language exceeded Compute Shade because it gives developers access to the GPU's virtual instruction set and parallel computational elements, while also providing control on memory management. CUDA can easily be utilized on any Windows, Linux, or macOS system that uses NVIDIA GPUs.

CUDA is a parallel programming language; the application executes the program simultaneously instead of only once. The language separates the code that runs on the *host* and the *device*. The *host* is the CPU that coordinates the GPUs (*device*). CUDA separates programs by their functions, called kernels. In this code sample (Figure 8), a kernel for vector addition is launched N times in parallel by different threads. Each thread is processed by one CUDA core (processing cores on an NVIDIA GPU).

```
// CUDA kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}
```

```
int main()
{
    ...
    // Kernel invocation with N threads
    VecAdd << <1, N >> > (A, B, C);
   ...
}
```

Figure 8. A simple CUDA kernel that executes vector addition in parallel.

CUDA uses NVCC (NVIDIA CUDA Compiler) to compile [24]. NVCC forwards host codes to a C compiler such as GCC, Intel C++ Compiler, or Microsoft Visual C++ Compiler. The remaining code, which is device code, is further compiled by NVCC.

NVCC can then compile device code into two different methods: offline compilation and just-in-time compilation. The offline compilation is a straightforward process. It compiles device code into PTX code or binary code.

The just-in-time compilation is a more advanced technique for low-level programming. The application uses the device driver on runtime to compile PTX code into a hardware instruction set [25]. The just-in-time method may benefit from compiler improvements on newer drivers and may also lead to device-specific optimizations or compiling for upcoming hardware.

The ray tracing engine OptiX, which is a dependency in our case study, uses NVCC's just-in-time compilation method. OptiX feeds PTX code into the application that will be compiled on runtime. The final compilation will be saved in a cache and only requires recompilation if there is a driver or hardware update. For example, applications do not need to be recompiled from source code to run on newer NVIDIA devices with the Turing architecture. The just-in-time compiled application may benefit from the device's Tensor or RT Cores, a hardware feature that was not available before [26].

3.5 OpenCL

OpenCL is language extension to C, developed to compete with CUDA [27, 4]. It aims to offer further support to a broad array of parallel hardware. OpenCL can also be used to run parallel implementations on non-GPU devices such as CPUs with SIMD (single instruction, multiple data) instructions, FPGAs (field-programmable gate array), and DSPs (digital signal processor).

Support on numerous devices makes OpenCL a programming language with high portability. OpenCL can be run on most systems as the typical consumer hardware at least utilizes a CPU that supports SIMD instruction. Because of OpenCL's portability, there is some initial overhead during setup. It must list and choose platforms first to get the desired device. The device can be automatically chosen or manually by the user.

```
// OpenCL kernel definition
__kernel void VecAdd(__global float* A, __global float* B, __global float* C)
{
    const int i = get_global_id(0);
    C[i] = B[i] + A[i];
}
```

Figure 9. A sample of OpenCL code of the same function as the CUDA code in figure 2. OpenCL uses the same structure with some different function types and data types.

OpenCL language structure and architecture are relatively similar to CUDA (Figure 9). It uses an equivalent representation of *device*, *host*, and *kernel*.

OpenCL kernel code is compiled on runtime because it is platform-specific. Host code handling is the same as in CUDA; it is compiled using the corresponding compiler. In our case, the host code written in C++ uses GCC, Intel C++ Compiler, or Microsoft Visual C++ Compiler.

Currently, OpenCL's latest version is 2.2. New functions are introduced to enhance parallel programming. Unfortunately, not a wide selection of devices supports OpenCL 2.2. Most devices still use the OpenCL 1.2 standard, which makes this version the lowest common denominator.

3.6 HIP

HIP (Heterogeneous-compute Interface for Portability) is a portable C++ programming environment for the GPU developed recently by AMD. This environment enables users to develop in a language that can be compiled seamlessly to both AMD and NVIDIA GPUs. HIP aims to reduce or even remove manual source porting.

HIP was developed because most GPGPU applications are still bound to the CUDA language and structure. CUDA, as a proprietary language, is closed-source and is maintained only by NVIDIA. It limits the user's freedom of using CUDA language on non-NVIDIA devices. HIP's goal is to provide an open-source solution to convert CUDA code to common C++. The converted code can then be compiled with NVCC or HCC (AMD's Heterogeneous Compute Compiler).

```
cudaMalloc((void **)&m_cuda, Size * Size * sizeof(float));
cudaMemcpy(m_cuda, m, Size * Size * sizeof(float), cudaMemcpyHostToDevice);
gpu_kernel<<<dimGridXY, dimBlockXY>>>(m_cuda, a_cuda, b_cuda, Size, Size - t, t);
cudaThreadSynchronize();
cudaMemcpy(m, m_cuda, Size * Size * sizeof(float), cudaMemcpyDeviceToHost);
cudaFree(m_cuda);

__global__ void gpu_kernel(float *m_cuda, float *a_cuda, float *b_cuda, int Size, int j1, int t)
{
    …
}
```

```
hipMalloc((void **)&m_cuda, Size * Size * sizeof(float));
hipMemcpy(m_cuda, m, Size * Size * sizeof(float), hipMemcpyHostToDevice);
hipLaunchKernel(gpu_kernel, dim3(dimGridXY), dim3(dimBlockXY), 0, 0,
                                    m_cuda, a_cuda, b_cuda, Size, Size - t, t);
hipDeviceSynchronize();
hipMemcpy(m, m_cuda, Size * Size * sizeof(float), hipMemcpyDeviceToHost);
hipFree(m_cuda);

__global__ void HIP_FUNCTION(gpu_kernel, float *m_cuda, float *a_cuda, float *b_ cuda, int Size, int j1, int t)
{
    …
}
HIP_FUNCTION_END
```

Figure 10. A sample conversion from CUDA code (above) into HIP code (below).

The HIP approach is more practical and less verbose than source porting CUDA code into OpenCL. The reason is that HIP and CUDA are C++ languages that share a similar language structure which can be seen in Figure 10. The language similarities as enable CUDA developers to program in HIP C++ with ease. On an NVIDIA environment, Nsight (NVIDIA's GPU profiler) can even be used to profile and debug.

Although HIP can support many devices, currently operating system support is limited to the Linux environment. The reason why it has not been ported to other operating systems is that ROCm (the software suite that includes HIP) kernel driver is highly tied to Linux and the HSA (Heterogeneous System Architecture) runtime is highly tied to the driver. Currently, porting LightHouse 2 with HIP is not a feasible option because of this limitation.

3.7 Automatic Porting

Automation tools were created to port CUDA to OpenCL because manual porting is a very tedious and time-consuming task. A noteworthy source-to-source porting tool is CU2CL. Another approach is to port CUDA to an intermediate language automatically (e.g., HIP) that can be compiled on the OpenCL environment.

3.7.1 CU2CL

CU2CL is a framework that was made to fully automate source code porting from CUDA to OpenCL [28]. The framework translates most commonly CUDA API calls so that few lines need to be ported manually after the process. This tool, however, is limited only to the Linux environment and can only serve as an offline source translator. Our best option is to port CUDA codes to OpenCL manually.

3.7.2 Hipify

Hipify is a tool that can automatically detect and port CUDA code into HIP code. On a case study made by AMD [29], the tool was able to port 99.6% code automatically. The software unit was CAFFE, a CUDA machine learning framework with more than 55,000 line of C++ code, more than 70 CUDA kernels, and uses NVIDIA's proprietary neural network library CUDNN. The resulting HIP code was able to be offline compiled on both AMD and NVIDIA GPUs.

3.8 Ray Tracing API

As demand increases for ray tracing development, commercial GPU vendors designed ray tracing APIs that can be used on personal computers. NVIDIA released OptiX in 2009. Later in 2013, Intel released Embree, a ray tracing engine optimized for Intel CPUs [30]. AMD finally released FireRays in 2015. A year later, FireRays became one of AMD's GPUOpen tools and was renamed into Radeon Rays [31]. These APIs are interactive and are not based on the shading language. In the search for a robust open-source GPGPU ray tracing engine to solve the closed-source part in our case study, Radeon Rays is currently the viable option.

3.8.1 Radeon Rays

Radeon Rays is AMD's ray tracing API that is developed with portability in mind. The API is implemented using C++ based on OpenCL version 1.2. Radeon Rays is optimized and guaranteed to perform on AMD devices, but it can also be used on other environments as long it complies the OpenCL 1.2 standard. Using OpenCL 1.2 means that NVIDIA and Intel devices can benefit from this API.

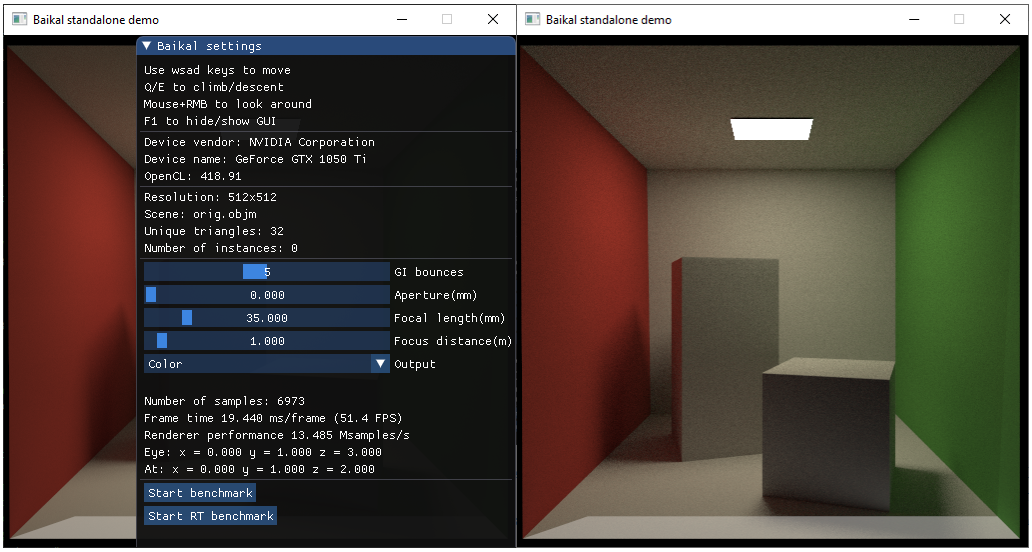


Figure 11. Baikal using Radeon Rays to render a Cornell Box using an NVIDIA GeForce GTX 1050 Ti GPU.

Radeon Rays demonstrates its ray tracing capability using Baikal, a global illumination renderer. The renderer runs out of the box on an NVIDIA device (Figure 11) and does not require changing or installing additional components.

Radeon Rays features an accelerated GPU intersection library. The library uses massive batches of rays as input to hide latency and increase GPU occupancy. It has two types of intersection queries; closest hit and any hit. Closest hit triggers when there is an intersection. Any hit returns an array of integers (1 for intersection and -1 for no intersection) that is useful for terminating shadow rays and occlusions.

Radeon Rays also supports acceleration structures, with Bounding Volume Hierarchy (BVH) as the default structure. The default BVH uses spatial median splits as it has fast built times while performing decent intersection performance. To improve intersection performance, an option to enable Surface Area Heuristic (SAH) is available. SAH will increase BVH build time in trade for a fast intersection on runtime. To support scenes with moving geometries, two-level BVH is also available to use.

The Radeon Rays SDK has requirements that almost resembles OpenCL.

- A Linux, Windows, or macOS working environment.
- A device that is capable of running OpenCL version 1.2.

Radeon Rays satisfies the target environment that we are aiming for, and it is also a strong candidate to replace OptiX functions that are specific for NVIDIA devices. The downside of using Radeon Rays is that the API documentation is scarce and not well established compared to OptiX. There may be bugs and glitches on devices that were never tested before. These problems may cause overhead and increase the porting cost if not treated carefully.

4. Research Methodology

4.1 Porting Implementation

In this section, we explain the port core implementation method. We use OpenCL to port CUDA kernels and RadeonRays version 2.0 as a replacement for OptiX functions. Radeon Rays version 2.0 is the latest version that can work directly with OpenCL. By using Radeon Rays means that we do not have to manage the acceleration structure ourselves and can focus on shading the scene. We also chose to use OpenCL version 1.2 as this is the highest version that is supported on most devices. The source core from LightHouse 2 that we use is the CUDA core that uses OptiX Prime, a variant of OptiX that does not require NVIDIA RTX hardware.

By using these porting methods, the degree of our porting implementation is considered as source level. The implementation is used to carry several experiments that can answer our proposed research questions. We conduct three different kinds of experiments: portability, features, and performance.

4.2 Portability Experiment

The experiment in this section aims to answer the first research question. Since porting does not have an exact metric measurement, data must be gathered from the porting implementation. Once the OpenCL core base is implemented, we measure porting of LightHouse 2 exclusive features.

Since OpenCL has a similar structure to CUDA, we measure source portability by making code comparison between version. Difference between the modified line of codes with the source line of codes indicates the porting effort.

In the test, we run the ported application on OpenCL 1.2 enabled devices. These devices include NVIDIA GPUs, AMD GPUs, Intel CPUs, and Intel integrated graphics. A perfect port would give the same rendering output and does not need any adjustments to run on these devices.

We also conduct a source code statement experiment by enabling automatic function switching based on the available device (CUDA or OpenCL). This approach is for testing maintainability for source code changes.

4.3 Features Experiment

To answer the second research question, we must verify if the OpenCL core has similar functionality as the source render core. We use three configurations for this test. First, we run the OpenCL core and the source core on the same machine so we can get an accurate core comparison. Second, we run the OpenCL core on different GPUs to check if different devices produce different results. Lastly, we run the source Core on an NVIDIA device and compare it to the render from the OpenCL core on a non-NVIDIA device.

We set up scenes that are identical across cores. These scenes include SEED's PICA PICA Warehouse and Crytek's Sponza. Scene camera viewports are set with the same point and pointing direction across cores. We then render images between fixed frame count intervals; 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, and 1024 frame samples. Real-time measurements for features do not yield the desired results because there are

converging variances between cores. The cores may not be in the same rendering state after running for a specified period.

$$\text{MSE} = \frac{1}{mn}\sum_{i=0}^{m-1}\sum_{j=0}^{n-1}[I(i,j) - K(i,j)]^2 \tag{9}$$

Images rendered between cores are compared using the Mean Squared Error (Equation 9). This measurement method objectively quantifies the strength of the error signal. The resulting value indicates how different are the pixel at position (i, j) of image I compared image K (the images must have the same dimension $m \times n$ pixels). Two identical images will have an MSE value of 0 and will increase as the difference between the pixel intensities increase.

Because Mean Squared Error (MSE) only measures absolute error, we also have to use the Structure Similarity (SSIM) method to measure the perceived differences observed by the human eye [32].

$$\text{SSIM}(x, y) = \frac{(2\mu_x\mu_y + C_1)(2\sigma_{xy} + C_2)}{(\mu_x^2 + \mu_y^2 + C_1)(\sigma_x^2 + \sigma_y^2 + C_2)} \tag{10}$$

The SSIM method attempts to measure the structural information from two images. The method (Equation 10) is based on a weighted combination from the results of comparing the luminance, contrast, and structure. The term μ is the average of an image, σ is the variance an image, and C as a variable to stabilize the equation. An SSIM index of 1 means that the images have perceived identical structures. The score will decrease the more structural information degrades.

The same feature tests are also be conducted on devices from different manufacturers. Testing on various devices gives us insight into how the port behaves on non-NVIDIA devices. With this experiment, we can deduce whether the OpenCL port is a head to head representation of the source core. We can also understand what features are supported in the OpenCL version.

4.4 Performance Experiment

This experiment tries to answer the third research question. Our OpenCL core is performance tested against the source render core. We run the main experiment on the same system environment such that we eliminate hardware configuration differences. We also run an additional experiment on a variety of devices, which let us know the core performance proportional to the device's computing capability.

Render performance is measuring the time it takes for a core to render a frame. In this render time, we separate it into trace and shade time. Trace time is the amount of time for the core's ray tracing engine to traverse a scene and return the intersections. Shade time is the time for the core to process the intersection data until it returns the pixel's color. During tracing and shading, we also measure the number of processed rays during both steps.

5. Implementation

In this chapter, we explain the procedures for implementing the OpenCL core. The following section also explains solutions to problems that we found during porting. Section 5.1 defines the core device selection process. Section 5.2 describes data types and data structures that are different between the source language and the target language. Section 5.3 contains information of the interoperability between the ray tracing engine and OpenCL. In Section 5.4, we explain scene data preparation for the target core. Section 5.5 discusses the shading process, covering ray generation and intersection handling. Section 5.6 provides persistent thread implementation in the OpenCL kernels.

5.1 Device Selection

Before a core can start its rendering routine, a device must be selected for the program to run on. Selecting a device on the CUDA core is rather straightforward because it is only limited to NVIDIA devices. The core selects the device with the highest floating-point operations per second (FLOPS). Calculating FLOPS requires access to three important information regarding the candidate devices; the number of Shading Multiprocessor, number of threads per Shading Multiprocessor (SM), and the clock frequency. The number of SM and the clock frequency can be easily retrieved by doing a device information lookup through CUDA. The CUDA core also implicitly store information about the number of threads per SM.

$$\text{FLOPS} = \text{SMs} \times \frac{\text{threads}}{\text{SM}} \times \text{clock frequency} \tag{11}$$

The fastest CUDA device can then be determined using Equation 11. This equation does not give the true FLOPS count but is sufficient enough for device selection.

Our target core, on the other hand, can run on a wide variety of hardware that is not limited by the hardware manufacturer. The device type is also not restricted to GPUs only as OpenCL can also utilize CPUs and FPGAs. The ability to use a wide range of hardware introduces the problem of selecting the optimal device for the core to run on as a system may contain several capable devices. OpenCL can retrieve a device's clock frequency and the number of Compute Units (equivalent to Shading Multiprocessors on NVIDIA devices). Unfortunately, it cannot look up information about the number of threads per Compute Unit. FLOPS thus cannot be calculated using Equation 11. In an example, a GPU and a CPU with the same number of Compute Units may not have the same thread count. Even in some cases, a CPU might have a higher clock frequency than a GPU.

To overcome the device selection limitation, we opt to prioritize device by type and manufacturer. A GPU will have more priority than a CPU and device manufacturers are prioritized based on the most common device available in the market; NVIDIA, AMD, to Intel. Priority sorting ensures that the core selects the fastest device on systems that have more than one GPUs such as discrete and integrated GPUs. Although this device selection solution should work in general cases, it is not robust as there can be cases where the lower priority manufacturer may have better performance if a system has both (i.e., a combination of NVIDIA and AMD discrete GPUs in one system).

When a device is successfully selected, an OpenCL context object will be created using the device's properties. The device-specific context object handles memory, program, and kernel objects and acts as the

entry point to run kernels on the device. An OpenCL command queue is also created based on the device to ensure that the kernels are correctly running in order.

5.1.1 OpenCL-OpenGL Interoperability

Because LightHouse 2 uses an OpenGL texture to display the rendered frames, it is significant for the selected device to have interoperability between OpenCL and OpenGL. Having interoperability means that the device can directly access and modify the OpenGL texture through OpenCL to avoid device-to-host copy latency.

If OpenCL-OpenGL interoperability is available, OpenCL can directly control the OpenGL texture provided by the render system using the `clCreateFromGLTexture` function. The texture's ownership is then able to be passed between the APIs with zero-copy latency as the actual data stays at the same memory location.

In the case where the OpenCL-OpenGL interoperability is unavailable, the rendered image on the device are copied over to host memory first and then to the OpenGL texture every frame, which causes significant performance degradation. An example where interoperability is not available is when the OpenGL texture resides in a physically different device than the OpenCL device selected.

5.1.2 Use of Persistent Threads

Another essential aspect that is determined by the device selected is the use of persistent threads (see Section 5.6). Persistent threads can only be enabled when the device's Compute Unit has more than one thread physically. Because the OpenCL cannot determine the actual number of threads per Compute Units, our solution is to enable persistent threads only on GPUs that are manufactured by NVIDIA and AMD because their Compute Units contains multiple of 32 threads. Integrated GPUs manufactured by Intel does not share the same thread count multiplication and does not benefit from persistent threads because the total number of threads is low.

5.2 Data Structure

Buffers are classes that stores data elements in various data types and structures on the host or the device. When these buffers are stored on the device, they can pass be passed between kernels without having to be copied over first to host memory.

By referring to the source core, buffers for CUDA are encapsulated in the `CUDABuffer` class which contains a pointer to host data and another to device data. Because CUDA uses pointers to represent both host and device data, the logic of the pointers is shared between the host functions and CUDA kernels.

We port the buffer class to the target core using a similar buffer structure called `CLBuffer` as we still separate data between host and device. Instead of using pointers for device data, OpenCL uses buffer objects. Buffer objects are one-dimensional arrays that are allocated on the device.

```
// location is ON_DEVICE; allocate room on device
CUDACHECK("cudaMalloc", cudaMalloc(&devPtr, sizeInBytes));
owner |= ON_DEVICE;
```

```
// location is ON_DEVICE; allocate room on device
cl_int err = CL_SUCCESS;
delete devPtr;
devPtr = new cl::Buffer(RenderCore::context, CL_MEM_READ_WRITE, sizeInBytes, NULL, &err);
CLCHECK("Buffer", err);
owner |= ON_DEVICE;
```

Figure 12. Allocating a buffer in the CUDA core (above) and in the OpenCL core (below).

Initialization of OpenCL buffer objects are similar to CUDA device pointers, requiring a device (implicit in CUDA, explicit in OpenCL using OpenCL context) and an array size based on the number of elements times the size of the data type (Figure 12). Additional flags are also required to give buffers access permissions. In our implementation, we allow buffer modification by the device using the `CL_MEM_READ_WRITE` flag.

A slight disadvantage of using objects instead of pointers for buffers is the inability to offset kernel input directly. We have to pass additional function parameters for offsetting buffer objects when calling an OpenCL kernel.

```
generateEyeRays(SMcount, extensionRayBuffer[inBuffer]->DevPtr(), extensionRayExBuffer[inBuffer]->DevPtr(),
    blueNoise->DevPtr() + blueSlot * 65536, RandomUInt(camRNGseed), (vars.filterEnabled ? 0 : -5), 0, samplesTaken,
    jitteredView.aperture, jitteredView.pos, right, up, jitteredView.p1, GetScreenParams());
```

Figure 13. Call to a CUDA kernel that passes buffers using device pointers. In this function, the `blueSlot` variable is different on every call which modifies the `blueNoise` pointer sent.

An example in the source core that uses offset when passing buffers can be found when calling the kernel that generates primary rays (Figure 13). In this kernel call, we pass a blue noise buffer that uses a different offset on each call.

5.2.1 Data Alignment

Another problem that we encounter when implementing data structures in the target core buffer is data alignment. The problem arises when we initialize a buffer with a data type that has a different structure between the host and the device.

In the source core and target core, `float3` data type in the host code is a structure with three `float` members that are tightly packed (three times 4 bytes). CUDA share the same packed `float3` structure in the device code, but OpenCL does not and will cause misalignment.

data[0].x	data[0].y	data[0].z	data[1].x	data[1].y	data[1].z	data[2].x	data[2].y	float[2].z
read[0].x	read[0].y	read[0].z	read[0].w	read[1].x	read[1].y	read[1].z	read[1].w	read[2].x

Figure 14. The top row is how the host initializes a `float3` array. The bottom row is how OpenCL accesses the array. Some data are skipped on access and reading the third `float3` members returns an error as it exceeds the array bounds.

A `float3` buffer initialized in our target core will yield the correct packed array. However, on the device, some data will be skipped and unavailable because it accesses the buffer as a `float4` array. By default, OpenCL assumes every `float3` structure as a `float4` for data access performance. This assumption is made to satisfy the size of the device's memory cache that is `float4` aligned and reduces the chance of cache misses on access [33]. A `float3` array is not preferable, as most of the time, one or two members of the data type may not be loaded on the cache. The problem is illustrated in Figure 14.

```
skyPixelBuffer = new CUDABuffer<float3>(width * height, ON_DEVICE, pixels);
skyPixelBuffer = new CLBuffer<float>(3 * width * height, ON_DEVICE, (float*)pixels);
```

Figure 15. Initializing a `float3` array on the CUDA core with the size of the screen's resolution (above) and initializing a `float` array on the OpenCL core with the size of three times the screen's resolution (below).

A workaround for this problem is to initialize `float3` buffers using `float4` for the data type. Using a `float4` array gives faster data access on the device but increases the size of the array by 33%. If the buffer has to be tightly packed (for example the size of the array is large), we can change the buffer from a `float3` data type into a `float` with three times the original array size. With this solution, OpenCL can safely access the `float` array without misalignment on the device side.

5.2.2 Half Data Type

Half-precision floating-point (`half`) is a 16-bit `float` data type. The primary usage of half-precision data type in LightHouse 2 is to store decimal values that do not require high precision compactly. On the source core, CUDA can support loading, storing, and arithmetic calculations with half-precision floating points by including the `cuda_fp16.h` header [34].

By default, OpenCL only allows `half` data type for storage and does not allow for `half` data type variables to be processed in arithmetic operations on the device. Half-precision data can only be initialized on the host when creating a buffer. Processing `half` data can be supported if the OpenCL device has the `cl_khr_fp16` extension enabled from its driver [33].

```
const float blue = __ushort_as_half(part1 & 0xffff);
```

Figure 16. Loading a `half` type using masking and reinterpretation in the CUDA core.

A case where a function requires `half` variables is available in the source core's material processing kernel (Figure 16). Diffuse material color is stored as a half-precision floating-point on the host to reduce the material structure size. The source core kernel loads the diffuse color data as a 32-bit unsigned integer to avoid a cache miss. The loaded data is then masked to get the 16-bit unsigned integer and reinterpreted back as `half` by using the `__ushort_as_half` function.

The same function to reinterpret data as `half` is also available in OpenCL under the function `as_half`. Procedure to load `half` type data can theoretically be applied using the exact steps as the source core if the device has the `cl_khr_fp16` extension enabled. However, this extension is mostly not available on consumer-grade NVIDIA or Intel hardware running on Windows operating system, even if the hardware is actually capable (available in recent AMD devices). The unavailability of reduced hardware extension on NVIDIA devices is under the suspicion that the corporation is pushing developers to use CUDA instead of OpenCL [35].

```
const float blue = vload_half(0, &materials[TRI_MATERIAL].diffuse_b);
```

Figure 17. Loading a `half` type using an intrinsic function in the OpenCL core.

Instead of replicating the same procedure as the source core, we directly load the half type data without reinterpreting it into a different format (Figure 17). We use `vload_half` to load and losslessly convert the `half` type data into a `float`. To store the data into `half` type, we use the function `vstore_half`, which converts `float` into `half` type data with denormalization to reduce 32-bit precision into 16-bit precision. Handling `half` data type may incur some performance loss in the OpenCL core because of type conversions, whereas in the source core, data is only bit shifted, masked, and reinterpreted.

5.2.3 Buffer with Device Pointers

For easy scene management, the source core initializes an instance descriptor buffer that contains the collection of all the geometry instances. The core initializes the CUDA buffer on render time such that it reflects scene changes on the next render frame. Each member of the CUDA buffer contains the pointer to the instance's triangle CUDA buffer and the instance's three-dimensional transformation. Two dummy variables are provided to avoid having a cache miss on access. Having a CUDA buffer that contains another CUDA buffer is possible because, by definition, a CUDA buffer is a pointer to a memory part on the device [36]. Based on the pointer characteristic of a CUDA buffer, the kernel on the device can access a buffer inside a buffer.

OpenCL buffers, on the contrary, are memory objects on the host [33]. These objects point to device memory but cannot be used directly as pointers. Thus, an OpenCL buffer inside an OpenCL buffer will not be recognized when accessed by the kernel on the device. The proper way to pass an OpenCL buffer to a kernel is by using the `clSetKernelArg` function (encapsulated as `Kernel::setArg` by the OpenCL C++ wrapper) when initializing the kernel from the host.

Workaround for this data structure problem is by avoiding buffer pointers inside buffers by directly appending the pointed buffer or by using other methods such as creating an additional buffer with a continuous array and offsets. Both of these solutions will slightly reduce performance because there are actual data moved between buffers on initialization.

```
struct GPUInstanceDesc
{
    GPUTri4* triangles;           // device pointer to model triangle array
    int dummy1, dummy2;           // padding; 80 byte object
    float4x4 invTransform;        // inverse transform for the instance
};
```

```
struct GPUInstanceDescCL
{
    // OpenCL buffers cannot use device pointers. Instead we use
    a continuous array of triangles with mesh offsets.
    int mesh;                     // mesh id of the instance
    int triangleOffset;           // global triangle array offset index
    int dummy;                    // padding; 80 byte object
    struct float4x4 invTransform; // inverse transform for the instance
};
```

Figure 18. GPU Instance descriptor structure of the CUDA code (above) and the OpenCL code (below).

One visible example of the problem lies within the instance descriptor buffer. This buffer includes pointers to mesh triangle buffers, a structure that we cannot use in OpenCL. To overcome this problem, we create a triangle descriptor buffer. The triangle descriptor buffer will contain a continuous array of triangles from all the meshes. For the instance descriptor buffer, we will pass the mesh identification number, the three-dimensional transformation, and the triangle offset. The triangle offset points to the start of the mesh triangle in the continuous triangle array.

5.2.4 Pass-by-Reference

Pass-by-reference is a way to pass a variable reference onto a function call parameter. The function can thus access and modify the variable's value as if it is the function's local variable. Referencing is handy to avoid unnecessary data copies between functions as the actual location of the variable is kept at the same memory address (only the values are changed). The practice of pass-by-reference is substantial throughout the source core kernels as CUDA can utilize this method.

OpenCL C is based on the ISO/IEC 9899:1999 C language specification where one of its specifications is that a function cannot change the actual parameters value [37]. The limitation drives us to simulate function pass-by-reference because we want to keep the ported function as close as possible to the source core. By using pass-by-reference, we avoid unnecessary data movement between functions that could lessen performance.

```
H2_DEVFUNC void GetShadingData(
    …
    ShadingData& retVal,               // OUT: material properties of the intersection point
    float3& N, float3& iN, float3& fN, //      geometric normal, interpolated normal, final normal (normal mapped)
    float3& T,                         //      tangent vector
    const float waveLength = -1.0f     // IN:  wavelength (optional)
)
{
    …
    ShadingData4& retVal4 = (ShadingData4&)retVal;
    retVal4.data0 = make_float4(redgreen.x, redgreen.y, blue, (float)(etarough0 >> 8) * (1.0f / 255.0f));
    …
    N = iN = fN = TRI_N;
    T = TRI_T;
    …
}
```

```
LH2_DEVFUNC void GetShadingData(
    …
    struct ShadingData* retVal,        // OUT: material properties of the intersection point
    float3* N, float3* iN, float3* fN, //      geometric normal, interpolated normal, final normal (normal mapped)
    float3* T,                         //      tangent vector
    const float waveLength,            // IN:  wavelength (optional)
    …
)
{
    …
    struct ShadingData4* retVal4 = (struct ShadingData4*)retVal;
    retVal4->data0 = make_float4(red, green, blue, (float)(etarough0 >> 8) * (1.0f / 255.0f));
    …
    *N = *iN = *fN = TRI_N;
    *T = TRI_T;
    …
}
```

Figure 19. Comparison between the same function that uses pass-by-references in CUDA C (above) and OpenCL C (below).

When tried using the same code structure as the source core on target core, the kernel will not compile as the OpenCL compiler does not recognize the syntax. Passing a reference, however, can be simulated using pointers. Pointers (address of the variables) are passed on the function parameters and dereferenced within the function which allows the function to read and write the actual variable, similar to passing by reference.

5.2.5 Constant Variables and Buffers

Constant data is a variable or a buffer that is shared between kernels running on the same device. There is a big difference in how CUDA and OpenCL handle constant data. Constant data on CUDA can be changed on runtime, whereas in OpenCL, constant data is hardcoded on the device and cannot be modified from both the host and device. Inability to change constant data on runtime possesses slight trouble when passing a lot of the same variables and buffers to different kernels in OpenCL.

We solve this problem by adding constant data parameters to every kernel. Because a kernel must have its function parameter to pass constant data, a kernel call can quickly become cluttered with the number of parameters needed. For now, this is the most straightforward approach that does not require kernel structure changes.

5.2.6 Fixed-point Arithmetic

Functions that contain fixed-point arithmetic are problematic when tested on AMD and Intel hardware. On AMD, OpenCL fails on program compilation without any warnings, and on Intel, the core produces shadows with strange patterns. This problem is caused when operation between a 4-byte variable with an 8-byte variable has a different numerical structure.

The case that we have found in our port is the multiplication of a `float` with an `unsigned long`. This problematic operation can be found in the function that returns the random barycentric coordinates in `lights.cl`. The function uses triangle sampling that only needs one random `float` number as the input [38]. The random number is then converted using fixed-point arithmetic to get an integer sample. Because this uses the aforementioned different byte variable multiplication, the OpenCL program becomes uncompilable on AMD devices and problematic on Intel devices.

As trying to change fixed-point calculation gives us unintended render output, we opted to change using a recent triangle sampling method [39]. This method is fast and does not require any fixed-point calculation; however, it does not produce results as accurate as of the source core. The difference is mostly visible on the softness of the shadows (Figure 20).

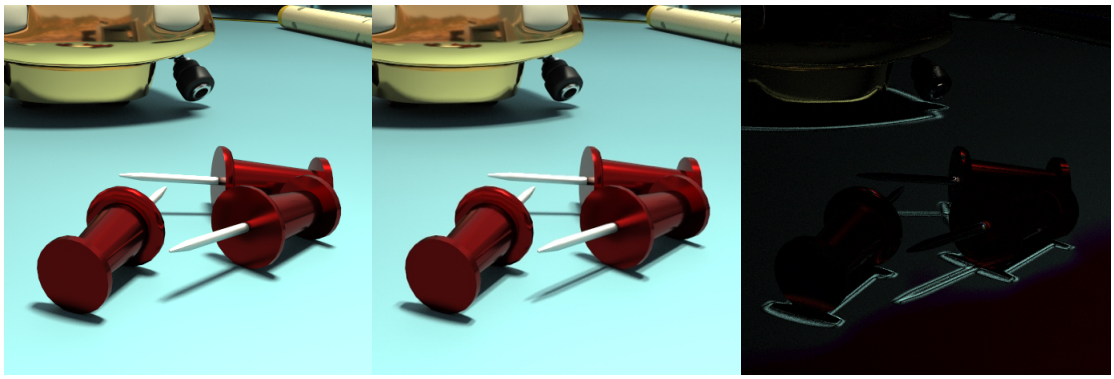


Figure 20. Result of shadows using different random barycentric coordinate methods. From left to right: Render using the source method, render using the alternative method, and pixel differences (intensified five times) between the methods.

We enable switching to the alternative random barycentric coordinate method when the port is run on AMD, Intel, or other non-NVIDIA devices. Method switching ensures high compatibility between devices while having the same behavior when the port is run on NVIDIA devices.

5.3 Ray Tracing Engine

The ray tracing engine is the API that allows geometry intersection calculation by using ray tracing methods. In the port, we use Radeon Rays as our intersection API to substitute the closed-source ray tracing engine OptiX. The engine must first choose a device that is capable of performing intersection queries on initialization. The available device listing can be processed directly through the intersection API or can be manually assigned using the already initialized OpenCL context. Using the automatic approach will create a new OpenCL context for the API to run. We do not want to have another OpenCL context as we have already initialized the context through our device selection method on core initialization.

```
// setup Radeon Rays
EXPECT_NO_THROW(api = RadeonRays::CreateFromOpenClContext(context(), device(), queue()));
```
Figure 21. Initializing Radeon Rays using the initialized OpenCL context, device, and command queue.

Our approach is using the manual device selection for the intersection API. The intersection API requires the initialized OpenCL context, the chosen device, and the command queue. Using the correct device ensures that the intersection API has interoperability with OpenCL. The interoperability is vital because Radeon Rays must access rays and intersection data that are stored in OpenCL buffers.

```
ASSERT_NO_THROW(extensionHitsDesc = RadeonRays::CreateFromOpenClBuffer(api, extensionHitBuffer->DevPtr()()));
ASSERT_NO_THROW(shadowRaysDesc = RadeonRays::CreateFromOpenClBuffer(api, shadowRayBuffer->DevPtr()()));
ASSERT_NO_THROW(shadowHitsDesc = RadeonRays::CreateFromOpenClBuffer(api, shadowHitBuffer->DevPtr()()));
```
Figure 22. Initializing Radeon Rays buffers using existing OpenCL buffers to enable interoperability.

Radeon Rays uses a different type of buffer; however, the buffers can handle interop if initialized from an existing OpenCL buffer. This allows Radeon Rays to assume control of the OpenCL buffer and make changes without any actual data movement.

5.3.1 Ray Structure

The ray tracing engine's primary purpose is to process rays. These rays primarily contain the vector origin and direction data. The main difference between the source and the target core is the ray structure as the structure is bound to the engine. Additional data may also be stored on the ray depending on the intersection requirement.

```
struct Ray { float3 O; float tmin; float3 D; float tmax; };
struct Ray { float4 o; float4 d; int2 extra; int doBackfaceCulling; int padding; };
```
Figure 23. Ray structure of OptiX (above) compared to Radeon Rays (below). OptiX only takes 32 bytes for a ray while Radeon Rays needs 48 bytes because of the extra data.

The source core uses a ray that contains the origin, the direction, and the distance to the hit geometry. This structure is one of the predefined ray structures from OptiX with the size of two `float4` members.

Radeon Rays currently only has one ray structure, containing extra data for its intersection calculation. Because of the extra data, one ray uses three `float4` members. By these definitions alone, the target core needs more storage for a single ray thus can be deducted that ray read and write performance might be lower than of the source core.

5.3.2 Intersection Structure

Once the ray tracing engine has processed a ray, it returns an intersection. Data in an intersection contains the hit distance, hit geometry, the triangle in the hit geometry, and the barycentric coordinates of the said triangle. All of these data determine the shading path taken.

```
struct Intersection { float t; int triid, instid; float u, v; };
struct Intersection { int shapeid; int primid; int padding0; int padding1; float4 uvwt; };
```
Figure 24. The intersection data structure in the CUDA core (above) and the OpenCL core (below).

As with the case of the ray structure, the intersection structure is also different between ray tracing engines. Intersection data on the source core, based on OptiX, contains the distance, hit triangle, hit instance, and the barycentric coordinates. By having five data members, OptiX intersection data takes 20 bytes of storage.

Radeon Rays intersection data has the same members but with padding for data alignment purposes. Padding for alignment makes the intersection data in the size of two `float4` members (32 bytes). Because

of the more significant storage needed for the intersection data, the target may have lower performance than the source core.

5.4 Scene Geometry Loading

Once the core initializes all the required APIs, the next step is to prepare scene data to a format that is usable by the core. LightHouse 2's rendering system loads all the necessary files to create a scene to the host. In the current state, LightHouse 2 support loading wavefront object (.obj), GL Transmission Format (.gltf), and partial loading for FBX format. The rendering system then passes the loaded scene onto the render core in a universal format regardless of the original file format or the core.

Data classes that are loaded into the rendering system includes the skydome, textures, materials, meshes, instances, and lights. All of these data classes are stored in the host scene with each having their separate list. On a core initialization, the rendering system sends all the host scene data to the core that allows the core to hold its representation of the scene data. The rendering system is also capable of detecting changes in the host scene that automatically sends a synchronization flag to the render core. The flag enables the rendering system to resend modified scene data, such as mesh deformation and instance changes, are up to date on render time.

By using the same principle of preparing scene data as the source core, each of the data class is handled by its function in the target render core. These functions convert host scene representation to core-specific data. Functions to convert the skydome, textures, materials, and lights are straightforward ports from the source core with minimal changes. Most of the changes are made to satisfy the OpenCL data structure that is explained in Section 5.2. Meshes and instances are treated a bit different because it uses the Radeon Rays geometry API calls.

Radeon Rays uses shape objects to represent geometry in the scene. For each geometry object, the rendering system provides an array of vertices, the number of vertices, the number of triangles, and an array that contains each triangle data such as materials, textures, and normals.

```
shape = RenderCore::api->CreateMesh(buffers.vertices, vertexCount, 3 * sizeof(float), buffers.indices, 3 * sizeof(int),
nullptr, triCount);
```

Figure 25. Generating a shape using Radeon Rays API call.

The host vertices array that is sent from the rendering system is a `float4` array, which means that each member of the array contains four `float` data. Only the first three members contain actual data while the fourth member contains a dummy. We compact the vertices array to reduce the size by discarding the dummy. The compacted array will be the input to the Radeon Rays shape call (Figure 25). We avoid using a `float3` array because Radeon Rays uses OpenCL as its basis; the data type alignment problem explained in section 5.2.1 can occur as well. We chose to use a `float` array that has three times the members of the host vertices.

Radeon Rays shape uses an index system to point vertices of a triangle. The rendering system expands all the vertices into a continuous vertex array sorted correspondingly to the triangle array order. The required indices list is initialized as an integer array containing dummy data; containing elements from zero to three times the number of triangles.

Additional data that must be provided to the shape API is the array strides and the number of vertices per face. Array strides allow the shape API to know the required array elements for one triangle. In our case, strides of both arrays are three times the element. The shape API needs the number of vertices per face because it can accept a geometry in a mix of quad and triangle meshes. As the mesh sent from the rendering system contain only triangles, we set the number of vertices per face as three.

Once a shape has been initialized, the shape can be directly committed to the scene or use instancing to allow multiple entities of the same mesh without creating a new shape object. We follow the same logic as the source core when applying geometry into the scene, sending only instances from the original geometry. A transformation is also applied when geometry instances are committed to the scene, enabling copies of the mesh with different position, orientation, and scale.

5.5 Rendering

After a core has successfully processed all the host scene data, the rendering routine will start. The routine consists of preparing on-render buffers, call to a kernel that generates the primary rays, geometry intersection using the ray tracing engine, shading, and connecting traced shadow rays. All of these steps are considered as one render cycle that produces a frame with one sample per pixel. Between the source core and the target core, only a few structural designs were changed.

5.5.1 On-render Buffer Initialization

The source core's rendering function starts by initializing an instance descriptor buffer that collects all the committed geometry instances. The purpose of this buffer is to give instance shading information on the intersected triangle. The buffer must be initialized on-render time to be able to handle on render mesh changes such as rigid movements. The core will detect for scene changes, and if there are changes, the core reinitializes the buffers for the next render cycle.

The source core only passes the pointers to the triangle buffers when initializing the instance descriptor buffer. These triangle buffers are initialized on the device directly on core initialization. Changes in the scene will only change the affected triangle buffer and transformation.

In the target core, we must add a triangle descriptor buffer because the unavailability of direct device pointers in OpenCL explained in section 5.2.3. Instead of directly initializing triangle buffers on the device as the source core, we create the triangle buffers on the host. These triangle buffers will be copied over to the device on triangle descriptor initialization. Changes in the scene mean reinitialization of the whole continuous triangle descriptor array. This approach will take more time, especially if there are changes in every frame.

5.5.2 Primary Rays

Prior tracing the scene, the camera kernel generates rays that are going to be the input of the ray tracing engine. These rays are the primary rays, originating from the viewport and heading towards the scene. Rays are stored in a buffer, which the case of the target core is an OpenCL buffer. For each ray, there is also extra ray data that is stored in the path state buffer. The kernel generates primary rays and extra ray data once every render cycle by utilizing persistent threads (see Section 5.6).

Only the ray structure is different between the source kernel and the target kernel. The target kernel is adjusted to work with Radeon Rays ray structure (see Section 5.3.1). Other than the ray structure, the camera kernel is a straightforward port.

5.5.3 Shading

LightHouse 2 separates each ray path iteration using a *wavefront loop*. The loop allows the kernel to separate work in smaller chunks to ensure that each group of threads in a GPU are fully utilized (see Section 5.6). In a wavefront loop, the core extends the ray intersection and processes the intersection data with the shade kernel. By default, the maximum of ray bounces is three; thus, each render cycle will have at most three wavefront loops.

The first stage within a wavefront loop is extending the rays with the ray tracing engine to generate intersection data. The first loop will trace all the primary rays, and subsequent loops will trace the number of active rays that have been processed by the shade kernel.

```
// extend
CHK_PRIME(rtpBufferDescSetRange(extensionRaysDesc[inBuffer], 0, pathCount));
CHK_PRIME(rtpBufferDescSetRange(extensionHitsDesc, 0, pathCount));
CHK_PRIME(rtpQuerySetRays(query, extensionRaysDesc[inBuffer]));
CHK_PRIME(rtpQuerySetHits(query, extensionHitsDesc));
CHK_PRIME(rtpQueryExecute(query, RTP_QUERY_HINT_NONE /* or RTP_QUERY_HINT_ASYNC */));
```

```
// extend
if (pathCount != 0)
{
    EXPECT_NO_THROW(api->QueryIntersection(extensionRaysDesc[inBuffer], pathCount, extensionHitsDesc,
        nullptr, &query));
    ASSERT_NO_THROW(query->Complete());
    ASSERT_NO_THROW(query->Wait());
}
```

Figure 26. Extending the rays using OptiX in the CUDA core (above) and Radeon Rays in the OpenCL core (below).

Both the core and target ray tracing engine use the same numbers of buffers as the input and the output. The ray tracing engine takes the ray buffer as an input, intersect the rays with the scene, and returns the intersection results to the ray hit buffer. The main difference between tracing in the source core and the target core is on handling the number of rays to be traced. OptiX in the source core allows the ray count to be zero as the input (no active rays to trace) and returns no errors when given such case. Radeon Rays in the target core, on the other hand, cannot handle the case when active rays are zero. It returns an exception and breaks the program with no active rays. We have to add a conditional operator to prevent Radeon Rays from crashing. Handling active rays in a wavefront loop is essential because there will be cases when there are no ray bounces after the first trace.

After extending the rays, the shade kernel uses the active rays, extra ray data, and the intersection data to process the scene. The shade kernel calls for other kernels to process different conditional paths which results in a random ray bounce and writes new data into the ray buffer, extra ray data buffer, shadow ray buffer, and shadow potential buffer. The other kernels are grouped and separated by function; this includes material handling, texture sampling, BxDF processing, light calculation, and a kernel that contains various helper functions.

Because the shade kernel contains a lot of conditional operations, some rays may take paths that terminate them and marks them as not active. The core uses compaction to avoid processing unnecessary rays by only writing active rays to the buffer. The ray buffer does not have an active ray counter, such that we must use

an additional static ray counter buffer. Due to the nature of how GPU threads process data at the same time (highly parallel), it is crucial to keep track of active rays by using atomic counters. Tracking the rays using atomic counters prevents threads from writing to the ray counter buffer at the same time, and by doing so, keeps the active ray number correct.

Porting the shade kernel and all the kernel dependencies from CUDA into OpenCL is rather straightforward once the structural problems explained in Section 5.2 are addressed. All applied CUDA intrinsic functions are available in OpenCL as well so that we do not have to create our functions to replace them. Frequently used OpenCL API intrinsic function calls with the same function as the CUDA counterpart are redefined using CUDA API function names. Function name redefinition enables less redevelopment by allowing us to copy more lines of codes without modifying them, meaning high portability.

5.5.4 Shadow Rays

At every shade kernel call in a wavefront loop, rays that have a geometry hit are intersected against a random light source to generate shadow rays and shadow ray potentials. Once the shading wavefront loop has reached its maximum ray bounce, the connections kernel traces shadow rays against the scene for intersections. If there are no occlusions, the kernel writes the ray throughput from the shadow ray potentials buffer to the accumulator.

```
// trace the shadow rays using OptiX Prime
RTPquery query;
CHK_PRIME(rtpQueryCreate(*topLevel, RTP_QUERY_TYPE_ANY, &query));
CHK_PRIME(rtpBufferDescSetRange(shadowRaysDesc, 0, counters.shadowRays));
CHK_PRIME(rtpBufferDescSetRange(shadowHitsDesc, 0, counters.shadowRays));
CHK_PRIME(rtpQuerySetRays(query, shadowRaysDesc));
CHK_PRIME(rtpQuerySetHits(query, shadowHitsDesc));
CHK_PRIME(rtpQueryExecute(query, RTP_QUERY_HINT_NONE /* or RTP_QUERY_HINT_ASYNC */));
CHK_PRIME(rtpQueryDestroy(query));
```

```
// trace the shadow rays using Radeon Rays
RadeonRays::Event* query = 0;
EXPECT_NO_THROW(api->QueryOcclusion(shadowRaysDesc, counters.shadowRays, shadowHitsDesc, nullptr, &query));
ASSERT_NO_THROW(query->Complete());
ASSERT_NO_THROW(query->Wait());
ASSERT_NO_THROW(api->DeleteEvent(query));
```

Figure 27. Any hit API function call for OptiX (above) and Radeon Rays (below). These functions only return a Boolean for each ray and are useful for checking occlusions.

Tracing shadow rays does not use the full intersection API call; instead, it uses a lightweight function that only returns a Boolean whether the ray intersects or not. OptiX in the source core uses the predefined `RTP_BUFFER_FORMAT_HIT_BITMASK` intersection structure, with one bit per ray for misses and hits. The structure allows OptiX to have a densely packed shadow hit buffer (32 intersection data in a 4-byte data). By having a compact data, the shadow hit buffer uses less storage size and is less prone to cache misses when accessed.

Tracing shadow rays on the target core using Radeon Rays uses the same principle of only having the hit or miss in the intersection data. The significant difference compared to OptiX is that in Radeon Rays, it uses a signed integer for each intersection data, where each data is -1 for misses or 1 for hits. Using a signed integer causes the target core shadow hit buffer to use more storage than the source core (1 intersection data in a 4-byte data). By using more storage per data, it is possible that accessing the shadow hit buffer is slower in the target core.

5.6 Persistent Threads

In order to maximize thread utilization in path tracing, the rendering core is implemented using persistent threads [40]. Threads on a GPU are typically grouped in warps (NVIDIA) or wavefronts (AMD) that always process the same instruction in lockstep. If a thread branches out from the instruction, that thread will be masked out until the warp or wavefront starts processing a new instruction. Masked out threads will not do useful work and is considered as performance loss. Persistent threads optimize grouped threads workload by separating it into small specialized kernels.

To maximize persistent threads utilization, rendering in LightHouse 2 is separated into multiple specialized kernels which consist of the camera, shade, connection kernel. The shade kernel is even further specialized into smaller kernels considering the number of conditional paths. Having one large kernel that contains a lot of execution divergence in control flow could cause substantial performance penalties.

```
__shared__ volatile int baseIdx[32];
int lane = threadIdx.x & 31, warp = threadIdx.x >> 5;
__syncthreads();
while (1)
{
    if (lane == 0) baseIdx[warp] = atomicAdd(&counters->generated, 32);
    int jobIndex = baseIdx[warp] + lane;
    if (__all_sync(THREADMASK, jobIndex >= pathCount)) break;
    if (jobIndex < pathCount) generateEyeRaysKernel(jobIndex,
        rayBuffer, pathStateData,
        blueNoise, R0, j0, j1, pass,
        pos, right, up, aperture, p1,
        screenParams);
}
```

```
__local volatile int baseIdx[32];
int lane = get_global_id(0) & 31, warp = get_global_id(0) >> 5;
barrier(CLK_LOCAL_MEM_FENCE);
while (1)
{
    if (lane == 0) baseIdx[warp] = atomicAdd(&counters->generated, 32);
    int jobIndex = baseIdx[warp] + lane;
    if (jobIndex >= pathCount) break;
    if (jobIndex < pathCount) generateEyeRaysKernel(jobIndex,
        rayBuffer, pathStateData,
        blueNoise, blueSlot, R0, j0, j1, pass,
        pos, right, up, aperture, p1,
        screenParams, geometryEpsilon);
}
```

Figure 28. Host kernel for the camera kernel divides the total ray generation into jobs of 32 threads. The host kernel runs the actual camera kernel until there are no jobs left to process. The target host kernel is written in OpenCL (below) has a similar structure as the source CUDA host kernel, with the only visible difference in intrinsic calls that does the same function.

To implement persistent threads, the core uses a host kernel that calls the actual kernel. An example of a host kernel for the camera kernel can be seen in Figure 28. The host kernel divides the total work of the actual kernel into jobs of 32 threads, based on the number of threads grouped in one NVIDIA GPU warp.

The target core keeps the same structure for the host kernel because it only enables persistent threads on NVIDIA and AMD devices (threads are grouped in 64 threads on AMD devices and is divisible by 32). If initialized on an unsupported device, the core skips the host kernel and directly uses the actual kernel.

6. Results

This section contains all the results of experiments (explained in Section 4) that has been conducted on the source and the target core. To answer the first research question, which involves around code porting, can be deducted from measuring software similarity in Section 6.1. The second research question that concerns about the partial closed-source part of the program can be answered from the results of image comparison in Section 6.2. Lastly, answer to the last research question depends on the performance measurements in Section 6.2.4.

6.1 Measure of Software Similarity

To measure code similarity between the cores, we compare the core files using MOSS (Measure of Software Similarity), an automatic system for determining the similarity of programs [41]. This program was designed to detect code plagiarism in mind but proves to be useful to measure the similarity between software ports. The algorithm to detect similarity is based on winnowing, an efficient local document fingerprinting algorithm that detects matches of a certain length [42].

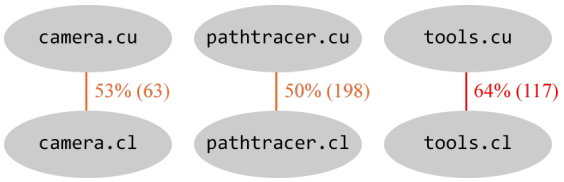
Comparing files using MOSS results in a percentage score and number of lines matched. The percentage score measures the amount of code of the current file that matches code to the opposite file. The comparison also gives us the approximate of the number of code lines that matched. The higher the percentage and lines match; the more similarity is between the codes.

	Total Lines / Match to the Other File		**Lines Matched**
	CUDA Core	**OpenCL Core**	
Root folder	1248 / 44%	1878 / 30%	690
Kernels	2133 / 25%	1418 / 43%	636
BSDFs	1427 / 6%	203 / 44%	59

Table 1. Results of core comparison using MOSS.

The results of comparing the files using MOSS gives us high scores across the different core modules (Table 1). We separate the files based on their physical location and function. The root folder contains mostly files that contain host function, the kernels folder contains device kernels, and the BSDFs contains functions to handle materials.

Although having high similarity and line matches, there is a substantial difference between the total number of code lines in each category. In the root folder, we can see that OpenCL has more code than the source core. OpenCL has more code than CUDA due to helper files inside the port core folder, while the respective CUDA helpers reside in the rendering system. As for kernels and BSDFs, the OpenCL core has much fewer lines of code because we do not port the advanced filtering and different BSDF handling functions as it is trivial for the research.



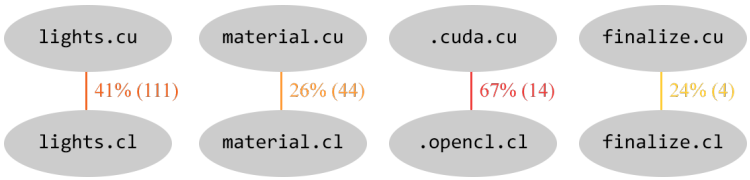


Figure 29. Comparison of individual kernel files from the source core and the target core using MOSS.

Comparing the kernel further by files, we can see that MOSS gives us a high number as well, with some exceeding 50% code similarity. Porting the kernels still needs a fair amount of redeveloping, but this is assisted by the guidelines explained in Section 5. Some of the necessary changes can also be automatically processed using text find and replace command.

Answering the first research question, minimizing the porting cost can be done by establishing clear procedures for repeating code patterns that needs redevelopment and code that can be directly used without modification. Having prior knowledge of these rules accelerates workflow as it also minimizes errors when porting new lines of code. Most line of codes, when conforming to the rules, can be easily ported without having to understand the exact function fully.

6.2 Image Comparison

We render each scene in 1280 x 720 pixels, a maximum of three ray bounces, and with 1024 SPP (samples per pixel). All of the scenes are rendered on an Intel Core i7 7700HQ CPU with 16 GB of RAM and an NVIDIA GeForce GTX 1050 Ti GPU with 4 GB of video RAM. The cores run on the same GPU, both utilizing persistent threads. All scenes are also rendered without any advanced filtering, meaning that the core uses pure path tracing to generate the images.

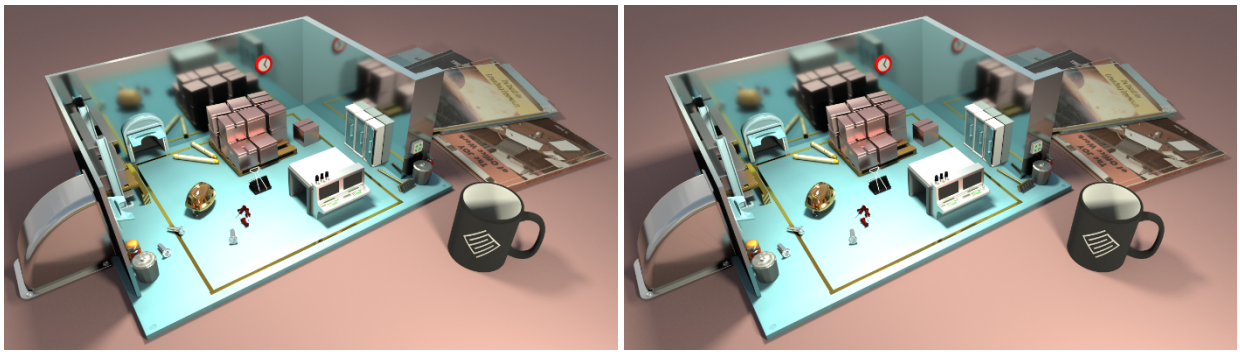


Figure 30. PICA PICA Warehouse Wide rendered with 1024 SPP using CUDA (left) and rendered using OpenCL (right).

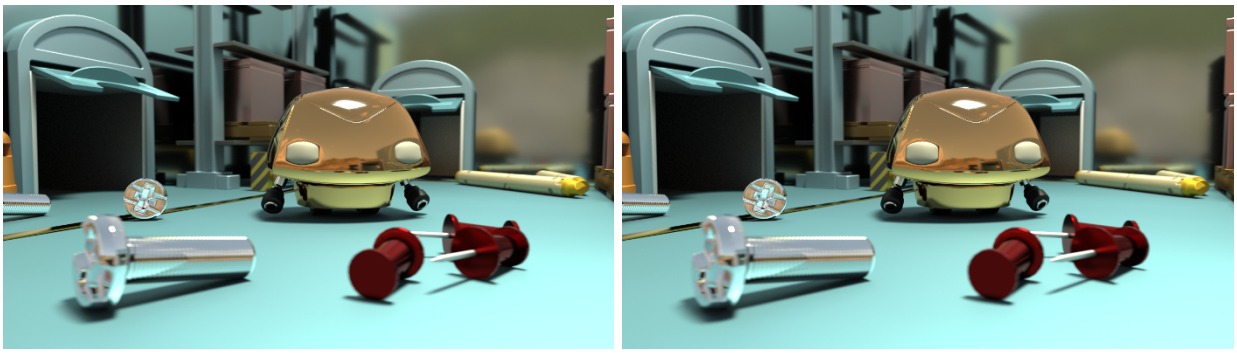


Figure 31. PICA PICA Warehouse Close rendered with 1024 SPP using CUDA (left) and rendered using OpenCL (right).

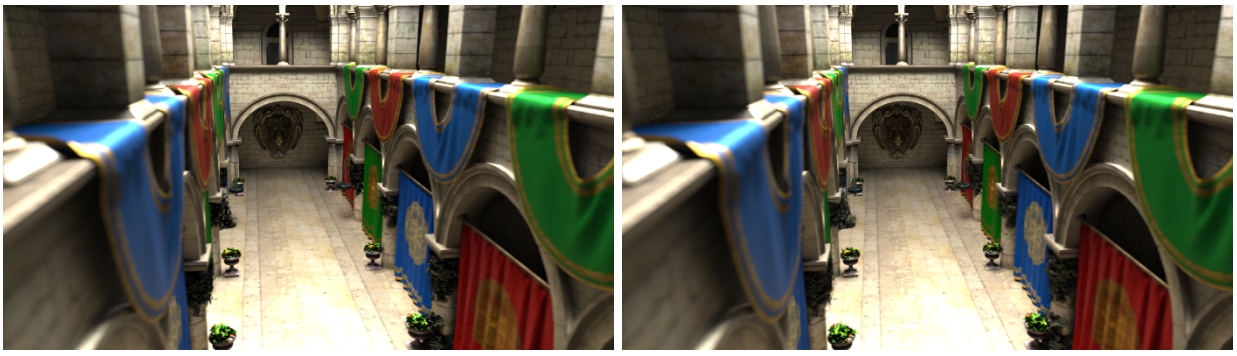


Figure 32. Sponza Wide rendered with 1024 SPP using CUDA (left) and rendered using OpenCL (right).

Figure 33. Sponza Close rendered with 1024 SPP using CUDA (left) and rendered using OpenCL (right). Note that some parts of the alpha textures are rendered different.

At a glance, the results of both cores look precisely alike, but on further inspection, there are some differences in alpha textures (transparent textures). The problem is most visible in Figure 33, where most of the mesh with alpha textures are apparent on screen. The problem does not seem to be perceptible in Figure 30, Figure 31, and Figure 32 as the view of these scenes do not have or show any mesh with alpha textures up close. To have a better understanding of the difference between the rendered images, we measure Mean Squared Error and Structure Similarity measurements in the next section.

6.2.1 Mean Squared Error

To obtain metrics, we compare rendered images using the same settings as the image comparison, but with different samples per pixel, as stated in Section 4.3.

	MSE			
SPP	**PICA PICA Warehouse Wide**	**PICA PICA Warehouse Close**	**Sponza Wide**	**Sponza Close**
1	5.40	0.83	53.61	174.46
2	2.07	0.33	40.62	144.36
4	0.99	0.22	28.46	113.41
8	0.59	0.11	19.72	86.81
16	0.44	0.05	14.27	70.18
32	0.34	0.04	11.09	60.14
64	0.32	0.03	9.35	54.14
128	0.30	0.02	8.35	50.61
256	0.28	0.02	7.82	48.76
512	0.28	0.01	7.52	47.74
1024	0.27	0.01	7.37	47.16

Table 2. Mean Squared Error by comparing rendered images with different samples per pixel.

By using MSE, we have a clear score of the absolute differences of the images (Table 2). The rendered image that is the closest to the source core render is the close-up shot of the PICA PICA Warehouse scene. The scene contains visible diffuse surfaces, fully-reflective surfaces, and non-alpha textures; which means functions that process these materials are working exactly as the source core. Error is mostly visible on the Sponza scene, which contains differences visible to the human eye. The problem is once again caused by the alpha textures not being rendered exactly as the source core.

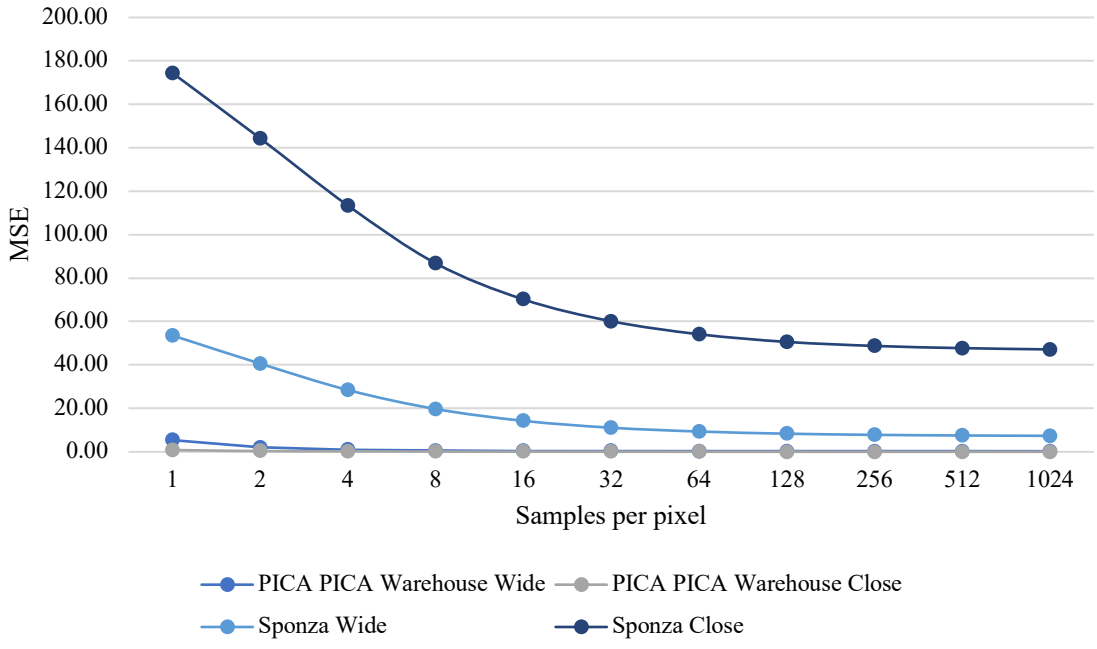


Figure 34. Mean Squared Error over the samples per pixel.

By having more sample per pixel, as the image converges, the Mean Squared Error is also decreased (Figure 34). By taking more samples, we can see that scenes will reach the error towards zero. The Sponza scene, however, converges to a higher error than zero, meaning it will never be rendered exactly as the source core.

To check the visual differences of the Sponza scene, we use per pixel subtraction from images rendered using CUDA and OpenCL to generate an image that only contains the different pixels and gives us visual of the absolute errors. A pure black pixel means that the pixels between the image rendered by CUDA and OpenCL are identical. An identical image will result in a black image with no visible pixels at all.

Figure 35. Difference between the alpha textures in the Sponza Wide scene rendered in CUDA and OpenCL.

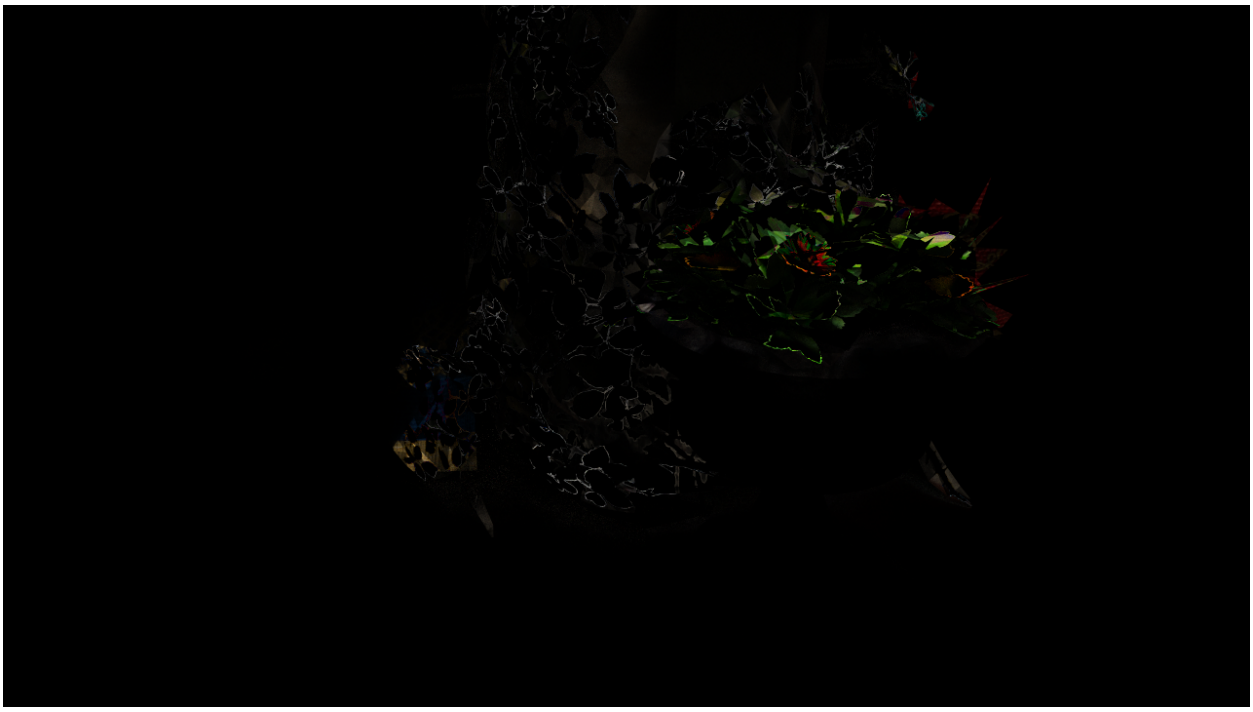


Figure 36. Difference between the alpha textures in the Sponza Close scene rendered in CUDA and OpenCL.

Subtracting the wide shot of the Sponza gives us an image with slight differences, mostly on meshes that contain alpha textures (Figure 35). On the close shot, the differences are more perceived as there are more pixels shown in the final image (Figure 36). These differences are the absolute errors, the cause of the high MSE obtained in both of the Sponza scenes.

We have precisely ported alpha material handling as the source core and can assure that the algorithm is the same in the target core. One possibility is that the difference in processing intersection using Radeon Rays causes the alpha texture problems.

In a quality perspective, Radeon Rays is a reliable replacement for the closed-source ray tracing engine. However, it has a different behavior than the source core's ray tracing engine that influences the render output. The engine is highly suitable for conditional cases, in systems where NVIDIA devices are not available or where the result of the image does not have to be highly accurate. It is possible to get the identical result as the source core when tracing with Radeon Rays, but a deeper understanding of the engine and redevelopment of the kernels are required. Because Radeon Rays is also open-source, modification of the engine is also possible when necessary.

6.2.2 Structural Similarity

As explained in Section 4.3, Mean Squared Error only measures the absolute errors. We have to use Structural Similarity to measure the perceived quality of the rendered images.

	SSIM			
SPP	**PICA PICA Warehouse Wide**	**PICA PICA Warehouse Close**	**Sponza Wide**	**Sponza Close**
1	1.00	1.00	0.99	0.96
2	1.00	1.00	0.99	0.95
4	1.00	1.00	0.99	0.96
8	1.00	1.00	0.99	0.96
16	1.00	1.00	0.99	0.96
32	1.00	1.00	0.99	0.97
64	1.00	1.00	0.99	0.97
128	1.00	1.00	0.99	0.97
256	1.00	1.00	0.99	0.97
512	1.00	1.00	0.99	0.97
1024	1.00	1.00	0.99	0.98

Table 3. Structural Similarity index by comparing rendered images with different samples per pixel.

When visually comparing the images, we perceive that Figure 30, Figure 31, and Figure 32 do not have visible differences. The perceived quality is backed with the SSIM index being or almost 1 for these scenes (Table 3), even on images with low samples per pixel. Only the fourth scene, although having the index close to 1, has visible differences.

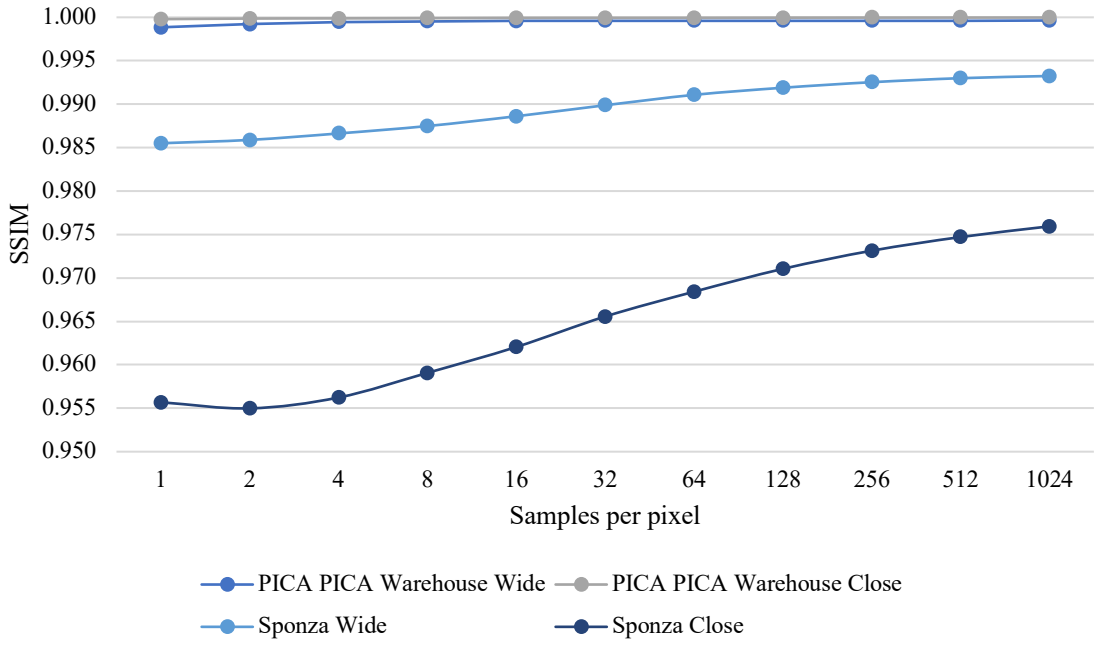


Figure 37. Structural Similarity index over the samples per pixel.

SSIM index also increases with the number of samples taken (Figure 37), although not as extreme as changes in MSE reduction. It is also seen that the quality of scenes with alpha textures will never reach an SSIM index of 1. When running on the same device, both of the cores render images that are perceived the same. How close the images are perceived depends on the scene as well. A scene with high visibility of alpha texture will deviate more than scenes that do not.

6.2.3 Cross-platform Image Comparison

We also run a render comparison on an image rendered on an NVIDIA Geforce GTX 1050 Ti and an AMD Radeon RX 560, both on the OpenCL core. These GPUs are roughly in the same performance class, with the GTX 1050 Ti having a slightly higher performance than the RX 560. Persistent threads are disabled when testing. We render two scenes with extreme material differences.

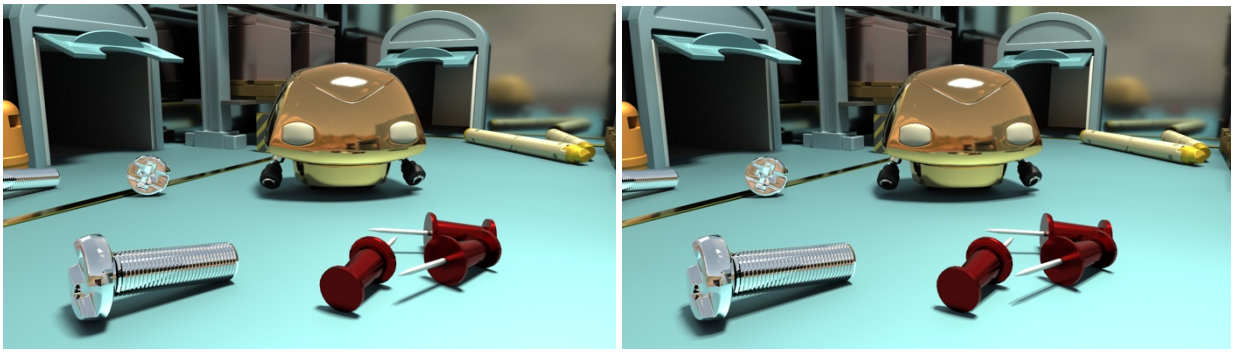


Figure 38. PICA PICA Warehouse Close rendered with 1024 SPP using the OpenCL core on an NVIDIA GeForce GTX 1050 Ti (left) and an AMD Radeon RX 560 (right).

Figure 39. Sponza Close rendered with 1024 SPP using the OpenCL core on an NVIDIA GeForce GTX 1050 Ti (left) and an AMD Radeon RX 560 (right).

When perceived, the images in Figure 38 shows a slight difference, which is caused by the alternative function used on AMD devices (see Section 5.2.6). In Figure 39, the difference is too subtle to notice and requires metrics to confirm. Even though there are visible differences, the perceived composition of the image remains the same. We run the images through the Mean Squared Error and Structural Similarity methods to obtain metrics.

PICA PICA Warehouse Close		
SPP	**MSE**	**SSIM**
1	361.88	0.85
2	179.93	0.87
4	89.07	0.90
8	47.12	0.92
16	27.13	0.94
32	16.86	0.96
64	11.81	0.97
128	9.17	0.98
256	7.82	0.99
512	7.14	0.99
1024	6.82	0.99

Sponza Close		
SPP	**MSE**	**SSIM**
1	2047.10	0.63
2	1058.51	0.70
4	558.48	0.74
8	295.36	0.78
16	165.76	0.83
32	101.97	0.87
64	69.41	0.90
128	52.58	0.93
256	43.81	0.95
512	39.37	0.97
1024	37.07	0.98

Table 4. MSE and SSIM values from comparing the rendered images of the scenes.

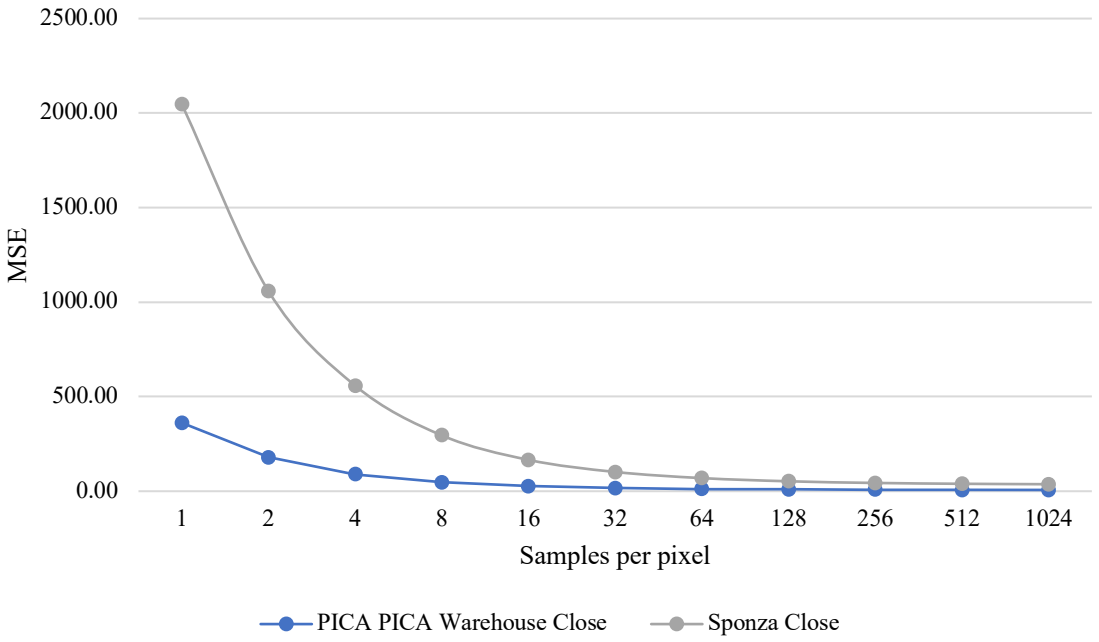


Figure 40. Mean Squared Error over the samples per pixel.

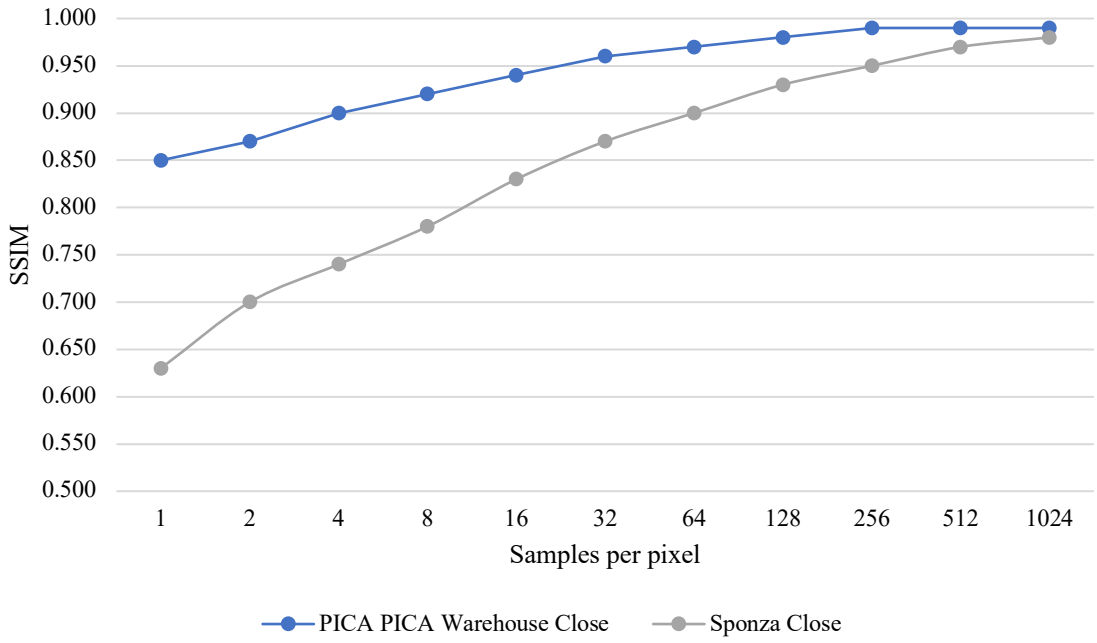


Figure 41. Structural Similarity index over the samples per pixel.

As expected from the results from Table 4, the absolute error from the MSE calculation is not near zero because of the visible differences across the images. Although of the visible differences, the SSIM index is nearing one, meaning that both the images provide almost the same structural information. The scenes converge to a smaller error (Figure 40) and higher structural similarity index (Figure 41) with more samples taken, even though the values starts highly biased. These results are correct because as we saw earlier, we have some difference, but still perceive the same information from the image.

Images rendered using the OpenCL core on NVIDIA and AMD devices are perceivably comparable, but not correct. While retaining the same structural information, there may be noticeable differences depending on the scene. Scenes with more shadows and alpha textures will have a higher MSE and may reduce the SSIM index when compared to the NVIDIA render.

6.2.4 Cross-core Image Comparison

The final quality experiment is measuring the differences between different cores on different devices. In this test, we run our CUDA core on an NVIDIA GeForce GTX 1050 Ti and the OpenCL core on an AMD Radeon RX 560. Persistent threads are disabled for the OpenCL core. We use the same scene in Section 6.2.3 as it gives the extreme case differences.

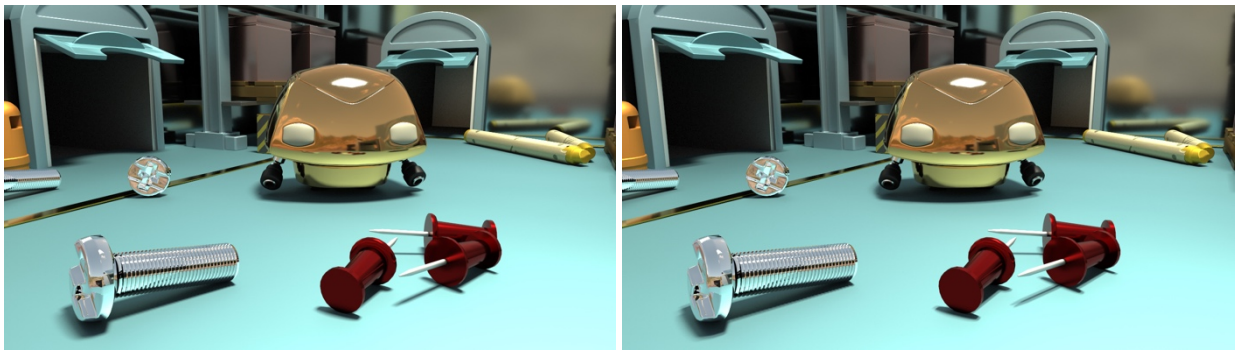


Figure 42. PICA PICA Warehouse Close rendered with 1024 SPP using the CUDA core on an NVIDIA GeForce GTX 1050 Ti (left) and the OpenCL core on an AMD Radeon RX 560 (right).

Figure 43. Sponza Close rendered with 1024 SPP using the CUDA core on an NVIDIA GeForce GTX 1050 Ti (left) and the OpenCL core on an AMD Radeon RX 560 (right).

As the perceived images are comparable to the test from Section 6.2.3, we must also measure Mean Squared Error and Structure Similarity.

PICA PICA Warehouse Close		
SPP	**MSE**	**SSIM**
1	362.11	0.85
2	179.96	0.87
4	89.07	0.90
8	47.14	0.92
16	27.13	0.94
32	16.86	0.96
64	11.82	0.97
128	9.17	0.98
256	7.83	0.99
512	7.15	0.99
1024	6.83	0.99

Sponza Close		
SPP	**MSE**	**SSIM**
1	2094.39	0.63
2	1101.00	0.68
4	598.49	0.73
8	335.13	0.77
16	205.57	0.81
32	142.20	0.85
64	109.12	0.89
128	92.00	0.91
256	82.89	0.94
512	78.26	0.95
1024	75.86	0.96

Table 5. MSE and SSIM values from comparing the rendered images of the scenes.

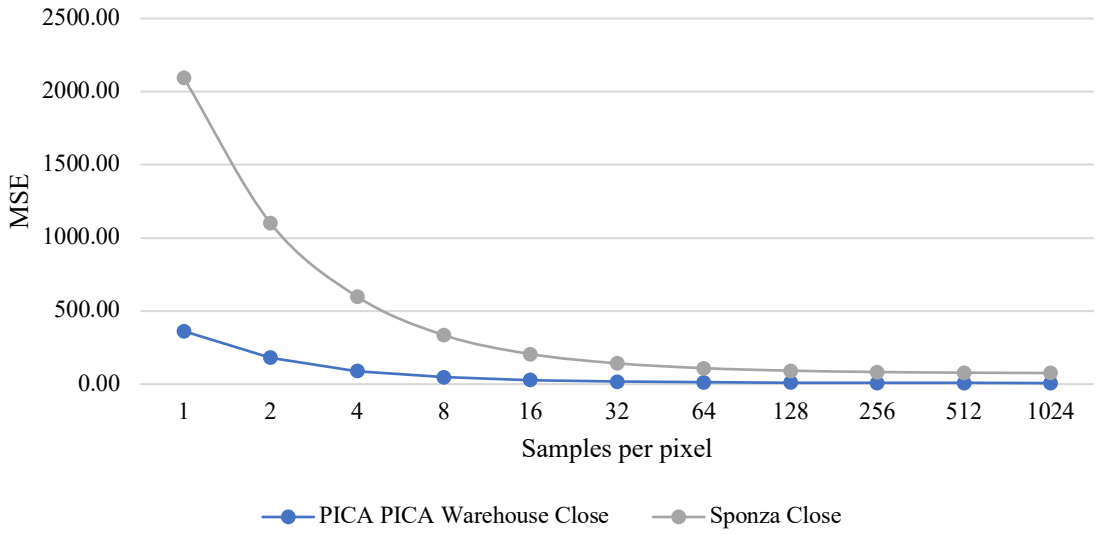


Figure 44. Mean Squared Error over the samples per pixel.

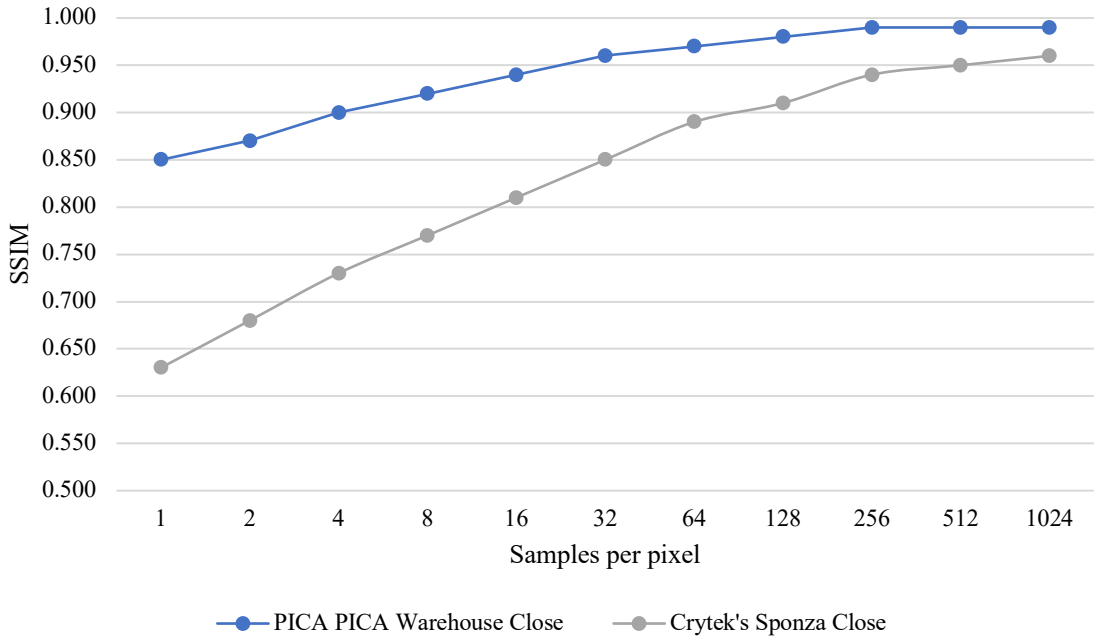


Figure 45. Structural Similarity index over the samples per pixel.

Our cross core test delivers similar results (Table 5) as in Section 6.2.3. The parts that cause differences in the results are the alpha material handling and the different method used on AMD devices (see Section 5.2.6). The Mean Squared Error depends much on scene complexity, but in general, are high between images and never reaches zero (Figure 44). With the high error, the images start to lose structure similarity (Figure 45) with apparent differences.

6.3 Performance

To answer the last research question, we measure the average render time for one render cycle and the number of rays processed per seconds. We use the same system settings, as explained in Section 6.2 to measure the performance of both cores.

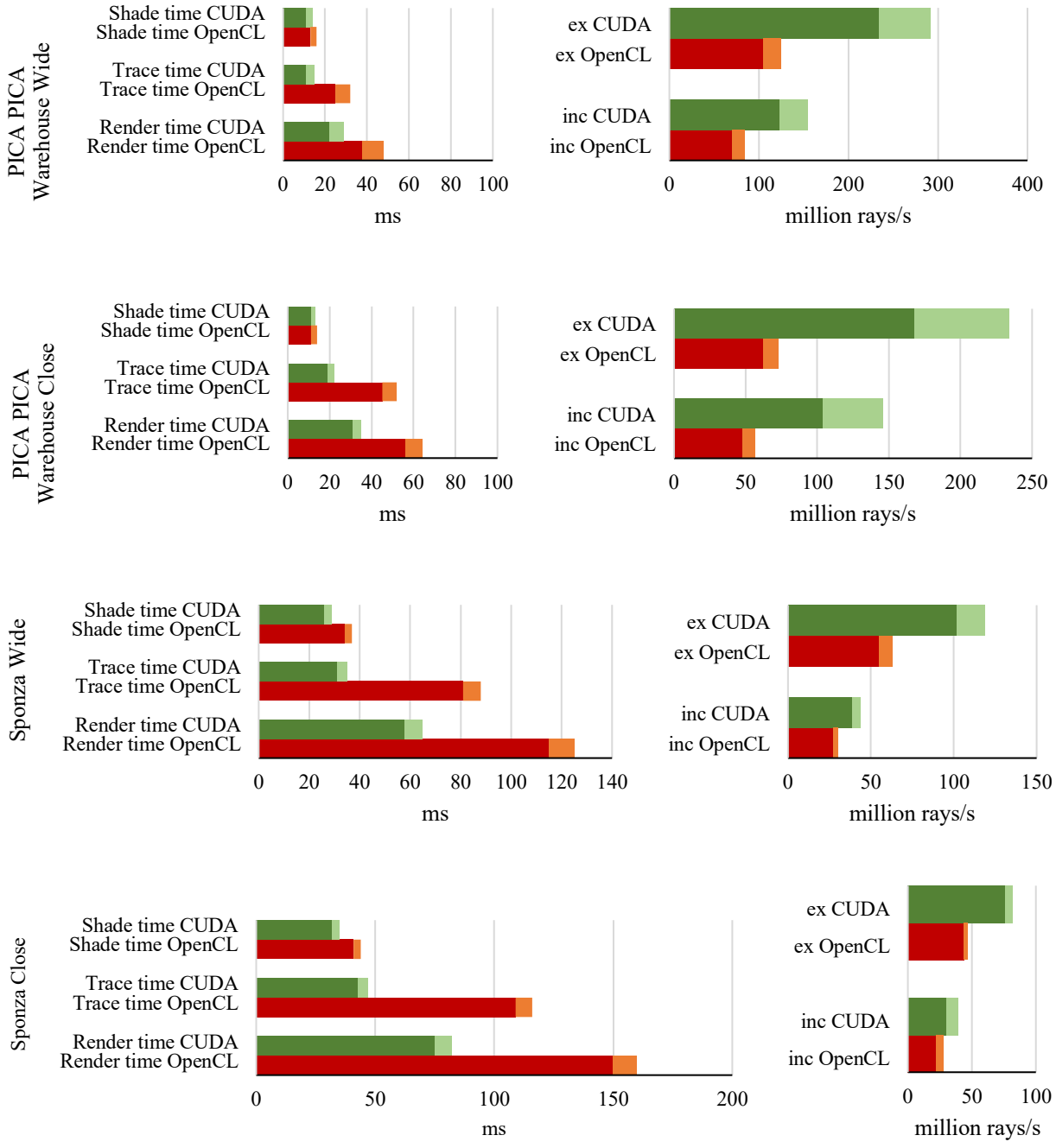


Figure 46. Render time and rays per second from all the predefined scenes. The lighter color in the graphs is the range of the average and the fastest processing that the core can do in one render cycle. Inclusive rays (inc) are the number of rays that are processed per render time. Exclusive rays (ex) are the number of rays that are processed per trace time.

From the performance measurements in Figure 46, we can deduce that there are two points of interest between the cores.

First, there is only a slight difference in the shading performance of the OpenCL core compared to the CUDA core. In our test, shading performance at most is 25% slower than the source core and means that pure OpenCL code is comparable to CUDA for path tracing purposes.

The second case is that the ray tracing engine in the OpenCL core is twice slower. The time to trace a scene in the target core always exceeds twice the amount needed by the source core, also with less processed rays per second. A slower tracing time means that Radeon Rays is not as efficient as OptiX for intersection geometry, which is also backed with the fact that the data structure for Radeon Rays needs significantly more storage, slowing the read-write process.

Performance-wise, the OpenCL core is not par with the CUDA core when tested using the same NVIDIA device. However, as our port is a cross-platform compatible application, we also compare performance against several GPUs from NVIDIA and AMD.

6.3.1 Cross-platform Performance

To compare cross-platform performance, we run the OpenCL core on different GPUs. Unfortunately, we cannot use the same CPU and RAM configuration across the tests as the GPUs are spread around different systems. Different configurations should not affect rendering performance too much as the CPU mostly handles data loading onto RAM. We use the PICA PICA Warehouse Close scene in this test and disabled persistent threads to ensure device compatibility.

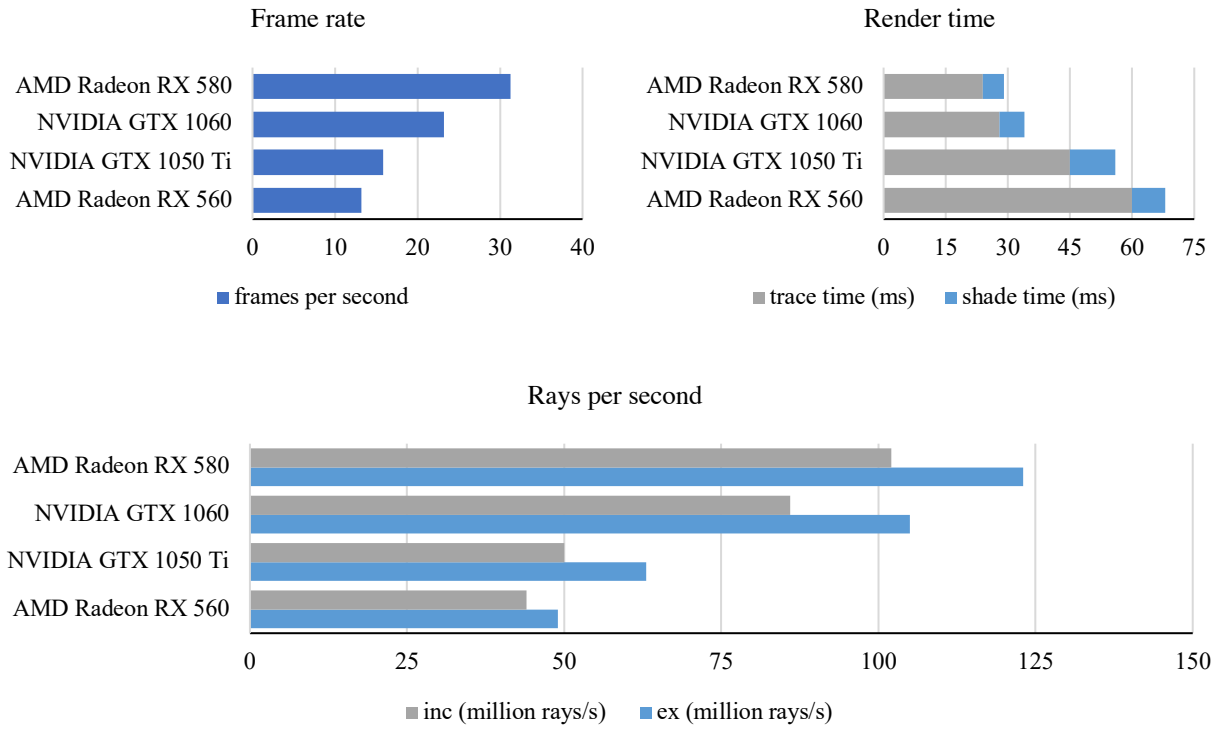


Figure 47. Frame rate, render time, and rays per second from different devices. Inclusive rays (inc) are the number of rays processed per render time. Exclusive rays (ex) are the number of rays processed per trace time.

Results from the cross-platform test provide the same trends as the core to core performance. Trace time for all the devices takes roughly 80% of the render time, whether the core is run on AMD or NVIDIA. The render engine works across these devices, but it does not provide performance boosts when running on AMD devices. Pure shading performance, however, may be faster on AMD hardware as the drivers are better optimized for OpenCL than NVIDIA.

We also test the OpenCL core on Intel devices, which provides poor performance results. Running on a consumer-grade processor (Intel Core i7 7700HQ) and its integrated graphics processing unit (Intel HD Graphics 630) gives frame rates under 5 FPS, unsuitable for interactive path tracing. On a workstation processor (Intel Xeon E5 2620), the core provides decent performance (13 FPS on average) and is comparable to our lowest-performing GPU device (AMD Radeon RX 560).

Devices from different manufacturers in the same class do not necessarily give a noticeable performance boost. Performance is affected mostly by device compute capability, proportional to the number of threads. As a path-tracing application, the core is preferably run on highly parallel devices, such as a GPU, to get the most performance.

6.3.2 Persistent Threads

This section contains the results of measuring render times using different persistent threads configurations.

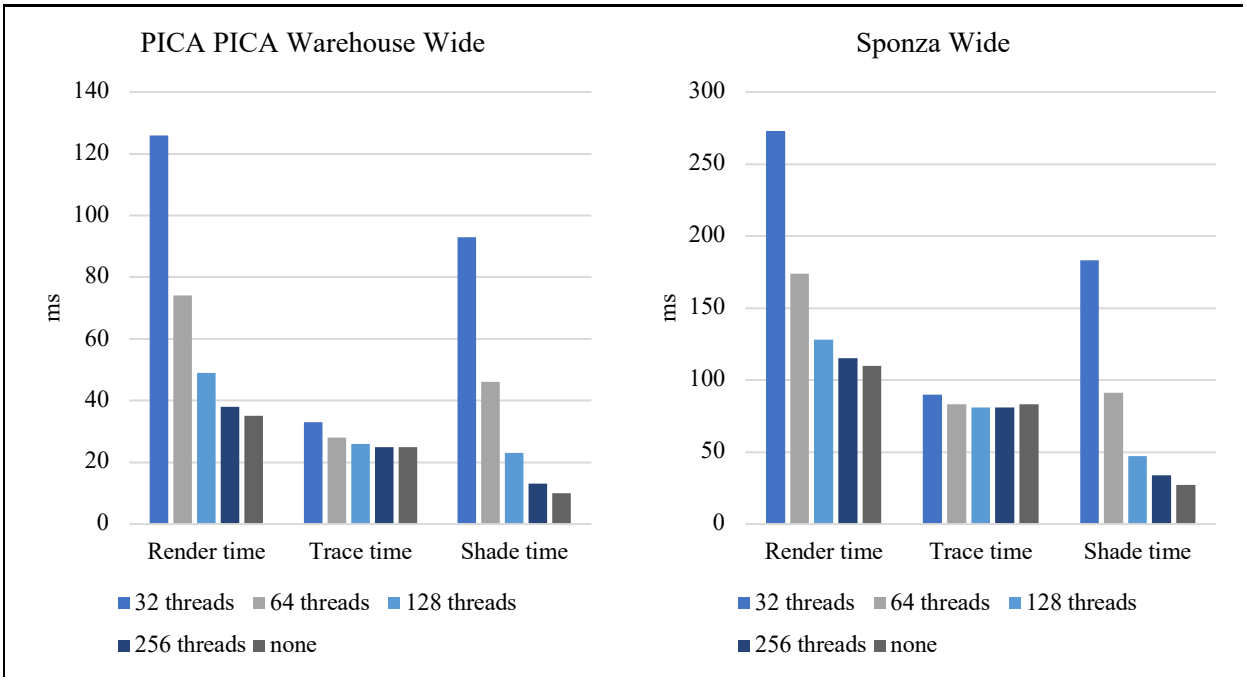


Figure 48. Performance test using different persistent threads configurations.

Measuring render times using different persistent threads configuration gives us unexpected results. According to the test results shown in Figure 48, increasing the number of threads which allows multiple jobs to run at the same time yields the desired results as the theory. However, rendering without using persistent threads gives us the fastest performance.

The maximum number of threads that can be allocated is dependent on the device, which in our case is 256 (should in multiple of grouped threads). When the number of threads exceeds the limit, OpenCL returns an exception and has to be reinitialized from the start. There is no definite number that works on most OpenCL devices and dynamically allocating the numbers does not work as the core crashes every time it exceeds the limit. Having a fixed number of threads may cause problems when running the core on untested devices.

To increase cross-platform compatibility, we disabled persistent threads and measured the core's performance. Disabling persistent threads makes the OpenCL core of performing faster, contradicting the persistent threads theory. The persistent thread algorithm that we have ported from the CUDA core might not work well with OpenCL or that the OpenCL scheduler might be efficient enough to run the kernels directly.

7. Conclusion

We introduce the problem of cross-platform compatibility as the original application can only run on specific devices. We divide the process into two phases: theoretical study and implementation.

For the first phase, we explain path tracing to give a better understanding of the source application primary function in Section 2. Understanding path tracing gives us knowledge of essential sections in the application that is critical to preserve when porting. A literature review in Section 3 provides us information about the different approaches and limitations on porting. This section introduces several methods for measuring porting cost and presents numerous techniques for implementation. The research methodology explained in Section 4, gives the research steps that we use in the thesis; porting implementation and experiments.

The second phase starts in Section 5, Implementation, which contains all the problems and solutions that we have tackled during porting. This section serves us as an essential guideline when implementing the port. Lastly, Section 6 gives us the results for all the experiments that we conducted using the source and the port. Experiments include running the port on different devices to measure cross-platform compatibility.

An efficient way to minimize porting cost is by having clear guidelines for redevelopment. Creating guidelines can be achieved by having a full understanding of the algorithm and underlying language of the source and the target port. As for the port, choosing a language that is widely used is also beneficial as there are established guidelines from other cases. These guidelines can help accelerate understanding the languages, which reduces the time for porting.

In our case, using OpenCL to port CUDA code is the most cost-effective approach. OpenCL has been the most used language for porting CUDA made applications, as it has roughly the same language structure and is targeted to run on a wide range of devices. Several cases of porting CUDA to OpenCL are also available, which mostly have positive results. Learning these cases provides useful information for common problems that are likely to be faced on porting.

We can approach closed-source parts of software by replacing it with open-source modules that have similar functions. However, not every closed-source library can be replaced. Replacement is case-specific, and there is no guarantee that an open-source counterpart exists. Device-specific libraries are mostly better optimized than the cross-device equivalent device, such it can fully utilize all the hardware features. Device-specific libraries are also better documented, with a guaranteed list of supported hardware.

In our case, we succeeded replacing NVIDIA OptiX, a closed-source ray tracing engine, with Radeon Rays, an open-source ray tracing engine by NVIDIA's competitor, AMD. Radeon Rays, however, do not behave precisely as OptiX and produces a slightly different result, which can be problematic in specific cases. Despite the render results, AMD's ray tracing engine can be run on a wide variety of hardware, which makes the port highly cross-platform compatible.

Port performance is lower than the original application when running on the same device. Even though the lower performance, it is not restricted to a specific device type. The port can run on high-performance GPUs manufactured by other vendors than NVIDIA and even on computer processors. The port was not meant to compete for performance on the same device, but rather to run on devices that the original application cannot.

The port has been engineered in such a way that it keeps the original application's structure intact. We use the same function algorithm as we possibly could, except for extreme cases that malfunctions the port. Some intrinsic function names have also been masked to ease porting. Having the same structure ensures that code changes in the source application can be easily implemented. A guideline has been provided in this thesis that should work in most cases.

We have successfully ported a CUDA path tracer that uses OptiX to run on non-NVIDIA devices by utilizing OpenCL and Radeon Rays. During porting, we also found limitations to the programming language and ray tracing engine that we use with the solutions to them. Purposefully, the port can serve as a backup core, where an NVIDIA device is not available, as well as a learning tool to port CUDA into OpenCL.

7.1 Future Work

Although we have achieved the primary goal of our thesis, the implemented port is far from perfect. First of is the quality degradation in the port caused by OpenCL problems in Section 5.2.6 and Radeon Rays in Section 6.2. By understanding how memory addresses work on a hardware-level, we should be able to fix the problems.

Achieving performance comparable to the source core is also visible. We directly port all the functions in our code but do not explicitly optimize code. We try not to change any algorithm that we port to ease code maintenance. If required, we can apply hardware-level optimizations to improve performance without sacrificing render quality.

An advanced feature from the original application that we have not yet ported is filtering. Filtering is an experimental feature that enables the core to use less path-tracing while resulting in a comparable output. Porting filtering in our research is trivial because it is not necessary for rendering and should be easily ported, if not for a large amount of source code it needed.

Implementing proper persistent threads is also a goal that we did not achieve in our research. Implementation requires further investigation on how persistent threads work in OpenCL and achieve a boost in performance as with the CUDA core.

Finally, without using intermediate code, it should be possible for CUDA and OpenCL code to coexist in one file by using a lot of redefinitions and code-switching. This coexistence will include overhead but will result in almost zero maintenance code. Changing or adding a function will automatically change for both languages; thus, no more porting is required.

8. References

[1] "Artificial Intelligence Computing Leadership from NVIDIA," NVIDIA Corporation, 2018. [Online]. Available: https://www.nvidia.com/en-us/. [Accessed 23 November 2018].

[2] NVIDIA Corporation Technical Staff, NVIDIA CUDA Programming Guide 2.2, NVIDIA Corporation, 2009.

[3] S. G. Parker, J. Bigler, A. Dietrich, H. Friedrich, J. Hoberock, D. Luebke, D. McAllister, M. McGuire, K. Morley, A. Robison and M. Stich, "OptiX: A General Purpose Ray Tracing Engine," in *ACM SIGGRAPH 2010 Papers*, New York, 2010.

[4] J. Peddie, "Jon Peddie Research releases its Q3, 2018 add-in board report," 28 11 2018. [Online]. Available: https://www.jonpeddie.com/press-releases/jon-peddie-research-releases-its-q3-2018-add-in-board-report. [Accessed 1 February 2019].

[5] "Welcome to AMD | Processors | Graphics and Technology | AMD," Advanced Micro Devices, Inc, 2019. [Online]. Available: https://www.amd.com/en. [Accessed 3 January 2019].

[6] "Intel | Data Center Solutions, IoT, and PC Innovation," Intel Corporation, 2019. [Online]. Available: https://www.intel.com/content/www/us/en/homepage.html.

[7] J. D. Mooney, "Issues in the Specification and Measurement of Software Portability," Dept. of Statistics and Computer Science, West Virginia University, Morgantown, 1993.

[8] T. Whitted, "An improved illumination model for shaded display," in *Communications of the ACM*, New York, ACM, 1980, pp. 343-349.

[9] D. S. Immel, M. F. Cohen and D. P. Greenberg, "A Radiosity Method for Non-diffuse Environments," *SIGGRAPH Comput. Graph.,* vol. 20, no. 4, p. 133–142, 1986.

[10] J. T. Kajiya, "The Rendering Equation," *SIGGRAPH Comput. Graph.,* vol. 20, no. 4, pp. 143-150, 1986.

[11] F. E. Nicodemus, "Directional Reflectance and Emissivity of an Opaque Surface," *Appl. Opt.,* vol. 4, no. 7, pp. 767-775, 1965.

[12] M. Pharr, W. Jakob and G. Humphreys, Physically Based Rendering: From Theory to Implementation, San Francisco: Morgan Kaufmann Publishers Inc., 2016.

[13] A. Chalmers, E. Reinhard and T. Davis, Practical Parallel Rendering, New York: A K Peters/CRC Press, 2002.

[14] "Create and Buy Bugaboo Fox - Bugaboo.com," Bugaboo International B.V., 2019. [Online]. Available: https://www.bugaboo.com/US/en_US/strollers/create?sId=FOX. [Accessed 28 January 2019].

[15] P. C. Poole and W. M. Waite, Portability and Adaptability, 1972.

[16] A. S. Tanenbaum, P. Klint and W. Bohm, "Guidelines for Software Portability," *Software, Practice and Experience,* vol. 8, no. 6, pp. 681-698, 1978.

[17] J. D. Mooney, "Bringing Portability to the Software Process," Dept. of Statistics and Computer Science, West Virginia University, Morgantown, 1997.

[18] J. D. Mooney, "Strategies for Supporting Application Portability," *Computer,* vol. 23, no. 11, pp. 59-70, 1990.

[19] J. D. Mooney, "Developing Portable Software," *Reis R. (eds) Information Technology. IFIP International Federation for Information Processing,* vol. 157, 2004.

[20] M. Hakuta and M. Ohminami, "A study of software portability evaluation," *Journal of Systems and Software,* vol. 38, no. 2, pp. 145-154, 1997.

[21] "GPUOpen - GPUOpen," Advanced Micro Devices, Inc, 2019. [Online]. Available: https://gpuopen.com/. [Accessed 15 January 2019].

[22] "ROCm, a New Era in GPU Computing," AMD Corporation, 2016. [Online]. Available: https://rocm.github.io/. [Accessed 15 January 2019].

[23] "CUDA, OpenCL - Explore - Google Trends," Google LLC, 2019. [Online]. Available: https://trends.google.com/trends/explore?cat=5&date=all&q=%2Fm%2F026kkml,%2Fm%2F047gb9r. [Accessed 10 February 2019].

[24] NVIDIA Corporation Technical Staff, CUDA COMPILER DRIVER NVCC Reference Guide, NVIDIA Corporation, 2018.

[25] NVIDIA Corporation Technical Staff, Parallel Thread Execution ISA Version 6.3, NVIDIA Corporation, 2018.

[26] NVIDIA Corporation Technical Staff, NVIDIA Turing GPU Architecture, NVIDIA Corporation, 2018.

[27] "OpenCL Overview - The Khronos Group Inc," The Khronos® Group Inc, 2019. [Online]. Available: https://www.khronos.org/opencl/. [Accessed 15 January 2018].

[28] G. Martinez, M. Gardner and W.-c. Feng, "CU2CL: A CUDA-to-OpenCL Translator for Multi- and Many-core Architectures," in *Proceedings of the 17th IEEE International Conference on Parallel and Distributed Systems*, Tainan, 2011.

[29] B. Sander, "We ported CAFFE to HIP - and here's what happened… - GPUOpen," Advanced Micro Devices, Inc., 1 July 2017. [Online]. Available: https://gpuopen.com/ported-caffe-hip-heres-happened/.

[30] "Intel® Embree," Intel Corporation, 2018. [Online]. Available: https://embree.github.io/.

[31] "Radeon-Rays - GPUOpen," Advanced Micro Devices, Inc, 2019. [Online]. Available: https://gpuopen.com/gaming-product/radeon-rays/.

[32] Z. Wang, A. C. Bovik, H. R. Sheikh and E. P. Simoncelli, "Image Quality Assessment: From Error Visibility to Structural Similarity," *IEEE Transactions on Image Processing,* vol. 13, no. 4, pp. 600-612, April 2004.

[33] Aaftab Munshi, "The OpenCL Extension Specification," Khronos OpenCL Working Group, 13 February 2018. [Online]. Available: https://www.khronos.org/registry/OpenCL/specs/opencl-1.2-extensions.pdf.

[34] NVIDIA Corporation, "CUDA MATH API," NVIDIA Corporation, July 2019. [Online]. Available: https://docs.nvidia.com/pdf/CUDA_Math_API.pdf.

[35] V. Hindriksen, "NVIDIA ended their industry-leading support for OpenCL in 2012 - Stream HPC," Stream HPC B.V., 10 September 2012. [Online]. Available: https://streamhpc.com/blog/2012-09-10/nvidias-industry-leading-support-for-opencl/. [Accessed 9 June 2019].

[36] NVIDIA Corporation, "CUDA RUNTIME API," NVIDIA Corporation, July 2019. [Online]. Available: https://docs.nvidia.com/pdf/CUDA_Runtime_API.pdf.

[37] "ISO/IEC 9899 - Programming languages -- C," International Organization for Standardization, 1999.

[38] K. Basu and A. Owen, "Low Discrepancy Constructions in the Triangle," *SIAM Journal on Numerical Analysis,* vol. 53, no. 2, pp. 743-761, 2015.

[39] E. Heitz, "A Low-Distortion Map Between Triangle and Square," 2019.

[40] S. Laine, T. Karras and T. Aila, "Megakernels considered harmful: wavefront path tracing on GPUs," in *Proceedings of the 5th High-Performance Graphics Conference*, New York, USA, 2013.

[41] A. Aiken, "Plagiarism Detection," Stanford University, 14 12 2018. [Online]. Available: https://theory.stanford.edu/~aiken/moss/.

[42] S. Schleimer, D. S. Wilkerson and A. Aiken, "Winnowing: Local Algorithms for Document Fingerprinting," in *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, New York, NY, USA, 2003.