Generating Renderers

Arsène Pérard-Gayot

Dissertation zur Erlangung des Grades des Doktors der
Ingenieurwissenschaften der Fakultät für Mathematik und Informatik
der Universität des Saarlandes

Saarbrücken, 2020

UNIVERSITÄT
DES
SAARLANDES

Tag des Kolloquiums:	27.11.20
Dekan der Fakultät:	Univ.- Prof. Dr. Thomas Schuster
Vorsitzender des Prüfungsausschusses:	Prof. Dr.-Ing. Thorsten Herfet
Erstgutachter/Doktorvater:	Prof. Dr.-Ing. Philipp Slusallek
Zweitgutachter:	Prof. Dr. Sebastian Hack
Drittgutachter:	Ass. Prof. Dipl-Ing. Dr. Techn. Markus Steinberger
Akademischer Beisitzer:	Dr.-Ing. Richard Membarth

Abstract

Most production renderers developed for the film industry are huge pieces of software that are able to render extremely complex scenes. Unfortunately, they are implemented using the currently available programming models that are not well suited to modern computing hardware like CPUs with vector units or GPUs. Thus, they have to deal with the added complexity of expressing parallelism and using hardware features in those models. Since compilers cannot alone optimize and generate efficient programs for any type of hardware, because of the large optimization spaces and the complexity of the underlying compiler problems, programmers have to rely on compiler-specific hardware intrinsics or write non-portable code. The consequence of these limitations is that programmers resort to writing the same code twice when they need to port their algorithm on a different architecture, and that the code itself becomes difficult to maintain, as algorithmic details are buried under hardware details.

Thankfully, there are solutions to this problem, taking the form of Domain-Specific Languages. As their name suggests, these languages are tailored for one domain, and compilers can therefore use domain-specific knowledge to optimize algorithms and choose the best execution policy for a given target hardware. In this thesis, we opt for another way of encoding domain-specific knowledge: We implement a generic, high-level, and declarative rendering and traversal library in a functional language, and later refine it for a target machine by providing partial evaluation annotations. The partial evaluator then specializes the entire renderer according to the available knowledge of the scene: Shaders are specialized when their inputs are known, and in general, all redundant computations are eliminated. Our results show that the generated renderers are faster and more portable than renderers written with state-of-the-art competing libraries, and that in comparison, our rendering library requires less implementation effort.

Zusammenfassung

Die meisten in der Filmindustrie zum Einsatz kommenden Renderer sind riesige Softwaresysteme, die in der Lage sind, extrem aufwendige Szenen zu rendern. Leider sind diese mit den aktuell verfügbaren Programmiermodellen implementiert, welche nicht gut geeignet sind für moderne Rechenhardware wie CPUs mit Vektoreinheiten oder GPUs. Deshalb müssen Entwickler sich mit der zusätzlichen Komplexität auseinandersetzen, Parallelismus und Hardwarefunktionen in diesen Programmiermodellen auszudrücken. Da Compiler nicht selbständig optimieren und effiziente Programme für jeglichen Typ Hardware generieren können, wegen des großen Optimierungsraumes und der Komplexität des unterliegenden Kompilierungsproblems, müssen Programmierer auf Compiler-spezifische Hardware-"Intrinsics" zurückgreifen, oder nicht portierbaren Code schreiben. Die Konsequenzen dieser Limitierungen sind, dass Programmierer darauf zurückgreifen den gleichen Code zweimal zu schreiben, wenn sie ihre Algorithmen für eine andere Architektur portieren müssen, und dass der Code selbst schwer zu warten wird, da algorithmische Details unter Hardwaredetails verloren gehen.

Glücklicherweise gibt es Lösungen für dieses Problem, in der Form von DSLs. Diese Sprachen sind maßgeschneidert für eine Domäne und Compiler können deshalb Domänenspezifisches Wissen nutzen, um Algorithmen zu optimieren und die beste Ausführungsstrategie für eine gegebene Zielhardware zu wählen. In dieser Dissertation wählen wir einen anderen Weg, Domänenspezifisches Wissen zu enkodieren: Wir implementieren eine generische, high-level und deklarative Rendering- und Traversierungsbibliothek in einer funktionalen Programmiersprache, und verfeinern sie später für eine Zielmaschine durch Bereitstellung von Annotationen für die partielle Auswertung. Der "Partial Evaluator" spezialisiert dann den kompletten Renderer, basierend auf dem verfügbaren Wissen über die Szene: Shader werden spezialisiert, wenn ihre Eingaben bekannt sind, und generell werden alle redundanten Berechnungen eliminiert. Unsere Ergebnisse zeigen, dass die generierten Renderer schneller und portierbarer sind, als Renderer geschrieben mit den aktuellen Techniken konkurrierender Bibliotheken und dass, im Vergleich, unsere Rendering Bibliothek weniger Implementierungsaufwand erfordert.

Résumé

La plupart des moteurs de rendu développés pour l'industrie du film sont de gigantesques logiciels qui sont capables de produire des images à partir de scènes très complexes. Malheureusement, ils sont implémentés avec les modèles de programmation courants qui ne sont pas vraiment adaptés pour programmer des ordinateurs modernes qui possèdent des CPUs avec des instructions vectorielles ou des GPUs. En conséquence, ces moteurs de rendu doivent faire face à la difficulté d'exprimer du parallélisme et d'utiliser les fonctionalités offertes par le matériel dans ces modèles. Les compilateurs seuls ne pouvant pas optimizer et générer des programmes efficaces pour n'importe quel type de machine—parce que la recherche du programme le plus efficace est difficile et complexe—les programmeurs doivent donc utiliser des fonctions intrinsèques au compilateur, ou de manière générale, écrire du code non portable. La conséquence de cette limitation est que les programmeurs doivent réécrire le même code à chaque fois qu'ils veulent le porter vers une architecture différente, et que donc la base de code grossit, devient difficile à maintenir et comprendre car les algorithmes sont dissimulés sous une large quantité de détails liés aux aspects matériels de la machine.

Heureusement, il y a des solutions à ce problème, sous la forme de languages spécialisés à un domaine. Ces languages sont comme leur nom l'indique spécialisés dans un domaine, dont la connaissance permet aux compilateurs de choisir la meilleure méthode d'exécution sur une machine donnée lors de l'optimisation de programme. Dans cette thèse, on choisit une autre façon d'encoder cette connaissance: On implémente une bibliothèque de rendu et de tracé de rayons générique et déclarative dans un langage fonctionnel, et on la transforme ensuite pour l'exécution sur une machine particulière à l'aide d'annotations pour une évaluation partielle. L'évaluateur partiel spécialise alors le moteur de rendu dans son intégralité, en tenant compte du contenu de la scène: Les "shaders" sont spécialisés lorsque leurs entrées sont connues, et de manière générale, tous les calculs superflus sont éliminés. Les résultats montrent que les moteurs de rendu ainsi générés sont plus rapides et portables que ceux écrits avec les bibliothèques proposées dans l'état de l'art, et que, de plus, l'implémentation de cette technique demande moins d'effort.

Contents

Introduction

Computer Graphics and Compilers are two fields that belong together. This has become even more apparent since the invention of programmable shaders in 1988. Introduced by Pixar in the third version of the *RenderMan Interface Specification* [88], shaders are small programs that control the appearance of an object in a 3D scene. Since then, shaders have become more powerful and generic: They can describe surfaces, volumes, lights, textures, or literally any rendering parameter. Because of their ubiquity, any type of rendering software—from offline renderers to graphic drivers—has to include a shader compiler or interpreter at its core. Compilation makes the shaders run faster than interpretation, but more importantly enables optimizations that are critical for performance.

The typical optimization for shaders is specialization [GKR95; MQP02; Son+14]: Given a scene specification, constants can be folded and control-flow branches can be pruned at compile-time. However, there has been relatively few attempts at compiling more than just shaders inside a renderer. In fact, early attempts to specialize ray-tracing programs [And96] focus solely on intersection routines and shading. Clearly, there is also potential for specialization in other parts of a renderer: The integrator, acceleration data structures, and in general any scene-dependent code could benefit from knowledge of the scene that is about to be rendered. For instance, spectral rendering or instancing can simply be removed from the final executable, along with all the code that handles spectral data and instance transformations, if the scene configuration states that these features can be turned off. Ideally, one could even imagine writing a *renderer-generator*: A program that, given the definition of a scene, would generate a renderer specialized for that scene. This would be viable for both offline rendering—where compilation times do not matter, or real-time rendering, provided the renderer is not specialized for dynamic objects.

Designing such a program is a non-trivial task, much like designing a new compiler: The input and output languages for this compiler would be the scene description, and the machine code for the generated renderer, respectively.

This thesis argues that, using *partial evaluation* [Fut99], writing a new compiler can be avoided. This powerful technique requires that the input data of a program is split in two categories: static and dynamic. From such a program and a particular static input data, the partial evaluator yields a residual program that only operates on dynamic data. In order to specialize a renderer, it is hence sufficient to partially evaluate [Jon96] the renderer with the scene description as static data.

Unfortunately, there are multiple challenges linked with partial evaluation as a technique.

In particular, it is generally undecidable whether partial evaluation will terminate, due to the halting problem. Moreover, efficient rendering is not only about specialization. High-performance rendering requires fast hardware: GPUs, CPUs with vector units, or any other type of accelerator. It is crucial to be able to take advantage of such hardware, when it is available, since it offers significantly larger performance. This involves applying transformations to the code so that it performs better on said hardware: Vectorization, for instance, requires to linearize control flow. Sadly, just like for partial evaluation, compilers are usually too conservative when optimizing programs, and for good reason: Even for a simple program transformation, it can be hard or even impossible to write an automatic procedure that proves its correctness or determines its profitability.

Therefore, this thesis places this burden on the programmers' shoulders: The decision of *what*, *when*, and *how* to transform is their responsibility. Should the annotations be incorrect, the compiler will at best bail out with an error message, or even fail to terminate. This does not mean that programmers have to perform the transformation *themselves*: The compiler will do it in their stead, but it still requires instructions on how to do it properly.

In this thesis, we first present related work on rendering and compilers in Chapter 1. Then, we show how to develop high-performance, portable, and configurable traversal kernels using partial evaluation and guided-vectorization in Chapter 2. These traversal kernels are then used in Chapter 3, where we present a renderer generator based on partial evaluation. The central idea in that chapter is that even though this system *generates* renderers, it is not itself implemented as a generator, which is a very desirable property. Finally, in Chapter 4, we discuss some relevant design decisions made in the AnyDSL compiler framework, which is the supporting structure enabling this renderer generator.

The work of this thesis is based on several publications [Pér+17; Pér+18; Pér+19]. Other publications not directly related to the topic discussed here [PKS17] have not been included. All the code presented in this thesis is Open-Source and available at `https://github.com/AnyDSL/rodent`. The AnyDSL framework is also Open-Source, and available at `https://github.com/AnyDSL/anydsl`.

Chapter 1

Background

This section gives an introduction to rendering and compilers. The goal is to give the reader an overview of the two topics, and present concepts relevant to the rest of the thesis.

1.1 Rendering

Rendering is the process of generating a two-dimensional picture from the description of a three-dimensional scene. Such a description encompasses the definition of the light sources and objects in the scene, as well as the associated physical properties that are relevant for light simulation. From this description, the *renderer* will then solve the *Rendering Equation* [Kaj86] in order to produce an image.

1.1.1 The Rendering Equation

The Rendering Equation describes the amount of outgoing radiance for a given point on a surface as a function of the emitted radiance and incoming radiance for that point, when the system reached equilibrium:

$$L_o(x, \omega_o) = L_e(x, \omega_o) + \int_\Omega L_i(x, \omega_i) f(\omega_i, x, \omega_o) \left|\cos(\theta_i)\right| \mathrm{d}\omega_i \tag{1.1}$$

The terms of this equation are as follows: x is a point, ω_i is the incoming light direction, ω_o is the outgoing light direction, θ_i is the angle between the surface normal at x and ω_i, L_o is the outgoing radiance, L_e is the emitted radiance, L_i is the incoming radiance, and f is the Bidirectional Scattering Distribution Function (BSDF).

For surfaces that emit light, the radiance emission function L_e is part of the scene description, as an area light source definition. For non-emitting surfaces, this term is simply zero.

The BSDF also comes from the scene description, usually in the form of an analytical representation. Most rendering programs support a finite set of BSDFs, and the artist can then combine them and tune their parameters to get the desired surface appearance. Some tools also allow to capture the reflectance properties of a surface using a gonioreflectometer and then use

the measured data during rendering. This unfortunately comes at a price in performance, as the measured data is large and more complex to evaluate than an analytical model.

Note that the rendering equation as stated above does not allow rendering of wavelength-dependent effects, like dispersion. There are techniques and algorithms to effectively render scenes without this approximation [Wil+14], but they are beyond the scope of this thesis. Instead, we use a standard color space such as sRGB to represent colors, and we evaluate the rendering equation separately for each component. Similarly, this equation does not include participating nor polarization.

In free space, it holds that $L_i(x, \omega) = L_o(x, -\omega)$ [Vea98]. However, the incoming radiance L_i for a point x *on a surface* is linked to the outgoing radiance by the *ray casting function* c. Assuming the surface of the objects in the scene is the set S, we give a definition of c, along with the *distance function* d:

$$\begin{aligned} d(x, \omega) &= \inf \{ t \mid (x + t\omega) \in S, t > 0 \} \\ c(x, \omega) &= x + d(x, \omega)\omega \end{aligned}$$

Note that with this definition, the distance function d evaluates to $+\infty$ if the ray defined by x and ω exits the scene. We can now express the incoming radiance $L_i(x, \omega_i)$ as:

$$L_i(x, \omega_i) = L_o(c(x, \omega_i), -\omega_i)$$

With this relation, it becomes visible that the rendering equation is a Fredholm equation of the second kind [Fre03], for which the resolution method is to use the Liouville-Neumann series. In his PhD thesis, Veach [Vea98] introduced an operator formulation of light transport to simplify the mathematical presentation of the Neumann expansion. The idea is to build a *light transport operator*, defined as a combination of the scattering and propagation operators $\mathbf{K}$ and $\mathbf{P}$. The definition of $\mathbf{K}$ follows strictly the rendering equation integrand:

$$(\mathbf{K}h)(x, \omega_o) = \int_\Omega f(\omega_i, x, \omega_o) h(x, \omega_i) \left|\cos(\theta_i)\right| \mathrm{d}\omega_i$$

Now, the propagation operator $\mathbf{P}$ can be defined with the help of the distance and ray casting functions defined earlier:

$$(\mathbf{P}h)(x, \omega_i) = \begin{cases} h(c(x, \omega_i), -\omega_i) & \text{if } d(x, \omega_i) < \infty, \\ 0 & \text{otherwise} \end{cases}$$

With these definitions, the rendering equation can be reformulated as:

$$L = L_e + \mathbf{KP}L$$

With $\mathbf{T} = \mathbf{KP}$, the solution to the functional equation above is then:

$$L = (\mathbf{I} - \mathbf{T})^{-1} L_e$$

Of course this is under the condition that $\mathbf{I} - \mathbf{T}$ is invertible. A *sufficient* condition is to have $\|\mathbf{T}\| < 1$ where $\|\mathbf{T}\|$ is the standard operator norm. This is condition is guaranteed to be

true if the scene is physically valid [Vea98], which means in particular that BSDFs must be energy-conserving, symmetrical, and absorb at least some energy. Given that condition, the inverse is given by the Neumann series:

$$(\mathbf{I} - \mathbf{T})^{-1} = \sum_{i=0}^{\infty} T^i$$

As a consequence, the solution of the rendering equation is:

$$L = Le + \mathbf{T}L_e + \mathbf{T}^2 L_e + \ldots$$

This means that the final radiance is the sum of the emitted radiance plus the emitted radiance scattered once, plus the emitted radiance scattered twice, and so on.

Another useful transformation of the rendering equation is its *surface form*, obtained by change of variables:

$$L_o(x, \omega_o) = L_e(x, \omega_o) + \int_{y \in S} L_i(x, x \longrightarrow y) f(x \longrightarrow y, x, \omega_o) G(x, y) \, \mathrm{d}A \tag{1.2}$$

In this equation, S is the set of all surfaces in the scene, and A is the *area measure*. Note the new notation $x \longrightarrow y$, which represents the direction formed by the two points x and y. The function G represents the change of variables from solid angle to surface area, called the *geometry term*:

$$G(x, y) = V(x, y) \frac{\cos(\theta_1)\cos(\theta_2)}{||x - y||^2}$$

Here, θ_1 and θ_2 are the angles between $x \longrightarrow y$ and the surface normal at x and y, respectively, and V is the visibility function defined as:

$$V(x, y) = \begin{cases} 1 & \text{if } x \text{ is visible from } y, \\ 0 & \text{otherwise} \end{cases}$$

This surface form is used in particular for Next Event Estimation, an addition to the Path Tracing algorithm that improves its convergence. The next section reviews the basic concepts of Monte Carlo integration used in Path Tracing.

1.1.2 Monte Carlo integration methods

In the general case, evaluating the rendering equation cannot be done analytically, because of the complexity of the models used for the scene surfaces and materials. An alternative is to use numerical methods: Monte Carlo (MC) integration methods, in particular, are used pervasively in rendering because they require very little knowledge of the integrand and have good convergence properties. The general idea in MC integration is to see the integral of interest

as the expected value of a random variable:

$$\begin{aligned} I &= \int f(x)\mathrm{d}x \\ &= \int \frac{f(x)}{p(x)} p(x)\mathrm{d}x \\ &= \mathbb{E}_p\left[\frac{f(X)}{p(X)}\right] \end{aligned}$$

The notation $\mathbb{E}_p[X]$ refers to the expected value of the random variable X with respect to the probability distribution p. This expected value can be estimated by computing the following empirical average, called the *Monte Carlo estimator*:

$$I_n = \frac{1}{n}\sum_{i=1}^{n} \frac{f(x_i)}{p(x_i)}, \text{ where } X_i \sim p$$

According to the Strong Law of Large Numbers [RC05], I_n is converging almost surely to I. One advantage of MC methods is that their rate of convergence is independent from the dimensionality of the integral, since the variance of I_n is proportional to $\frac{1}{n}$. Intuitively, this means that with n large, one needs to take 100 times more samples to diminish the error by a factor of 10.

In order to improve the convergence of an MC estimator, variance reduction techniques, such as Importance Sampling (IS) can be employed. First introduced in 1950 [Kah50], IS takes advantage of the fact that there is some freedom in the choice of sampling distribution p. For instance, if we choose $p(x) = \frac{f(x)}{I}$, we get:

$$I_n = \frac{1}{n}\sum_{1}^{n} I = I \tag{1.3}$$

In this case, only one sample is enough to compute the integral (i.e. $I_1 = I$)! However, in order to be able to compute this optimal sampling probability p, one must already have the value of the integral I, which is in general not possible. What this well-known equality tells us, however, is that we should try to make the sampling probability as close to the integrand (f) as possible.

This technique is essential to improve the convergence speed of any MC algorithm, including Path Tracing, which is described in the next section.

1.1.3 Path Tracing

Path Tracing, an algorithm introduced at the same time as the Rendering Equation, in 1986 [Kaj86], has remained useful in many scenarios, because of its speed and simplicity. The Path Tracing (PT) algorithm is an MC integration method that builds a path by sequential sampling, starting from the camera, and sampling new random directions at every surface interaction in order to compute a Monte Carlo estimator of Equation (1.1). Algorithm 1 gives an overview of the naive, recursive version of PT.

Given a ray representing the outgoing light direction, this function intersects the scene to find a surface. If no surface is found, the function returns a black color. Otherwise, the

Algorithm 1 Basic recursive version of Path Tracing.

```
function PATH_TRACE(ray, scene)
    if ray exits scene then
        return SRGB(0, 0, 0)
    end if

    hitpoint ⟵ INTERSECT(scene, ray)
    emission ⟵ EMISSION(hitpoint)
    normal ⟵ NORMAL(hitpoint)
    sample ⟵ SAMPLE_UNIFORM_HEMISPHERE(hitpoint)
    pdf ⟵ 1/(2π)
    cos ⟵ COS(normal, sample)
    f ⟵ BSDF(hitpoint, ray, sample)

    return emission + PATH_TRACE(sample, scene) × f × cos / pdf
end function
```

hemisphere at the surface point is sampled, and an incoming direction is obtained. Note that the probability density of uniform hemisphere sampling is constant and equal to $\frac{1}{2\pi}$, the inverse of the surface of the unit hemisphere. The algorithm then proceeds with the evaluation of the integrand in Equation (1.1), and divides the value by the probability density, as in any Monte Carlo integration algorithm.

There are however multiple issues with this description of the algorithm: First, it may not terminate if the ray does not escape the scene. It might be tempting to introduce a maximum path length, but doing so will make the algorithm biased (which in practice means that the rendered image might be significantly darker than the correct solution, or even miss some important features). Instead, the correct solution is to use a technique known as *Russian Roulette* [AK90]: The idea is to randomly choose to terminate (or *kill*) the path. The termination probability can be constant, but it is better to adapt it to the current path throughput, so that paths with low contribution terminate earlier—or even better, guide it [VK16]. The amount of energy lost by terminating the path has to be accounted for with surviving paths: If the probability of killing a path is p and the path survives the Russian Roulette, then its throughput has to be divided by $1 - p$.

The second problem with this algorithm is that, at each surface interaction, new directions are sampled uniformly on a hemisphere. This choice is not optimal: In fact, it is even incorrect since some BSDFs are zero almost everywhere, meaning that the probability of sampling a direction on a hemisphere for which the BSDF is non-zero is literally zero: This happens for instance with perfect mirror materials.

Fortunately, IS can be used to solve this problem: Directions can be sampled using a sampling function that is adequate for each material. In the case of a perfect mirror, the sampling function can just return the only valid reflection direction. In the general case, though, the ideal sampling distribution should be as close as possible to the integrand. However, in the rendering equation,

the integrand contains the incoming radiance, which is the function that we are trying to compute. Thus, a simple and effective choice is to consider a sampling procedure that is proportional to the other two terms: the BSDF, and the cosine. In terms of implementation, this means that in the scene description, every BSDF now has to come with an accompanying sampling function. For instance, if the BSDF is purely diffuse (i.e. of the form $f(\omega_i, x, \omega_o) = k/\pi$, with $k \in [0, 1)$), then cosine-weighted hemisphere sampling is an appropriate choice of sampling probability.

1.1.4 Next Event Estimation

In its naive formulation, PT samples a direction on the support of the BSDF (usually a hemisphere) around the current point. Then the path is continued by performing a ray tracing step to find the corresponding point in that direction. Another way to build a path would be to sample a point on the surfaces of the scene directly. It is in general a very inefficient way to produce paths, as such points will often be occluded, in which case the geometric term will be zero. However, this can be done for light sources, since if a non-occluded point can be found on a light source, then the generated path will likely be largely contributing to the final radiance. This is the general idea behind Next Event Estimation (NEE) [Kaj86]: Separate emitting from non-emitting objects in the scene, and opportunistically sample a light source at every surface interaction. There are other variants where some heuristic or importance function is used to control when NEE is performed, but this is out of the scope of this introduction.

In order to include this in the PT algorithm, we need to sample a point y on the emitting surfaces of the scene, and then add the following contribution:

$$L_{NEE} = L_e(y) f(x \longrightarrow y, x, \omega_o) G(x, y) \frac{1}{p(y)} \tag{1.4}$$

Note that purely specular surfaces (those for which f is 0 almost everywhere) should not use NEE, as Equation (1.4) is always 0 in this case. Additionally, computing the geometric term $G(x, y)$ involves an evalution of the visibility function V, which in terms of implementation involves tracing a *shadow* ray. Shadow rays are in general faster to trace than normal rays, because here we do not care about finding the closest intersection: $V(x, y)$ is zero if there is *any* surface in between x and y.

Of course, because the original PT algorithm samples directions around the current point, and can thus end up building a path that hits a light source, we have to make sure that the contribution of a light source is not counted twice, as the result would be incorrect. A simple solution to this problem is to ignore the contribution of light sources when they are hit after a surface bounce, unless that bounce was specular (because NEE is not performed in that case, following the remark made above). Applying the simple solution means that paths will always have their last vertex sampled with NEE, which might not be optimal for some scenes, in particular when the angle between the light source normal and the outgoing ray is very shallow (see Figure 1.1). A much better solution is to use Multiple Importance Sampling to combine direct light hits with NEE.

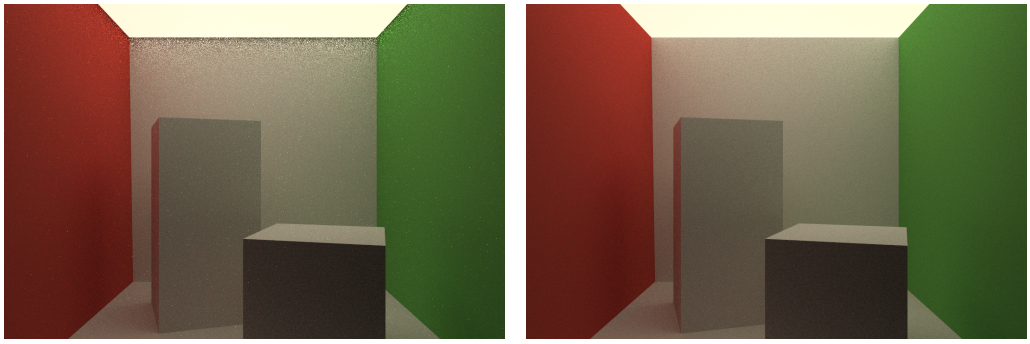


Figure 1.1: PT with NEE without MIS (left), and with MIS (right). On the top of the walls, it is apparent that NEE is inefficient at sampling the large light source. Scene inspired by the original Cornell box.

1.1.5 Multiple Importance Sampling

The idea behind Multiple Importance Sampling (MIS) [VG95] is to weight the contribution of direct light hits and NEE such that the total contribution is unbiased. Considering the last segment $x_{n-1} \longrightarrow x_n$ of a path whose last vertex x_n lies on a light source, it is possible to combine the contributions of NEE L_{NEE} and direct light hits L_D using their respective weights w_{NEE} and w_D to form the final estimator L:

$$L(x_{n-1} \longrightarrow x_n) = w_{NEE}\, L_{NEE}(x_{n-1} \longrightarrow x_n) + w_D\, L_D(x_{n-1} \longrightarrow x_n)$$

For this estimator to be unbiased, it is necessary that the weights w_{NEE} and w_D sum to 1 [VG95]. The balance heuristic [VG95] is a particular way of choosing those weights that gives more importance to the technique that has a higher probability of sampling the point on the light source. The weights for this heuristic are given here:

$$w_D = \frac{p_D(x_{n-1} \longrightarrow x_n)}{p_D(x_{n-1} \longrightarrow x_n) + p_{NEE}(x_{n-1} \longrightarrow x_n)}$$
$$w_{NEE} = \frac{p_{NEE}(x_{n-1} \longrightarrow x_n)}{p_{NEE}(x_{n-1} \longrightarrow x_n) + p_D(x_{n-1} \longrightarrow x_n)}$$

The terms $p_{NEE}(x_{n-1} \longrightarrow x_n)$ and $p_D(x_{n-1} \longrightarrow x_n)$ denote the probability densities for sampling the point x_n from x_{n-1}, using NEE and direct light hits only, respectively. In other words, $p_{NEE}(x_{n-1} \longrightarrow x_n)$ corresponds to sampling the set of emitting surfaces in the scene to find x_n, and $p_D(x_{n-1} \longrightarrow x_n)$ corresponds to sampling the BSDF at x_{n-1} and hitting the light source at x_n.

Typically, light sources are represented as a collection of objects in the scene. In order to perfom NEE, the renderer selects a light source S_i in this collection at random, using a discrete probability distribution $p(S_i|x_{n-1})$. In the remaining text, we assume that this probability is independent from the current surface point x_{n-1}, which means that $p(S_i|x_{n-1}) = p(S_i)$. Once a

light source is selected, the point x_n is then sampled on the surface of the light S_i, using the probability distribution function $p(x_n|S_i)$. In this case, the probability $p_{NEE}(x_{n-1} \longrightarrow x_n)$ for a point x_n can be decomposed into:

$$\begin{aligned} p_{NEE}(x_{n-1} \longrightarrow x_n) &= p(S_i|x_{n-1})p(x_n|S_i) \\ &= p(S_i)p(x_n|S_i) \end{aligned}$$

Note that the probability $p_{NEE}(x_{n-1} \longrightarrow x_n)$ is expressed in area form, whereas the path probability $p_D(x_{n-1} \longrightarrow x_n)$ is expressed in terms of solid angle [VG95]. Therefore, we need to transform $p_D(x_{n-1} \longrightarrow x_n)$ to the area measure by multiplying it by the geometry term $G'(x_{n-1}, x_n)$:

$$G'(x_{n-1}, x_n) = \frac{\cos(\theta_n)}{||x_n - x_{n-1}||^2}$$

Where θ_n is the angle between the normal at x_n and the vector $x_{n-1} - x_n$.

If we notice that the weight w_D can be simplified into:

$$\begin{aligned} w_D &= \frac{p_D(x_{n-1} \longrightarrow x_n)}{p_D(x_{n-1} \longrightarrow x_n) + p_{NEE}(x_{n-1} \longrightarrow x_n)} \\ &= \frac{1}{1 + \frac{p_{NEE}(x_{n-1} \longrightarrow x_n)}{p_D(x_{n-1} \longrightarrow x_n)}} \end{aligned}$$

And that similarly,

$$w_{NEE} = \frac{1}{1 + \frac{p_D(x_{n-1} \longrightarrow x_n)}{p_{NEE}(x_{n-1} \longrightarrow x_n)}}$$

Then, the final balance heuristic weights are:

$$\begin{aligned} w_D &= \frac{1}{1 + \frac{p(S_i)p(x_n|S_i)}{p_D(x_{n-1} \longrightarrow x_n)G'(x_{n-1}, x_n)}} \\ w_{NEE} &= \frac{1}{1 + \frac{p_D(x_{n-1} \longrightarrow x_n)G'(x_{n-1}, x_n)}{p(S_i)p(x_n|S_i)}} \end{aligned}$$

In terms of implementation, this only requires to attach one floating point value to the path, to keep track of $p_D(x_{n-1} \longrightarrow x_n)$ [Geo13].

In this thesis, we focus on rendering with PT combined with all the techniques mentioned in this section, but additional techniques such as guiding [VK16; MGN17], or different rendering algorithms [Vea98; Geo+12] can provide better convergence rates for complex scenes, at the cost of being more expensive per sample. Obviously, a complex algorithm may indeed provide better quality samples, but a decent Path Tracer will generate lower quality samples at a much higher rate. Another way to produce rendered images faster is by applying software optimization techniques, as shown in the following sections.

1.1.6 Traversal

Any algorithm based on ray tracing requires to *trace* rays in the scene. For instance, Algorithm 1 evaluates Equation (1.1) by finding the closest intersection between a sampled ray and any object in the scene. Similarly, Equation (1.2) contains a visibility term $V(x, y)$ which is 0 if the ray between x and y intersects any object in the scene, and otherwise 1. Since this operation is performed repeatedly during rendering, it follows that a large part of the rendering time is spent tracing rays, even in optimized renderers [Áfr+16; Lee+17].

High-performance renderers typically rely on an acceleration data structure to perform those ray queries. For instance, Bounding Volume Hierarchies (BVHs) represent the scene as a tree in which each node is associated with a bounding volume, and each leaf contains a list of primitives. In order to intersect a ray with the scene, this tree is traversed recursively from the root, processing only the nodes whose bounding volumes intersect the ray. Other data structures include grids [FTI86], Kd-trees [Ben75] or even octrees [Mea82], but major high-performance libraries like Embree [Wal+14] or OptiX [Par+10] typically prefer BVHs. One reason for this is that regular grids tend to perform poorly when the primitives are not distributed uniformly in the scene, a problem known as the *teapot in the stadium* [Wal04]. There exist solutions to this problem [KBS11; PKS17], but they often mean changing the construction and traversal algorithms considerably. Additionally, grids, Kd-trees, and octrees are spatial data structures: They subdivide *space*—unlike BVHs, which partition the set of objects in the scene. In a spatial data structure, primitives can appear in different nodes or cells, which requires to store *references* to the original primitive data. Consequently, spatial data structures use more memory and are often slower to build than BVHs, although high-performance construction algorithms do exist, e.g. [WH06].

On CPUs with vector units, BVH traversal routines are often vectorized to maximize their performance. There are four main categories of vectorized traversal algorithms: packet, single-ray, hybrid, and streaming traversal. Packet traversal is an algorithm that traverses the BVH with a group of N rays—a packet—with N being a multiple of the vector width [ORM08; Wal+01]. Intersection routines are then modified to intersect the N rays with a single primitive or bounding volume. When traversing the data structure, the children of the current node are pushed onto the stack if at least one ray in the packet intersects them. For this reason, this type of traversal is often deemed *speculative* [AL09], since only a subset of the rays in the packet might be active for a given subtree. Naturally, this makes packet traversal inefficient for *incoherent* rays: If the rays in the packet differ significantly in their origins and directions, they will most likely traverse different parts of the tree, meaning that only a few rays in the packet will be active at a time. Single-ray traversal algorithms try to solve this problem by optimizing the intersection of a single ray with the BVH. Typically, this is done by making the BVH wider and using vector instructions to perform intersection tests [DHK08; WBB08; EG08]. The BVH is built so that each node has N children, with N being the number of vector lanes for the target hardware. During traversal, the ray is then intersected with N bounding volumes at a time using Single Instruction Multiple Data (SIMD) instructions. Unfortunately, single-ray traversal is not as efficient as packet traversal for coherent rays: It fails to take advantage of the fact that similar rays traverse similar parts of the tree. The solution to this problem is to combine packet traversal with single-ray traversal [Ben+12] to form a hybrid algorithm. The idea is to start

traversal with packets, and switch to single-ray traversal when too many rays in the packet are inactive. This algorithm can therefore handle any kind of ray distribution, and is faster than both packet and single-ray traversal. Another way to extract coherence from ray distributions is to trace large groups of rays. Streaming traversal algorithms exploit this idea, by reordering or partitioning rays during traversal [Pha+97; Wal+07; Tsa09; BA14]. These algorithms can sometimes outperform hybrid ray traversal, but they require to process larger groups of rays.

On GPUs, BVH traversal routines should minimize memory traffic, and maximize SIMD efficiency [AL09]. Memory latency can also be a problem, which can be addressed by using wider BVHs [Gut14].

In order to minimize the amount of memory used per ray, and make BVH traversal more hardware-friendly, traversal algorithms can be made stackless. Sadly, stackless algorithms are slower than their stack-based counterparts [Lai10; BA13; Hap+13; ÁS14], since they need to perform additional work to remember the path taken in the hierarchy.

Even if traversal is often one of the most expensive parts of a renderer, shading can become the bottleneck in scenes containing complex materials. Fortunately, there are techniques to make shading more efficient.

1.1.7 Shading

Shading is a vague term that initially referred to the action of assigning a color for a given pixel: Early renderers would take the angle between the light source and the object visible from that pixel to produce a *shade* of gray [Pho75]. Since the introduction of complex material models, the term has been extended to refer to any computation that involves the material model. For instance, evaluating or sampling the BSDF in Algorithm 1 would be considered shading.

In most renderers, shading is programmable, and each material is assigned a small program called a *shader*. Depending on the type of material used, the shader might need to perform texture lookups, or evaluate some complex mathematical model. Shader execution can hence be either memory-bound or compute-bound, depending on the type of shader.

If the rays to shade are coherent, shader vectorization will be profitable [Áfr+16]: In this case, every SIMD execution thread of a vectorized shader will access similar parts of memory, and perform similar computations, maximizing both SIMD utilization and cache usage. However, for PT or any algorithm that produces incoherent workloads, sorting of large batches of rays may be necessary to extract enough coherence [Eis+13; Áfr+16].

Another way to speed up shaders is to *specialize* them. Shaders are typically written separately from the renderer, possibly even in a different programming language: This makes them reusable and independent from the rest of the renderer. However, this modularity comes at a cost, since the interface between the renderer and the shading system has to be generic enough to handle every possible shader. This interface introduces an overhead, but this can be avoided if the shaders are specialized to the type of scene and rendering algorithm [GKR95; MQP02; Son+14]. Of course, this requires to design a shader compiler that is tightly integrated with the rest of the renderer.

Implementing and optimizing shading systems or traversal algorithms is a tedious process, since compiler optimizations are often unreliable. Renderer developers therefore avoid costly

abstractions and use low-level, platform-dependent compiler intrinsics to maximize performance [Lee+17]. A similar observation can be made for GPU renderers, since they are written in platform-specific languages and duplicate parts (if not all) of their CPU counterparts. As a result, implementations become difficult to maintain and non-portable, which is undesirable. A first approach to this problem is to build reusable software libraries that abstract some or even all of the low-level details of rendering, as discussed in the next section.

1.1.8 Libraries

Two major types of libraries for rendering exist: High-level libraries target rendering as a whole, and let users of the library design an entire renderer [SS95; DH02]. Their focus is on ease of use, reusability, and genericity: Using object orientation, these libraries define classes for every rendering concept and define clean interfaces between them. This type of library is however difficult to optimize for a traditional compiler: Most virtual calls cannot be eliminated and the layout of objects in memory prevents effective vectorization.

Lower-level libraries like OptiX [Par+10] or Embree [Wal+14] define a scene management and traversal library, and contain heavily optimized implementations for ray traversal and intersection. Users of such libraries have to develop their own shading system and infrastructure around this base functionality. While such libraries do not suffer from the performance problems of higher-level libraries, they do not address most of the functionality required to build a complete renderer. Additionally, they offer only limited configurability, since they have been optimized for a particular usage scenario: Adding more features will bring the performance down for others scenarios.

Ultimately, renderers should be portable, fast, and generic. With traditional programming models and libraries, developers are forced to choose at most two out of those three: For instance, Vision [SS95] is portable and generic, and OptiX or Embree are fast but neither really generic nor portable. Modern compilers and Domain-Specific Languages (DSLs) offer an alternative to this conundrum: With domain-specific knowledge, compiler optimizations can be made more aggressive and transform high-level constructs into high-performance code. Previous works on compilers and DSLs for ray tracing have attempted to specialize [And96] ray tracers, or provide a vectorization and traversal library for renderers [GS08; ZWL17], but they were either too restricted in their scope, or limited by the choice of host language. Thus, the novelty of our approach is both in its scope—we implement an entire rendering library—and the method used—we leverage partial evaluation to generate high-performance code.

1.2 Compilers

Compilers traditionally transform or translate a program written in a programming language into another one (possibly in a different language). In this process, they might apply optimizations, in order to maximize the performance of the compiled program.

Some high-level optimizations try to *execute* as much code as possible during compilation (like constant folding), and others try to tune the code for a particular architecture (this includes transformations like parallelisation, vectorization, or even loop blocking). In any case, optimiza-

tions should be *safe*: They should not impact the correctness of the program. In the presence of non-termination, this becomes difficult to ensure, and compilers must be conservative in their decisions, because of the *halting problem*. Even when non-termination is not a problem, search spaces are often so large that heuristics must be used to make the problem tractable [ASU86]. As a consequence, some code that could have been executed at compile-time or folded into constants will remain unoptimized, and some beneficial transformations will not be performed automatically.

There are typically two solutions for these problems: Better heuristics, or user-driven annotations. Note that these approaches are not mutually exclusive, as heuristics can always help the user write annotations. This work will focus on the second approach, and is therefore connected to Domain-Specific Languages and techniques that perform guided optimization of the source program.

1.2.1 Domain-Specific Languages

Domain-Specific Languages (DSLs) are languages that are specifically designed and constrained for a particular domain. Compiling such a language is therefore easier than compiling a general-purpose language, and domain-specific knowledge can be exploited in order to make optimizations more aggressive. DSLs can either be implemented from the ground up, or embedded inside a host language, in which case most of the existing infrastructure for that language can be reused. We distinguish between two different types of DSL embedding: *deep*, and *shallow*.

Deep embedding refers to DSLs that *generate* specialized, high-performance programs: The user of such a DSL essentially builds a program generator. In order to generate programs, these DSLs rely on the ability to execute code at compile-time in the host language. This feature is also known as *staging*, since it allows to *stage* expressions of the source program for later evaluation (at runtime, for instance). Examples of deeply embedded DSLs are Spiral in Scala [Ofe+13], OptiML [Suj+11], or Liszt [DeV+11]. The first one uses Lightweight Modular Staging [RO10], a Scala library for building code generators, and the other two rely on Delite [Bro+11], a framework for parallel embedded languages. Another example is Halide [Rag+13], a DSL for image processing that leverages C++ as a host language.

On the other hand, shallowly embedded DSLs can be compiled by a standard, unchanged compiler for the host language. Nonetheless, these DSLs require a specialized compiler to produce efficient executables. A specialized compiler can optimize the program representation using domain-specific knowledge for the DSL, something that a general-purpose compiler cannot do. HIPA^{cc} [Mem+16] and SYCL[1] are two examples of this approach.

In this thesis, we use a form of shallow embedding that relies on partial evaluation to remove any abstraction overhead. Fundamentally, we are harnessing the first Futamura projection [Fut83], a famous result of meta-programming. In that setting, we write a renderer as a program $R(S, D)$, that given the description of a scene S (the *type* of lights, materials, geometries) and the scene data D (buffers, vertex arrays or textures), produces an image $R(S, D)$. This program can be viewed as a scene interpreter, taking the program S that describes how to render the scene, and D the data needed by program S. According to the first Futamura

[1] `https://www.khronos.org/sycl/`

projection, it is possible to partially evaluate R for S and yield a specialized renderer R_S that is identical to the result of executing a hypothetical *renderer generator* with the scene S as input.

With this technique, we get the best of both worlds: We do not have to reimplement a domain-specific compiler, as with traditional shallow embedding, nor do we write a program generator, as with deep embedding. Our code will get specialized at compile-time by a partial evaluator, requiring only a minimum amount of annotations. Conceptually, this is not really different from building a domain-specific compiler for rendering. While we do not write compiler code *per se*, we add domain-specific knowledge to a *generic* compiler in the form of annotations.

1.2.2 Vectorization

Vectorization is the process of transforming a *scalar* program where instructions operate on one value at a time, into a *vector* program in which each operation processes N elements of a specific data type simultaneously [NT98]. The transformed program may run faster if the target machine has vector instructions, which is the case for almost all modern processors. It is possible to automatically vectorize code, with recursive program vectorization [Ren+15], or loop vectorization [NZ08], for instance. These techniques are limited to certain classes of programs and will fail if the source program does not match their requirements.

An alternative solution is to add support for vectorization in the source language: `ispc` [PM12], a compiler for a C-based language with support for vector programming, and Sierra [LHH14], an extension to C++, use vector types and overloaded control-flow statements. In these languages, a variable can be *varying*, meaning that each vector lane may have a different value for that variable, or *uniform*, in which case every vector lane contains the same value. The problem is that types are a poor choice to represent this information: Variables can become varying either explicitly because of data-flow, or implicitly because of control-flow. It is therefore often difficult to write the most precise type for a variable, and subtle changes in the code, including those resulting from program optimizations, will make type information less accurate.

In regular programming languages with vector types where control-flow statements are not overloaded, the code has to be *linearized*: Non-scalar control flow must be replaced with data flow. The library RaTrace [Pér+17] performs this transformation manually, and uses type inference to avoid writing complex vector types. However, this forces the programmer to mask out inactive lanes by hand in conditionals, and clutters APIs and functions signatures with execution masks.

The semantics of the language can also be modified to enable an implicit vector execution model, as in CUDA or OpenCL. In this thesis, we follow this idea: We let the programmer annotate regions in which the program behaves as if it was executing on a SIMD processor. We rely on RV [MH18], a vectorization framework based on LLVM, to analyze the region and determine what can remain scalar, and what should be vectorized. The result of this analysis is a map from the variables and statements of the original program to a *vector shape*. RV vector shapes are affine[CDZ10], and represent the fact that a value is uniform or varying, along with additional information such as alignment and type of memory access, if any.

1.2.3 AnyDSL

AnyDSL [Lei+18] is the compiler framework that we will be using in this thesis. It consists of a language named *Impala*, and an intermediate representation named *Thorin* with a partial evaluator, and a backend targetting LLVM. Impala is based on an earlier version of Rust, and allows both an imperative and functional programming style.

Partial Evaluation

The AnyDSL partial evaluator can be controlled in Impala by specifying a *filter*, a small boolean expression, on the signature of a function. When the partial evaluator encounters a call to that function, it *instantiates* the filter by replacing occurences of the parameters of the function by the supplied arguments in the call. If the resulting expression evaluates to `true` after this substitution, then the partial evaluator inlines the call. As an example, consider the following function, whose filter is highlighted in orange:

```
fn @(?n) pow(x: i32, n: i32) -> i32 {
    if n == 0 {
        1
    } else if n % 2 == 0 {
        let y = pow(x, n / 2);
        y * y
    } else {
        x * pow(x, n - 1)
    }
}
```

The filter for this function is the expression `?n`, prefixed with the symbol `@` to indicate a filter. When the partial evaluator reaches the call:

```
pow(z, 5)
```

It replaces the symbolic parameter `n` by the constant `5` in the expression `?n`. The result is the symbolic expression `?5`, which in turns evaluate to `true`, because the operator `?` returns `true` if its argument is a constant at compile-time. Therefore, the call will be inlined, resulting in:

```
z * pow(z, 4)
```

Note that the partial evaluator executed the conditionals, and only the recursive call remains. Then, the procedure is repeated on the remaining call to `pow` until the final program reads:

```
let w = z * z;
z * w * w
```

When no filter is present, the partial evaluator does nothing. If the filter symbol `@` is present but no condition is given, it is as if the filter was always true. This means that the function `f` in the listing below will always be *recursively* inlined:

```
fn @f() -> i32 { 42 }
```

It is possible to obtain the same effect by annotating the *call site* of a function:

```
@@g()
```

This inlines the call to g, and that, regardless of its filter.

In Impala, the following `for` loop:

```
for i in range(0, n) {
    print(i)
}
```

is actually syntactic sugar for a call to `range`:

```
range(0, n, |i| {
    print(i)
})
```

In other terms, the `for` loop body is the third argument in a call to `range`, under the form of an anonymous function taking the loop counter `i` as parameter. In this way, the programmer can design his own iteration functions for other domains, such as a 2D image:

```
let img = Img { width: 640, height: 480, pixels: /* ... */ };
for x, y in iterate_img(img) {
    img.pixels(x, y) = 0;
}
```

In this case, a simple implementation of `iterate_img` will simply iterate through the rows and columns of the image:

```
fn iterate_img(img: Img, body: fn (i32, i32) -> ()) -> () {
    for y in range(0, img.height) {
        for x in range(0, img.width) {
            body(x, y)
        }
    }
}
```

The body of the for loop is available as a function parameter in `iterate_img`, allowing the programmer to apply various loop transformations. In general, this idea of passing the body of a loop as a function to a higher-order function that iterates over some domain is not really new and can in fact be linked to the `map` function of many programming languages [McC60].

Sometimes, it is necessary to modify the flow of the program: For instance, if the loop body contains an early exit or an error condition that terminates the loop. This is no longer possible to implement using regular functions, unless control-flow is encoded as data-flow (i.e. by placing the error condition in a boolean variable and testing it at every iteration). In order to solve this problem, Impala allows the use of *continuations* (informally, "functions that never return") to modify the control-flow of the program.

Continuations

First discovered in 1964 [Rey93], continuations represent control flow in a program, and can be used to encode loops, exceptions, or any control structure in a programming language. In Impala, continuations are first class citizens, and regular functions are encoded using continuation passing style. In fact, the following two functions are equivalent:

```
fn foo(i: i32) -> bool {
    i < 3
}
```

```
fn foo(i: i32, return: fn (bool) -> !) -> ! {
    return(i < 3)
}
```

Note how the return type `!` indicates a continuation.

Programmers can use explicit continuations to write functions that can return different types, just like with checked exceptions:

```
fn safe_div( x: i32, y: i32
           , return: fn (i32) -> !
           , error: fn () -> !
           ) -> ! {
    if y != 0 {
        return(x / y)
    } else {
        error()
    }
}
```

Additionally, standard continuations like **`return`**, **`break`** or **`continue`** can be captured and used like any other function:

```
for i in range(0, 100) {
    let exit_outer = break;
    for j in range(0, 100) {
        exit_outer() // Breaks out of the outer loop
    }
}
```

In this example, the **`break`** continuation is captured by the variable `exit_outer`, and later invoked inside the inner loop to exit from the outer one. In C, the same behavior could be obtained by placing a label at the end of the outer loop and using the **`goto`** keyword to jump to that label from the inner loop.

Vectorization and Device Code Generation

Impala also offers the possibility to vectorize and parallelize code. For instance, a more efficient implementation of `iterate_img` for CPUs with multiple cores and vector units could be:

```
fn iterate_img(img: Img, body: fn (i32, i32) -> ()) -> () {
    let num_cores = 4;
    let simd_width = 4;
    for y in parallel(num_cores, 0, img.height) {
        for x in range_step(0, img.width, simd_width) {
            for i in vectorize(simd_width) {
                body(x + i, y)
            }
        }
    }
}
```

The functions **`vectorize`** and **`parallel`** are provided by the compiler and perform vectorization using RV (see Section 1.2.2), and parallelization using Intel TBB, respectively. In the example above, vectorization is performed by extracting the region to vectorize, highlighted in orange, and marking every variable that is defined outside the vectorized region, like `y`, as uniform. RV will then propagate this information to determine the vector shapes of all the variables and statements in the region, and then linearize the control-flow graph accordingly, producing the vectorized program. As a consequence, there is no need to annotate types with vector information, as with `ispc` or other vectorizing compilers. This also means that the same function

can be first used with vector arguments, and can then be called later in the same program with scalar arguments. In this instance, RV will generate two versions of the function, one for each use case: We call this *polyvariant vectorization*. Both **parallel** and **vectorize** expect a compile-time known value for their first argument: the number of threads for **parallel**, and the number of vector lanes for **vectorize**. As an example, using the call `pow(2, 2)` with the definition of `pow` given above as the number of vector lanes (or threads) for **vectorize** (or **parallel**), is perfectly acceptable, since the call will be reduced to `4` at compile-time. Additionally, and in contrast to **#pragma** in C-based languages, these functions can be passed around to other functions, just like any regular Impala function.

Impala provides additional vectorization intrinsics: **any**, **all**, **shuffle**, **extract**, and **ballot**. These work in a similar way to the CUDA [20] intrinsics of the same name: **any** and **all** are functions that return **true** if any lane, or all lanes of their arguments are true, respectively. **shuffle** performs a circular permutation whose order is given by its second argument, **extract** extracts the contents of one vector lane, and **ballot** returns a bit mask containing ones for each lane in which the argument evaluates to **true**. When used outside of a vectorized region, these intrinsics behave as if there was only one vector lane. Here is an example showing the behavior of these functions:

```
for i in vectorize(4) {
    // i is a vector containing 0, 1, 2, 3
    let a = any(i > 3);     // a = false
    let b = any(i > 2);     // b = true
    let c = all(i > 2);     // c = false
    let d = all(i >= 0);    // d = true
    let e = ballot(i > 1); // e = 0b1100
    let f = shuffle(i, 2); // f = vector containing 2, 3, 0, 1
    let g = extract(i, 2); // g = 2
}
```

Other built-in functions are available: **cuda** triggers code generation for CUDA, and **amdgpu** does the same for AMD GPUs. These two functions extract a GPU kernel from a **for** loop body:

```
let grid = (64, 64, 64);
let block = (8, 8, 8);
/* Creates an 8x8x8 grid with block of size 8x8x8 */
for work_item in cuda(grid, block) {
    // same as threadIdx.x in CUDA or get_local_id(0) in OpenCL
    let thread_id_x = work_item.tidx();
    // ...
}
```

With this, a programmer can write a version of `iterate_img` that runs on a CUDA-enabled GPU:

```
fn iterate_img(img: Img, body: fn (i32, i32) -> ()) -> () {
    let grid = (img.width, img.height, 1);
    let block = (8, 8, 1);
    for work_item in cuda(grid, block) {
        let x = work_item.gidx();
        let y = work_item.gidy();
        body(x, y)
    }
}
```

Using such a method, the programmer can cleanly separate the *algorithm*—the **for** loop body—from its *mapping*—the way it is executed, specified as in `iterate_img`. In this thesis, we strive

to formulate algorithms in this way, using a very descriptive style of writing programs.

Vectorization, GPU execution, and specialization are extremely useful tools in general, and are in fact essential to write efficient BVH traversal kernels [Wal+14; Par+10]. Since AnyDSL offers these transformations as *primitives*, it becomes possible to write such kernels as high-level code, and then use those primitives to optimize them for the target architecture. The next chapter will explain in detail how to do this without losing genericity, portability, or performance.

Chapter 2

Generating BVH Traversal Kernels

Ray traversal is an essential operation in Path Tracing (PT) and every rendering algorithm based on ray tracing. Depending on the scene and rendering algorithm, more than half of the time can be spent traversing rays [Áfr+16; Lee+17]: Making ray traversal faster is thus essential to performance. In this chapter, we investigate a method to generate efficient, hardware-aware traversal kernels for different platforms and use cases, all from a common code base. The techniques and ideas presented here are based on a previous publication presenting traversal kernels written in Impala [Pér+17], adapted to reflect the changes in the compiler and renderer infrastructure that were made in a later publication [Pér+19].

2.1 Motivation

Libraries such as Embree [Wal+14], or OptiX [Par+10] expose highly-tuned traversal *kernels* developed by hardware vendors. These libraries contain different variants of the same algorithm, tuned for different use cases and different architectures.

However, these variants are implemented manually, which eventually means that the code will be duplicated instead of shared. An example of this problem is shown in Figure 2.1: This extract from Embree's implementation shows the same ray-triangle intersection algorithm implemented twice, once for packets of rays, and once for single rays. In general, this problem is not limited to the intersection routines, since the interface to the traversal has to expose vector types. The user of Embree, for instance, has to fill a packet of rays of a specific size (4, 8, or 16) and pass it to the traversal routines. If for some reason the user wants to port his implementation to another packet size, code has to be rewritten.

The reason why developers prefer to implement each version separately is mostly due to compilers. The current generation of compilers is not able to properly perform vectorization, and does not allow a good programming model to perform guided or semi-automatic vectorization. As a result, programmers have to rely on target-specific compiler intrinsics and manual vectorization to implement vectorized traversal kernels. Vectorizing compilers such as `ispc` do not offer the feature set required to implement complex traversal algorithms. Worse, they do not solve the code duplication problem as they force the programmer to annotate variable types with vector qualifiers.

```
__forceinline bool intersect(
  const vbool<M>& valid0,
  Ray& ray,
  const Vec3vf<M>& tri_v0,
  const Vec3vf<M>& tri_e1,
  const Vec3vf<M>& tri_e2,
  const Vec3vf<M>& tri_Ng,
  MoellerTrumboreHitM<M>& hit) const
{
  vbool<M> valid = valid0;
  const Vec3vf<M> O = Vec3vf<M>(ray.org);
  const Vec3vf<M> D = Vec3vf<M>(ray.dir);
  const Vec3vf<M> C = Vec3vf<M>(tri_v0) - O;
  const Vec3vf<M> R = cross(D,C);
  const vfloat<M> den = dot(Vec3vf<M>(tri_Ng),D);
  const vfloat<M> absDen = abs(den);
  const vfloat<M> sgnDen = signmsk(den);

  const vfloat<M> U = dot(R,Vec3vf<M>(tri_e2)) ^ sgnDen
      ;
  const vfloat<M> V = dot(R,Vec3vf<M>(tri_e1)) ^ sgnDen
      ;

  valid &= (den != vfloat<M>(zero)) &
            (U >= 0.0f) & (V >= 0.0f) &
            (U+V<=absDen);

  /* ... */
}
```

(a) Uniform ray, varying triangle

```
__forceinline bool intersectN(
  const Vec3vf<M>& ray_org,
  const Vec3vf<M>& ray_dir,
  const vfloat<M>& ray_tnear,
  const vfloat<M>& ray_tfar,
  const Vec3vf<M>& tri_v0,
  const Vec3vf<M>& tri_e1,
  const Vec3vf<M>& tri_e2,
  MoellerTrumboreHitM<M>& hit) const
{
  const Vec3vf<M> tri_Ng = cross(tri_e1,tri_e2);
  const Vec3vf<M> &O = ray_org;
  const Vec3vf<M> &D = ray_dir;
  const Vec3vf<M> C = Vec3vf<M>(tri_v0) - O;
  const Vec3vf<M> R = cross(D,C);
  const vfloat<M> den = dot(Vec3vf<M>(tri_Ng),D);
  const vfloat<M> absDen = abs(den);
  const vfloat<M> sgnDen = signmsk(den);

  const vfloat<M> U = dot(R,Vec3vf<M>(tri_e2)) ^ sgnDen
      ;
  const vfloat<M> V = dot(R,Vec3vf<M>(tri_e1)) ^ sgnDen
      ;

  vbool<M> valid = (den != vfloat<M>(zero)) &
                   (U >= 0.0f) & (V >= 0.0f) &
                   (U+V<=absDen);

  /* ... */
}
```

(b) Varying ray, varying triangle

Figure 2.1: Excerpt from the manually vectorized ray-triangle intersection routines in Embree 2.16.

Sadly, this is only part a small part of the picture: Since programming models and compilers used for GPU programming are profoundly different from their CPU counterparts, developers who want to port their traversal routine to GPUs will have to re-implement them from scratch. Evidently, current programming models for CPUs and GPUs are lacking in that regard.

Ideally, traversal libraries should be implemented as a set of reusable components that can be instanciated in different vectorization contexts. This would allow the CPU and GPU implementation to be derived from the same code base, with minor changes specifying target-specific details such as vector width or scheduling strategy.

This chapter presents an approach that allows for doing exactly this. Using AnyDSL [Lei+18], a compiler framework for high-performance applications, we describe a set of BVH traversal kernels that can be configured to a particular use case. Vectorization and GPU execution are performed transparently, from the same code base, and traversal algorithms share most of their logic. We start by presenting the common infrastructure, and then describe the traversal kernels themselves.

2.2 Common Infrastructure

Since the data layouts for BVH traversal vary depending on the architecture—e.g. GPU [AL09] or CPU [Wal+14]—and instruction set—e.g. AVX2 vs. SSE4.2 [Wal+14], it makes sense to abstract the data types over which a traversal algorithm operates. In particular, the data type representing a BVH should not enforce a particular arity, nor should it force a particular node

layout. In object-oriented languages, this could be achieved with an *interface* that specifies a *behavior*, and a set of implementations for this interface. However, in those languages, treating the object as an interface means that the compiler loses information about the concrete type of the object, and thus any call to a method will result in an expensive virtual function call. In Impala, we can build a generic data type that does not suffer from this drawback, by creating structures with function members. Take for instance the `Bvh` data type in our traversal kernels:

```
struct Bvh {
    node:     fn (i32) -> Node,
    prim:     fn (i32) -> Prim,
    prefetch: fn (i32) -> (),
    arity:    i32
}
```

With this structure, a traversal algorithm can access a particular node or a primitive by its index, and prefetch it if necessary. The `arity` member defines the number of children per node in the BVH (wider BVHs are used for single-ray traversal algorithms, as explained in Section 2.3). Note that each variable of type `Bvh` has its own fields, and that thus the members `node`, `prim`, `prefetch` and `arity` are *not* methods in the traditional sense.

Every concrete implementation of this abstract data type makes sure that calls to one of its fields are turned into regular function calls—not virtual function calls—using partial evaluation annotations. In practice, this means that any function that returns (or takes as an argument) a `Bvh` object has to be annotated with an `@` sign, to force its inlining. This is not as restrictive as it may seem, as such a function would have to be inlined anyway for performance: In a language that supports higher-order functions, closures are generated if the function passed or returned is not known at compile-time. Chapter 3 will discuss those annotations in more detail, and give a general rule to decide where to place them.

Another example is the node structure returned by the `node` member of a BVH:

```
struct Node {
    bbox:         fn (i32) -> BBox,
    ordered_bbox: fn (i32, RayOctant) -> BBox,
    child:        fn (i32) -> i32
}
```

Each node has a bounding box, accessible with the `bbox` field. When traversing a single ray, one possible optimization is to precompute the ray octant in order to minimize the number of comparisons in the ray-box test. To allow this optimization, a node can also return a bounding box with its near and far planes ordered according to a given octant via the `ordered_bbox` function. Finally, each node has at least two children whose indices are given by the `child` member. If this index is a negative number, it represents a leaf whose first primitive is located at the absolute value of the index plus one (so that 0 is a valid *inner* node index). Otherwise, it represents an inner node and points to the array of nodes of the associated `Bvh`.

Finally, one common aspect of all BVH traversal algorithms is that they eventually need to intersect a ray with a primitive. However, primitives are often grouped by packets in the BVH leaves. Therefore, we represent a primitive packet with the following structure:

```
struct Prim {
    intersect: fn (i32, Ray) -> Hit,
    is_valid:  fn (i32) -> bool,
    is_last:   bool,
    size:      i32
}
```

To intersect a primitive contained in the packet `prim`, the programmer must provide the index of the primitive in the packet, from 0 to `size - 1`, and a ray. The result of the intersection is a `Hit` structure containing the distance along the ray, along with the surface coordinates. Since the packet might not be full, it is possible to check if a primitive is valid or not by calling `is_valid`. There can be several packets in a leaf, and the last one to be processed has the `is_last` member set to **`true`**. If the implementation wishes to avoid using primitive packets, it is possible to set the `size` to 1, and get the expected behavior.

With this common design, intersection routines and BVH data layouts can be exchanged between traversal algorithms, at no performance cost, since the code will be specialized by the partial evaluator of AnyDSL. Therefore, our BVH traversal implementations act as generators, that take the description of a BVH data layout and a set of intersection routines as input, and produces an optimized traversal algorithm as an output. The flexibility of this approach is essential, since traversal algorithms may perform better or worse when paired with different data layouts and intersection routines (see Table 2.1 to see the impact of changing the BVH and ray data layouts, for instance).

2.3 CPU Kernels

Most modern processors offer a small vector instruction set, like SSE or AVX for x86 processors, or NEON for ARM architectures. Traversal kernels optimized for CPUs take advantage of these instructions by vectorizing intersection routines, and processing several rays or primitives together. In this section, we look at three vectorization strategies: ray packet, single-ray, and hybrid traversal.

2.3.1 Ray Packet Traversal

Ray packet BVH traversal is a form of speculative traversal: Several rays are traversed together, and a node is pushed onto the stack if *any* ray in the packet intersects it. Consequently, this type of algorithm performs better if the rays in the packet agree on the nodes to intersect, in which case the rays are said to be coherent (see Section 1.1.6). In order to simplify the presentation, we first discuss the implementation of a speculatively vectorized binary tree search, but as we explain later, this can easily be generalized to packet traversal. The following listing is an Impala implementation of this binary tree search, taking a tree and the element to search for, and returning **`true`** if the element is found:

```
fn traverse(stack: Stack, tree: Tree, elem: int) -> bool {
    let mut found = false;
    stack.push(tree.root)

    while !stack.is_empty() && !all(found) {
        let node = tree.node(stack.pop());

        if elem == node.elem {
            found = true;
        }

        if !node.is_leaf {
            if any(elem < node.elem) {
                stack.push(node.left)
            }

            if any(elem > node.elem) {
                stack.push(node.right)
            }
        }
    }

    found
}
```

In a binary search tree, for an inner node, all the elements of the left subtree are smaller than the current node, and all the elements of the right one are greater. The binary tree search algorithm exploits this property to only process the nodes that can potentially contain the element that is being searched for. This version of the algorithm is very similar: It pops the current node from the stack, tests if the element we are looking for is contained in the node, and if not, pushes the child of the current node that potentially contains the element on the stack. The only difference is the presence of calls to `any` and `all` to inform the compiler that a condition should be reduced over all lanes.

To trigger vectorization, the programmer has to place the call to `traverse` in a vectorized region:

```
let elems = [0, 1, 2, 3];
let mut found = [false, false, false, false];
let tree = binary_tree();
let stack = alloc_stack();
for i in vectorize(4) {
    found(i) = traverse(stack, tree, elems(i));
}
```

Let us now detail how RV [MH18] will determine the shapes of the variables in this example. Initially, the vectorized loop counter `i` is marked as affine with an offset of 0 and a stride of 1, meaning that the lane k of `i` is equal to $0 + 1 \times k$: In other words, `i` is the vector `[0, 1, 2, 3]`. Additionally, the variables `elems`, `found`, `tree`, and `stack` are marked uniform (identical for all lanes), because they are outside the vectorized region.

From this initial state, RV will infer that `found(i)` and `elems(i)` are varying, and that the memory accesses are contiguous and aligned, because of the shape of `i`. With this, it follows that for this call to `traverse`, the parameters `stack` and `tree` are uniform, and `elem` is varying. Inside `traverse`, every call to a reduction intrinsic like `any` or `all` will be considered uniform. Therefore, the calls to `push` will be uniform, as all lanes agree on when to push, and consequently, the

stack and traversal loop will remain uniform. only the variable `found` will be varying, because of the conditional assignment highlighted in orange. Indeed, since the condition is varying, the assignment to `found` is not executed by lanes for which the condition **`false`**.

By replacing elements to search in the tree by rays, and element comparisons by bounding volume intersections, we obtain a packet traversal algorithm. This simple analogy does not take into account the fact that BVH traversal order matters for performance, since we are often interested in the closest intersection. To solve this problem, we can store the distance with the packet of rays for each node in the stack. Then, a reasonable heuristic is to push a node under the top of the stack if its distance is larger than the distance of the top node, for **`all`** lanes:

```
for i in range(0, bvh.arity) {
    let (tentry, texit) = intersect_ray_box(packet, node.bbox(i));
    let hit = tentry <= texit;
    if any(hit) {
        let distance = if hit { tentry } else { inf };
        if all(stack.top().distance < distance) {
            // This pushes the node under the top
            stack.push_under(node.child(i), distance)
        } else {
            stack.push(node.child(i), distance)
        }
    }
}
```

Note that in this example, since the ray is varying, the entry and exit distances `tentry` and `texit` are also varying. The algorithm must hence ensure that lanes for which the ray does not hit the box get valid distance values using an **`if`**-expression.

As mentioned earlier, we do not have to annotate anything else than the vectorization region. This means that any function, including intersection routines like `intersect_ray_box`, can have its parameters vectorized differently without having to rewrite anything: It only depends on the shapes of the arguments at the *call site*.

2.3.2 Single-ray Traversal

Instead of processing several rays together, single-ray traversal extracts parallelism from the BVH itself. For this to work, the BVH must be made *wider*: Inner nodes must be large enough to contain N children, where N is the number of vector lanes. Similarly, leaves of the tree contain packets of N primitives, grouped together to make intersections faster.

The following listing intersects a single ray with multiple children of an inner node in a BVH: Varying or uniform vector shapes are given in the comments to simplify the exposition:

```
for i in vectorize(bvh.width()) {
    let (tentry, texit) = intersect_ray_box(/*uniform*/ ray, /*varying*/ node.bbox(i));
    let hit = /*varying*/ tentry <= texit;
    let mask = /*uniform*/ ballot(hit);
    for j in one_bits(mask) {
        stack.push(/*uniform*/ node.child(j), /*uniform*/ extract(tentry, j));
    }
}
```

In this listing, a ray is intersected with several boxes at a time, and the **`ballot`** intrinsic is used to get a mask indicating which children have been intersected. The `one_bits` function then

iterates through the set bits in this mask, using bit manipulation instructions. This mechanism is similar to `foreach_active` in ISPC.

Once the intersected children are pushed onto the stack, they need to be sorted according to their distance, or rely on some precomputed traversal order. When opting for exact sorting, the typical solution is to use a fast, in-register algorithm if the number of intersected children is small, and resort to a slow algorithm like insertion sort in the general case. Instead of implementing this logic manually, we rely on *sorting networks*, also known as comparator networks. These networks take a fixed number of input values in any given order, and sort them using a fixed sequence of compare-and-swap operations. Since the sequence of comparisons is fixed, a sorting network may be less efficient than other sorting algorithms like insertion sort or merge sort. However, this is also an advantage because this allows the implementation of a sorting network to operate only in registers, and avoid any costly branch.

In our traversal algorithm, we use both bitonic sort and the Bose-Nelson sorting algorithm to generate sorting networks, depending on the BVH arity. We achieve this by giving every sorting network the same type, a function that takes the number of elements to sort and a function to perform the compare-and-swap operation:

```
type SortingNetwork = fn (i32, fn (i32, i32) -> ()) -> ();
```

Since every sorting network now has the same type, we can choose the appropriate sorting network at compile-time in the traversal routine:

```
let sorting_network : SortingNetwork = match bvh.arity {
    8 => bitonic_sort,
    _ => bose_nelson_sort
};
```

This particular choice of networks has been determined based on performance measurements. The reason for this is that the performance of a sorting network is not only a function of its *size*—the number of comparisons it contains, but it also depends on its *depth*—the number of steps required to execute a network, assuming all non-conflicting comparisons can run in parallel. In turn, these parameters impact the final run-time performance, depending on the amount of Instruction-Level Parallelism that is exploited by the target machine, or the number of registers. Even if an accurate model for this was available, it would still be required to know how many registers a given compiler assigns to a sorting network and how the generated machine instructions are scheduled, based on those two parameters alone, which is difficult to predict as the comparison function might do more than just comparing and swapping two registers.

Sorting networks themselves are implemented so that they can be specialized for a particular size, by forcing the inlining of functions when the range of elements to sort is known. We give the Impala implementation of a specializable bitonic sort here, for instance:

```
fn @bitonic_sort(n: i32, cmp_swap: fn (i32, i32) -> ()) -> () {
    fn @(?i & ?len) merge(i: i32, len: i32, dir: bool) -> () {
        if len > 1 {
            let m = 1 << (ilog2(len) - 1); // Greatest power of two lower than len
            for j in unroll(i, i + len - m) {
                cmp_swap(select(dir, j, j + m), select(dir, j + m, j));
            }
            merge(i, m, dir);
            merge(i + m, len - m, dir);
        }
    }
    fn @(?i & ?len) sort(i: i32, len: i32, dir: bool) -> () {
        if len > 1 {
            let m = len / 2;
            sort(i, m, !dir);
            sort(i + m, len - m, dir);
            merge(i, len, dir);
        }
    }
    sort(0, n, true)
}
```

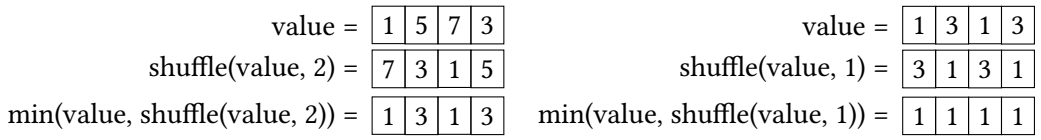
A call to `bitonic_sort(n, cmp_swap)` will generate a sorting network optimized for `n` elements and the given compare-and-swap function. It is important to note that writing similar code with C++ templates would require to duplicate the implementation: In our case, the algorithm works even if `n` is not known at compile-time, thanks to the filters attached to `merge` and `sort`, but a C++ implementation would have to choose whether `n` has to be a template or function parameter.

With all this machinery, we can generate and specialize sorting networks for each possible number of intersected children:

```
for n in unroll(0, bvh.arity + 1) {
    if n == bvh.arity || n_intr == n {
        sorting_network(n, @ |i, j| {
            if stack.elem(i).distance <
               stack.elem(j).distance {
                stack.swap(i, j)
            }
        });
        break()
    }
}
```

Since the number of intersected children `n_intr` is only known at run-time, it cannot be used to specialize sorting networks. Instead, the call to `unroll` will create `bvh.arity + 1` copies of the loop body, each instanciated with a different loop index `n` that is known at compile time. Thus, this code generates several sorting networks with different sizes, one for each possible case. Essentially, this snippet is equivalent to a manually written sequence of **`if`**/**`else`** statements, except it works with any BVH width.

A related problem arises when we intersect primitives: We only need the intersection with the smallest distance along the ray. Hopefully, this does not require sorting, but only finding the minimum value in a vector. This can be implemented with a vector reduction, either in hardware, if there is such an instruction, or emulated in software with a series of applications of the reduction operator. Since both SSE and AVX do not offer hardware support for finding



(a) First reduction step (b) Second reduction step

Figure 2.2: Finding the minimum of a vector using a logarithmic vector reduction.

the minimum value among the vector lanes of a register, we have to rely on software emulation. In Impala, a generic reduction operator can be implemented like this:

```
fn @reduce(width: i32, value: f32, op: fn (f32, f32) -> f32) -> f32 {
    fn @(?width) reduce_inner(width: i32, value: f32, op: fn (f32, f32) -> f32) -> f32
    {
        if width >= 2 {
            let shift = width / 2;
            let res = op(value, shuffle(value, shift));
            reduce_inner(shift, res, op)
        } else {
            value
        }
    }
    extract(reduce_inner(width, value, op), 0)
}
```

This function works by shifting the lanes of a vector by half the vector width, and applying the reduction operator on the result of the shift and the original vector, and repeating the operation until the vector width is just one. Because of the filters attached to `reduce` and `reduce_inner`, no recursive call will remain in the program after partial evaluation if the vector width is known, which is the case in our setting. The diagram in Figure 2.2 illustrates a reduction to find the minimum value of a vector register.

The astute reader will notice that the vectorization analysis of RV marks the result of a call to `reduce_inner` as varying. Indeed, if `value` is a vector, then the result of the **`if`** must also be a vector. This is unfortunate, as we know by definition of a reduction that the result should be uniform. RV cannot prove this property, because the return value is uniform only after the last recursive call: A compiler wanting to infer this would have to execute the function! We solve this problem simply by calling **`extract`** in the wrapper function `reduce`.

2.3.3 Hybrid Traversal

Hybrid traversal switches between ray packet and single-ray traversal depending on SIMD utilization, and is therefore an adaptive algorithm: Its performance does not degrade as much as ray packet traversal for incoherent rays, but it can still take advantage of ray coherence, when present. The algorithm starts by traversing the BVH with a ray packet, and checks, for every traversal step, that the number of rays intersecting the current node is above some threshold. When the threshold is reached, the algorithm switches to single-ray traversal for each active ray in the packet. A sketch of the algorithm is given below:

```
for i in vectorize(packet_size) {
    // Packet traversal loop
    /* ... */
    while !stack.is_empty() {
        // Test SIMD utilization
        let mask = ballot(active);
        if popcount(mask) < switch_threshold {
            for lane in one_bits(mask) {
                // Vectorize using a different vector width
                let ray = extract(packet, lane);
                for j in vectorize(bvh.arity) {
                    // Single-ray traversal
                    /* ... */
                }
            }
        } else {
            // Continue with packet traversal
            /* ... */
        }
    }
}
```

In this sample code, there are two calls to **`vectorize`**: One is outside the packet traversal loop, and the other is inside. This form of *nested* vectorization, where the vector width can be changed from inside a vectorized region, is something that other vectorization frameworks, like `ispc` [PM12] or Sierra [LHH14], cannot do naturally. `ispc` would require separate compilation of the packet and single-ray traversal, and Sierra would require the programmer to manually annotate types everywhere in order to make the switch possible.

On a machine with the AVX2 instruction set, we derived experimentally the best threshold for every combination of packet size and BVH arity:

```
let switch_threshold = match packet_size {
    4  => 3,
    8  => if bvh.arity == 4 { 4 } else { 6 },
    _  => packet_size / 2 // default case, not possible on AVX2
};
```

Note that these values might not be optimal for a different machine, since they depend on many factors, including the speed of the vector units, or the cache hit ratio and memory performance of each individual part of the algorithm. Nonetheless, it seems reasonable to expect that, for other machines, these values are not too far off from the optimum, since they are in fact very close or even identical to those used in Embree, even though they were derived independently.

Finally, note that the packet traversal part of the hybrid algorithm should operate on the same (wide) BVH as the single-ray part: This prevents cache pollution resulting from switching data structures [Ben+12]. In fact, in our implementation, we apply other memory-aware optimizations to maximize performance. More specifically, we tune the node or triangle data layout [DHK08] so that the size of a node or primitive is a multiple of the cache line size. For instance, the size of our 4-wide BVH node is 128 bytes, which is the same size as two cache lines on most CPUs: In that case, prefetching a 4-wide node can be done with exactly two prefetching instructions. Prefetching itself is only used in the single-ray variant, and there, the traversal rountine just calls the `prefetch` member of the node right after the ray-box test, in order to prefetch the children that were intersected.

2.4 GPU Kernels

Traversal kernels for GPUs do not require explicit vectorization: SIMD processors automatically take care of masking. Instead, other aspects will drive the performance of the traversal algorithm, like the traversal loop shape, or the use of texture memory [AL09]. Therefore, we use Aila and Laine's *while-while* loop layout [AL09], and read every node and triangle from the read-only texture cache. For this reason, and because the CUDA compiler and the NVPTX backend in LLVM are both extremely sensitive to the input program, we found that it was easier to design another traversal loop, instead of sharing the same one with the CPU traversal variants. Most importantly, we can still reuse the rest of the infrastructure, including intersection routines or sorting networks. Only the traversal loop, which amounts to around 60 lines of code, differs between implementations.

Another improvement is to use the PTX video instructions `vmin` and `vmax` to speed up the ray-box test on NVIDIA hardware [ALK12]. Those instructions allow to perform two minimum or maximum operations together, and can therefore be employed in the ray-box test of the traversal code. The ray-box intersection function supports this feature by taking user-specified comparison functions as arguments:

```
struct MinMax {
    fmaxmaxf:   fn (f32, f32, f32) -> f32,
    fminminf:   fn (f32, f32, f32) -> f32,
    fminmaxf:   fn (f32, f32, f32) -> f32,
    fmaxminf:   fn (f32, f32, f32) -> f32,
    fmaxf:      fn (f32, f32) -> f32,
    fminf:      fn (f32, f32) -> f32
}
```

This structure contains enough information to encode the classical ray-box slabs test [KK86], and the three-operand minimum and maximum functions can be implemented with `vmin` or `vmax` instructions on hardware that supports them.

The kernels are generated using either **`cuda`** or **`amdgpu`**, depending on the target machine, and we launch the traversal kernel with a 1-dimensional grid large enough to hold all rays:

```
let accelerator = match gpu_type {
    GpuType::AMD    => amdgpu,
    GpuType::NVIDIA => cuda
};

let grid  = (round_up(num_rays, block_width), 1, 1);
let block = (block_width, 1, 1);

for work_item in accelerator(device, grid, block) {
    let i = work_item.gidx();
    if i >= num_rays { continue() }

    hits(i) = traverse(rays(i), bvh);
}
```

In this example, rays and hits are loaded from memory, but it is also possible to generate them on the fly inside the kernel itself. The `device` variable holds the index of the target device. The `block_width` variable is used to specify the width of each block in the grid. Finding the optimal block width usually depends on the type of workload and the limitations of the kernel.

(a) Sponza (262K tris.) (b) Crown (3.5M tris.) (c) San Miguel (6.5M tris.) (d) Powerplant (12.8M tris)

Figure 2.3: Scenes used in our comparison.

In practice, we used a profiler to determine the optimal value for each traversal variant.

2.5 Results

In this section, we compare the performance and complexity of the traversal kernels presented above with hand-tuned implementations by hardware vendors.

2.5.1 Performance

Performance on x86-64 with AVX2

We present the evaluation of our traversal kernels on an Intel Skylake i7 6700K CPU with AVX2 in Table 2.1. As a reference, we use Embree 2.7.15, a manually optimized ray-tracing library by Intel written in C++. Both our kernels and Embree's use the same BVHs, obtained by using Embree's own high-quality BVH builder. We do not provide any number for packet tracing with Embree since it does not expose this variant. For a fair comparison, Embree is compiled with clang 7.0.1, which is based on the same LLVM version as the one used in the backend of Impala. The ray distributions used in this test come from a real renderer, but we only evaluate raw traversal times in these benchmarks, and exclude any shading. We stress different aspects of the traversal routines: As mentioned before, packet tracing is more efficient with primary rays than incoherent rays, for instance. Additionally, the test scenes (Figure 2.3) have different geometric complexity, and range from 262K to 13M triangles.

Overall, except for a few outliers, the performance of our traversal routines is always between −20% and +10% of Embree's. Considering that Embree is written with architecture-specific intrinsics, and has been optimized over the course of several years, this result shows that the vectorization features of Impala are competitive and can get really close to the gold standard.

The performance differences and the irregularities of the results observed in this evaluation are mostly due to the quality of the machine code generated by LLVM. One of the main reasons for this is that LLVM 7.0.1 generates lower quality assembly when the source code does not contain SIMD intrinsics: In particular, the register allocation and instruction schedule are often sub-par. In Figure 2.4, we give the assembly generated for the ray-box test of our kernels and Embree. The number of register spills and reloads and the overall instruction schedule is much worse in the assembly generated for our kernels than it is for Embree, except for the ray-packet

			BVH4					
			Primary		AO		Diffuse	
Scene	Fig.		Ours	Embree	Ours	Embree	Ours	Embree
Sponza	(2.3a)	Single	2.88 (-7%)	3.11	5.56 (-13%)	6.42	1.60 (-14%)	1.86
		Packet	7.72	–	17.14	–	1.39	–
		Hybrid	6.93 (-0%)	6.94	16.15 (+7%)	15.03	1.63 (-10%)	1.82
Crown	(2.3b)	Single	9.87 (-14%)	11.52	5.82 (-16%)	6.93	3.03 (-18%)	3.71
		Packet	22.24	–	7.46	–	2.79	–
		Hybrid	20.61 (+11%)	18.57	8.19 (-10%)	9.12	3.34 (-14%)	3.88
San-Miguel	(2.3c)	Single	2.17 (-21%)	2.74	3.01 (-23%)	3.93	1.08 (-23%)	1.41
		Packet	4.90	–	2.81	–	0.88	–
		Hybrid	4.41 (-2%)	4.51	3.31 (-18%)	4.05	1.03 (-24%)	1.37
Powerplant	(2.3d)	Single	4.77 (-21%)	6.03	9.10 (-16%)	10.78	2.05 (-18%)	2.50
		Packet	10.80	–	25.39	–	1.65	–
		Hybrid	9.75 (+0%)	9.75	22.86 (+10%)	20.74	1.98 (-20%)	2.49

			BVH8					
			Primary		AO		Diffuse	
Scene	Fig.		Ours	Embree	Ours	Embree	Ours	Embree
Sponza	(2.3a)	Single	3.95 (-6%)	4.19	7.75 (-6%)	8.25	2.08 (-9%)	2.29
		Packet	8.28	–	16.58	–	1.48	–
		Hybrid	7.13 (+1%)	7.08	16.03 (+9%)	14.70	1.98 (-8%)	2.14
Crown	(2.3b)	Single	12.08 (-7%)	13.06	7.37 (-10%)	8.22	3.68 (-16%)	4.40
		Packet	22.34	–	7.83	–	2.92	–
		Hybrid	19.81 (+11%)	17.80	8.78 (-7%)	9.47	3.76 (-12%)	4.27
San-Miguel	(2.3c)	Single	2.82 (-15%)	3.31	3.87 (-19%)	4.79	1.34 (-20%)	1.67
		Packet	4.36	–	3.00	–	0.90	–
		Hybrid	3.83 (-3%)	3.97	3.69 (-23%)	4.78	1.17 (-26%)	1.59
Powerplant	(2.3d)	Single	6.03 (-6%)	6.43	12.76 (-5%)	13.48	2.56 (-14%)	2.99
		Packet	9.91	–	23.98	–	1.77	–
		Hybrid	8.67 (+3%)	8.44	21.71 (+11%)	19.61	2.36 (-14%)	2.76

Table 2.1: Performance of every variant of our traversal kernels for two BVH branching factors compared with Embree, in **Mrays/s** (higher is better), measured on a Skylake i7 6700K. Speed-ups (slow-downs) with respect to Embree are indicated in parentheses. A dash (–) indicates that a traversal variant is not available in Embree. We perform 5 warm-up iterations and report the average of 20 runs. *Primary* rays start from the camera, *AO* rays compute Ambient Occlusion [Mil94], and *Diffuse* rays compute purely diffuse reflections.

intersection code, where our assembly appears mostly similar to that of Embree, or even slightly better. This explains why our single-ray variant is slower than Embree's in every case, and why our hybrid variant is most of the times marginally faster than Embree's for coherent rays: Remember that the hybrid variant is made of the packet and single ray traversal routines, and switches between the two based on SIMD efficiency. When rays are coherent, SIMD efficiency stays high and the traversal does not switch to single rays, so the results showcase mostly the performance of the packet traversal part, hence the overall good results in that scenario. The reverse observation can be made for fully incoherent rays, where our hybrid kernel performs the worst compared to Embree: There, the results are mostly due to the performance of the single-ray traversal part. For ambient occlusion [Mil94] rays, there is a mixed picture where our hybrid kernel performs better on some scenes (Sponza and Powerplant) than on others. This can be explained by the fact that ambient occlusion rays are more or less coherent depending on the scene and point of view [AL09].

Performance on ARMv8-A

Our routines can also be compiled for ARM CPUs with vector units. We list the performance of our routines on the CPU of the Jetson TX1 board in Table 2.2. This CPU is a Cortex A-57 with the NEON instruction set, which introduces 128-bit vector registers that can hold 4 single-precision floating point numbers. In that sense, this instruction set is similar to SSE for x86-64, except that it has 32 registers to work with, instead of 16 on x86-64. Additionally, like VEX-encoded SSE or AVX instructions, NEON instructions can have three operands.

Since, at the time of writing, there was no reference high-performance ray-tracing library on ARM, we compare the performance of the vectorized packet kernel against the scalar (non-vectorized) one. As side note, we obtain the scalar version simply by setting the vector with to 1 in the packet kernel. Due to the lack of instruction to perform a call to `ballot`, and because emulating this behavior with a sequence of instructions is too expensive, we do not list the performance of the hybrid or single-ray kernels.

The performance of the packet traversal is at its peak for coherent rays, including ambient occlusion rays, for which coherence is still present although usually lower than for primary rays. However, for completely incoherent rays generated by diffuse bounces, performance drops to the level of the scalar traversal, as expected.

Performance on NVIDIA GPUs

We evaluated the performance of our kernel on an NVIDIA GPU, and compared it with the state-of-the-art traversal kernel from Aila et al. [ALK12]. We present the results of our benchmarks on a GTX Titan X (Maxwell) GPU in Table 2.3. Overall, the performance of our kernel is between −10% and +14% of the reference. Since we chose to implement the same single-ray traversal algorithm as the one used in the reference traversal routine [ALK12], there is not much of a lesson to learn here: The different performance is not due to fundamental algorithmic reasons, or a different data layout, but is rather an artefact of the different compiler stacks used—`nvcc` for the reference, and the NVPTX backend of LLVM for our kernels. However, it is important to point out that we can reach the same level of performance with high-level, generic code,

```
vmovaps      64(%rcx,%r11), %ymm1
vfmadd132ps  %ymm5, %ymm4, %ymm1
vmovaps      64(%rcx,%rbx), %ymm2
vfmadd132ps  %ymm6, %ymm9, %ymm2
vpmaxsd      %ymm2, %ymm1, %ymm1
vmovaps      64(%rcx,%r10), %ymm2
vfmadd132ps  %ymm7, %ymm10, %ymm2
vpmaxsd      %ymm2, %ymm8, %ymm2
vpmaxsd      %ymm2, %ymm1, %ymm1
vmovaps      64(%rcx,%r15), %ymm2
vfmadd132ps  %ymm5, %ymm4, %ymm2
vmovaps      64(%rcx,%r13), %ymm3
vfmadd132ps  %ymm6, %ymm9, %ymm3
vpminsd      %ymm3, %ymm2, %ymm2
vmovaps      64(%rcx,%r14), %ymm3
vfmadd132ps  %ymm7, %ymm10, %ymm3
vpminsd      %ymm3, %ymm0, %ymm3
vpminsd      %ymm3, %ymm2, %ymm2
vpcmpgtd     %ymm2, %ymm1, %ymm2
vmovmskps    %ymm2, %esi
xorl         $255, %esi
vmovdqa      %ymm1, 320(%rsp)
je           .LBB98_76
```

(a) Embree, single-ray vs. multiple boxes

```
movslq       %r12d, %rbx
addq         $-1, %rbx
movslq       %r15d, %r15
movl         488(%rsp,%r15,4), %eax
addl         $-1, %r15d
shlq         $8, %rbx
addq         %r9, %rbx
vmovaps      (%rbx,%rdx,4), %ymm0
vfmadd213ps   %ymm8, %ymm13, %ymm0
vmovaps      (%rbx,%r8,4), %ymm1
vfmadd213ps  %ymm8, %ymm13, %ymm1
vmovaps      (%rbx,%r10,4), %ymm2
vfmadd213ps  %ymm9, %ymm14, %ymm2
vpminsd      %ymm2, %ymm0, %ymm0
vmovaps      (%rbx,%r11,4), %ymm2
vfmadd213ps  %ymm9, %ymm14, %ymm2
vpmaxsd      %ymm1, %ymm2, %ymm1
vmovaps      (%rbx,%rcx,4), %ymm2
vfmadd213ps  %ymm10, %ymm15, %ymm2
vpmaxsd      %ymm2, %ymm4, %ymm2
vpmaxsd      %ymm1, %ymm2, %ymm5
vmovaps      (%rbx,%rdi,4), %ymm1
vfmadd213ps  %ymm10, %ymm15, %ymm1
vpminsd      %ymm12, %ymm1, %ymm1
vpminsd      %ymm1, %ymm0, %ymm0
vpcmpgtd     %ymm0, %ymm5, %ymm0
vmovmskps    %ymm0, %esi
xorl         $255, %esi
je           .LBB13_101
```

(b) Our kernel, single-ray vs. multiple boxes

```
vbroadcastss 64(%r12,%rbx,4), %ymm2
vmovaps      1088(%rsp), %ymm0
vmovaps      1120(%rsp), %ymm3
vmovaps      1152(%rsp), %ymm4
vmovaps      1184(%rsp), %ymm5
vfmsub213ps  %ymm5, %ymm0, %ymm2
vbroadcastss 128(%r12,%rbx,4), %ymm6
vmovaps      1216(%rsp), %ymm7
vfmsub213ps  %ymm7, %ymm3, %ymm6
vbroadcastss 192(%r12,%rbx,4), %ymm8
vmovaps      1248(%rsp), %ymm9
vfmsub213ps  %ymm9, %ymm4, %ymm8
vbroadcastss 96(%r12,%rbx,4), %ymm10
vfmsub213ps  %ymm5, %ymm0, %ymm10
vbroadcastss 160(%r12,%rbx,4), %ymm5
vfmsub213ps  %ymm7, %ymm3, %ymm5
vbroadcastss 224(%r12,%rbx,4), %ymm3
vfmsub213ps  %ymm9, %ymm4, %ymm3
vpminsd      %ymm10, %ymm2, %ymm0
vpminsd      %ymm5, %ymm6, %ymm4
vpmaxsd      %ymm4, %ymm0, %ymm0
vpminsd      %ymm3, %ymm8, %ymm4
vpmaxsd      %ymm4, %ymm0, %ymm0
vpmaxsd      %ymm10, %ymm2, %ymm2
vpmaxsd      %ymm5, %ymm6, %ymm4
vpminsd      %ymm4, %ymm2, %ymm2
vpmaxsd      %ymm3, %ymm8, %ymm3
vpminsd      %ymm3, %ymm2, %ymm2
vpmaxsd      1376(%rsp), %ymm0, %ymm3
vpminsd      1408(%rsp), %ymm2, %ymm2
vcmpleps     %ymm2, %ymm3, %ymm2
vtestps      %ymm2, %ymm2
je           .LBB1_41
```

(c) Embree, ray packet vs. single box

```
vbroadcastss (%r9,%rcx), %ymm0
vfmsub213ps  %ymm15, %ymm10, %ymm0
vbroadcastss 64(%r9,%rcx), %ymm3
vmovaps      416(%rsp), %ymm7
vfmsub213ps  %ymm7, %ymm11, %ymm3
vbroadcastss 128(%r9,%rcx), %ymm4
vmovaps      448(%rsp), %ymm12
vfmsub213ps  %ymm12, %ymm14, %ymm4
vbroadcastss 32(%r9,%rcx), %ymm5
vfmsub213ps  %ymm15, %ymm10, %ymm5
vbroadcastss 96(%r9,%rcx), %ymm6
vfmsub213ps  %ymm7, %ymm11, %ymm6
vbroadcastss 160(%r9,%rcx), %ymm7
vpmaxsd      %ymm0, %ymm5, %ymm8
vpminsd      %ymm5, %ymm0, %ymm0
vfmsub213ps  %ymm12, %ymm14, %ymm7
vpmaxsd      %ymm3, %ymm6, %ymm5
vpminsd      %ymm5, %ymm8, %ymm5
vpminsd      %ymm6, %ymm3, %ymm3
vpmaxsd      %ymm0, %ymm3, %ymm0
vpmaxsd      %ymm4, %ymm7, %ymm3
vpminsd      %ymm7, %ymm4, %ymm4
vpminsd      %ymm1, %ymm3, %ymm3
vpminsd      %ymm3, %ymm5, %ymm3
vpmaxsd      %ymm4, %ymm2, %ymm4
vpmaxsd      %ymm0, %ymm4, %ymm0
vpcmpgtd     %ymm3, %ymm0, %ymm3
vmovmskps    %ymm3, %ebx
cmpl         $255, %ebx
jne          .LBB13_33
```

(d) Our kernel, ray packet vs. single box

Figure 2.4: Assembly generated by clang/LLVM 7.0.1 for the two ray-box intersection routines (single-ray/packet) of the hybrid ray-tracing algorithm in Embree (2.4a, 2.4c) and our implementation (2.4b, 2.4d).

			BVH4		
Scene	Fig.		Primary	AO	Diffuse
Sponza	(2.3a)	Scalar	0.22	0.44	0.19
		Packet	0.60 (+165%)	1.15 (+162%)	0.21 (+10%)
Crown	(2.3b)	Scalar	1.05	0.66	0.38
		Packet	2.08 (+98%)	0.85 (+28%)	0.42 (+9%)
San-Miguel	(2.3c)	Scalar	0.18	0.38	0.16
		Packet	0.45 (+152%)	0.42 (+10%)	0.16 (+6%)
Powerplant	(2.3d)	Scalar	0.45	0.72	0.24
		Packet	0.98 (+116%)	1.95 (+169%)	0.24 (+1%)

Table 2.2: Performance of a scalar (without vectorization) traversal kernel and a packet tracing kernel generated from our traversal code, in **Mrays/s** (higher is better), measured on a Cortex-A57 CPU. Speed-ups (slow-downs) with respect to the scalar traversal are indicated in parentheses. We perform 5 warm-up iterations and report the average of 20 runs. *Primary* rays start from the camera, *AO* rays compute Ambient Occlusion [Mil94], and *Diffuse* rays compute purely diffuse reflections.

whereas the reference often relies on non-portable constructs like inline PTX assembly and CUDA intrinsics.

Performance on AMD GPUs

We measured performance of our GPU kernel on an AMD R9 Nano, using the AMDGPU backend of Impala. Since there is no reference traversal library for this GPU, we can only provide absolute numbers. The results of this evaluation are listed in Table 2.4.

For primary rays, the performance of the traversal routine on this GPU is consistently between −20% and −28% of its performance on the GTX Titan X. With ambient occlusion rays, the performance comparably improves and is between −11% and −22%. Finally, on diffuse rays, the performance varies wildly, sometimes better (+25% on San-Miguel), worse (−23% on Powerplant), or even similar (−4% on Sponza).

In general, comparing these two GPUs is difficult, since their hardware architecture is different: The R9 Nano has wavefronts containing 64 work items, and uses HBM memory, while the GTX Titan X has warps made of 32 threads, and uses GDDR5 memory. Moreover, the software stack used in both cases is also not directly comparable: CUDA has established itself since 2007 as a platform to program NVIDIA GPUs, but ROCm, the software stack of AMD that we run our traversal kernels on, was only released in 2016. However, the numbers obtained in this evaluation are in line with benchmarks of these two GPUs: Folding@home, for instance, also shows a similar −20% difference between the two GPUs [Smi15b; Smi15a].

	BVH2					
	Primary		AO		Diffuse	
Scene	Ours	Aila et al.	Ours	Aila et al.	Ours	Aila et al.
Sponza	373.10 (+3%)	363.22	1031.68 (+6%)	975.01	146.28 (+2%)	143.59
Crown	788.19 (-3%)	816.36	372.40 (-7%)	401.61	157.34 (-4%)	164.53
San-Miguel	194.70 (-5%)	204.34	149.80 (-2%)	153.25	67.61 (+14%)	59.08
Powerplant	473.34 (-10%)	525.02	1086.86 (-2%)	1112.64	130.62 (-8%)	142.31

Table 2.3: Performance of the GPU traversal kernel on a NVIDIA GTX Titan X (Maxwell) GPU, in **Mray/s** (higher is better). Speed-ups (slow-downs) with respect to the traversal kernel of Aila et al. [AL09] are indicated in parentheses. We perform 100 warm-up iterations and report the average of 500 runs. *Primary* rays start from the camera, *AO* rays compute Ambient Occlusion [Mil94], and *Diffuse* rays compute purely diffuse reflections.

	BVH2		
Scene	Primary	AO	Diffuse
Sponza	297.27	903.33	137.39
Crown	599.86	289.89	136.37
San-Miguel	146.53	133.80	74.69
Powerplant	342.19	923.91	109.48

Table 2.4: Performance of the GPU traversal kernel on an AMD R9 Nano GPU, in **Mray/s** (higher is better). We perform 100 warm-up iterations and report the average of 500 runs. *Primary* rays start from the camera, *AO* rays compute Ambient Occlusion, and *Diffuse* rays compute purely diffuse reflections.

2.5.2 Implementation Effort

Since Embree provides more features than our library, a direct comparison with its code base is out of the question. However, it is clear that using SIMD intrinsics to vectorize code—either directly or by wrapping them inside structures—is not a practical way of vectorizing algorithms. In Embree, almost all intersection routines are duplicated at least twice: once for packets of rays, and a second time for single rays. Interestingly, the packet and single-ray implementations of each routine are mostly identical, except of course for the type of the rays. These routines alone represent 22% of the total number of lines of code in the whole library. Using the approach presented here, we could halve this number of lines of code.

Compared to existing auto-vectorization approaches, we can nest calls to `vectorize`: By doing so, we can easily change the vector width during execution. In `ispc` , this would require separate compilation and code duplication to enforce proper type signatures for variables and functions. Using Sierra, a programmer would have to use templates specifying the vector length and `varying`-ness of every function parameter. This is a direct consequence of the fact that in these frameworks, vector information is contained in the types, while Impala infers this information automatically.

Additionally, every variant of our CPU and GPU kernels share most of their code, except for the traversal loop. This means fewer bugs and less maintenance effort, but also allows porting the code to another platform easily. For instance, the source code provided by Aila et al. only works for NVIDIA GPUs, whereas our code can also run on AMD GPUs without any change.

In conclusion, our kernels are configurable, efficient, and *performance* portable. It is worth clarifying that the last point is *not* provided by frameworks like OpenCL. While it is true that code written using OpenCL is portable—provided the OpenCL driver follows the standard, the performance of an OpenCL kernel is not guaranteed to be similar across different machines. In fact, vendors often advocate completely different ways of writing high-performance kernels [Car18; AMD15]. Frameworks such as SYCL or Thrust that are built on top of OpenCL or CUDA are not the solution either, as they inherit all the problems of the underlying solution, mainly the lack of portability (if the SYCL implementation is not standard-compliant), or performance portability. Moreover, all these systems fail to provide a unified way to represent programs for any architecture: Software has to be written with one system or target architecture in mind, and it is difficult to predict its performance on any other platform, if it runs there at all. For that reason, designing a fully portable, high-performance renderer with those frameworks is an arduous task. As we show in the next chapter, AnyDSL is a compelling alternative for this, as it provides a common infrastructure to program different types of hardware, and apply domain- or hardware-specific optimizations with partial evaluation, triggered code generation, and vectorization.

Chapter 3

Generating Renderers

This chapter elaborates on the core idea of this thesis: generating renderers. It is mostly based on previously published work [Pér+19], and has been adapted and extended to better fit the format of this thesis. In the traversal kernels described in the previous chapter, partial evaluation was rarely used, except perhaps to generate sorting networks or to force the inlining of specific functions. Here, we will show how to use partial evaluation to generate an *entire renderer*. While this idea may sound surprising, it is in fact a very logical evolution of the current trends in renderer design: More and more renderers incorporate compilers for their shading systems, sometimes using existing frameworks [18b; 18a]. Such frameworks typically include optimizations that specialize shaders at run-time, removing redundant computations whenever possible. In our setting, we apply this technique to the entire renderer: We compile, optimize and specialize everything using as much of the (available) knowledge of the scene as possible.

3.1 Motivation

Rendering systems have never been more complex: Commercial renderers offer a huge set of features, all designed to improve the artist's creativity. This complexity is often getting in the way of performance, since the renderer has to test which features are enabled, and must contain code paths to handle each case, losing performance on the way. Hopefully, this problem can be solved for shaders: Specialization can be applied at run-time to generate a more efficient version of the shader in which unnecessary tests are elided [GKR95; MQP02; Son+14].

Naturally, shaders are not the only place where specialization can benefit performance. In fact, we argue here that the entire renderer can benefit from the knowledge of the scene. For instance, optimizing interfaces between rendering modules is essential: Knowing which mesh or vertex attributes are used, the size of each shader, or the total number of shaders in the scene will allow the renderer to make intelligent decisions for scheduling of different parts of the renderer, like shader fusion.

In order to specialize an entire renderer, we build a library of functions that describe the scene, following similar high-level APIs for rendering [SS95; DH02]. Thanks to partial evaluation, we have the same freedom and expressivity as any other functional language, at no cost: By placing annotations on constructors and the functions they return, closures are

guaranteed to be removed, along with any overhead. Because we separate the rendering code from the low-level aspects, we can repurpose the same algorithm on different machines, and with different parallelization strategies.

3.2 Rendering Library

Rodent, our rendering library, is a collection of interfaces with accompanying implementations, which we will just refer to as *abstractions*. The design of the API is influenced by domain-specific languages like Halide [Rag+13], or GraphIt [Zha+18], where the algorithm—*what* is computed—is decoupled from its execution—*how* it is computed. This is achieved by making interfaces generic enough to express different behaviors, in a very similar way to the various BVH-related data types introduced in the previous chapter. Additionally, the user of the library does not need to know how any of those abstractions work: Once combined into a scene description, they will be automatically specialized by the partial evaluator, following the annotations inside the library itself.

For clarity, this presentation only covers Path Tracing with MIS, but more complex algorithms can be implemented in the same way. We list abstractions by order of increasing complexity, separating rendering-related ones (Section 3.2.1 through Section 3.2.6), from those defining execution on the target hardware (Section 3.2.7).

3.2.1 Images and Textures

Let us first describe the ideas that underpin the design of this library by giving the example of textures and images. In Rodent, images are defined as a two-dimensional discrete collection of pixels:

```
struct Image {
    pixels: fn (i32, i32) -> Color,
    width:  i32,
    height: i32,
}
```

An important detail of this data type is that no particular layout is enforced on the image pixels. In fact, this image could even be procedural, as the `pixels` field is just a regular function, that can either return pixel data from memory, or compute some color based on the pixel coordinates.

Then, we can define an image filter as a function that maps an image and a position in the unit square $[0, 1]^2$ to a color:

```
type ImageFilter = fn (Image, Vec2) -> Color;
```

Note how the following constructor yields a *nearest neighbor* filter:

```
fn @make_nearest_filter() -> ImageFilter {
    @ |img, uv| {
        let x = min((uv.x * img.width  as f32) as i32, img.width -1);
        let y = min((uv.y * img.height as f32) as i32, img.height-1);
        img.pixels(x, y)
    }
}
```

The placement of partial evaluation annotations (introduced with `@`) is worth a couple of notes. Recall that our goal is to ensure performance, which in this case involves two things: avoiding closures, and allowing caller-callee optimizations. Closures have to be avoided since they require dynamic memory allocations for the captured environment. Unfortunately for us, returning a function from another function requires a closure. We avoid this by placing an annotation on `make_nearest_filter` that forces inlining: This is logical, after all, since constructors usually create objects and do not perform large computations—particularly in this example, where it just returns a function. Note that closures will also be generated for functions that capture variables from their environment, but this does not happen in our example. Finally, a good practice is to force inlining of small functions or functions that take other functions as arguments: This will allow caller-callee optimizations, which can be extremely beneficial. Consider the case where `img` is a constant color:

```
let black_img = Image {
    pixels: |_, _| black, // always black, regardless of position
    width: 1024,
    height: 1024
};
```

Because we annotated the anonymous function returned by `make_nearest_filter`, the `img` parameter is known inside the filter. Thus, when we call the filter with `black_img` as an argument, bounds check will be eliminated since the compiler will determine that `x` and `y` are not used in `img.pixels`. In Rodent, we follow this strategy to the letter: We annotate constructors and the functions they return with a force-inline (`@`) annotation.

There is an analogy to be made between this API design and object-oriented approaches: In an object-oriented language, a filter would be represented as an `ImageFilter` interface, and a nearest filter would be a class that derives from `ImageFilter`. In our library, returning a function is similar to creating an object that contains a *vtable*. The notable difference with object-oriented languages is that we give the compiler enough information to remove calls to functions, instead of relying on heuristics and devirtualization.

In Rodent, we define a border handling mode with a pair of functions that bring back texture coordinates into the unit square:

```
struct BorderHandling {
    horz: fn (f32) -> f32,
    vert: fn (f32) -> f32,
}
```

We apply the `horz` border mode for the first texture coordinate and `vert` for the second one. For example, the following constructor yields a *clamping* border handling mode:

```
fn @make_clamp_border() -> BorderHandling {
    let clamp = @ |x| min(1.0f, max(0.0f, x));
    BorderHandling {
        horz: clamp,
        vert: clamp
    }
}
```

Rodent defines textures as a mapping from 2D points to colors:

```
type Texture = fn (Vec2) -> Color;
```

From an `Image`, an `ImageFilter` and a `BorderHandling` object, we can build a texture:

```
fn @make_texture( border: BorderHandling
                , filter: ImageFilter
                , image: Image
                ) -> Texture {
    @ |uv| {
        let (u, v) = (border.horz(uv.x), border.vert(uv.y));
        filter(image, make_vec2(u, v))
    }
}
```

Application developers define textures by calling this constructor with the desired border handling mode and filter:

```
let tex = make_texture(make_repeat_border(), make_bilinear_filter(), image);
```

Evaluating the color for a particular texture coordinate is just a matter of invoking the texture function:

```
let color = tex(make_vec2(0.5f, 0.7f));
```

Because Rodent defines textures as ordinary functions, we can also implement procedural textures:

```
fn @make_checkerboard_texture() -> Texture {
    @ |uv| {
        let (x, y) = (uv.x as i32, uv.y as i32);
        if (x + y) % 2 == 0 { white } else { black }
    }
}
```

3.2.2 Materials and BSDFs

Recall the rendering equation from Section 1.1.1:

$$L_o(x, \omega_o) = L_e(x, \omega_o) + \int_\Omega L_i(x, \omega_i) f(\omega_i, x, \omega_o) \cos(\theta_i) \, \mathrm{d}\omega_i$$

This equation defines how light is scattered on a surface point x in the direction ω_o. The BSDF f controls scattering at point x, and can be represented with analytical models or measured data. In practice, artists define f by combining simple predefined models whose inputs are connected to textures or mesh attributes.

From the point of view of a MC simulation, though, a BSDF is an abstract object that can be evaluated and sampled. It might be useful to also provide other properties such as sampling probability, or flags that specify the type of material, so that the renderer can adapt its sampling strategy. In Rodent, we define BSDFs as follows:

```
struct Bsdf {
    eval:        fn (Vec3, Vec3) -> Color,
    pdf:         fn (Vec3, Vec3) -> f32,
    sample:      fn (&mut RndState, Vec3) -> BsdfSample,
    is_specular: bool
}
```

The `sample` function takes a random number generator state, an outgoing direction, and returns a `BsdfSample` containing the sample value, direction, sampling probability, and the cosine between the surface normal and the sample:

```
struct BsdfSample {
    color:  Color,
    in_dir: Vec3,
    pdf:    f32,
    cos:    f32
}
```

As an example, the purely diffuse BSDF $f_{k_d}(\omega_i, x, \omega_o) = k_d/\pi$ is defined like so:

```
fn @make_diffuse_bsdf(elem: SurfaceElement, kd: Color) -> Bsdf {
    let color = kd * (1.0f / pi);
    Bsdf {
        eval: @ |in_dir, out_dir| color,
        pdf:  @ |in_dir, out_dir|
            cosine_hemisphere_pdf(positive_cos(in_dir, elem.normal)),
        sample: @ |rnd, out_dir| {
            let sample = sample_cosine_hemisphere(rnd);
            BsdfSample {
                color:  color,
                in_dir: mat3x3_mul(elem.local, sample.dir),
                pdf:    sample.pdf,
                cos:    sample.dir.z
            }
        },
        is_specular: false
    }
}
```

This implementation follows a textbook definition: Evaluation always return k_d/π, and sampling on the hemisphere is weighted by the cosine between the direction and the surface normal [Dut03]. Here, `sample_cosine_hemisphere` returns a sample on the hemisphere oriented around the Z axis, so we have to transform it back into the local coordinate system at the hit point to obtain the final sampling direction `in_dir`. This also means that the cosine between the normal and the sampled direction is just the Z coordinate of the sample returned by `sample_cosine_hemisphere`. Note that in this example, when the parameter of the BSDF `kd` is known, the returned color evaluates to a constant, but this cannot happen if `kd` is obtained by looking up a texture or if it is an expression that is not resolved at compile-time. There is also a bit of redundancy in this code as both the value returned by `eval` and the `color` member of the returned `BsdfSample` are the same. That is not the case for BSDFs that are perfectly specular, because those have an `eval` function that always returns zero, even if the color returned by their sampling function is non-zero.

Additionally to the BSDF, the right-hand side of the rendering equation also specifies an emission function $L_e(x, \omega_o)$ that returns the emitted radiance at point x in direction ω_o. In Rodent, we model this concept by combining an emission function and a BSDF into a `Material` structure:

```
struct Material {
    bsdf:        Bsdf,
    emission:    fn (Vec3) -> EmissionValue,
    is_emissive: bool,
}
```

If the material does not emit light, the `is_emissive` flag is set to **`false`**. Alternatively, one could test if the emission value is black, but that would be more expensive than testing a single boolean flag if the test has to be done at run-time (e.g. when the emission value is not known).

The `emission` function returns an `EmissionValue` made of the intensity and probabilities for the given direction:

```
struct EmissionValue {
    intensity: Color,
    pdf_area:  f32,
    pdf_dir:   f32,
}
```

The `pdf_area` member represents the probability of sampling the current point on the surface, and the `pdf_dir` member corresponds to the probability of sampling the direction using emission sampling (sampling from the light source). These probabilities are necessary to implement MIS.

In order to build simple, non-emissive materials, we provide a handy constructor:

```
fn @make_material(bsdf: Bsdf) -> Material {
    Material {
        bsdf: bsdf,
        emission: @ |dir| EmissionValue {
            intensity: black,
            pdf_area:  1.0f,
            pdf_dir:   1.0f
        },
        is_emissive: false
    }
}
```

Note that emission probabilities are irrelevant, as the material is marked as non-emissive. Setting them to 1 lets the compiler optimize away any division or multiplication if the programmer forgot to check the `is_emissive` flag.

3.2.3 Lights

Light sources are usually kept distinct from other surfaces, in order to make NEE possible. Depending on the rendering algorithm, it might be necessary to sample them and produce a position and direction on the light source, as is for instance the case in Photon Mapping (PM) or any algorithm which includes tracing paths from the light sources. Furthermore, lights may not have an area and may be handled differently by the integrator: This is the case for point light sources, for instance.

In order to support these features, we represent lights with the following structure:

```
struct Light {
    sample_direct:   fn (&mut RndState, Vec3) -> DirectLightSample,
    sample_emission: fn (&mut RndState) -> EmissionSample,
    emission:        fn (Vec3, Vec2) -> EmissionValue,
    has_area:        bool,
}
```

Direct emission sampling is done during NEE: The integrator invokes `sample_direct` with a random number generator state and a point on a surface, and in return gets a sample containing a position on the light source, the light intensity and cosine on the light source for the

corresponding light direction, and a pair of probabilities:

```
struct DirectLightSample {
    pos:       Vec3,
    intensity: Color,
    cos:       f32,
    pdf_area:  f32,
    pdf_dir:   f32,
}
```

The probability `pdf_area` represents the probability of sampling the point on the surface of the light source. The other probability `pdf_dir` represents the probability of sampling the direction between the point on the surface and the point on the light source using direction sampling.

The `sample_emission` function generates a point and a direction (along with the probabilities) from a random number generator state:

```
struct EmissionSample {
    pos:       Vec3,
    dir:       Vec3,
    intensity: Color,
    cos:       f32,
    pdf_area:  f32,
    pdf_dir:   f32
}
```

This structure is mostly identical to `DirectLightSample` except that it additionally contains a direction. The probabilities are computed in a different way, though: When calling `sample_direct`, the `pdf_dir` member of the returned `DirectLightSample` value is the probability of sampling the direction between the given surface point and the sampled point on the light source, as if it had been sampled using emission sampling. When calling `sample_emission`, the probability is the actual probability of sampling the returned direction.

As an example, a point light source could be implemented by connecting the point on the surface to the light position to create a direction in `sample_direct`, and sampling a direction uniformly around a sphere centered on the light position in `sample_emission`.

3.2.4 Geometric Objects

Inside a rendering algorithm, geometric objects act like proxies to the surface information, and are associated with a *shader*. With this in mind, Rodent represents geometric objects like this:

```
struct Geometry {
    surface_element: fn (Ray, Hit) -> SurfaceElement,
    shader: Shader
}
```

The `SurfaceElement` structure represents the surface element at the hit point, with a normal, tangent, bitangent, and interpolated vertex attributes, if any. The renderer invokes `surface_element` on the geometric object with the current ray and hit to obtain the surface information at the hit point. Then, it passes this information to the shader contained in the object, along with the ray and hit, in order to produce a `Material`. Note that the shader is an ordinary Impala function, and not a shader program written in another shading language. Other languages can of course be supported by implementing a translator or compiler between from that language and Impala.

3.2.5 Shaders

In Rodent, shaders produce a `Material` from a ray, a hit point, and the surface information around the hit point:

```
type Shader = fn (Ray, Hit, SurfaceElement) -> Material;
```

Unlike in libraries like OptiX, these shaders are only describing a material and do not have to return a new ray to continue the path: Only the renderer decides how paths are traced and defined.

As an example, here is a physically-corrected *Phong* shader [Dut03] controlled by a texture:

```
let image = device.load_image("data/textures/wall.png");
fn phong_like_shader( ray: Ray
                    , hit: Hit
                    , elem: SurfaceElement
                    ) -> Material {
    let tex = make_texture(make_repeat_border(), make_bilinear_filter(), image);
    let diffuse = make_diffuse_bsdf(elem, tex(elem.uv_coords));
    let ks = make_color(0.9f, 0.9f, 0.9f);
    let ns = 150f;
    let specular = make_phong_bsdf(elem, ks, ns);
    let bsdf = make_mix_bsdf(diffuse, specular, 0.5f);
    make_material(bsdf)
}
```

In this example, we use the `device` (see Section 3.2.7) to load an image, and then create a material from the combination of two BSDFs. The diffuse component is driven by a texture created from the previously loaded image, with the repeat border mode and bilinear filtering.

An interesting feature of this listing is that it contains no partial evaluation annotations. This property is vital because shaders are part of the user interface of the renderer: They are often written with little to no knowledge of its internal design. To avoid writing annotations in the shader, it is sufficient to place annotations on the *calls* to the shaders used in the renderer using the call-site annotation syntax `@@` introduced in Chapter 1.

3.2.6 Renderers

The traditional description of a ray-tracing algorithm like Path Tracing (PT) is often decomposed into several steps (see Figure 3.1): emission of camera rays, acceleration data structure traversal, shading computations, emission of shadow ray, and continuation or termination of a path [Gla89]. It turns out that most variants of PT can be represented using this framework, even bidirectional algorithms like Bidirectional Path Tracing—in this case, by starting the process both from the camera and light sources and adding a final connection step to form full paths.

In Rodent, we build on this idea and represent a renderer using the following structure to represent rendering algorithms:

```
struct Tracer {
    on_emit:   OnEmitFn,
    on_hit:    OnHitFn,
    on_shadow: OnShadowFn,
    on_bounce: OnBounceFn
}
```

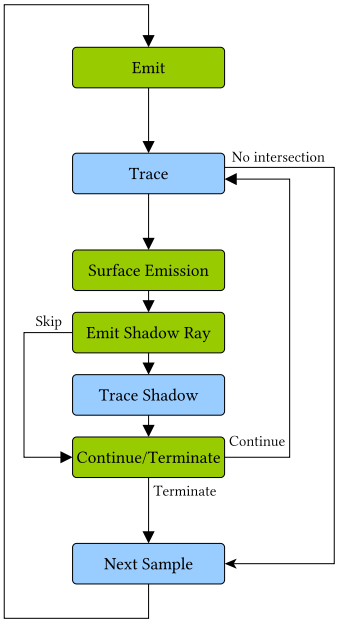


Figure 3.1: Diagram of a standard ray-tracing algorithm like Path Tracing. Green nodes are represented by the members of the `Tracer` structure. Blue nodes are scheduling nodes that are implemented in the rendering device (see Section 3.2.7).

This structure represents the actions the renderer will take when emitting rays, hitting a surface, tracing shadow rays, or bouncing off a surface. Most of those member functions contain continuations in their type, so as to allow the renderer to perform different actions, depending on the situation. For instance, the type `OnShadowFn` is in fact defined as the following function type:

```
type OnShadowFn = fn (
    Ray, Hit, &mut RayState, SurfaceElement, Material,
    fn (Ray, Color) -> !,
    fn () -> !
    ) -> !;
```

Intuitively, this type can be read as: "A function that takes a ray, a hit, a state for the ray, the surface information, and a material, and either returns a ray and a color, or nothing". In fact, this function type indicates that there are two *return continuations*: one of type `fn (Ray, Color) -> !`, and the other of type `fn () -> !`. While a normal function only has one return continuation, having more than one allows to return different values depending on the situation. This is similar to checked exceptions in Java [Gos+14], or sum types in Haskell [Mar10], where the programmer can encode several possible outcomes in the return value of a function. The other members types `OnEmitFn`, `OnHitFn`, and `OnBounceFn` follow the same idea.

Once a renderer is implemented with this structure, it becomes possible to reorder and schedule the emission, tracing, or shading of rays in many different ways. Since there are many options to consider, domain knowledge is necessary to choose the particular schedule that will maximize device utilization and parallelism, which is why the concept of rendering device is

presented next.

3.2.7 Rendering Devices

In Rodent, rendering devices define the actual *mapping* of a rendering algorithm to the hardware device. Essentially, a rendering device is a scheduler that specifies when to run each part of the renderer, and on which rays. It also contains helper functions to load data (such as an image or a mesh) into the device, which is necessary for GPU execution, and maintains acceleration data structures. Rendering devices are represented with the following structure:

```
struct Device {
  trace:      fn (Scene, Tracer) -> (),
  load_image: fn (&[u8]) -> Image,
  // ...
}
```

The `trace` function schedules a rendering algorithm defined by a `Tracer` object on the device, and effectively renders the given scene.

Thanks to this library design, the user of the library can easily port all his rendering algorithms to a new platform: He only has to implement one new device to perform scheduling. Similarly, rendering algorithm developers can harness high-performance devices already implemented in the library and focus on writing high-level rendering code instead of low-level scheduling code.

Currently, three rendering devices are implemented:

1. A tile-based CPU device that uses vectorization (and counting sort to order rays by material and maximize SIMD utilization) to shade rays, and parallelizes tiles over several CPU cores,

2. A megakernel-based GPU device,

3. A wavefront-based GPU device [LKA13].

Each of these devices is designed to allow optimization of all rendering subsystems and the interface between them. In particular, they perform shader specialization, either by using a static dispatch mechanism (as in the megakernel GPU device), or by sorting rays by material and processing contiguous ranges of rays (as in the CPU and wavefront GPU devices).

As an example, in the `trace` function of the CPU device, we render tiles of the image in parallel. For each of these tiles, we maintain a stream (wavefront) of rays that we traverse and shade together. Before shading, we sort rays by increasing geometric object ID—and thus, also by shader—and process ranges of rays within the stream with the same shader:

```
fn trace(scene: Scene, tracer: Tracer) -> () {
    // ...
    for geom_id in unroll(0, scene.num_geometries) {
        // Get the range of rays that hit this geometric object
        let (begin, end) = geometry_range(geom_id);
        for i in vectorize_range(vector_width, begin, end) {
            // Use tracer.on_hit/on_bounce/...
        }
    }
    // ...
}
```

This extract assumes that rays are already traced and sorted, and that the ranges of rays that hit the same geometric object are returned by `geometry_range`. The function used to iterate over a range of rays, `vectorize_range`, is merely a wrapper around the **`vectorize`** built-in function. Unlike **`vectorize`**, this function can process any range, including those whose size is not a multiple of the vector size.

In order to specialize the shaders, we need to unroll the outer loop running over geometric objects. Using the `unroll` function, we produce n different copies of the inner loop, where n is the number of geometric objects in the scene. Thanks to partial evaluation annotations placed inside `unroll`, each of these copies is specialized for a different object, and any computation depending on the type of material or geometry is transparently optimized.

The wavefront-based GPU device functions is implemented in a comparable manner, except that the ray streams correspond to an entire image, and that the inner loop uses **`cuda`** instead of `vectorize_range`. Thanks to this, we generate different, specialized compute kernels for each geometric object.

The megakernel-based GPU device, on the other hand, does not use streams at all. One unique kernel contains the entire rendering loop, in which each execution thread is in charge of tracing one path:

```
fn trace(scene: Scene, tracer: Tracer) -> () {
    for work_item in cuda(grid, block) {
        let (x, y) = (work_item.gidx(), work_item.gidy());
        let (ray, state) = tracer.on_emit(x, y);
        let mut terminated = false;
        while !terminated {
            // Intersect, shade and continue path
        }
    }
}
```

Typical megakernel-based renderers often exhibit high register pressure and important execution divergence. In a wavefront-based renderer, the individual register requirements of each shader do not impact each other. But when all shaders are merged together into one megakernel, the total number of registers used by the kernel is equal to the maximum register requirement among all shaders. Since, in general, registers limit occupancy, and therefore parallelism, every shader in a megakernel becomes as expensive as the most expensive shader. However, wavefront-based renderers have to transfer results from kernel to kernel, resulting in increased global memory traffic compared to a megakernel. The trade-off between the two designs is discussed further in the literature [LKA13], and the consensus is that with the current architectures, a wavefront-based renderer will scale better than a megakernel-based one—a

conclusion corroborated by our results.

Regardless of performance considerations, a notable property of our library is that we do not enforce any particular schedule. Other, not yet invented schedules will not be difficult to add, because we clearly separated the algorithm—the `Tracer`, from its *mapping*—the `Device`. On the contrary, in OptiX, the programmer has no choice but to develop a megakernel renderer. We argue that a clean API design should focus on the high-level concepts of rendering, and not force the programmer into a programming model, like what OptiX, `ispc` , or Embree do. Without proper *separation of concerns*, developers are bound to duplicate code and maintain different code bases in order to provide performance portable software.

3.3 Results

In this section, we evaluate the performance and code complexity of renderers written with Rodent, compared to renderers written with state-of-the-art, high-performance rendering libraries.

3.3.1 Experimental Setup

Reference Renderers

Embree [Wal+14] and OptiX [Par+10] offer low-level APIs to design renderers. In order to compare them against Rodent, we have implemented renderers with both of them, following the examples provided in their respective documentation (the Embree example renderer[1] and the OptiX path tracer [2]). These reference renderers only support triangle meshes of a certain type and a small set of predefined shaders corresponding to the set of possible materials in our tested scenes, so as to make them more efficient for the tested scenes and avoid unecessary computations. Rodent supports more features and is more flexible than those renderers, and relies on partial evaluation to remove redundant computations automatically. Moreover, our Embree implementation uses `ispc` [PM12] to vectorize shading code. Acceleration data structures performance hints suggested in the documentation have been applied for both renderer implementations: We use the SBVH data structure in OptiX and Embree, and make sure to have a minimum amount of divergence for both implementations.

Generating Renderers with Rodent

Since Rodent expects a scene description written in Impala, we have written a small converter from a 3D scene format to Rodent's scene representation. In order to generate a renderer, we only need to run the converter on a scene of our choice, choose a device (e.g. by inserting a line like `let device = make_cpu_device()` in the converted scene or specifying a command line option in the converter), and then compile the result using Impala. The final result is a renderer specialized for that scene that runs on the chosen device. A real application can embed the Impala compiler and convert a scene at run-time, using Just-In-Time compilation.

[1] see `https://embree.github.io/renderer.html`

[2] https://developer.nvidia.com/designworks/optix/download

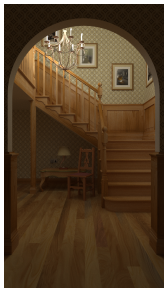
(a) Living Room

(b) Bathroom

(c) Bedroom

(d) Dining Room

(e) Kitchen



(f) Staircase

Figure 3.2: Scenes used in the performance evaluation.

	CPU (i7 6700K)		GPU (Titan X)			GPU (R9 Nano)	
Scene	Rodent	Embree	Rodent[1]	Rodent[2]	OptiX	Rodent[1]	Rodent[2]
Living Room	9.77 (+23%)	7.94	38.59 (+25%)	43.52 (+42%)	30.75	24.22	33.42
Bathroom	6.65 (+13%)	5.90	27.06 (+31%)	35.32 (+42%)	20.64	14.48	26.19
Bedroom	7.55 (+ 4%)	7.24	30.25 (+ 9%)	38.88 (+29%)	27.72	18.49	30.72
Dining Room	7.08 (+ 1%)	7.01	30.07 (+ 5%)	40.37 (+29%)	28.58	16.10	28.71
Kitchen	6.64 (+12%)	5.92	22.73 (+ 2%)	32.09 (+31%)	22.22	16.32	24.65
Staircase	4.86 (+ 8%)	4.48	20.00 (+18%)	27.53 (+39%)	16.89	11.20	20.71

Table 3.1: Performance comparison between the renderers generated by Rodent and the reference renderers (not available for the R9 Nano) in Msamples/s (higher is better). On the GPUs, we list the numbers for both the *megakernel* device (1) and the *wavefront* device (2).

3.3.2 Performance

For the performance evaluation, we use a workstation with a Intel i7-6700K CPU, an NVIDIA Titan X (Maxwell) GPU, and an AMD R9 Nano GPU. The numbers for our test scenes are listed in Table 3.1, and the associated reference images are presented in Figure 3.2. Note that OptiX does not run on the R9 Nano, which is why there is no reference for this GPU.

In all tested scenes, the renderers generated by Rodent outperform the reference renderers. On both GPUs, the megakernel-based renderers generated by Rodent are less efficient than the wavefront-based ones, replicating the results of existing literature [LKA13].

Distribution of Execution Time

We present the distribution of time spent in each part of the CPU renderers in Figure 3.3. In this graph, it appears that the time spent tracing rays is nearly identical between the renderer generated by Rodent and the Embree-based reference. This is not surprising, considering that on the CPU, we use the same traversal algorithm as Embree. However, shading is much less expensive, sometimes more than 2× faster than the reference. In Rodent, both specialization and sorting improve vectorization: Specialization decreases the number of branches, and sorting allows similar rays to be processed together [Áfr+16].

We designed a synthetic benchmark to explain which part of the performance increase is due to sorting, and which part is due to specialization. In this benchmark, we shade a 4096 rays using a physically-corrected Phong BSDF. Each ray is associated with an integer $S \in [0..3]$ that defines how the parameters k_d, k_s, and n_s of the BSDF are chosen:

- For $S = 0$, every parameter is a constant.
- For $S = 1$, k_d is a texture, k_s and n_s are constants.
- For $S = 2$, k_s is a texture, k_d and n_s are constants.
- For $S = 3$, k_s and k_d are textures, n_s is a constant.

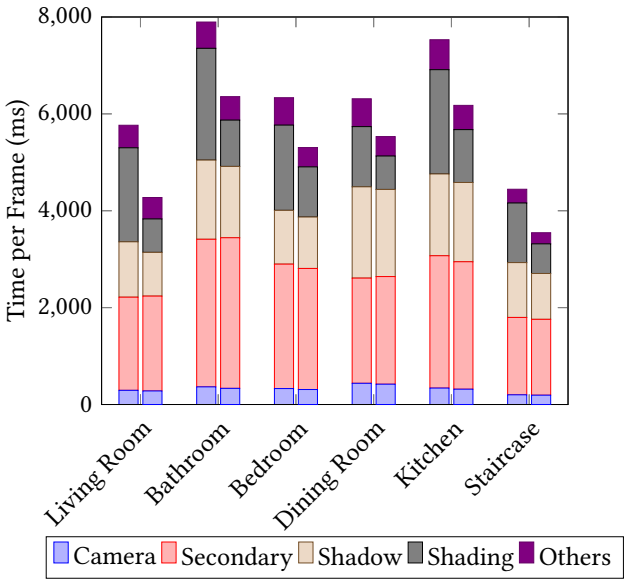


Figure 3.3: Average distribution of the execution time for renderers generated by Rodent (right) and the Embree-based reference (left) over all tested scenes on an i7 6700K. We measure time tracing camera/secondary/shadow rays, and performing shading. Other tasks include generating, sorting or compacting rays.

We implemented this shader using both Rodent and `ispc` , and measured performance before and after specialization. With `ispc` , specialization is a difficult, manual process, and relies on macros to duplicate some functions in order to make sure everything remains uniform whenever possible. This is not practical in any way, and is only used for demonstration purposes, because it breaks every good programming practice by forcing the programmer to break existing abstractions. With Rodent, specialization is done transparently, when the number of shaders is known. When specialization is enabled, we let the compiler know that there are 4 shaders. As a result, the generated program will contain 4 different paths, one for each value of S. Another big advantage over `ispc` is that we do not have to write annotations to specify what is scalar or vectorial: The compiler will infer the best possible annotation for us [MH18].

The results of this benchmark, in Figure 3.4, show that specialization brings a performance improvement of around 25% in both the Rodent-generated version and the manually-specialized `ispc` version. This confirms the intuition that specialization is, even in such a simple example with only 4 trivial shaders, an efficient optimization technique.

Cross-Layer Optimizations

Rodent does not only specialize shaders, but also the interfaces between each rendering component. In Rodent, these interfaces materialize as functions calls, but in traditional programming languages, modules or class hierarchies could play that role. When Rodent specializes these interfaces, it is doing a form of *cross-layer* optimization. Traditional compilers usually only

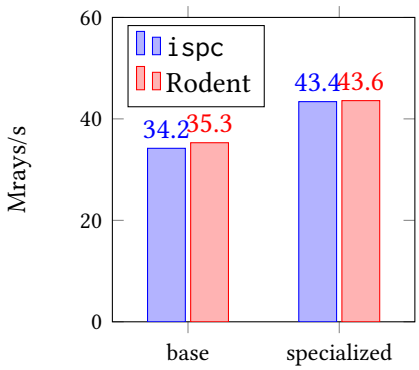


Figure 3.4: Shading performance on an i7 6700K for the simple shader described in Section 3.3.2 implemented using Rodent and `ispc` (higher is better). Specialization is done manually in the `ispc` version.

optimize very conservatively around interfaces, and cross-layer optimizations are often viewed as *dangerous*. The reason for this is that cross-layer optimizations, just like other optimizations, may change the performance of the target program. However, because of their global nature, they often impact performance dramatically, and it is hard to understand why some transformation happenened. For instance, aggressive inlining may increase register pressure, which can impact performance negatively if the code is running on a GPU and is limited by occupancy.

Even though they might be scare some, we argue here that cross-layer optimizations are *essential*. In fact, they are critical for shader specialization too since the information has to flow from the top level of the program, where the user defines the scene, to the innermost part of the renderer, where the shader is executed. If this flow of information is interrupted because some interface cannot be optimized, then specialization cannot occur at the deeper levels, where it is most necessary. Take for example the filtering mode of a texture, which is usually given at the "highest level", in the scene description. If this information is not available in the innermost loop of the renderer, or inside the GPU kernel that performs shading, then the generated code will contain the equivalent of a switch statement to select the appropriate filtering function, which is suboptimal.

We designed another synthetic benchmark to test the claim that cross-layer optimizations are critical for performance. In this benchmark, we run a simple diffuse shader on a batch of rays. This time, we enabled or disabled specialization for two different interfaces: the one between the texture and the shader, and the one between the mesh attributes and the shader. An easy way to do this is to add a boolean parameter to every function that specifies whether or not to perform specialization, as in the following example:

```
fn @(specialize) f(specialize: bool, /* ... */) { /* ... */ }
```

Using this trick, the call `f(true, /*...*/)` will trigger specialization, but `f(false, /*...*/)` will not.

We show the results of this benchmark in Figure 3.5. There, we compute only very basic mesh attributes (shading normal, face normal, and texture coordinates), and use only an image

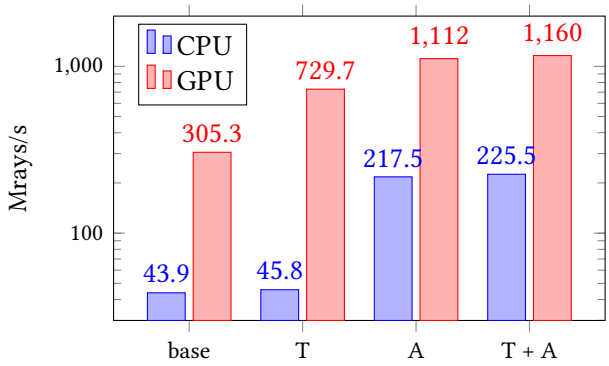


Figure 3.5: Performance when specializing the interfaces between the shader and the texturing function (T), or between the shader and attribute computation (A). We present the results for an i7 6700K CPU, using all cores, and on a Titan X GPU. Note the logarithmic scale on the vertical axis.

texture whose border handling mode and filter are controlled by the shader. These results show that specializing interfaces is beneficial, as expected, but also that some interfaces are less interesting to specialize than others. This indicates that some modules can be kept truly separate with almost no performance hit, while others should be optimized together. Our framework allows us to do exactly that: We can specify exactly where cross-layer optimizations happen using partial evaluation annotations. This, and unlike traditional cross-layer optimizations like LTO, is completely predictable and in the hands of the programmer.

3.3.3 Implementation Effort

Since we must compare the effort required to write the reference renderers with the effort required to write Rodent, we need a language-agnostic measure of implementation effort. Halstead's well-known software complexity measure [Hal77] is based on the number of operators and operands in the program, and serves exactly this purpose. We list Halstead's effort along with numbers of Lines of Code (LoC) for all the renderers in Table 3.2. For Rodent, we separate the core library from the rendering devices to allow a better comparison with the reference renderers. In this comparison, we only included the relevant parts of each renderer: The scene loader, for instance, has been omitted for every renderer.

Even though Rodent is more generic, has more features, and allows scenes that are not supported by the reference renderers—we even provide a wavefront-based GPU implementation not provided by OptiX—the effort required to code Rodent and all its devices is lower than the effort required to write the two references.

It is also worth noting that every renderer written with Rodent uses the same core concepts. For the reference renderers, everything had to be reimplemented from scratch: Even though both CUDA and `ispc` use a C-based language, they are too different—syntax-wise and semantics-wise—to allow sharing parts of the implementation.

Finally, remember that none of our references uses specialization. The only tools available

	Rodent				
	Core	CPU	GPU	Embree	OptiX
LoC	1086	471	600	869	930
Effort	3.798M	4.403M	5.724M	7.856M	9.605M
		13.93M		17.46M	

Table 3.2: Implementation effort for Rodent's core rendering library, the CPU device, the GPU devices (including megakernel *and* wavefront), and for the Embree-based and OptiX-based references. The effort is measured using Halstead's metric.

with `ispc` or the CUDA compiler are macros and template meta-programming. These are completely impractical in our setting: Dynamic (run-time) and static (compile-time) data should be represented in the same language because the scene may be only *partially* known. In fact, the set of scene parameters that are static can vary from scene to scene, and thus multiple implementations of the same renderer would be required, some with those scene parameters represented with macro or template parameters for the case where those parameters are static, and others using regular function parameters for the case where those parameters are dynamic.

3.3.4 Compilation Times

Depending on the use case, the time spent compiling and generating renderers might also be relevant. We measured compilation times for all three devices on our test scenes and present the results in Figure 3.6. Overall, the compiler takes between tens of seconds to 2mins, depending on the number of shaders and the device used. The CPU device is the slowest to compile due to the calls to `vectorize` and the complex transformations that ensue.

Unfortunately, these results mean that our approach is currently reserved to situations where render times dominate. However, the compiler itself is not particularly well-optimized, and runs in a single-thread, so there is hope that compilation can be made faster by distributing partial evaluation work to different cores.

Another approach to overcome this problem is to pre-specialize parts of the code that do not change, or simply avoid specialization of certain parts of the renderer. For instance, we already do not specialize the renderer for the camera position, which allows us to move the camera without having to recompile the renderer.

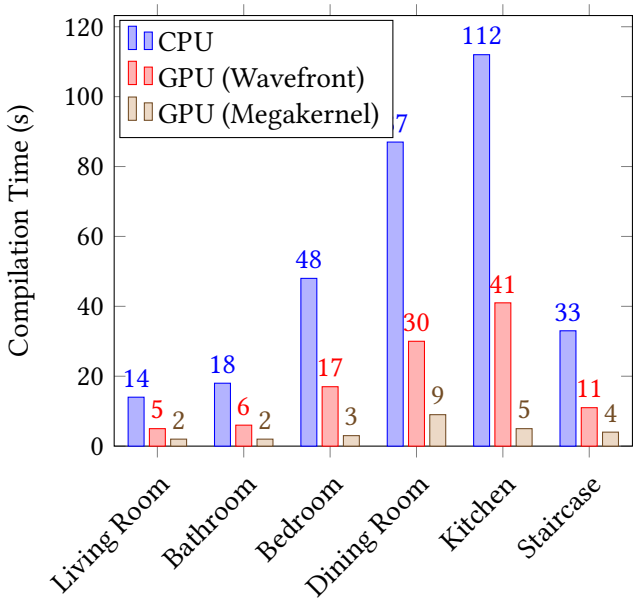


Figure 3.6: Compilation times for various scenes and rendering devices.

Chapter 4

Compiling Generators

This chapter presents a small set of interesting current and future design decisions in the implementation of AnyDSL, and discusses alternatives. Investigating them allows us to draw conclusions and present potential new directions for the project.

4.1 Type Inference

The type system of Impala is fundamentally a simple ML-style (or Hindley-Milner) type system: A type system in which polymorphic types can only be introduced by the keyword *let* [Pie02] (or the equivalent version of that keyword in Impala, `fn`). For this reason, this type of polymorphism is often called *let-polymorphism*. There are two main types of algorithms to infer types in such systems: constraint-based algorithms, like algorithm $\mathcal{W}$ [DM82], and local type inference.

4.1.1 Algorithm $\mathcal{W}$ and Constraint-based Inference Algorithms

Algorithm $\mathcal{W}$, the type inference algorithm by Impala, was initially designed for an ML-style language [DM82]. The version used in Impala is adapted to work with explicit type annotations [Pie02]. The basic idea of this algorithm is to collect and solve type equality constraints when running over the program. Equality constraints are solved by unification [Rob65], an algorithm that given a set of constraints, generates a type substitution that solves those constraints or fails with an error message to indicate that the program does not type-check. This process of collecting constraints and solving them can either be done in two distinct passes, or performed in tandem, alternating collection and solving iteratively [Pie02], which has the benefit of generating better error locations. Other constraint-based algorithms follow the same principle: They generate constraints that are then solved to produce the types of each term.

One major drawback of algorithm $\mathcal{W}$ in particular is that it cannot be extended easily, as adding more features often make complete type reconstruction much harder or even undecidable [Pie02]. Take subtyping, for instance, a feature of a type system in which a binary relation $S <: T$ between types allows to use a value of type S where a value of a type T is expected. In Impala, there a very limited form of subtyping that allows mutable pointer types to be treated as immutable ones. In other words, the rule `&mut T <: &T` is present in the type system. However,

the implementation of algorithm $\mathcal{W}$ in Impala handles subtyping using ad-hoc rules that have to be carefully crafted to ensure termination. They also tend to fail if the program is in an unusual (but in principle, correct) form. The following program, for instance, is rejected:

```
fn foo(y: &i32) -> () {}
fn bar() -> () {
    let mut x = 1;
    let id = |i| i;
    let y = id(&mut x);
    foo(y);
    *y = 3;
}
```

Because the variable `y` in the body of `bar` is not annotated with a type, it is initially assigned an unknown type variable by the algorithm, and the call to `foo` immediately constrains that variable to the type `&i32`, which causes the compiler to report an error when later assigning the value `3` to that pointer, as it is not mutable. Even though this can be solved by a user-placed annotation on `y` to tell the compiler that its type is in fact `&mut i32`, this excerpt shows that ad-hoc rules are not enough to infer the types correctly in all cases, as the order in which they are applied impacts the final result [TBN11]. Properly adding subtyping to algorithm $\mathcal{W}$ requires to change the algorithm to propagate type *inequality* constraints [TBN11]—as opposed to type *equality* constraints, in the original form of the algorithm. This change is unfortunately not as simple as adding ad-hoc rules, and mandates that constraint collection happens separately from constraint solving.

Another issue with algorithm $\mathcal{W}$ is the location of errors. When type inference is performed globally, type conflicts are often reported at the wrong location. For instance, take the following Impala program:

```
let a = |x| x;
// ...
let b = a(5);
// ...
let c = a(1.0);
```

There is a type conflict here, because `a` will be assigned the type `fn (i32) -> i32` when checking the declaration for `b`, but later, when checking `c`, it will be assigned the incompatible type `fn (f64) -> f64`. The original intent was perhaps to make `a` a function that takes floating point values, but the decimal point is missing in the first invocation. Depending on the order in which the type checker goes over the source code, it is possible that the conflict is reported either at the location of `a`, `b`, or `c`. Obviously, only one of those locations is correct. This problem does not have a simple solution, and there are many publications that try to improve the situation by using heuristics, changing the order of unification, or listing all options [Hee05].

All these drawbacks make algorithm $\mathcal{W}$ and other constraint-based algorithms impractical. Thankfully, there is another class of inference algorithms that supports ML-style type systems with various extensions, including subtyping, and that produces good error messages: *local type inference* [PT00].

4.1.2 Local Type Inference

Recall that with algorithm $\mathcal{W}$, constraints are generated when going over the *entire* program, and that they might thus be connecting types that are linked to completely different parts of the AST. In contrast, local type inference algorithms only propagate type information across neighboring AST nodes.

In the original publication on local type inference [PT00], two techniques were introduced: local type argument synthesis and bidirectional type-checking. The first one only infers type arguments in polymorphic function applications, and can be combined with the second. Bidirectional type-checking, the second one, is according to Pierce, a well-known method that is part of the "folklore" on programming language design. It infers both annotations on local variable bindings and anonymous function abstractions, by switching between two modes: inference, and checking modes. In the inference mode, the algorithm synthesizes types, while in the checking mode, it verifies that an AST node can be assigned a given type.

The traditional typing relation is given as $\Sigma \vdash e : T$, which informally means "e types as T in context Σ". Associated with that relation are rules that define the type of an expression. For instance the rule to type a tuple expression $(e_1, e_2, \dots e_n)$ could be written as:

$$\frac{\Sigma \vdash e_1 : T_1 \qquad \Sigma \vdash e_2 : T_2 \qquad \cdots \qquad \Sigma \vdash e_n : T_n}{\Sigma \vdash (e_1, e_2, \dots e_n) : (T_1, T_2, \dots T_n)} \text{ Tuple}$$

This rule states that the tuple $(e_1, e_2, \dots e_n)$ has type $(T_1, T_2, \dots T_n)$ in context Σ if all the hypotheses above the rule are fulfilled, that is, if each $e_{i \in [1,n]}$ types as T_i in context Σ. Bidirectional type inference essentially splits typing judgments such as these in two: $\Sigma \vdash e : T$ becomes $\Sigma \vdash e \Rightarrow T$ and $\Sigma \vdash e \Leftarrow T$. The judgment $\Sigma \vdash e \Rightarrow T$ synthesizes type T from expression e in context Σ. Here, T is an *output* variable. On the contrary, $\Sigma \vdash e \Leftarrow T$ checks if e has type T in context Σ. In this case, T is an *input* variable. For instance, the rule for tuples presented above is split like this:

$$\frac{\Sigma \vdash e_1 \Rightarrow T_1 \qquad \Sigma \vdash e_2 \Rightarrow T_2 \qquad \cdots \qquad \Sigma \vdash e_n \Rightarrow T_n}{\Sigma \vdash (e_1, e_2, \dots e_n) \Rightarrow (T_1, T_2, \dots T_n)} \text{ Tuple-Infer}$$

$$\frac{\Sigma \vdash e_1 \Leftarrow T_1 \qquad \Sigma \vdash e_2 \Leftarrow T_2 \qquad \cdots \qquad \Sigma \vdash e_n \Leftarrow T_n}{\Sigma \vdash (e_1, e_2, \dots e_n) \Leftarrow (T_1, T_2, \dots T_n)} \text{ Tuple-Check}$$

The nice property about these judgments is that they are algorithmic. In other words, they can be translated directly into code. The checking relation, for instance, is usually implemented as a function that takes an AST node and a type, and returns a boolean to indicate when the node type-checks:

```
bool check(AST ast, Type type) { /* ... */ }
```

The synthesis relation, on the other hand, will look like this:

```
std::optional<Type> infer(AST ast) { /* ... */ }
```

This function just takes an AST node, and returns a type for that node or nothing if the type cannot be inferred.

Bidirectional type-checking then just traverses the AST in one of those two modes, switching from one to the other depending on the context. For instance, upon encountering a typing annotation, like $a : i32$, the algorithm switches from synthesis to checking mode. The corresponding typing rule is as follows:

$$\frac{\Sigma \vdash e \Leftarrow T}{\Sigma \vdash e : T \Rightarrow T}$$

The following pseudo-C++ code shows a possible implementation:

```
std::optional<Type> infer(AST ast) {
    switch (ast) {
        // ...
        case Annotation:
            // Expression of the form "<expression> : <annotated_type>"
            if (!check(ast.expression, ast.annotated_type))
                return std::nullopt;
            return std::make_optional(ast.annotated_type);
        // ...
    }
}
```

In the above code, the implementation switches from inference mode to checking mode when seeing a type annotation. This is in practice done by calling the function `check` with the annotated expression, and the annotated type. When `check` fails (after reporting an error to the user), the function returns an `std::nullopt`, indicating that inference failed. When `check` succeeds, the function returns the annotated type, since the call to `check` made sure that the expression has that type.

With bidirectional type checking, the type checker is simpler to implement and maintain, since there is a direct connection between the typing rules and the type-checker implementation. In turn, this simplicity allows to implement more advanced type system extensions such as subtyping or higher-order polymorphism [Pey+05]. Moreover, because knowledge is propagated only locally in the AST, error locations are unambiguous.

Sadly, local type inference algorithms may fail on programs that do not have enough type annotations. Take for example this Impala program:

```
let f = |i| i;
f(1);
```

Here, a constraint-based algorithm would have generated a constraint that binds the type of `i` to the type of `1`, which is **`i32`**, and thus `f` would be assigned the type **`fn`** `(`**`i32`**`) -> `**`i32`**. Bidirectional type-checking would fail in this instance because the type of `f` cannot be inferred just by analyzing the **`let`** statement.

Fortunately for us, we have found such programs to be very rare. In fact, when porting the benchmark test suite of Impala to a local type inference-based front-end[1], not a single time did we have to add an explicit type annotation. This result is not surprising anyway, since the syntax of Impala already forces users to annotate top-level functions. Therefore, most calls can already be inferred using only local knowledge. For Impala, we hence believe that local type inference is a very compelling alternative to constraint-based type inference algorithms, particularly in terms of user-friendliness, robustness, and implementation effort: In fact, we

[1] `https://github.com/AnyDSL/artic`

are planning to replace the old constraint-based type inference completely in favor of the new, local type inference-based front-end.

4.2 Pattern Matching

Pattern matching is an essential feature of any functional language, as it allows to express algorithms that operate on sum and product types (tuples, structures, or enumerations in Impala) very concisely. However, generating code for such patterns is not obvious. Consider the following snippet:

```
enum E { A, B(i32) }
fn foo(x: (bool, E))) -> i32 {
    match x {
        (true,  E::A   ) => 1,       // P1
        (true,  E::B(1)) => 2,       // P2
        (false, E::B(y)) => y + 2,   // P3
        (_,     E::A   ) => 3,       // P4
        (_,     _      ) => 4        // P5
    }
}
```

In this small example, it becomes obvious that generating code for this expression requires to inspect the structure of the value contained in `x`. The naive way to tackle that problem is to test each pattern in turn, until there is a match. This is what the current version of Impala is doing, but this is clearly suboptimal, since some patterns will generate redundant tests: In the example above, Impala will generate one test per pattern, everytime checking that the pattern matches the argument (for instance, with $P1$, this means checking that the first element is `true` and that the second one is `E::A`). As a consequence, if the function `foo` above is called very often with the value `(true, E::B(2))`, then, the code generated by Impala will be very slow: It will have to go through all the patterns from $P1$ to $P4$—that all fail to match the value, before eventually reaching the pattern $P5$—which finally accepts the value. In fact, part of the reason enumerations are not used very often in Rodent is that Impala generates poor quality code when dealing with complex match expressions. Mercifully, the literature already contains methods to approach that problem, all based on either *decision trees* or *backtracking automata*.

4.2.1 Backtracking Automata

Backtracking automata, introduced as early as 1985 [Aug85], are a simple way to compile patterns into code, by eliminating data type constructors (e.g. `true`, `5`, `E::A`, ...) from the patterns present in the `match` expression until the pattern contains no more constructors. Prior to constructor elimination, patterns are grouped by constructor, and the algorithm recurses with each group to compile the rest of the expression. When a pattern fails to match and there are no more patterns to test, the algorithm backtracks and tests the patterns of the parent expression. For instance, the `match` expression above would be translated to:

```
let (x0, x1) = x;
fn case_true() -> i32 {
    match x1 {
        E::A    => 1, // P1
        E::B(y) => match y {
            1 => 2,   // P2
            _ => case_default()
        }
    }
}
fn case_false() -> i32 {
    match x1 {
        E::B(y) => y + 2, // P3
        _       => case_default()
    }
}
fn case_default() -> i32 {
    match x1 {
        E::A => 3, // P4
        _    => 4  // P5
    }
}
match x0 {
    true  => case_true(),
    false => case_false()
}
```

This code starts by inspecting the first element of the pair, `x0`, and depending on the value calls either one of `case_true` or `case_false`. Now, for the **`true`** case, there are two patterns that have that constructor: The patterns commented as $P1$ and $P2$. Therefore, the `case_true` function tests those two patterns, and calls the default case when none match. In the **`false`** case, there is only one pattern that has that constructor, $P3$, and `case_false` therefore tests if it matches and switches to the default case when it does not. Finally, `case_default` corresponds to $P4$, the only default pattern (the one with a first element *not* being a data type constructor). Note that this example is a simplification of what would happen in practice, because cases are usually modelled as basic blocks in the generated program, not functions.

This code generation method, on top of reducing the number of tests compared to the naive method used in Impala, guarantees that the generated code size is linear in the number of arms of the original **`match`** expression. However, because it applies backtracking when patterns do not match, it has the disadvantage that a particular element may be inspected several times: In the example above, `x1` can be tested up to two times, if the input value is of the form `(`**`true`**`, E::B(y))` with `y != 2`, or if it is of the form `(`**`false`**`, E::A)`. Consequently, if performance is the primary goal of the code generator, decision trees should be used instead.

4.2.2 Decision Trees

Decision trees were introduced in the context of pattern matching in 1985 [BM85], at the same time as backtracking automata. The principle behind this compilation technique is to inspect an element of a pattern only once, and *decide* on an action based on the result. A pattern matching compiler using this technique selects a column of the pattern (e.g. the element of a pair, or the member of a structure) based on some heuristic, and then groups patterns based on their data type constructor in that column: This is so far similar to backtracking automata,

in that we have now a set of patterns for each data type constructor. However, there is no backtracking involved: For each data type constructor, we then add to the associated list of patterns the patterns that have a variable instead of a constructor on that column, so that every case is covered without having to test the enclosing **match** expression. Finally, if the data type constructors that are considered form a *signature* (they cover all possible values for the element in the chosen column), then no default case is generated. If they do not, then the default case is generated by gathering patterns that have a variable in the chosen column. For instance, in the **match** expression above, the compiler may choose the first column—the first element of the pair—to make a decision: In that case, the patterns are grouped by their data type constructor in that column, **true** or **false**. For the **true** case, we get a list of patterns containing $P1$ and $P2$, to which we add $P4$ and $P5$ since they have a variable (not a data type constructor) in the first column. Likewise, we get the list made of $P3$, $P4$, and $P5$ in the **false** case. For those two cases, we can now eliminate the chosen column of each pattern in the associated list and recurse on the resulting list of patterns. Since **true** and **false** form a signature for **bool**, no default case is generated. To sum things up, the **match** expression above would be compiled to:

```
let (x0, x1) = x;
fn P1() -> i32 { 1 }
fn P2() -> i32 { 2 }
fn P3(y: i32) -> i32 { y + 2 }
fn P4() -> i32 { 3 }
fn P5() -> i32 { 4 }
fn case_true() -> i32 {
    match x1 {
        E::A => P1(),
        E::B(y) => match y {
            1 => P2(),
            _ => P5()
        }
    }
}
fn case_false() -> i32 {
    match x1 {
        E::B(y) => P3(y),
        E::A    => P4()
    }
}
match x0 {
    true  => case_true(),
    false => case_false()
}
```

One obvious difference with the generated code from backtracking automata is that each column—`x0` or `x1`—is inspected only once. Additionally, because patterns that contain variables may be duplicated several times (once per data type constructor), the value associated with each pattern must be wrapped into one function to avoid creating too much code. For instance if we were compiling the following expression by choosing the first column as a decision variable:

```
match x {
    (true,  E::B(_)) => 1,
    (false, E::A) => 2,
    (_, _) => {
        /* Lots of code */
    }
}
```

The generated code would contain the last pattern in both the `true` and `false` cases. Thus, wrapping each pattern in its own function makes sure that the code contained in the last pattern is not replicated everytime it is used, but that instead, a call to its corresponding function is made. Note that the remark made in the previous section still applies: This presentation is a simplification as in practice those functions would in fact be basic blocks of the function being generated.

Another advantage of using decision trees is that it is easy to see if a `match` expression is exhaustive, or if some cases are redundant: Since we have the generated decision tree at hand, we can easily explore it to see if cases are not covered. With backtracking automata, it is not so obvious to check the exhaustivity of a `match` expression, because the generated code might backtrack in the case where a data type constructor is not covered in the current pattern.

Finally, generating the best possible decision tree, a problem also known as the *dispatching problem*, is NP-complete [BM85]. However, the choice of column from which to create the next level of the decision tree is open to many heuristics [Mar08]. This allows to optimize the shape of the final tree, since matching on some columns might create shallower trees.

To conclude, decision trees seem to be a superior alternative to backtracking automata, both in terms of performance of the generated code and implementation effort, since exhaustivity checks can be integrated into the pattern matching compiler. Their only disadvantage is in the size of the generated code, which is mitigated by the fact that we wrap each pattern in its own function. Thus, we have implemented this compilation technique, along with heuristics favoring the first row of a pattern, small amounts of variable patterns, and a small branching factor (known as f, d, and b, respectively, in previous work [Mar08]), in another front-end for AnyDSL [2]. Hopefully, this should now allow users to rely more often on sum types and pattern matching when writing high-performance code, something that we were reluctant to do in the past, given the poor quality of the code generated by Impala.

4.3 Memory Management

Managing memory is always a major problem in programming language design. Some languages allow the user to manipulate pointers to memory directly, disregarding any safety concern, like C. Others try to make allocating memory safe by using either garbage collection or a complex type system, like Haskell or Rust, respectively. For AnyDSL, the question is even broader than traditional programming languages. We are dealing with memory that can reside on different devices, possibly even in different address spaces (shared versus private memory in CUDA, for example).

[2] https://github.com/AnyDSL/artic

In general, memory management techniques fall into two categories: automatic memory management, and manual memory management. The latter is simply referring to the idea of letting the programmer decide when to allocate and deallocate data structures, and thus can be at the same time tedious and error-prone: This is the approach chosen in the current version of Impala. The former is describing a wide array of techniques, which automatically decide when memory should be allocated and deallocated, either at compile-time, using static program analysis, or at runtime, using garbage collection.

4.3.1 Manual Memory Management

In Impala, there is currently no automated system to allocate or deallocate memory. In particular, local variables are placed on the stack, but their lifetime might be extended by partial evaluation. Take the following piece of Impala code, for example:

```
fn @f() -> &mut i32 { let mut a = 0; &mut a }
*f() = 5;
```

In this example, `f` returns the address of `a`, which is a variable private to `f`. If `f` is not inlined, the value returned by `f` is a local stack variable that is no longer alive after the call, and hence this code will not work. The important note to make here is that after partial evaluation, the optimized version of this program does not exhibit this issue anymore. The call to `f` will be specialized, and `a` will be pulled out of its scope, into the scope of the callee. In general, partial evaluation can (and eventually will) change the lifetime of variables. Clearly, a proper language design would either declare that local variables have a local lifetime and reject this program, or would handle such cases with a system that ensures that the behavior is the same with and without partial evaluation.

Moreover, in Impala, manual memory management forces the user to allocate and deallocate host or device memory with specific functions for each device: `alloc_cpu`, `alloc_cuda`, or `alloc_opencl`, for instance. These functions return a pointer or some opaque object that represents the allocated memory that is only valid on the device for which it is allocated. For instance, the following code would crash at runtime:

```
let dev   = /* ... */;
let grid  = /* ... */;
let block = /* ... */;
let ptr = alloc_cpu(/* ... */);
with cuda(dev, grid, block) {
    *ptr(id) = 42;
}
```

Listing 4.1: Incorrect use of a host pointer on a device

Indeed, the pointer `ptr` obtained from `alloc_cpu` is only valid on the CPU. Trying to use it on a GPU using the **`cuda`** intrinsic will cause the GPU driver to report an access violation error. As said previously, two options exist to handle this case: Rejecting the program, or accepting it, but making sure the pointer is transformed to a device pointer before launching the kernel. Rejecting problematic programs such as the one above usually requires to add some level of type-checking. Accepting them, on the other hand, requires to dynamically track ownership of memory: In the previous example, the pointer `ptr` was allocated right before the call, but

in general, this pointer might come from a function parameter, or could be stored in a data structure, which makes compile-time tracking impossible in general.

4.3.2 Automatic Memory Management

Automatic memory management systems can run at compile-time, at runtime, or a combination of both. Compile-time solutions for memory management include effect and region-based type-systems [GL86; TT94], or compile-time analyses [KM05]. Runtime memory management, on the other hand, is based on various forms of *garbage collection*, deallocating unreachable memory—*garbage*—when it is no longer used.

Region-based Memory Management

One way to control how memory is allocated and deallocated is to use a stack of *regions*, which are blocks of memory that contain allocated objects. Then, all the allocations inside one region can be reclaimed by reclaiming the region itself and its children—the regions that appear above it in the region stack. This is in essence extending the stack-based allocation technique [Dij60] to arbitrary data and lifetimes that are potentially larger than the body of a function.

In order to integrate regions to a programming language, one can either add them as syntactic constructs, or rely on static analysis to perform *region inference* [TT94]. The issue with the first approach is that it requires to change the way programs are written by explicitly annotating the lifetimes of variables, which is tedious. The second method does not require user intervention, but may generate lifetimes that are not optimal in some cases, potentially covering the entire duration of the program. For such problematic allocations, it is possible to fall back to garbage collection [HET02].

Linear Type Systems

Instead of using regions, it is also possible to use a *linear type system* [Laf88], a form of substructural type system in which the rules of weakening and contraction are disallowed. By removing those two rules, linear type systems ensure that variables are used at least once (no weakening), and at most once (no contraction). Thus, in those systems, variables can only be used once, and this means that their lifetimes can be determined at compile-time, since the points of the program where they are introduced and eliminated are known. Compared to languages with region inference, languages with linear type systems require more user intervention: Programmers have to be careful in the way they design their data structures and algorithms, otherwise the type-checker will complain that variables are not used, or that they are used too many times.

In the context of Impala, using a linear type system or implementing region-based memory management would not completely solve our problems: While those systems would handle allocation and deallocation of objects, they would not solve the problem of rejecting or transforming programs that access host memory on a device, like the one in Listing 4.1.

Effect Type Systems

In order to disallow or detect programs such as the one in Listing 4.1, the type system would have to be enhanced with *effects* representing memory allocation, deallocation, and access. Such a type system is called an effect type system [GL86], and has the following type judgment:

$$\Sigma \vdash e :^{\phi} T$$

Which, intuitively means that, under the assumptions in Σ, the expression e evaluates to a value of type T, producing effects ϕ in the process. With a typing judgment like this, it is now possible to encode rules that forbid memory accesses to host memory from device code, and vice-versa, since the well-typedness of an expression can depend on the effects performed in that expression.

Applying these type-based techniques in Impala conflicts with the fact that partial evaluation might drastically change the lifetime of objects, as noted in the previous section. Since partial evaluation is run only *after* type-checking, any analysis done at type-checking time will miss those changes. The other issue is that these systems either require user intervention or need a garbage collector to handle corner cases. If a garbage collector has to be implemented anyway, a combination of static analyses with a garbage collector seem to be a superior alternative for Impala. In particular, escape analysis [KM05], an analysis that tries to determine what can be allocated on the stack and what requires dynamic memory management, provides an addition that would be lightweight to implement, only requiring an additional pass in the compiler.

Garbage Collection

Garbage collection is an umbrella term that can refer to many algorithms, including, but not limited to *reference counting* [Col60], *mark and sweep* [McC60], *copying collection* [FY69]. In general, the idea is to dynamically trace objects created in the *mutator*—the program whose memory the garbage collector is managing—and to deallocate them when they are no longer reachable.

Reference counting is a particular case of garbage collection for non-cyclic data structures, in which a counter is associated with an object. That counter is initialized with 1, and is incremented when the object enters a scope, and decremented once the object goes out of a scope. Thus, when the counter becomes 0, the object is no longer reachable at any point in the program and can then be safely deallocated.

Algorithms like mark and sweep and copying collection are more general and work by tracing live objects at a given point in the execution of a program, called the *roots*. With mark and sweep, the set of roots is inspected in a *marking phase*, and objects that are not reachable anymore from that set are marked for deallocation. In a second *sweeping* phase, objects that were marked for deallocation are deallocated. This algorithm is simple and fast, but suffers from the fact that it can introduce fragmentation.

In the copying garbage collection algorithm, there are two areas of memory, called *semi-spaces*. One contains the current, valid objects, and the second contains old, no longer valid ones. At each collection, the two spaces are exchanged and the objects of the program are moved from one space to the other, by copying them. This algorithm is also relatively simple

and does not exhibit fragmentation, but has the drawback that it has to copy a potentially large amount of data at each collection.

On top of these basic algorithms, there are variants that allow parallel [ETY97] (using multiple threads in the garbage collector), concurrent [AEL88] (running the garbage collector in parallel with the mutator), or incremental collections [Pir98] (interleaving mutator execution and garbage collection). These additions can improve either the latency or throughput of the mutator, or, in the case of parallel collections, both.

The traditional complaints made by opponents of garbage collection concern its speed. Garbage collection is seen as slow, ineffective and sometimes even wasteful—in the case of techniques such as copying garbage collection. There are several reasons for this, but they usually all boil down to one thing: implementation choices. In AnyDSL, the philosophy is to give the programmer full control, and that should include any runtime garbage collection system. With that in mind, it makes sense to expose a garbage collection API that allows the programmer to give performance hints to the garbage collector or trigger collections. Beyond that, the major problem with garbage collection systems is not garbage collection *per se*. Languages that feature a garbage collector typically allocate a lot of small objects and recursive data structures, making collection pauses very long, and this is simply not the case for the typical application written in Impala. The traversal library in Chapter 2, for instance, only allocates large arrays containing non-recursive data types such as rays, BVH nodes (with *indices* instead of pointers) and hit points.

To conclude, it seems that garbage collection is a valid option for AnyDSL programs: It allows to track memory along with the devices on which said memory is valid, and thus would allow the runtime to automatically transfer memory between devices when necessary. There are many variants that could be implemented in the runtime, but it seems that non-copying collectors are better for Impala programs, since copying and compacting data that resides on a device can be slow. If a garbage collector that may exhibit fragmentation is used (e.g. a mark and sweep collector), AnyDSL programmers should be advised to use arrays instead of small objects containing pointers. As noted earlier, Rodent, the renderer presented in this thesis, already uses large arrays whenever possible, and hence we believe this is a reasonable restriction. In any case, avoiding small objects connected by pointers is a good programming practice for high-performance programs, because dereferencing a pointer is an expensive operation, even more so when the data pointed to is not in cache.

Conclusion

In this thesis, we have presented a way to generate entire renderers. Based on partial evaluation, our technique allows to generate from traversal kernels to complete renderers. Previous attempts at using meta-programming for rendering were either limited in their scope [And96] or targeting rendering subsystems alone [GKR95].

While generating renderers might seem an outlandish idea, there are in fact very strong incentives behind our work. First, the ever increasing complexity of computing hardware, from CPUs and GPUs to FPGAs, has lead us into an ever increasing set of incompatible technologies that prevent software developers from writing high-level, high-performance and portable code. Second, there is potential to optimize renderers beyond the shading system. Renderers are very logically configurable tools that act like interpreters of scene data. In this context, it is then natural to use partial evaluation to get the result of *compiling* a renderer for a scene, instead of relying on slow *interpretation*.

Consequently, we believe this work addresses a void in the high-performance rendering community. In other domains, like image processing, custom solutions to the two problems mentioned above have been developed, under the form of DSLs, such as Halide [Rag+13]. This allows the compiler to take advantage of domain-specific knowledge to optimize the program for a particular algorithm on some specific hardware platform. Our work essentially addresses the same issues for rendering, but it is novel in the sense that it presents an alternative to building custom compilers for rendering. All of the knowledge specific to rendering is encoded as *code*: partial evaluation annotations and custom accelerator *mappings*. In the compiler used in this thesis, there is no heuristic or automatic algorithm to select the best code path for a given target hardware: The hardware expert knows the best strategy, and the compiler offer tools to implement that strategy effortlessly, such as guided vectorization or triggered code-generation.

A key benefit of this design is that we do not have to write a renderer generator, as required with other meta-programming techniques. All of our code is standard, text-book *rendering* code, with a clean API. This is a consequence of the first Futamura [Fut99] projection: In essence, we obtain the *compiled* version of a program, by partially evaluating an interpreter—in our case, the renderer—for its input—here, the scene data. Our results demonstrate that we can at the same time solve the performance-portability issue, and specialize renderers to extract even more performance.

We provide all the code presented in this thesis as Open-Source at `https://github.com/AnyDSL/rodent`. The compiler framework used in this thesis is also distributed as Open-Source, and is available as `https://github.com/AnyDSL/anydsl`.

Thanks

I would like to thank my entire family and my friends for their strong support all these years, and all the people in Saarland University. I would also like to thank my all my co-workers in the Computer Graphics Lab and Compiler Design Lab, in particular Richard Membarth, Roland Leißa, Pascal Grittmann, and Javor Kalojanov. Finally, I would like to thank both Philipp Slusallek and Sebastian Hack for supporting this work and sharing their knowledge with me.

Bibliography

[18a] *NVIDIA Material Definition Language Specification*. version 1.4.4. NVIDIA. 2018.

[18b] *Open Shading Language Specification*. version 1.10. Sony Pictures Imageworks. 2018.

[20] *NVIDIA CUDA C Programming Guide*. version 10.2. NVIDIA. 2020.

[88] *RenderMan Interface Specification*. version 3.0. Pixar. 1988.

[AEL88] A. W. Appel, J. R. Ellis, and K. Li. "Real-Time Concurrent Collection on Stock Multiprocessors". In: *SIGPLAN Not.* 23.7 (June 1988), pp. 11–20. ISSN: 0362-1340. DOI: 10.1145/960116.53992.

[Áfr+16] Attila T. Áfra et al. "Local Shading Coherence Extraction for SIMD-efficient Path Tracing on CPUs". In: *Proceedings of High Performance Graphics*. HPG '16. Dublin, Ireland: Eurographics Association, 2016, pp. 119–128. ISBN: 978-3-03868-008-6. DOI: 10.2312/hpg.20161198.

[AK90] James Arvo and David Kirk. "Particle Transport and Image Synthesis". In: *SIGGRAPH Comput. Graph.* 24.4 (1990), pp. 63–66. ISSN: 0097-8930. DOI: 10.1145/97880.97886.

[AL09] Timo Aila and Samuli Laine. "Understanding the Efficiency of Ray Traversal on GPUs". In: *Proceedings of High-Performance Graphics 2009*. 2009.

[ALK12] Timo Aila, Samuli Laine, and Tero Karras. *Understanding the Efficiency of Ray Traversal on GPUs - Kepler and Fermi Addendum*. Tech. rep. 2012.

[AMD15] AMD. *OpenCL Optimization Guide*. Tech. rep. 2015.

[And96] Peter Holst Andersen. "Partial evaluation applied to ray tracing". In: *Software Engineering im Scientific Computing*. Springer, 1996, pp. 78–85.

[ÁS14] Attila T. Áfra and László Szirmay-Kalos. "Stackless Multi-BVH Traversal for CPU, MIC and GPU Ray Tracing". In: *Computer Graphics Forum* 33.1 (2014), pp. 129–140. DOI: 10.1111/cgf.12259.

[ASU86] Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. *Compilers, principles, techniques*. Vol. 7. 8. Addison Wesley, 1986, p. 9.

[Aug85] Lennart Augustsson. "Compiling Pattern Matching". In: *Proceedings of a Conference on Functional Programming Languages and Computer Architecture*. Nancy, France: Springer-Verlag, 1985, pp. 368–381. ISBN: 3-540-15975-4.

[BA13] Rasmus Barringer and Tomas Akenine-Möller. "Dynamic Stackless Binary Tree Traversal". In: *Journal of Computer Graphics Techniques (JCGT)* 2.1 (2013), pp. 38–49. ISSN: 2331-7418. URL: http://jcgt.org/published/0002/01/03/.

[BA14] Rasmus Barringer and Tomas Akenine-Möller. "Dynamic Ray Stream Traversal". In: *ACM Trans. Graph.* 33.4 (July 2014), 151:1–151:9. ISSN: 0730-0301. DOI: 10.1145/2601097.2601222.

[Ben+12] Carsten Benthin et al. "Combining Single and Packet-Ray Tracing for Arbitrary Ray Distributions on the Intel MIC Architecture". In: *IEEE Transactions on Visualization and Computer Graphics* 18.9 (2012), pp. 1438–1448. ISSN: 1077-2626. DOI: 10.1109/TVCG.2011.277.

[Ben75] Jon Louis Bentley. "Multidimensional Binary Search Trees Used for Associative Searching". In: *Commun. ACM* 18.9 (Sept. 1975), pp. 509–517. ISSN: 0001-0782. DOI: 10.1145/361002.361007.

[BM85] Marianne Baudinet and David Macqueen. *Tree Pattern Matching for ML (Extended Abstract)*. Tech. rep. 1985.

[Bro+11] Kevin J. Brown et al. "A Heterogeneous Parallel Framework for Domain-Specific Languages". In: *12th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 2011, pp. 89–100. DOI: 10.1109/PACT.2011.15.

[Car18] Michael Royce Carroll. "Improving Performance of OpenCL Workloads on Intel Processors with Profiling Tools". In: *Proceedings of the International Workshop on OpenCL*. IWOCL '18. Oxford, United Kingdom: ACM, 2018, 6:1–6:1. ISBN: 978-1-4503-6439-3. DOI: 10.1145/3204919.3204925.

[CDZ10] Sylvain Collange, David Defour, and Yao Zhang. "Dynamic Detection of Uniform and Affine Vectors in GPGPU Computations". In: *Proceedings of the 2009 International Conference on Parallel Processing*. Euro-Par'09. Delft, The Netherlands: Springer-Verlag, 2010, pp. 46–55. ISBN: 978-3-642-14121-8. URL: http://dl.acm.org/citation.cfm?id=1884795.1884804.

[Col60] George E. Collins. "A Method for Overlapping and Erasure of Lists". In: *Commun. ACM* 3.12 (Dec. 1960), pp. 655–657. ISSN: 0001-0782. DOI: 10.1145/367487.367501.

[DeV+11] Zach DeVito et al. "Liszt: a domain specific language for building portable mesh-based PDE solvers". In: *Conference on High Performance Computing Networking, Storage and Analysis (SC)*. 2011, 9:1–9:12. DOI: 10.1145/2063384.2063396.

[DH02] J. Döllner and K. Hinrichs. "A Generic Rendering System". In: *IEEE Transactions on Visualization and Computer Graphics* 8.2 (2002), pp. 99–118. ISSN: 1077-2626. DOI: 10.1109/2945.998664.

[DHK08] Holger Dammertz, Johannes Hanika, and Alexander Keller. "Shallow Bounding Volume Hierarchies for Fast SIMD Ray Tracing of Incoherent Rays". In: *Proceedings of the Nineteenth Eurographics Conference on Rendering*. Sarajevo, Bosnia and Herzegovina: Eurographics Association, 2008, pp. 1225–1233. DOI: 10.1111/j.1467-8659.2008.01261.x.

[Dij60] E. W. Dijkstra. "Recursive Programming". In: *Numerische Mathematik* 2.1 (Dec. 1960), pp. 312–318. ISSN: 0029-599X. DOI: 10.1007/BF01386232.

[DM82] Luis Damas and Robin Milner. "Principal Type-Schemes for Functional Programs". In: *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '82. Albuquerque, New Mexico: Association for Computing Machinery, 1982, pp. 207–212. ISBN: 0897910656. DOI: 10.1145/582153.582176. URL: https://doi.org/10.1145/582153.582176.

[Dut03] Philip Dutré. "Global illumination compendium". In: (2003).

[EG08] Manfred Ernst and Gunther Greiner. "Multi bounding volume hierarchies". In: *Interactive Ray Tracing, 2008. RT 2008. IEEE Symposium on*. IEEE. 2008, pp. 35–40.

[Eis+13] Christian Eisenacher et al. "Sorted Deferred Shading for Production Path Tracing". In: *Computer Graphics Forum* 32.4 (2013), pp. 125–132. DOI: 10.1111/cgf.12158.

[ETY97] Toshio Endo, Kenjiro Taura, and Akinori Yonezawa. "A Scalable Mark-Sweep Garbage Collector on Large-Scale Shared-Memory Machines". In: *Proceedings of the 1997 ACM/IEEE Conference on Supercomputing*. SC '97. San Jose, CA: Association for Computing Machinery, 1997, pp. 1–14. ISBN: 0897919858. DOI: 10.1145/509593.509641.

[Fre03] Ivar Fredholm. "Sur une classe d'équations fonctionelles". In: *Acta Mathematica* 27 (1903), pp. 365–390. DOI: 10.1007/BF02421317.

[FTI86] A. Fujimoto, T. Tanaka, and K. Iwata. "ARTS: Accelerated Ray-Tracing System". In: *IEEE Computer Graphics and Applications* 6.4 (1986), pp. 16–26. ISSN: 0272-1716. DOI: 10.1109/MCG.1986.276715.

[Fut83] Yoshihiko Futamura. "Partial computation of programs". In: *RIMS Symposia on Software Science and Engineering*. Ed. by Eiichi Goto et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 1983, pp. 1–35. ISBN: 978-3-540-39442-6.

[Fut99] Yoshihiko Futamura. "Partial Evaluation of Computation Process–An Approach to a Compiler-Compiler". In: *Higher Order Symbol. Comput.* 12.4 (1999), pp. 381–391. ISSN: 1388-3690. DOI: 10.1023/A:1010095604496.

[FY69] Robert R. Fenichel and Jerome C. Yochelson. "A LISP Garbage-Collector for Virtual-Memory Computer Systems". In: *Commun. ACM* 12.11 (Nov. 1969), pp. 611–612. ISSN: 0001-0782. DOI: 10.1145/363269.363280.

[Geo+12] Iliyan Georgiev et al. "Light transport simulation with vertex connection and merging". In: *ACM Trans. Graph.* 31.6 (2012), 192:1–192:10. ISSN: 0730-0301. DOI: 10.1145/2366145.2366211.

[Geo13] Iliyan Georgiev. *Implementing Vertex Connection and Merging*. Tech. rep. 2013.

[GKR95] Brian Guenter, Todd B. Knoblock, and Erik Ruf. "Specializing Shaders". In: *Proceedings of the 22Nd Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '95. New York, NY, USA: ACM, 1995, pp. 343–350. ISBN: 0-89791-701-4. DOI: 10.1145/218380.218470.

[GL86] David K. Gifford and John M. Lucassen. "Integrating Functional and Imperative Programming". In: *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*. LFP '86. Cambridge, Massachusetts, USA: Association for Computing Machinery, 1986, pp. 28–38. ISBN: 0897912004. DOI: 10.1145/319838.319848.

[Gla89] Andrew S. Glassner, ed. *An Introduction to Ray Tracing*. London, UK, UK: Academic Press Ltd., 1989. ISBN: 0-12-286160-4.

[Gos+14] James Gosling et al. *The Java Language Specification, Java SE 8 Edition*. 1st. Addison-Wesley Professional, 2014. ISBN: 013390069X.

[GS08] Iliyan Georgiev and Philipp Slusallek. "RTfact: Generic concepts for flexible and high performance ray tracing". In: *2008 IEEE Symposium on Interactive Ray Tracing*. 2008, pp. 115–122. DOI: 10.1109/RT.2008.4634631.

[Gut14] Michael Guthe. "Latency Considerations of Depth-first GPU Ray Tracing". In: *Eurographics 2014 - Short Papers*. Ed. by Eric Galin and Michael Wand. The Eurographics Association, 2014. DOI: 10.2312/egsh.20141013.

[Hal77] Maurice H. Halstead. *Elements of Software Science (Operating and Programming Systems Series)*. New York, NY, USA: Elsevier Science Inc., 1977. ISBN: 0444002057.

[Hap+13] Michal Hapala et al. "Efficient Stack-less BVH Traversal for Ray Tracing". In: *Proceedings of the 27th Spring Conference on Computer Graphics*. SCCG '11. Viničné, Slovak Republic: ACM, 2013, pp. 7–12. ISBN: 978-1-4503-1978-2. DOI: 10.1145/2461217.2461219.

[Hee05] Bastiaan J Heeren. "Top quality type error messages". PhD thesis. Utrecht University, 2005.

[HET02] Niels Hallenberg, Martin Elsman, and Mads Tofte. "Combining Region Inference and Garbage Collection". In: *SIGPLAN Not.* 37.5 (May 2002), pp. 141–152. ISSN: 0362-1340. DOI: 10.1145/543552.512547.

[Jon96] Neil D. Jones. "An Introduction to Partial Evaluation". In: *ACM Comput. Surv.* 28.3 (1996), pp. 480–503. ISSN: 0360-0300. DOI: 10.1145/243439.243447.

[Kah50] Herman Kahn. "Random sampling (Monte Carlo) techniques in neutron attenuation problems–I." In: *Nucleonics* 6.5 (1950).

[Kaj86] James T. Kajiya. "The Rendering Equation". In: *SIGGRAPH Comput. Graph.* 20.4 (1986), pp. 143–150. ISSN: 0097-8930. DOI: 10.1145/15886.15902.

[KBS11] Javor Kalojanov, Markus Billeter, and Philipp Slusallek. "Two-Level Grids for Ray Tracing on GPUs". In: *Computer Graphics Forum* 30.2 (2011), pp. 307–314. DOI: 10.1111/j.1467-8659.2011.01862.x.

[KK86] Timothy L. Kay and James T. Kajiya. "Ray Tracing Complex Scenes". In: *SIGGRAPH Comput. Graph.* 20.4 (1986), pp. 269–278. ISSN: 0097-8930. DOI: 10.1145/15886.15916.

[KM05] Thomas Kotzmann and Hanspeter Mössenböck. "Escape Analysis in the Context of Dynamic Compilation and Deoptimization". In: *Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments*. VEE '05. Chicago, IL, USA: ACM, 2005, pp. 111–120. ISBN: 1-59593-047-7. DOI: 10.1145/1064979.1064996.

[Laf88] Y. Lafont. "The linear abstract machine". In: *Theoretical Computer Science* 59.1 (1988), pp. 157–180. ISSN: 0304-3975. DOI: 10.1016/0304-3975(88)90100-4.

[Lai10] Samuli Laine. "Restart Trail for Stackless BVH Traversal". In: *Proceedings of High Performance Graphics*. HPG '10. Saarbrucken, Germany: Eurographics Association, 2010, pp. 107–111.

[Lee+17] Mark Lee et al. "Vectorized Production Path Tracing". In: *Proceedings of High Performance Graphics*. HPG '17. Los Angeles, California: ACM, 2017, 10:1–10:11. ISBN: 978-1-4503-5101-0. DOI: 10.1145/3105762.3105768.

[Lei+18] Roland Leißa et al. "AnyDSL: A Partial Evaluation Framework for Programming High-performance Libraries". In: *Proceedings of ACM Program. Lang.* 2.OOPSLA (2018), 119:1–119:30. ISSN: 2475-1421. DOI: 10.1145/3276489.

[LHH14] Roland Leißa, Immanuel Haffner, and Sebastian Hack. "Sierra: a SIMD extension for C++". In: *Proceedings of the 2014 Workshop on Programming models for SIMD/Vector processing*. ACM. 2014, pp. 17–24.

[LKA13] Samuli Laine, Tero Karras, and Timo Aila. "Megakernels Considered Harmful: Wavefront Path Tracing on GPUs". In: *Proceedings of the 5th High-Performance Graphics Conference (HPG)*. ACM, July 2013, pp. 137–143. DOI: 10.1145/2492045.2492060.

[Mar08] Luc Maranget. "Compiling Pattern Matching to Good Decision Trees". In: *Proceedings of the 2008 ACM SIGPLAN Workshop on ML*. ML '08. Victoria, BC, Canada: Association for Computing Machinery, 2008, pp. 35–46. ISBN: 9781605580623. DOI: 10.1145/1411304.1411311. URL: https://doi.org/10.1145/1411304.1411311.

[Mar10] Simon Marlow. *Haskell 2010 Language Report*. 2010.

[McC60] John McCarthy. "Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I". In: *Commun. ACM* 3.4 (Apr. 1960), pp. 184–195. ISSN: 0001-0782. DOI: 10.1145/367177.367199.

[Mea82] Donald Meagher. "Geometric modeling using octree encoding". In: *Computer Graphics and Image Processing* 19.2 (1982), pp. 129–147. ISSN: 0146-664X. DOI: 10.1016/0146-664X(82)90104-6.

[Mem+16] Richard Membarth et al. "HIPA^{cc}: A Domain-Specific Language and Compiler for Image Processing". In: *IEEE Trans. Parallel Distrib. Syst.* 27.1 (2016), pp. 210–224. DOI: 10.1109/TPDS.2015.2394802.

[MGN17] Thomas Müller, Markus Gross, and Jan Novák. "Practical Path Guiding for Efficient Light-Transport Simulation". In: *Computer Graphics Forum (Proceedings of EGSR)* 36.4 (2017), pp. 91–100. DOI: 10.1111/cgf.13227.

[MH18] Simon Moll and Sebastian Hack. "Partial Control-flow Linearization". In: *SIGPLAN Not.* 53.4 (2018), pp. 543–556. ISSN: 0362-1340. DOI: 10.1145/3296979.3192413.

[Mil94] Gavin Miller. "Efficient Algorithms for Local and Global Accessibility Shading". In: *Proceedings of the 21st Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '94. New York, NY, USA: Association for Computing Machinery, 1994, pp. 319–326. ISBN: 0897916670. DOI: 10.1145/192161.192244.

[MQP02] Michael D. McCool, Zheng Qin, and Tiberiu S. Popa. "Shader Metaprogramming". In: *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*. HWWS '02. Saarbrucken, Germany: Eurographics Association, 2002, pp. 57–68. ISBN: 1-58113-580-7.

[NT98] Viet N Ngo and Wei-Tek Tsai. *Outer loop vectorization*. US Patent 5,802,375. 1998.

[NZ08] Dorit Nuzman and Ayal Zaks. "Outer-loop Vectorization: Revisited for Short SIMD Architectures". In: *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*. PACT '08. Toronto, Ontario, Canada: ACM, 2008, pp. 2–11. ISBN: 978-1-60558-282-5. DOI: 10.1145/1454115.1454119.

[Ofe+13] Georg Ofenbeck et al. "Spiral in Scala: Towards the Systematic Construction of Generators for Performance Libraries". In: *International Conference on Generative Programming: Concepts & Experiences (GPCE)*. 2013, pp. 125–134. DOI: 10.1145/2517208.2517228.

[ORM08] Ryan Overbeck, Ravi Ramamoorthi, and William R. Mark. "Large Ray Packets for Real-time Whitted Ray Tracing". In: *2008 IEEE Symposium on Interactive Ray Tracing*. 2008, pp. 41–48. DOI: 10.1109/RT.2008.4634619.

[Par+10] Steven G. Parker et al. "OptiX: A General Purpose Ray Tracing Engine". In: *ACM Transactions on Graphics* (2010). DOI: 10.1145/1778765.1778803.

[Pér+17] Arsène Pérard-Gayot et al. "RaTrace: Simple and Efficient Abstractions for BVH Ray Traversal Algorithms". In: *Proceedings of the 16th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*. GPCE 2017. Vancouver, BC, Canada: ACM, 2017, pp. 157–168. ISBN: 978-1-4503-5524-7. DOI: 10.1145/3136040.3136044.

[Pér+18] Arsène Pérard-Gayot et al. "A Data Layout Transformation for Vectorizing Compilers". In: *Proceedings of the 2018 4th Workshop on Programming Models for SIMD/Vector Processing*. WPMVP'18. Vienna, Austria: ACM, 2018, 7:1–7:8. ISBN: 978-1-4503-5646-6. DOI: 10.1145/3178433.3178440.

[Pér+19] Arsène Pérard-Gayot et al. "Rodent: Generating Renderers without Writing a Generator". In: *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2019)* 38.4 (July 28–Aug. 1, 2019), 40:1–40:12. DOI: 10.1145/3306346.3322955.

[Pey+05] Simon Peyton Jones et al. “Practical type inference for arbitrary-rank types”. In: *Journal of Functional Programming* 17 (Jan. 2005). Submitted to the Journal of Functional Programming, pp. 1–82. URL: https://www.microsoft.com/en-us/research/publication/practical-type-inference-for-arbitrary-rank-types/.

[Pha+97] Matt Pharr et al. “Rendering Complex Scenes with Memory-coherent Ray Tracing”. In: *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH ’97. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1997, pp. 101–108. ISBN: 0-89791-896-7. DOI: 10.1145/258734.258791.

[Pho75] Bui Tuong Phong. “Illumination for Computer Generated Pictures”. In: *Commun. ACM* 18.6 (1975), pp. 311–317. ISSN: 0001-0782. DOI: 10.1145/360825.360839.

[Pie02] Benjamin C. Pierce. *Types and Programming Languages*. 1st. The MIT Press, 2002. ISBN: 0262162091.

[Pir98] Pekka P. Pirinen. “Barrier Techniques for Incremental Tracing”. In: *Proceedings of the 1st International Symposium on Memory Management*. ISMM ’98. Vancouver, British Columbia, Canada: Association for Computing Machinery, 1998, pp. 20–25. ISBN: 1581131143. DOI: 10.1145/286860.286863.

[PKS17] Arsène Pérard-Gayot, Javor Kalojanov, and Philipp Slusallek. “GPU Ray Tracing using Irregular Grids”. In: *Computer Graphics Forum* 36.2 (2017), pp. 477–486. DOI: 10.1111/cgf.13142.

[PM12] Matt Pharr and William R Mark. “ispc: A SPMD compiler for high-performance CPU programming”. In: *2012 Innovative Parallel Computing (InPar)*. IEEE. 2012, pp. 1–13.

[PT00] Benjamin C. Pierce and David N. Turner. “Local Type Inference”. In: *ACM Trans. Program. Lang. Syst.* 22.1 (Jan. 2000), pp. 1–44. ISSN: 0164-0925. DOI: 10.1145/345099.345100.

[Rag+13] Jonathan Ragan-Kelley et al. “Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines”. In: *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 2013, pp. 519–530. DOI: 10.1145/2462156.2462176.

[RC05] Christian P. Robert and George Casella. *Monte Carlo Statistical Methods (Springer Texts in Statistics)*. Berlin, Heidelberg: Springer-Verlag, 2005. ISBN: 0387212396.

[Ren+15] Bin Ren et al. “Efficient Execution of Recursive Programs on Commodity Vector Hardware”. In: *SIGPLAN Not.* 50.6 (2015), pp. 509–520. ISSN: 0362-1340. DOI: 10.1145/2813885.2738004.

[Rey93] John C. Reynolds. “The discoveries of continuations”. In: *LISP and Symbolic Computation* 6.3 (1993), pp. 233–247. ISSN: 1573-0557. DOI: 10.1007/BF01019459.

[RO10] Tiark Rompf and Martin Odersky. "Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs". In: *Proceedings of the 10th International Conference on Generative Programming and Component Engineering (GPCE)*. 2010, pp. 127–136. DOI: 10.1145/1868294.1868314.

[Rob65] J. A. Robinson. "A Machine-Oriented Logic Based on the Resolution Principle". In: *J. ACM* 12.1 (Jan. 1965), pp. 23–41. ISSN: 0004-5411. DOI: 10.1145/321250.321253. URL: https://doi.org/10.1145/321250.321253.

[Smi15a] Ryan Smith. *The AMD R9 Nano Review: The Power of Size*. 2015. URL: https://www.anandtech.com/show/9621/the-amd-radeon-r9-nano-review/15.

[Smi15b] Ryan Smith. *The NVIDIA GeForce GTX Titan X Review*. 2015. URL: https://www.anandtech.com/show/9059/the-nvidia-geforce-gtx-titan-x-review/15.

[Son+14] Kristian Sons et al. "shade.js: Adaptive Material Descriptions". In: *Computer Graphics Forum* 33.7 (2014), pp. 51–60. ISSN: 1467-8659. DOI: 10.1111/cgf.12473.

[SS95] Philipp Slusallek and Hans-Peter Seidel. "Vision - An Architecture for Global Illumination Calculations". In: *IEEE Transactions on Visualization & Computer Graphics* 1 (1995), pp. 77–96. ISSN: 1077-2626. DOI: 10.1109/2945.468387.

[Suj+11] Arvind K. Sujeeth et al. "OptiML: An Implicitly Parallel Domain-Specific Language for Machine Learning". In: *Proceedings of the 28th International Conference on Machine Learning (ICML)*. 2011, pp. 609–616.

[TBN11] Dmitriy Traytel, Stefan Berghofer, and Tobias Nipkow. "Extending Hindley-Milner Type Inference with Coercive Structural Subtyping". In: *Programming Languages and Systems*. Ed. by Hongseok Yang. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 89–104. ISBN: 978-3-642-25318-8.

[Tsa09] John A. Tsakok. "Faster Incoherent Rays: Multi-BVH Ray Stream Tracing". In: *Proceedings of High Performance Graphics*. HPG '09. New Orleans, Louisiana: ACM, 2009, pp. 151–158. ISBN: 978-1-60558-603-8. DOI: 10.1145/1572769.1572793.

[TT94] Mads Tofte and Jean-Pierre Talpin. "Implementation of the Typed Call-by-Value λ-Calculus Using a Stack of Regions". In: *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '94. Portland, Oregon, USA: Association for Computing Machinery, 1994, pp. 188–201. ISBN: 0897916360. DOI: 10.1145/174675.177855.

[Vea98] Eric Veach. "Robust Monte Carlo Methods for Light Transport Simulation". AAI9837162. PhD thesis. Stanford, CA, USA, 1998. ISBN: 0-591-90780-1.

[VG95] Eric Veach and Leonidas J Guibas. "Optimally combining sampling techniques for Monte Carlo rendering". In: *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*. ACM. 1995, pp. 419–428.

[VK16] Jiří Vorba and Jaroslav Křivánek. "Adjoint-driven Russian Roulette and Splitting in Light Transport Simulation". In: *ACM Trans. Graph.* 35.4 (2016), 42:1–42:11. ISSN: 0730-0301. DOI: 10.1145/2897824.2925912.

[Wal+01] Ingo Wald et al. “Interactive Rendering with Coherent Ray Tracing”. In: *Computer Graphics Forum* (2001). ISSN: 1467-8659. DOI: 10.1111/1467-8659.00508.

[Wal+07] Ingo Wald et al. *SIMD Ray Stream Tracing - SIMD Ray Traversal with Generalized Ray Packets and On-the-fly Re-Ordering*. Tech. rep. 2007.

[Wal+14] Ingo Wald et al. “Embree: A Kernel Framework for Efficient CPU Ray Tracing”. In: *ACM Trans. Graph.* 33.4 (2014), 143:1–143:8. ISSN: 0730-0301. DOI: 10.1145/2601097.2601199.

[Wal04] Ingo Wald. “Realtime Ray Tracing and Interactive Global Illumination”. PhD thesis. Computer Graphics Group, Saarland University, 2004.

[WBB08] Ingo Wald, Carsten Benthin, and Solomon Boulos. “Getting Rid of Packets: Efficient SIMD Single-ray Traversal Using Multibranching BVHs”. In: *IEEE/Eurographics Symposium on Interactive Ray Tracing*. 2008, pp. 49–57. DOI: 10.1109/RT.2008.4634620.

[WH06] Ingo Wald and Vlastimil Havran. “On building fast kd-Trees for Ray Tracing, and on doing that in O(N log N)”. In: *2006 IEEE Symposium on Interactive Ray Tracing*. 2006, pp. 61–69. DOI: 10.1109/RT.2006.280216.

[Wil+14] A. Wilkie et al. “Hero Wavelength Spectral Sampling”. In: *Proceedings of the 25th Eurographics Symposium on Rendering*. EGSR ’14. Lyon, France: Eurographics Association, 2014, pp. 123–131. DOI: 10.1111/cgf.12419.

[Zha+18] Yunming Zhang et al. “GraphIt: A High-Performance Graph DSL”. In: *PACMPL* 2.OOPSLA (2018), 121:1–121:30. DOI: 10.1145/3276491.

[ZWL17] Stefan Zellmann, Daniel Wickeroth, and Ulrich Lang. “Visionaray: A Cross-Platform Ray Tracing Template Library”. In: *Proceedings of the 10th Workshop on Software Engineering and Architectures for Realtime Interactive Systems (IEEE SEARIS 2017)*. Mar. 2017, pp. 1–8.

Acronyms

API Application Programming Interface

AST Abstract Syntax Tree

AVX Advanced Vector Extensions

AVX2 Advanced Vector Extensions, version 2

BPT Bidirectional Path Tracing

BSDF Bidirectional Scattering Distribution Function

BVH Bounding Volume Hierarchy

CPU Central Processing Unit

DSL Domain-Specific Language

FLOPS FLoating point OPerations per Second

FPGA Field-Programmable Gate Array

GPU Graphics Processing Unit

ILP Instruction-Level Parallelism

IS Importance Sampling

LLVM Low-Level Virtual Machine (`https://llvm.org`)

LoC Lines of Code

LTO Link-Time Optimization

LMS Lightweight Modular Staging

MC Monte Carlo

MIS Multiple Importance Sampling

NEE Next Event Estimation

OpenCL Open Computing Language

PT Path Tracing

PM Photon Mapping

PPM Progressive Photon Mapping

SIMD Single Instruction Multiple Data

SSA Static Single Assignment

SSE Streaming SIMD Extensions