

Spear: Across the Streaming Multiprocessors

Porting a Production Renderer to the GPU

Clifford Stein

cstein@imageworks.com Sony Pictures Imageworks Culver City, USA Chris Hellmuth hellmuth@imageworks.com Sony Pictures Imageworks Vancouver, Canada Alejandro Conty Estevez aconty@imageworks.com Sony Pictures Imageworks Vancouver, Canada

Pascal Lecocq plecocq@imageworks.com Sony Pictures Imageworks Vancouver, Canada Larry Gritz lg@imageworks.com Sony Pictures Imageworks Vancouver, Canada



(a) Ecto-1 render on CPU

(b) Ecto-1 render on GPU

Figure 1: An equal sample comparison (256 samples per pixel) of CPU vs GPU renders of the Ecto-1 vehicle from *Ghostbusters: Frozen Empire* ©2023 CTMG, Inc. All Rights Reserved. The GPU has noticeably brighter reflections in the windows. The GPU render was approximately 8x faster.

ABSTRACT

We ported the Sony Pictures Imageworks version of the Arnold Renderer to the GPU using NVIDIA's OptiX ray tracing toolkit. This required modifying algorithms to run efficiently on the GPU, the use of new software methodologies to better share source code between the host and device renderers, and a reevaluation of what contributes to poor performance on the device. We share here the key decisions we made to overcome these challenges and the valuable lessons we learned during our journey in implementing the Sony Pictures Evolved Arnold Renderer (Spear) on the GPU.

CCS CONCEPTS

• Computing methodologies → Ray tracing.

DigiPro '24, July 27, 2024, Denver, CO, USA

© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM. ACM ISBN 979-8-4007-0690-5/24/07 https://doi.org/10.1145/3665320.3670988

KEYWORDS

ray tracing, GPU, rendering, path tracing, shading

ACM Reference Format:

Clifford Stein, Chris Hellmuth, Alejandro Conty Estevez, Pascal Lecocq, and Larry Gritz. 2024. Spear: Across the Streaming Multiprocessors: Porting a Production Renderer to the GPU. In *The Digital Production Symposium* (*DigiPro '24*), *July 27, 2024, Denver, CO, USA*. ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3665320.3670988

1 INTRODUCTION

The Sony Pictures Imageworks (SPI) version of the Arnold Renderer [Kulla et al. 2018] is a highly parallel and efficient CPU-based path tracer that has been used in over fifty feature motion pictures. Artists at Imageworks use Arnold to perform look-development (a.k.a. "lookdev") and lighting in an interactive/live-rendering environment from within the Katana lighting package. While we had monitored the "pulse" of ray tracing on the GPU over the years, it was the introduction of NVIDIA's Turing microarchitecture with hardware ray tracing in 2018 that demonstrated to us compelling evidence to port the SPI Arnold renderer to the GPU.

We saw a number of benefits of porting SPI Arnold to the GPU. We wanted to build upon the productivity increase we saw when

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

our artists started using SPI Arnold's live-rendering features by providing them with even faster tools. And as USD/Hydra [Pixar 2016] becomes more widespread at our facility via DCC integration, we wanted to offer faster and better renders in these contexts that were also more consistent throughout the pipeline. There was also the possibility that as we reworked algorithms to be more amenable to running on the GPU that the CPU-side might benefit as well.

Early during the development of our port to the GPU we set the goal that our first users would be lookdev artists because of the simpler lighting set-ups and smaller environments they use compared to lighters. We also decided to add GPU support to our renderer instead of writing an entirely new tool because we felt it gave us the greatest chance of success in matching the look of the CPU renderer since artists are very sensitive to changes in looks. We believed if we could share code and algorithms between CPU and GPU, then that would give us the greatest chance of matching feature and image parity between the two renderers (a strategy taken by "XPU" renderers as described in Section 2). While we knew we could not reach 100% feature parity with the software renderer, there were some features which were non-negotiable: texturing support, hair rendering, AOVs, homogeneous participating media, and the OSL [Gritz et al. 2010] shading system upon which our facility shading library is built.

SPI's version of Arnold is based on a fork of the commercial Arnold Renderer [Georgiev et al. 2018], independently developed and evolved at SPI since around 2009. As such, our renderer contains a novel GPU implementation which we are calling *Spear*, which stands for Sony Pictures Evolved Arnold Renderer.

This article describes the journey we took in adding GPU support to our renderer. We will cover some of the lessons we learned to maximize performance and maintain feature parity. Some of this necessitated the rethinking of core algorithms to make them more GPU-friendly and also making contributions to 3rd party open source projects.

Throughout this work, we will try to adhere to the following terms: "host" refers to the CPU, "device" refers to the GPU, and we will use "platform" to denote the environment where code runs, such as on the host (CPU) or device (GPU).

2 BACKGROUND

Many GPU-based ray tracing renderers, ranging from academic research tools to applications for film production, are based on NVIDIA's CUDA [Kirk 2007] toolkit, which is a set of libraries, compilers, and development tools for writing kernels that execute on GPU hardware.

CUDA was followed by OptiX [Parker et al. 2010], a ray tracing API specifically for NVIDIA's GPUs. OptiX provides ray tracing "building blocks" such as BVH builders for triangles and curves, ray-geometry intersectors, the ability to call user-supplied kernels for launching rays, processing ray-geometry intersections, misses, etc. The kernels are written with CUDA and run on the GPU device and can take advantage of NVIDIA's ray tracing hardware.

DXR [Microsoft 2018] and Vulkan [Khronos Group 2016] are two other ray tracing APIs with hardware support, but they were quickly ruled out due to our specific needs: DXR only runs on Windows and we are a Linux facility; and while Vulkan [Khronos Group 2016] runs on Linux it does not have native motion blur support. Therefore, we settled on CUDA and OptiX.

Several feature film production ray tracers such as Pixar's RenderMan [Christensen et al. 2018] and Dreamworks Animation's MoonRay [Lee et al. 2017] have "XPU" technology meaning that both the CPU(s) and GPU(s) can render a frame collaboratively to improve performance. But we decided against this approach because we felt the relatively small performance bump one would get by adding the system's CPU to a GPU with hardware-based ray tracing would not be worth the added code complexity; the faster the GPU implementation relative to the CPU, the less of an impact a CPU contribution would have. We feel this decision was validated by the results discussed in Section 13.

3 SHARING CODE IS HARD

We set a mandate that Spear would be a single application capable of rendering either on the CPU or GPU (but not necessarily both simultaneously). As such, we began this project with the noble intention of sharing as much code as possible between the two rendering engines, with the expectation that sharing code would help us attain our goal of 1:1 look parity (modulo pixel color differences caused by LSB floating-point differences in the calculations) by sharing algorithms between the rendering platforms. Because our CPU rendering engine is written in C++, we thought sharing code with the GPU, which is built with CUDA C++, would be straightforward-one simply puts platform-agnostic code in header files which are included by platform-specific . cpp and . cu files! However, we quickly found it was much faster and easier to rewrite, or cut-and-paste, rendering code than it was to disentangle algorithms from their associated data structures, especially since many of those structures contained data which did not need to be resident on the device and would occupy precious RAM. Over time, however, we have been cleaning up our data structures and adopting software patterns to help our code consolidation efforts.

First, we needed to remove virtual methods and recursion from many of our host-side designs due to the difficulty, or inability, of supporting those features in device-side kernels. Those were replaced with switch/case alternatives where we made sure they were not re-entrant. The same new pattern was also adopted on the host-side codepaths to avoid code duplication. The second challenge was that many elements/algorithms, which would be the same on both the CPU and GPU, differ only in their lower level operations.

We found the CRTP ("curiously recurring template pattern") idiom, shown in Listing 1, to be useful for specifying shared algorithms and data structures in some base class, and then specifying the platform-specific code in a derived class (i.e. host or device) which the shared algorithms could call. This gives us the effect of having virtual function calls for our different architectures which are resolved at compile time without incurring any runtime penalty or memory overhead. The result is a collection of templates that expand to slightly different implementations for the CPU- and GPUcodepaths. This way, developers write the code only once and the compiler tailors it to the particular platform for us.

Finally, there is a recurring design pattern used for nearly every object and data structure we need to upload to GPU due to our desire to minimize their use of explicit pointers. We used an approach Spear: Across the Streaming Multiprocessors

```
template <typename T>
class Common
{
    // shared/common methods and data, call
    // functions like this:
    float3 lookup_texture(float2 uv)
        { return T::lookup_texture_impl(uv); }
}
class CPUPlatform : public Common<Platform>
{
    // platform (CPU/GPU) specific code
    float3 lookup_texture_impl(float2 uv)
        { ... } // Actual implementation
}
Common<CPUPlatform> cpu; // In CPU code
Common<GPUPlatform> gpu; // In GPU code
```

Listing 1: CRTP example in C++

where we store data in contiguous blocks of memory, and pointers to that data are computed on the fly via offsets. It could be defined as data-oriented, trivially relocatable design. Data structures such as trees, sampling tables, and even hash tables, can be packed into a single block of memory when coupled with methods that provide the correct offsets to desired fields. These blocks can be allocated and easily copied between platforms (or just elsewhere in memory) with a single operation and be expected to work without tediously patching pointers.

4 DATA COHERENCY FOR THE WIN

Early in the development for the GPU's rendering engine we had both "megakernel" and "wavefront" [Laine et al. 2013] ray tracing implementations. The megakernel-based engine was similar to our host-side implementation where each thread iteratively traces a ray, evaluates shaders, trace more rays, etc. The wavefront implementation, in contrast, would trace a "wave" of rays, sort the hits based on some criterion, evaluate shaders, then trace another wave of rays, etc. We found the wavefront implementation was substantially faster, by a factor of up to 2x. Eventually we learned that this was because computation time per path segment was invariant to scene depth in the wavefront architecture, but would suffer in the megakernel as we increased maximum depth. This was due to more and more threads being idle from their paths ending as depth increased. In fact, one thread in a warp with high depth could hold the other 31 completed threads idle as it completed its integration.

However, maintaining two different ray tracing codepaths was cumbersome. We removed the wavefront implementation after finding that we were able to attain its performance in our megakernel codepath by introducing sample depth coherency to nearby pixels, which substantially sped up renders at the cost of some increase in noise. To do this, our Russian Roulette implementation uses similar "base" random numbers for blocks of pixels during each rendering pass, and then "micro-jitters" each individual pixel slightly in the same spirit as [Dufay et al. 2016].

4.1 Shader Execution Reordering

Version 8 of the OptiX toolkit introduced Shader Execution Reordering (SER) which allows the GPU to "sort" rays after a ray tracing "traverse". The hope is that by grouping "similar" rays together data and execution divergence will be minimized during shader execution. Indeed, we found OptiX's SER, with the default coherency heuristics, increased performance by as much as 2.45x in some of our production assets. We saw the most benefit in large outdoor environments where some rays would immediately bounce into the skydome and terminate, leaving their threads idle while other rays in the warp could bounce many times.

Unfortunately, most of our artists are not able to utilize SER due to a lack of available Ada-generation cards in our facility. We expect to experiment further with OptiX coherency hints, and to explore the relationship between SER and our sample depth coherency technique, as more of our users acquire SER-compatible hardware.

5 COMPILE TIMES

Long compile times are a perennial gripe about developing for the GPU. Unlike traditional host-side software which is compiled a single time when the software is released to the facility, device-side software is compiled twice: first, into a device-agnostic intermediate representation (IR); and second, from this IR into device-specific code. The first stage can be performed when the application is built, but the second cannot because its output is specific to the microarchitecture of the device it will be running on. This second stage, performed by the device driver, occurs during the execution of the application when it uploads the IR to the device. This "Just in Time" (JIT) compilation can incur a several minute delay on start-up because the entire device-side renderer, effectively, is being compiled.

Hardware vendors attempt to ameliorate this JIT penalty by caching the compiled device code into a database which is keyed by the fingerprint of the IR. This works well as long as programs always compile to the same IR. We also found that globally disabling inlining reduced JIT times by 2-4x. This would hurt performance by as much as 10x, but worked very well for iterative developer builds.

5.1 Character String Hashes

Our facility's shading library is built on top of OSL. When we initially added OSL support to Spear, one component of the shading library that we uploaded to the GPU was a global character strings file which contained the device addresses of all the strings used by shaders and the rendering system. Because its content depended on the rendered scene, the file was constructed and compiled on-the-fly at render time. This could introduce a start-up JIT delay since the IR could change for each render launch due to the non-deterministic addresses of the character strings in device memory. In other words, we were defeating the driver's caching mechanism!

We eliminated this start-up delay by changing how OSL dealt with character strings. For GPU compilation, we changed OSL's behavior to pass 64-bit character string hashes (based on the ustring hash of the string) instead of char pointers. In addition to addressing the start-up delay, this change yielded a few other benefits: string hashes become compile time constants (via constexptr functions) which gives the optimizer more information to work with; and we no longer need to upload strings to the GPU which saves device space and reduces host-side code complexity. This work has since been extended to apply to code generated for the CPU as well.



(a) Slimer render on CPU

(b) Slimer render on GPU

Figure 2: An equal sample comparison (256 samples per pixel) of CPU vs GPU renders of Slimer from *Ghostbusters: Frozen Empire* ©2023 CTMG, Inc. All Rights Reserved. Note that the GPU does not always match the CPU 1:1 yet, as seen in the softer specular highlights in the CPU render. The GPU render was approximately 6.9x faster.

The likelihood of experiencing a collision between 64-bit hashes is extremely low, especially in a context where it would lead to an actual image difference (such as due to texture names colliding). We have run tests on production scenes and have not experienced any collisions, but we do not enable the check by default to avoid slowing down facility renders. However, we are aware of the possibility and have plans to address it.

6 OSL REPARAMETERIZATION

Our artists light and lookdev interactively with "live rendering" from within Katana. While our CPU-based renderer is fairly quick, the GPU's interactivity is so much faster that it revealed some of the shortcomings in our Katana-to-Arnold translation plug-in and forced us to rethink some of the decisions we made years ago,

When we first wrote the material handling portion of our Katanato-Arnold plug-in, we did the "easy" thing of deleting old material shading nodes each time a material changed (either due to changes in the shading graph or due to simple parameter edits) because Katana lacks useful introspection to give us insight into what has changed when an artist edits a material. This was a reasonable choice at the time because we could avoid tracking material changes. And, it did not seem to have much of an impact on performance because re-JITing shading networks for the CPU was fast.

On the GPU, however, things were different—a shading edit would trigger shader recompilation and device re-JITing which introduced a noticeable delay for artists. So, we finally added material tracking logic that could distinguish between material network changes and simple parameter edits. This allowed us to use OSL's "reparameterize" facility where changes to parameters which have been marked as "editable" are copied to the appropriate slot in memory without the need to recompile the shaders. Changes on the GPU-side became near instantaneous, and changes on the CPU-side were noticeably faster, too. Shading network changes still need to go through the usual compile/JIT cycle, however.

An important feature of OSL is its late-stage optimization after the shader graph is connected and values of all shader parameters are known. This runtime optimization is key to performance, and since marking parameters as "editable" prevents them from being constant-folded, it is important to keep the working set of editable parameters at any one time restricted to only those parameters likely to be interactively manipulated. Changing the set of currently editable parameters, just like changing the connectivity of the shader node graph, requires a full re-JIT of the shader group.

7 STACK SIZE

One of the items which surprised us the most was the impact of stack size on GPU renders. For host-side rendering, stack size is something we never think about with the exception of occasionally increasing the OS default. In a virtual memory system, the amount of RAM dedicated to the stack by a host-side application with tens, or even hundreds, of parallel threads is generally negligible. However, when device-side code has, effectively, tens of thousands of threads in flight, the memory dedicated to stack space can run into the gigabytes—we were observing that device-side stack space for a render of a production asset was taking over 12GB of device RAM! Even if we were willing to pay that cost, our most complex OSL shaders would occasionally cause us to surpass CUDA's 64KB stack limit, leading to un-renderable scenes.

7.1 Rethinking Data Structures

We tracked down significant stack consumption to some particular software design patterns in the shading library used for passing texturing parameters between shading nodes in a material network which led to some large data structures. After working with the shading team to implement a few fixes, we saw a reduction in shader footprint 2-3x, which also sped up our renders by a remarkable 1.5-4x. We have continued to pare this down even further.

7.2 Updating OSL Device Entry Points

When we first added OSL support to Spear, the device-side entry point to compiled OSL shaders required pointers to two buffers which were provide by the renderer: a "group data" buffer where OSL would store shader parameters, and a buffer where the shading network returned its expression tree of closures. For performance

DigiPro '24, July 27, 2024, Denver, CO, USA



(a) Carol model rendered at 256 samples per pixel on GPU



(b) Carol model warp-cost heatmap

Figure 3: A GPU render of the Carol model from *The Marvels* ©2023 Marvel compared against our warp-cost heatmap visualization on the right. The visualization shows that the most expensive areas of the image are where the hair and skin start overlapping, leading to execution divergence even on primary rays.

reasons, we stored these two buffers on the stack to use the thread's local memory. This meant the buffers' sizes needed to be specified at compile time, and that forced us to plan for the worst case scenario that we would expect to encounter when executing shaders.

We found some of our more complicated shading networks required a group data buffer of size 48-64KB which, when multiplied by the huge number of in-flight threads, led to vast amounts of memory set aside for shader execution. This was a performance hit "twofer": this reduced the amount of memory available for geometry and texture storage, and it could also slow the execution of shaders as it increased the amount of memory copying that happens as threads are "shuffled" around during execution.

As we implemented more features on the device side and were able to render more sophisticated production assets, we encountered shading networks requiring group data buffers on the order of 96KB. At that point we finally accepted that fixed-size buffers were not tenable, and we implemented a technique which we called device-side "growable buffers".

7.3 Growable Buffers

The basic concept of growable buffers is to reserve some small amount of memory on the stack (per-thread) which could be used for the group data buffer or closure storage. If more memory is required, then we would overflow into a buffer allocated in the device's heap. If that heap pool is exhausted during rendering then we will halt ray tracing and shading execution on the device, double the size of the overflow buffer, and then relaunch the rendering kernel.

The two growable buffer use cases, the closure pool and the group data buffer, have slightly different specifications which affects their implementations. In the case of closures, shaders allocate space in the closure buffer during execution whenever they add a new closure to the Cout output variable. This happens through a function call into the device-side renderer services library which makes it easy to control when to overflow into the heap-located buffer and to signal that the heap buffer has been exhausted. In contrast, the group data problem required a more complicated solution which necessitated us making a few additions to OSL:

- OSL can now return the amount of group data space required for a shader group;
- (2) OSL now has responsibility for allocating the shading network's group data on the stack instead of the renderer; and
- (3) the renderer can set an OSL attribute that will cap how much group data space OSL will allocate on the stack and also pass in a pointer to a heap-allocated "overflow" buffer via the shading network's entry point.

The group data change cut our memory requirements in half for most scenes, since we no longer had to statically reserve enough memory for our most complex shaders. This moved our memory bottleneck away from the shaders, and onto the renderer itself.

8 STOCHASTIC SHADING

The development of Spear caused us to reevaluate some of the original decisions we made while implementing the CPU renderer. One of these was how to evaluate the closure trees returned by shaders for illumination. The host-side renderer constructs BSDF objects for each of the returned closures in the expression tree, randomly selects one for next bounce sampling, and performs next event estimation (NEE) for all of them. This is a common practice.

On the GPU, we found it very expensive and memory costly to construct all of the BSDF objects returned in the closure expression tree. Therefore, we introduced the concept of "stochastic shading" where we randomly select one of the closures to construct from the closure tree based on their albedos. And we do this greedily using reservoir sampling so we only keep a single BSDF object at all times.

We use the selected BSDF to sample the next bounce and also to perform NEE. While this improved performance, it also increased noise too much. We found that extending it to two BSDFs, one for smooth specular lobes and one for rougher ones, provided a good balance of noise reduction and performance improvement over the single lobe method. We anticipate developing this approach further. Clifford Stein, Chris Hellmuth, Alejandro Conty Estevez, Pascal Lecocq, and Larry Gritz



(b) Warp visualization for a scene with high depth divergence

Figure 4: Our warp visualization tool helped build intuition about the relative costs of path computation and low thread utilization. The different colors denote different stages of integration. Both visualizations are generated with SER disabled.

9 VOLUMES

On the CPU side, Arnold renders volumes via ray marching and an MIS combination of density and equiangular samples [Kulla and Fajardo 2012]. During ray tracing, we accumulate a list of volumetric intersections, which is converted into a sorted list of intervals over fixed volume segments. For each interval, we allocate an array for accumulating density samples while marching along the interval. The array's size is proportional to the voxel density of the volume. Later in the pipeline, we sample a scattering point from the populated array.

In our GPU implementation of volumes, we opted to avoid growable arrays, and changed the integration calculations to work with a fixed memory footprint. This had two implementation consequences:

- Instead of tracing one ray and returning a vector of hits, on the GPU we trace and integrate one hit at a time.
- (2) Instead of allocating an array to sample density, we tightlycouple sampling and shading, and reservoir sample our shading point as we sample density. This works very well for single volumes, but means that we have to invert our forloop when integrating segments containing multiple volumes: instead of processing each volume in isolation, we only loop over the marching steps once. This means that the inner-loop samples each volume at each step, which is not cache-efficient.

We took advantage of NanoVDB [Museth 2021] to both upload and traverse voxels on the GPU. NanoVDB is a very successful example of reallocatable data-oriented design from which we learned a lot. Equiangular sampling is not yet implemented due to a shift in priorities, but we are able to render dense fields like clouds or smoke at a very high performance gain compared to the CPU.

10 AOVS

Arbitrary Output Variables (AOVs) allow artists to render different geometric or shading components of a scene, facilitating complex layer compositing in a single shot. On the CPU, each output allocates its own frame buffer, and multiple outputs can point to the same AOV, but with different destinations, such as image files or OpenGL textures. In addition, each output can have different filtering properties that require additional memory resources. For example, a depth filter retains the closest or farthest depth contributions by storing depth values in a frame.

Such replication of memory across outputs is undesirable on the GPU due to its limited memory capacity. To address this, we had to rethink our AOV model to better mutualize and conserve memory resources on the GPU. For example, two outputs pointing to the same AOV but different targets now share a single frame buffer. For depth filters, the memory resources are reduced to a single z-buffer, and the discard decision is made only once for all outputs using a depth filter. By rethinking our AOV model in this way, we have been able to conserve GPU memory resources while still providing artists with the functionality they need when rendering.

It should be noted that for outputs targeting CPU memory, we make sure the memory is page-locked. This enables efficient DMA (Direct Memory Access) transfers between the GPU and CPU, maximizing data transfer speed. This optimization is particularly critical in real-time rendering applications where direct access to the GPU memory might not be feasible.

11 DIAGNOSTIC TOOLING

When trying to analyze our GPU performance, NVIDIA's nsightcompute tool gave us valuable statistics related to occupancy, divergence, cache utilization, and memory usage. But we struggled to get top-down profiling analysis from the application similar to what we get from VTune on the CPU.

Occasionally we wrote custom tools to answer our questions about how our code was behaving on the GPU. One interesting visualization logged the clock cycle that each thread hit for different stages of integration (e.g., bouncing, shading, or shooting a shadow ray). Combined with information about that thread's SM, warp, and lane, we could visualize what the GPU utilization actually looked like over the duration of a frame's computation.

This technique helped us learn about the relative costs of shading versus ray tracing on the GPU, and further illuminated the negative effects of completed threads on sample throughput. Figure 4 compares a simple scene with a small set of materials and low bounce counts to a complex outdoor environment with many materials. Whitespace in the diagram indicates time where a thread has no assigned work. Figure 3 shows a custom warp-cost heatmap AOV assembled from the same data, highlighting which areas of the image are the most expensive to render.

11.1 Visualizing OSL Shaders

One tool that proved its worth multiple times was an OSL shader graph visualizer that let us examine the parameter costs of the connections between layers in a shadergroup. Although production graphs can be massive in size (see an example of a moderately sized graph in Figure 5), being able to visually identify clusters of



Figure 5: An example OSL shader graph. We annotated nodes with metrics such as parameter size to help track down clusters of heavy layers and identify GPU anti-patterns that we pruned from our shader generation pipeline.

layers with heavy parameters was invaluable. In one case, we were able to identify a large parameter array which could be reduced in size (Section 7.1). In another, we realized we were writing constant values over a parameter's initial values in a way that OSL was unable to optimize. In both cases, we saw significant performance boosts with adjustments to our shader generation.

EXPECTING THE UNEXPECTED 12

Over the course of any large software project one must be open to reconsidering choices made early in its development, and Spear was no different. In addition to reconsidering our position on wavefront ray tracing, as already mentioned in Section 4, there were several other decisions where we needed to reevaluate and pivot.

12.1 Device-pointer abstraction

Early during the development of Spear, we introduced a device pointer abstract data type. For device-side code, it would be interpreted as a typed pointer while on the CPU it would be interpreted as simple pointer-sized integer. The rationale was that the compiler could help enforce rules that pointers to device memory would raise compilation errors when dereferenced on the CPU. On paper, it seemed like a good idea but its use became such a hindrance that we ultimately removed it. It had a complicated API for performing "common" pointer operations such as fetching the typed pointer, fetching the void pointer, performing pointer arithmetic, getting a reference to the underlying pointer, etc. And it could not easily be used in data structures that were shared between the CPU and GPU platforms.

Shader Initiated Ray Tracing 12.2

Anecdotally, we heard that when first writing a GPU ray tracer, one is amazed with the speed-up compared to a CPU-based ray tracer. But as one slowly adds features to match CPU, the hardware renderer begins to slow down and it is difficult to claw back those performance gains. Thus, we told our shading team that we would not support OSL's trace() call on the GPU, thinking that shaders using it would have terrible performance.

SPI's facility shading library makes trace calls for features such as estimating surface curvature, attaching shadow rays to projections, and projecting room interiors for buildings. On a whim, we decided to evaluate the performance hit of performing a simple trace() call inside shaders on a variety of production assets, and we we were pleasantly surprised by the modest slowdown it introducedroughly 5-10% on the scenes we tested.

12.3 Dipole SSS

We were hoping to "gently" retire our dipole subsurface implementation [King et al. 2013] from the facility, first by not supporting it on the GPU followed by eventually removing it from CPU renderer, because it is problematic on high-curvature geometry and we thought we had a more robust SSS implementations, based on a real medium random walk, that could yield a similar look. We thought if artists became accustomed to not using it on the GPU due to the GPU's ray-tracing speed, then it would not be missed during offline CPU renders on the render farm. However, we eventually learned that some artists were preferring to lookdev on the CPU rather than the GPU because of the missing dipole SSS; either they preferred the "look" or convergence of the older SSS, or they were having trouble matching assets and characters from previous shows, or from outside clients, with the newer SSS models. For these reasons, we will port this feature to the GPU after all.



(a) CPU render

(c) Difference image

Figure 6: Comparison of the CPU and GPU rendered Ecto-1 vehicle from Figure 1. The values in the difference image have been scaled by 2x to accentuate the differences.

RESULTS 13

Objectively comparing performance between the CPU and GPU implementations is not always straightforward. Both share lighting and shading code, but the top-level integrator is still different. Hence they produce slightly different images. This is also due to the unimplemented functionality on the GPU. Once everything is supported on GPU this will be resolved by unifying everything. The CPU integrator often generates better but more expensive samples than the GPU. Comparing convergence rates is time-consuming, and on the GPU we prefer a quicker frame to a longer, more efficient frame in order to maximize interactivity. This is especially true when the OptiX denoiser (see Section 13.1) is on, as it can clean up much of the noise caused by the less efficient strategy.

Clifford Stein, Chris Hellmuth, Alejandro Conty Estevez, Pascal Lecocq, and Larry Gritz

Table 1: Frame time averages for various benchmark scenes

scene	cpu frame	gpu frame	speed-up
bucket	1525.1 ms	102.7 ms	14.9x
pot	1724.0 ms	103.8 ms	16.6x
elliot	1387.5 ms	110.0 ms	12.6x
breakfast	1402.5 ms	114.5 ms	12.3x
newsstand	766.4 ms	116.0 ms	6.6x
vehicle	2040.1 ms	127.3 ms	16.0x
resort	1402.7 ms	190.6 ms	7.4x
dining-room	1626.3 ms	194.9 ms	8.4x
hospital	2045.6 ms	229.8 ms	8.9x
classroom	2881.2 ms	274.2 ms	10.5x
pig	3609.5 ms	370.3 ms	9.8x
creek	3565.9 ms	426.4 ms	8.4x

Table 1 shows a direct comparison of frame times to get a general idea of the performance benefits we see on the GPU. Because the renders are not a strictly apples-to-apples comparison because of the differences in integrators (and elsewhere), internally we usually track the GPU's performance over time against its previous versions. One welcome side-effect of the GPU project has been porting cheaper GPU sampling strategies back to the CPU, and seeing efficiency gains there, as well (e.g., stochastic shading in Section 8). All timings in this paper are comparing an NVIDIA RTX 6000 (Ada Generation) to a dual-processor Intel Xeon Gold 6226R system with 64 threads which is typical of our lighters' workstations. We have also increased OptiX's per-thread register count to 192 registers.

Figures 1 and 2 illustrate some side-by-side comparisons of production assets rendered on the CPU and GPU. The images are qualitatively very similar but the differences are obvious when flipping between the two interactively. Some differences in the "Ecto-1" vehicle are highlighted in Figure 6. As we consolidate code and algorithms between the two rendering codepaths, we expect these differences to lessen over time.

13.1 Enhancing the visual experience.

The integration of a denoiser pass using the OptiX AI-based denoiser into Spear significantly enhances the user experience by eliminating noise on the fly during live editing scenarios. This enables artists to work more efficiently and make more informed decisions during the creative process. By leveraging the temporal and upscaling features of the OptiX denoiser, we further improve the denoising quality while doubling the rendering performance. This is achieved by rendering at half resolution and then upscaling the image to its original size, which trades minor detail loss for a 2x speed boost-this is slower than the expected 4x performance improvement due to the overhead of the denoiser itself and the incompressible execution time of a GPU thread. It is noteworthy that the denoiser pass is also available on the CPU, but the ability to do instant denoising on a complete frame with Spear on the GPU, as opposed to the legacy bucket rendering on the CPU, delivers a far more engaging experience for the lookdev artists.

14 CONCLUSION

We have enjoyed the process of creating our GPU-based renderer, Spear. The journey has been frustrating at times, partly due to development tools that are not as capable as their counterparts for CPU-side software development and partly due to our intuition sometimes failing us as we try to rationalize performance regressions. But we cannot say it has not been educational. We have re-architected algorithms to be GPU-friendly, embraced software patterns to maximize code reuse between our CPU and GPU renderering engines, and developed new diagnostic tools to measure performance.

While we continue to improve performance and match feature parity with CPU Arnold—the lack of deep output AOVs, pointcloud, and bump-to-roughness [Olano and Baker 2010] support are priorities we'd like to address—artists have begun using Spear for lookdev. Interestingly, we have learned that interactivity drives more interactivity; as artists use the GPU they are finding that the fast feedback they get quickly puts them in "the zone", and any delay which takes them out, such as having to restart the renderer to enable a particular feature, is highly distracting.

ACKNOWLEDGMENTS

There are a number of individuals and teams we would like to thank for their help during this development. We are indebted to the OptiX team at NVIDIA for their time and patience in explaining the intricacies of writing efficient GPU code. We would particularly like to express our gratitude to Tim Grant for his GPU-side contributions to the OSL codebase, and Mark Leone for the ondemand texture loading library. Alex Wells and Steena Monteiro from Intel contributed important OSL work related to the string representation and other architectural groundwork. We would also like to acknowledge Christopher Kulla for his early contributions to the Spear renderer as well as Lee Kerley for his discussions and improvements in making our SPI facility shading library more GPU "friendly". We would like to thank SPI management, especially Steven Vargas and Mike Ford for their support throughout this project. And lastly, we would like to thank Spear testers Orde Stevanoski, Brian Steiner, Israel Yang, Dan Lavender, and Thomas Devorsine for their patience, feedback, and understanding as we fixed bugs and addressed missing features; their high standards helped make Spear a stronger product.

REFERENCES

- Per Christensen, Julian Fong, Jonathan Shade, Wayne Wooten, Brenden Schubert, Andrew Kensler, Stephen Friedman, Charlie Kilpatrick, Cliff Ramshaw, Marc Bannister, Brenton Rayner, Jonathan Brouillat, and Max Liani. 2018. RenderMan: An Advanced Path-Tracing Architecture for Movie Rendering. ACM Trans. Graph. 37, 3, Article 30 (aug 2018), 21 pages. https://doi.org/10.1145/3182162
- Arthur Dufay, Pascal Lecocq, Romain Pacanowski, Jean-Eudes Marvie, and Xavier Granier. 2016. Cache-Friendly Micro-Jittered Sampling. In ACM SIGGRAPH 2016 Talks (Anaheim, California) (SIGGRAPH '16). Association for Computing Machinery, New York, NY, USA, Article 36, 2 pages. https://doi.org/10.1145/2897839.2927392
- Iliyan Georgiev, Thiago Ize, Mike Farnsworth, Ramón Montoya-Vozmediano, Alan King, Brecht Van Lommel, Angel Jimenez, Oscar Anson, Shinji Ogaki, Eric Johnston, Adrien Herubel, Declan Russell, Frédéric Servant, and Marcos Fajardo. 2018. Arnold: A Brute-Force Production Path Tracer. ACM Trans. Graph. 37, 3, Article 32 (aug 2018), 12 pages. https://doi.org/10.1145/3182160
- Larry Gritz, Clifford Stein, Chris Kulla, and Alejandro Conty. 2010. Open Shading Language. In ACM SIGGRAPH 2010 Talks (Los Angeles, California) (SIGGRAPH '10). Association for Computing Machinery, New York, NY, USA, Article 33, 1 pages. https://doi.org/10.1145/1837026.1837070

Khronos Group. 2016. Vulkan. https://www.vulkan.org/.

Spear: Across the Streaming Multiprocessors

- Alan King, Christopher Kulla, Alejandro Conty, and Marcos Fajardo. 2013. BSSRDF Importance Sampling. In ACM SIGGRAPH 2013 Talks (Anaheim, California) (SIG-GRAPH '13). Association for Computing Machinery, New York, NY, USA, Article 48, 1 pages. https://doi.org/10.1145/2504459.2504520
- David Kirk. 2007. NVIDIA Cuda Software and GPU Parallel Computing Architecture. In Proceedings of the 6th International Symposium on Memory Management (Montreal, Quebec, Canada) (ISMM '07). Association for Computing Machinery, New York, NY, USA, 103–104. https://doi.org/10.1145/1296907.1296909
- Christopher Kulla, Alejandro Conty, Clifford Stein, and Larry Gritz. 2018. Sony Pictures Imageworks Arnold. *ACM Trans. Graph.* 37, 3, Article 29 (aug 2018), 18 pages. https://doi.org/10.1145/3180495
- Christopher Kulla and Marcos Fajardo. 2012. Importance Sampling Techniques for Path Tracing in Participating Media. *Comput. Graph. Forum* 31, 4 (jun 2012), 1519–1528. https://doi.org/10.1111/j.1467-8659.2012.03148.x
- Samuli Laine, Tero Karras, and Timo Aila. 2013. Megakernels Considered Harmful: Wavefront Path Tracing on GPUs. In Proceedings of the 5th High-Performance Graphics Conference (Anaheim, California) (HPG '13). Association for Computing Machinery, New York, NY, USA, 137–143. https://doi.org/10.1145/2492045.2492060

Mark Lee, Brian Green, Feng Xie, and Eric Tabellion. 2017. Vectorized Production Path Tracing. In *Proceedings of High Performance Graphics* (Los Angeles, California) (*HPG '17*). Association for Computing Machinery, New York, NY, USA, Article 10, 11 pages. https://doi.org/10.1145/3105762.3105768

Microsoft. 2018. DXR. https://microsoft.github.io/DirectX-Specs/d3d/Raytracing.html.

- Ken Museth. 2021. NanoVDB: A GPU-Friendly and Portable VDB Data Structure For Real-Time Rendering And Simulation. In ACM SIGGRAPH 2021 Talks (Virtual Event, USA) (SIGGRAPH '21). Association for Computing Machinery, New York, NY, USA, Article 1, 2 pages. https://doi.org/10.1145/3450623.3464653
- Marc Olano and Dan Baker. 2010. LEAN Mapping. In Proceedings of the 2010 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games (Washington, D.C.) (I3D '10). Association for Computing Machinery, New York, NY, USA, 181–188. https://doi.org/10.1145/1730804.1730834
- Steven G. Parker, James Bigler, Andreas Dietrich, Heiko Friedrich, Jared Hoberock, David Luebke, David McAllister, Morgan McGuire, Keith Morley, Austin Robison, and Martin Stich. 2010. OptiX: A General Purpose Ray Tracing Engine. ACM Trans. Graph. 29, 4, Article 66 (jul 2010), 13 pages. https://doi.org/10.1145/1778765.1778803
- Pixar. 2016. Introduction to USD. http://openusd.org.