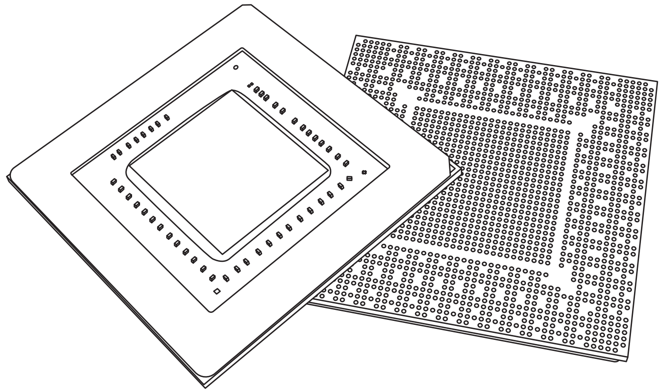


GUIDE TO GPU

chaosgroup.com | v-ray.com | labs.chaosgroup.com

CHAOSGROUP | LABS

version 1.0

contributors: Christopher Nichols, Blagovest Taskov
editors: Henry Winchester, Colin McLaughlin, David Tracy, Dariush Derakhshani
design and illustrations: Melissa Knight, Andrey Katanski, Desislava Bruseva
image contribution: Dabarti Studio - Tomasz Wyszolmirski

Introduction

When it comes to rendering, speed has always been important to artists, whether they are working in architecture, film, TV or marketing. What hasn't always been so constant is the hardware they use to achieve that speed. Nowadays, there are two distinct technologies available for ray tracing a computer generated scene - CPU (Central Processing Unit), the original force, and GPGPU (General Purpose Graphics Processing Unit), the up-and-comer. Both are essential parts of modern computers, but they work in completely different ways.

This guide seeks to dispel some of the myths around CPU and GPU workflows, while also providing a window into the history, advantages and disadvantages of each option. Although GPUs are quickly becoming a go-to choice for many cases, it's still not always the best choice or the only choice. V-Ray has been GPU-ready since 2008 and gives artists both options in one program – CPU or GPU. The last part of this guide will delve into what V-Ray provides for GPU users and how to get the most out of it.

Contents

History

```
D/C/W/M ?
MEMORY SIZE?
TERMINAL WIDTH?

   7423 BYTES FREE

OSI 6502 BASIC VERSION 1
.0 REV 3.2
COPYRIGHT 1977
OK
```

Understanding the rise of GPUs requires a little history lesson. For instance, many of the common applications of the present weren't anticipated by the original designers. Their intent was to correct a processing weakness in CPUs. Many graphic calculations had always been slow for CPUs, providing room for GPUs to fill the gap and focus on this specific aspect of the computing process. As the years went on, it was discovered that GPUs could bring more speed to new calculations – ray tracing, for instance.

In the timeline below, you'll see how GPUs have adapted and branched out over time. Something to note is that as companies like NVIDIA and AMD grow, so do the opportunities for artists and software developers.

1960s

Computers can only show alphanumeric characters. As their processing power increases, scientists realize they can be used for far more than just numbers and letters. However, this requires special hardware known as a framebuffer, which turns computer code into a visual representation a screen can represent using pixels. The first framebuffer is prototyped at Bell Labs in 1969, and becomes the original ancestor of today's GPU.

1970

In 1972 Xerox starts producing framebuffers commercially. From there, the graphical capabilities of computers evolve quickly. Screens go from monochrome, to displaying a few colors, to displaying millions of colors. More advanced imagery suddenly becomes possible, but also puts a heavy load on the CPU. To distribute the load, GPUs emerge. These specialized chips handle advanced graphical computations, and free up the CPU for other tasks.

1980s

Home computing and gaming become big businesses, and the power of the hardware snowballs. The advanced graphics made possible by GPUs help sell computers like the Commodore Amiga and IBM-compatible PCs.

1990s

The development process behind GPUs grows disorganized and inconsistent. Gaming and computing companies create their own solutions for 3D content. Different designs, approaches and implementations become the norm, instead of the emergence of an industry standard. While this flurry of competition helps push the industry forwards, its haphazard nature results in many companies giving up or going out of business.

1992

Silicon Graphics' Open Graphics Library (OpenGL) helps standardize graphics programming. This API for 2D and 3D is vendor agnostic, allowing it to run on any hardware.

1996

3dfx's Voodoo graphics card is one of the first pieces of hardware to unify the industry. This card slots into a computer's PCI slot and bumps up its graphical prowess. Along with the advent of OpenGL, the Voodoo fundamentally changes the way computers work, giving software companies a platform that would deliver impressive results across a range of hardware.

2000

3dfx's dominant position in the turbulent hardware industry is short-lived, and it's soon purchased by NVIDIA. After many other market acquisitions, two desktop GPU giants emerge: NVIDIA and ATi (now AMD). Two APIs also rise to the top: OpenGL, and Microsoft's DirectX.

2002

Purcell (et al) writes a paper called "Ray Tracing on Programmable Graphics Hardware". This paper discusses a breakthrough where fixed function pipelines were replaced with programmable vertex. In doing so, this paradigm shift opened up the possibility of raytracing using GPUs.

2004

Gelato is released by NVIDIA; it becomes the first commercially available renderer made for the GPU. That same year, Martin Christen of the University of Applied Science Basel, writes a GPU-based ray tracer for his thesis.

2005

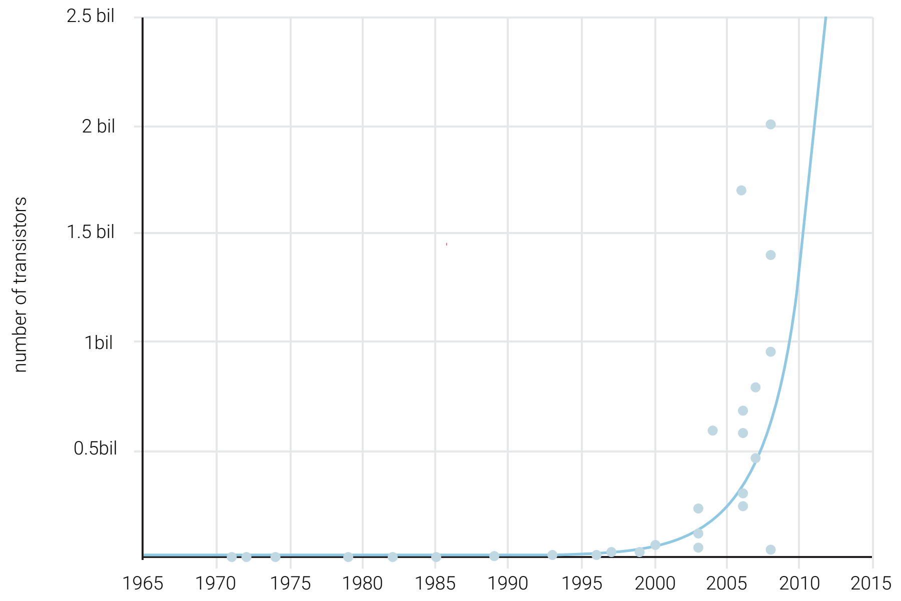
Game developers request more programmable shaders. Hardware vendors give them more freedom through extensions and new APIs to use the GPUs for purposes other than 3D graphics. This results in entire applications that can run on the graphics processor, turning them into general purpose GPUs (GPGPUs). We'll revisit these again later with respect to raytracing.

2008

Chaos Group creates a version of V-Ray called V-Ray RT, which runs on both the GPU or CPU (see page 36).

2013/2014

In 2013 NVIDIA announces the Quadro K6000 which includes 12GB of memory; AMD announces the FirePro W9100 with 16GB of memory soon after. This big jump in memory size enabled users to consider using GPUs for final frame rendering.



Moore's law *(curve) attempted to predict how the number of transistors on a processor would increase over time (dots)*

{fig 01}

Hardware

Choosing between a CPU or a GPU isn't always cut and dry; in fact, it often depends on where that specific hardware is in the development process, or which trends are about to make it more applicable for a specific use. The truth is CPUs and GPUs organically evolve in different ways. The closer we can get to understanding these elements, the easier it is to see the current advantages and disadvantages of both solutions.

Moore's law

Moore's law is named after Intel's Gordon Moore, who in 1975 theorized that every 18 to 24 months, the number of transistors on a processor will roughly double. *{fig 01}* For a long time, people assumed that Moore's law would predict the increase in processor speeds. This misconception, and the idea that chip speeds aren't "following the law" like they used to, have led some to proclaim Moore's law as dead. This is not true. In reality, companies now see performance improvements in other ways. There are three main areas that performance can be gained: speed, efficiency, and parallelism. We will explain how each of these can be attained.

Speed

The reason that speeds have stalled on the CPU is interesting. Initially, Moore's law predicted that processor manufacturers would halve the size of their transistors, so they could put twice as many on the die. Because these transistors are smaller, they can run at twice the speed, or frequency. At the same time, they require considerably less voltage, which results in about the same power consumption. However, because of a physics phenomenon called leakage, processor companies can't keep lowering the voltage. If they don't lower the voltage, but they increase the frequency, you get four times more power consumption.

As a result, a higher frequency processor will always require more power. Higher frequency processors also require more cooling, and cooling is not an infinite process – to be precise, no more than 250W per processor. All of this affects speed.

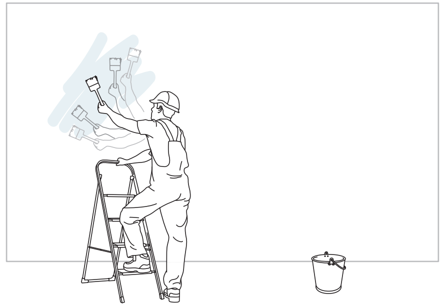
Efficiency

So if speed is limited, manufacturers have to start looking for other ways to gain performance. Here's an easy way to think about those options: Imagine you were painting a room. You basically have three options available to you if you want to get your project done faster. You can actually paint faster, make sure your paint bucket is as close as possible or ask for help.

Translated to processors, *faster* means higher frequencies. ***{fig 02}*** Single-core processor speed hit a power wall in 2005, at about 4GHz based on the limitations we discussed earlier.

Smarter means caches and complex processor logic, allowing more instructions to be executed. But this also has reached it maximum efficiency around the same time. To extend our painting metaphor, if the paint bucket was placed 10 meters from you, you would have to go back and forth a lot. ***{fig 03}*** At five meters it becomes a little easier. And once it is few inches from you, there's no way to improve the efficiency of getting your paint.

In the computer world, the bucket represents the cache, which is a small bank of memory built into the processor itself. It temporarily stores data before it's processed. Caches have now reached the point where making them bigger doesn't make any difference though.



*a **faster** processor means higher frequency*
- the painter works quickly

{fig 02}

***smarter** processing improves efficiency*
- the paint pot is close to the painter

{fig 03}

Parallelism

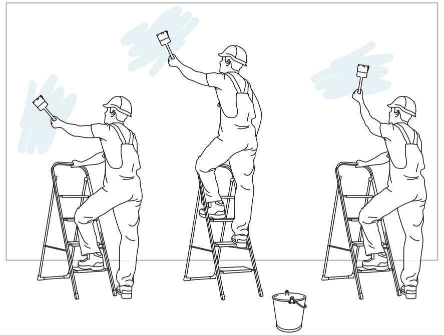
The last way to get more performance out of your hardware, the getting *help* principle, refers to parallelism, which means more cores. Parallel processors still work for us – we can insert more cores, because we can still shrink the transistors in size. Then they can work together as fast as we expect, and we can keep them all at the frequency limit we reached.

If we return to our room decorating metaphor, parallel processing is the equivalent of having more painters, one for each section of the wall. ***{fig 04}*** Something to note is they're all still taking paint from the same pot. Managing how the painters work together is still the only way to ensure maximum efficiency, which means some tasks are better suited for parallelism.

As parallel CPUs have risen up, programmers have made their software multi-core aware. When this software gets re-written in a parallel fashion, GPUs become suitable for some general purpose tasks. This has led some developers to only program for the GPU. And for some of them, this approach worked.

*parallel processors **help** increase performance*
- many painters work together

{fig 04}



Differences between CPU and GPU

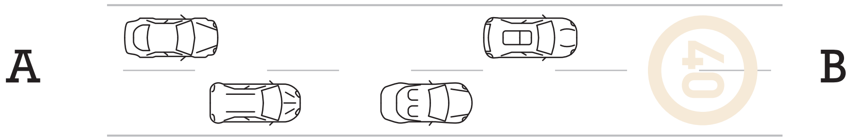
To better understand the difference between CPUs and GPUs, we have to start with latency and throughput.

Latency tells you how much time it will take to get an answer to a particular task. For example, if you have a car traveling from A to B, latency tells you how much time it will take the car to arrive.

Throughput, on the other hand tells you how many cars can make it from A to B within a particular amount of time. One way to increase throughput is to decrease the latency, by increasing the speed limit so that the cars can go faster. ***{fig 05}***

*a high **speed** limit results in a decrease in latency (time it takes to complete the task)*

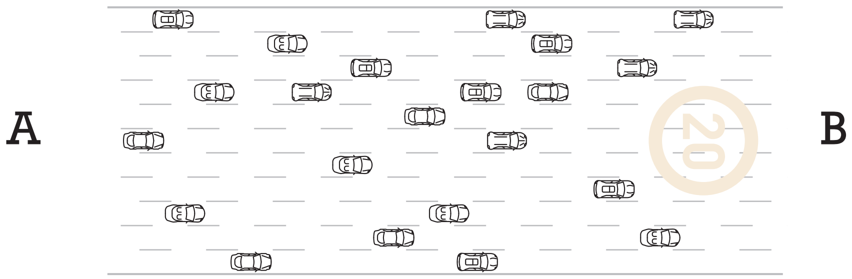
{fig 05}



Or we could let more cars on the roads by adding lanes (using parallelism). With more lanes you can achieve higher throughput even with lower speed limits, or higher latency. ***{fig 06}***

*increased **throughput** via parallelism (more lanes) also results in a decrease in latency, even if the speed is lower*

{fig 06}



CPUs are designed for low latency and were initially optimized for single tasks. GPUs, on the other hand, are designed for massively parallel and independent tasks, such as calculating the color of each pixel on the screen.

A GPU's "speed limit" is quite low compared to a CPU's, but they can carry many cars at the same time. Thus, they are better suited for problems with a massive amount of parallel and independent tasks. Luckily, there are many tasks which require minimal to no effort at all to be computed in parallel. These are known as *embarrassingly parallel* tasks. Ray tracing is such a task, as the rays that are being traced have no dependencies between one another.

There is a caveat, however. GPU hardware is designed for shaders and real time graphics, thus it is not as general-purpose as the CPU. It hits limitations, especially when it has to execute more complex tasks. For example, one limitation could be that the road is made so that all the cars on it can only travel in one direction. So even if you have an embarrassingly parallel problem, you may find that a CPU is still the way to go, because of its advanced and optimized general purpose nature.

Low-latency CPU processing comes at a cost, though. Usually such calculations are more expensive in terms of power, and GPUs tend to be more power efficient at certain tasks (which, as mentioned above, is very important). A GPGPU's clock is generally lower, which tends to require less power as well.

The way a GPGPU app is made, it abstracts the number of cores from the developer. Once the application is written to match certain criteria, the driver can run the GPGPU application on as many cores as there are available. This means that if the hardware changes after some time, and new GPUs with more cores are introduced, the GPGPU applications will scale automatically and use all the new cores.

Trends in GPU hardware

Going from 28nm to 14nm

A very important characteristic of microprocessors is the size of their transistors – the smaller they are, the more you can cram into a chip. Smaller transistors are usually faster and use less power, too. Since 2011, the biggest discrete GPU vendors – NVIDIA and AMD – were using 28 nanometer (nm) manufacturing process for their GPUs.

According to Moore's Law, NVIDIA and AMD should have switched to 22nm chips in 2014. These are about 70% smaller in each of the two dimensions, but their manufacture was prevented due to production issues. For the last five years they were stuck at 28nm, and only managed to eke improvements from producing better chip architecture and higher quality schemes from the current 28nm technology. The good news however, is that in the summer of 2016 instead of going to 22nm, they will jump directly to 16 or 14nm. Combined with 3D transistors, this could result in one of the biggest leaps in GPU performance since they were first manufactured.

GPU vendors will use these changes to introduce their new IP and architectures, named Pascal (NVIDIA) and Polaris (AMD). Because GPU applications aren't generally dependent on architecture types, GPUs can be much more flexible in drastically changing their architectures compared to CPUs.

Going from GDDR to HBM memory

NVIDIA engineer Bill Daily said that calculation units on the processor are like kids – they are easy to make and hard to feed. In other words, it is important to feed data to all the calculation units on the processor. Efficiently moving data also becomes more important – in fact, it's one of the most expensive operations in computing.

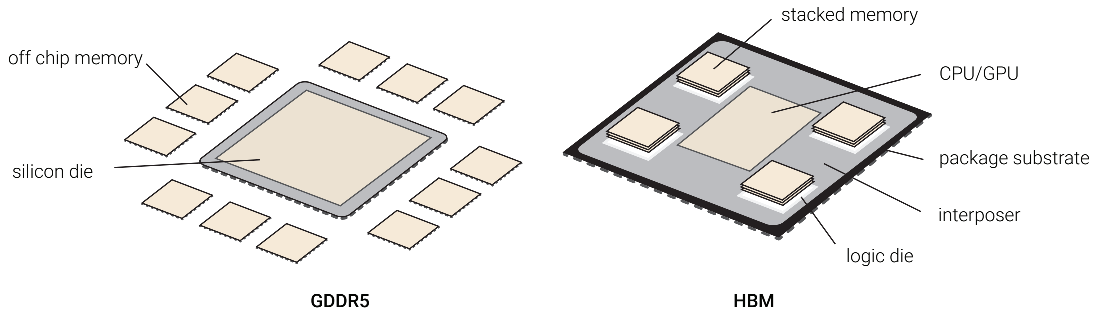
Imagine that you can feed 10GB/s of data to your GPU processor. If it gets two times more powerful by having two times as many calculation units, you will either need to feed it twice as much to keep the same data flow, or to reduce the data flow to it. The latter has little effect on performance, as there is no data to work on.

The solution to this problem is to move the memory as close as possible to the GPU. For a long time, this has been done with GDDR5 memory which is placed directly on the graphics card. Last summer AMD and Hynix memory designed a new kind of memory called high-bandwidth memory, or HBM. ***{fig 07}***

HBM consists of a stack of regular memory chips on top of one other. This allows the memory to be even closer to the GPU, and it uses less power while providing higher bandwidth. HBM memory is best suited for processors that require high bandwidth but not necessarily low latency, which makes it ideal for the GPU. The first version, called HBM 1, was introduced in the AMD Fury series, featured 500GB/s transfer speeds (compared to 384 GB/s of GDDR5) and used much less power. HBM 2 will be used in both AMD and NVIDIA's new GPUs, due in the imminent future, will have up to 1000GB/s.

High Bandwidth Memory (HBM) *configuration compared to GDDR5*

{fig 07}



Stacking memory over multiple GPUs with NVLink

Another approach for increasing GPU performance can be found in NVIDIA's Pascal architecture, which employs a new technology called NVLink. *{fig 08}* Until now, rendering on the GPU required the whole scene to be transferred to the GPU memory. There are techniques to render only the required parts of a scene. But when a render is done on multiple GPUs, the scene still has to be transferred to all of the GPUs' frame buffer, meaning that 2 GPUs with 12GB memory each still allows rendering without performance hits of a scenes that fit in 12GB, not 24GB.

NVLink provides connection between the GPUs, much like NVIDIA's SLI and AMD's CrossFire used to, but it is designed for GPGPU applications. It also provides cross-GPU memory access at very low latencies, which should enable "stacking" of the GPU memory. Stacking means that two GPUs with 12GB of memory each will have access to 24GB of memory, not 12.

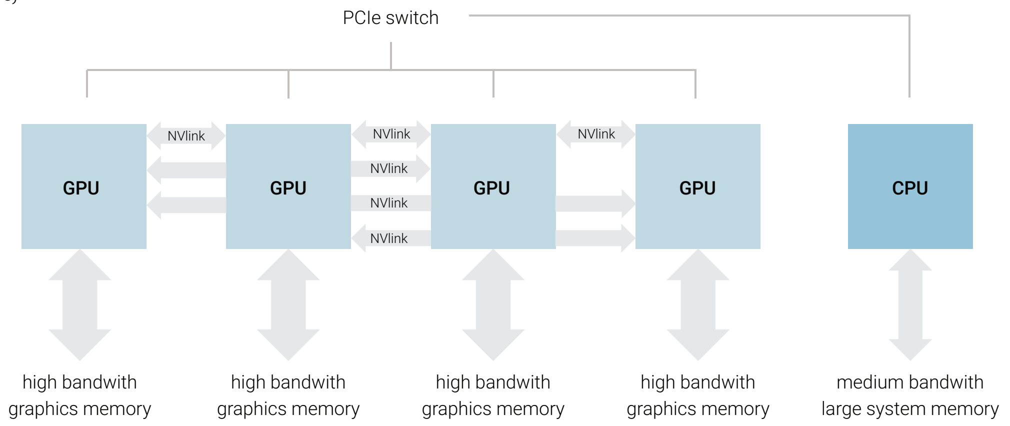
The maximum performance of NVLink is achieved with up to four GPUs, each with up to 16GB of memory (although this will expand in the future). This means around 64GB of cross GPU memory NVLink will be available in many future NVIDIA products.

Better performance from a single GPU

Another major feature for Pascal is the compute preemption. Nowadays, if a machine with only one GPU does rayt racing, the UI of the OS can become sluggish. This happens because the GPU gets too busy, and there is no time left for the OS. With compute preemption, automatic context switches between different applications should reduce this effect dramatically. In other words, you will be able to render on the GPU and still be able to work the interface.

a visual representation of ***NVlink***

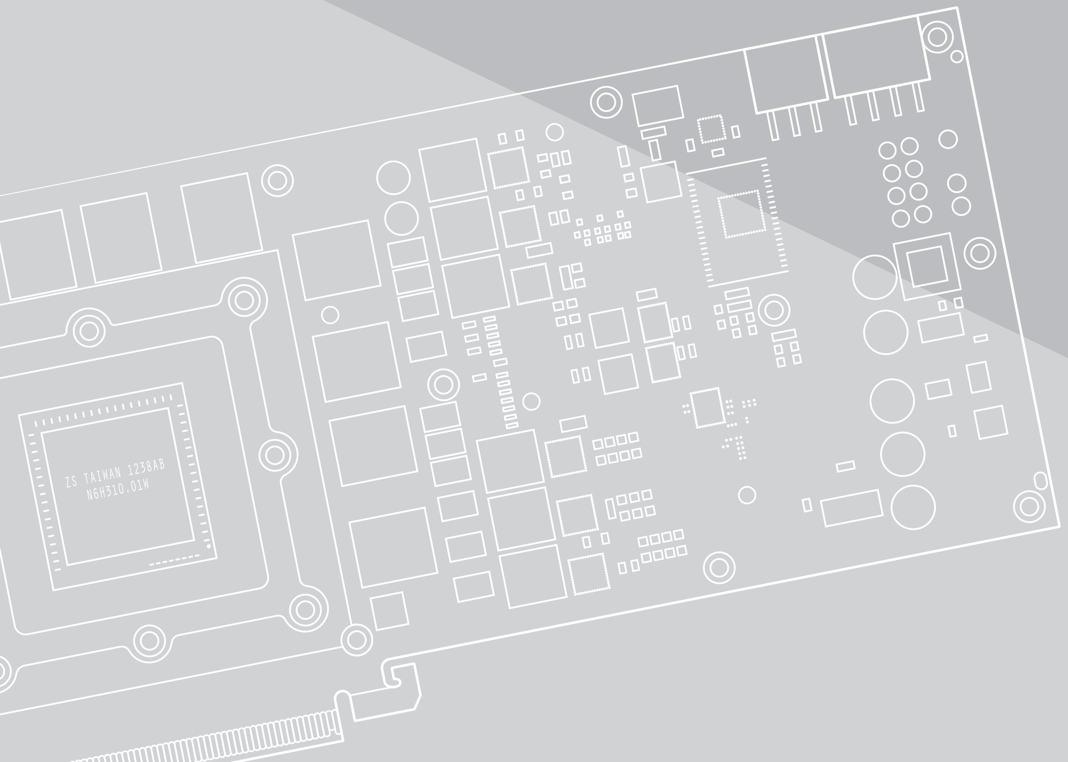
{fig 08}


PCIe switch
NVlink
NVlink
NVlink
NVlink
NVlink
NVlink
GPU
GPU
GPU
GPU
CPU
high bandwith graphics memory
high bandwith graphics memory
high bandwith graphics memory
high bandwith graphics memory
medium bandwith large system memory

Bright future through Pascal and Polaris

The micro architecture in Pascal is also better suited for GPGPU applications, meaning that each of the cores have better handling of complex tasks such as ray tracing, which results in higher performance. Even if Maxwell (NVIDIA's preceding architecture) and Pascal GPUs seem to have equal theoretical TFLOPs, the Pascal one should outperform it for ray tracing. Some of these changes include more registers, HBM2, and native half and double floating point numbers support. AMD's new Polaris architecture aims to improve GPGPU performance, and it's better suited to complex tasks than its current GCN architecture. Benefits include an improved memory cache system, a new memory controller and upgraded instruction dispatching mechanisms.

Drivers



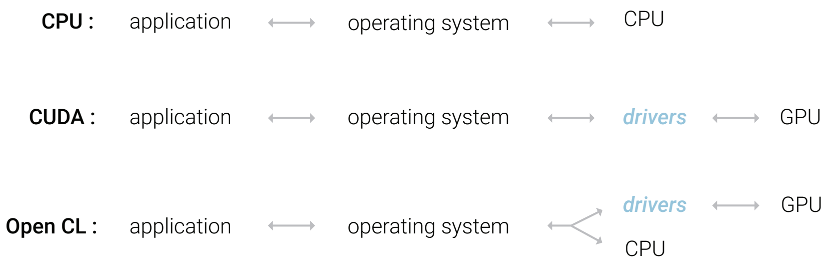
Drivers sit between the application, operating system and the hardware. *{fig 09}* Hardware is useless by itself, and it needs software to tell it what to do and how to do it. For the CPU, that role is filled by an operating system (OS), such as OS X, Windows, or Linux. The GPU relies on the driver, which links the hardware and the OS.

Non-GPGPU programs simply consist of the application and the OS. In the GPGPU world, there is the application, the OS, and the driver. The hierarchy that defines your system capability is Hardware-Driver-OS-App.

Hardware manufacturers make their own drivers, which either optimize or inhibit an artist's workflow depending on where they are in the release/upgrade process. An example: a freshly upgraded driver might not be compatible yet with software in an artist's pipeline. Until this situation is resolved, the efficiency and speed previously enjoyed by the artist might go down. However, after that resolution, those two attributes might go way up.

***drivers** play an important role in a GPU set up*

{fig 09}

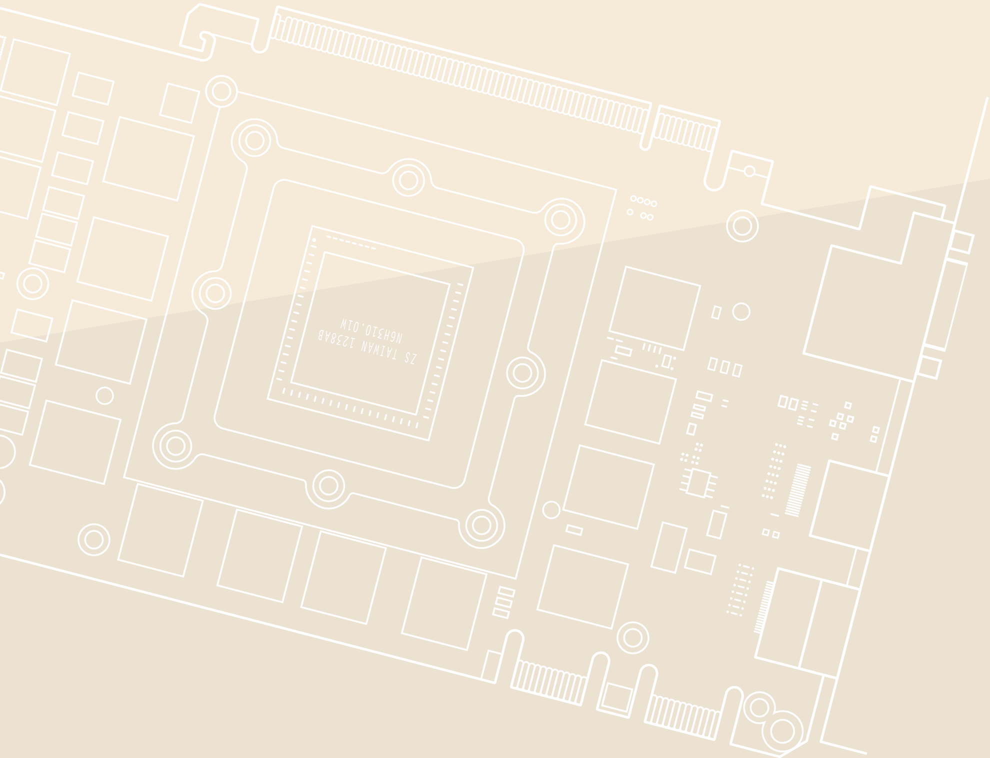


CPU : application ⟷ operating system ⟷ CPU

CUDA : application ⟷ operating system ⟷ *drivers* ⟷ GPU

Open CL : application ⟷ operating system ⟷ *drivers* ⟷ GPU / CPU

Rendering



Introduction

Now that we have a solid understanding of the different types of processors (CPUs and GPUs), we are ready to break down how this affects rendering. Ray tracing is traditionally done on the CPU, making ray tracers that work with GPGPUs a more recent event. Out of the many current options, V-Ray is unique in that it works with both. Something that is important to note is that while users can switch from CPU to GPU and apply the same shaders, they are in fact two completely different systems with their own advantages and disadvantages.

In this section we break down those pluses and minuses, and discuss how V-Ray has approached GPU rendering since 2008.

The history of V-Ray GPU

In 2008, Chaos Group started two separate GPU research projects: the first focused on an AMD GPU, the second on an FPGA processor. The AMD GPU project was led by a Chaos Group R&D developer named Deyan Ivanov. This became V-Ray RT, an interactive ray trace engine initially used for look development (mainly).

Developers tried to exploit the capabilities of OpenGL shaders for fully featured ray tracing. This was a standalone project, and not integrated into V-Ray.

These tests proved successful, but the Chaos team moved the development to BrookGPU API, a Stanford University project for GPGPU programming. It had many limitations, but it was better suited to GPGPU tasks than OpenGL shaders. Chaos Group used it to implement a path tracer which ran entirely on the GPU.

It became clear that the API, or language and hardware features could change, so the project was redesigned to so it could port fluidly to any future technology.

Tests continued with specialized ray tracing hardware by Caustic Graphics using a new language – OpenRL. These tests were based on a port of the BrookGPU implementation.

Around this time, the team started porting V-Ray RT to CUDA, and realized that the GPUs could work well for ray tracing. RT became an integral part of V-Ray.

As V-Ray RT was a ground-up design, built to be flexible, translating V-Ray RT from CUDA to OpenCL proved to be straightforward.

Since 2009, V-Ray RT's GPU development has been based on CUDA and OpenCL technologies.

In 2013 and 2014 V-Ray RT went from being a look development tool to a final production renderer. Features such as render elements support, light cache, displacement, subdivisions and more were added.

In 2014, Chaos Group Labs helped Kevin Margo produce a teaser for his short film CONSTRUCT. The entire short was to be produced with V-Ray on the GPU. It was a landmark project for GPU ray tracing. *{fig 10}*

V-Ray for MotionBuilder *in action during the production of the short film* ***CONSTRUCT*** (opposite)

{fig 10}

Kevin and Chaos Group Labs then collaborated with NVIDIA and the Chaos Group GPU and Cloud teams to see how far they could push the speed of GPU rendering. At SIGGRAPH 2014, the group was able to present V-Ray inside of MotionBuilder during Real-Time Live. This set-up also utilized the GPU.

In 2015, V-Ray RT was pushed even further with V-Ray Cloud. Using a standard internet connection, we were able to distribute and leverage 32 GPUs over 4 NVIDIA VCA machines. Using this technology, users are able to ray trace live motion capture at half HD frames at 24 frames a second.

Rendering on the CPU

Advantages:

- **Historical dominance** - Photorealistic computer rendering has a long history, and has conventionally been done on the CPU. Because of that, the CPU ray tracers are very advanced, and contain a lot of feature developments and improvements that have grown over time. Developing for the CPU is also easier most of the time, as it makes adding more features a simpler process. Additionally, developers are generally more familiar with programing on the CPU.

- **Best option for a single-threaded task** - As CPUs are very good at single-threaded tasks, they can implement algorithms that are not suited to parallelism, and therefore differing approaches are possible.

- **Superior at using all the resources on the computer** - The CPU has direct access to the main system memory and the hard drives, which gives it better flexibility in managing resources. Therefore, the CPU can usually hold a lot more data as system memory is not only cheaper, but also expandable. If you need more memory for your scene you can simply buy more. Additionally, you can use fast hard drives, such as SSDs, to cache more data out of memory.

- **Stability** - The current tools for CPU developers are more mature, which means CPU programs tend to be better tuned and more stable.

Drawbacks:

- **Hard to stack** - CPUs do not necessarily stack well. If you need more CPUs you often have to buy another computer. CPUs change their designs often as well, so you can't simply buy a new CPU and swap it with the old one, since it will not fit on the older motherboard. Thus, upgrades often require new a motherboard as well.

- **More power is more expensive** - Multi-CPU motherboards and NUMA (Non-uniform memory access) systems, promise more power, but they can be very expensive, and might not perform as expected in a lot of situations.

- **Inefficient** - A CPU often wastes a lot of power in order to deliver low latency results, thus for a massively parallel task such as ray tracing it tends to have worse performance per watt.

Remember: *CPUs are good for large tasks that don't fit on the GPU or complex tasks that are not yet supported by it. When you have more access to CPUs, which is frequently the case for most people/firms, CPUs might become the better bet.*

Special tip: *CPUs are very common for render farms, especially on the Cloud. GPUs are rarer.*

Rendering on the GPU

Advantages:

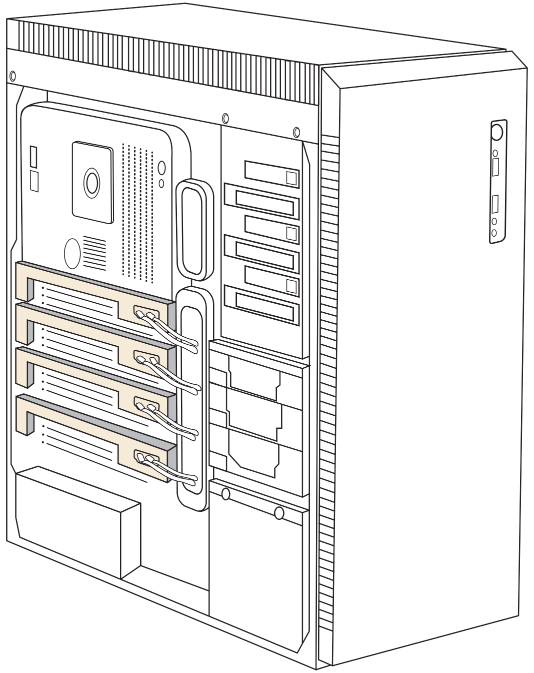
- **Scalability** - the most important advantage. The GPU works on the PCIe lane, which connects to the CPU. Most computer motherboards have more than one PCIe lane, so stacking multiple GPUs in one PC case is possible. *{fig 11}* This makes building a "personal farm" far easier – since you need fewer PC towers and licenses.

  If you are building your own computer and are considering adding multiple GPUs, consider this – PCIe bus speed comes in a few flavors in terms of width – x1, x4, x8, x12, x16 and x32. We recommend wider PCIe lanes as they have higher bandwidth. However, if the goal of the render is production and not interactivity, even narrower lanes can work for many GPU apps and ray tracers. V-Ray, for example, can be tuned to give tasks to the GPU in bigger chunks (bundle size / rays per pixel), so it will use the PCIe less often and it will not bottleneck.

- **Independent scaling** - GPU apps such as V-Ray are designed to scale independently to the number of active GPU threads, so stacking multiple GPUs gives a linear speedup. If you add a second GPU, the render will happen twice as fast. Eight GPUs will give you close to eight times the speed. And so forth. Usually you don't need a powerful CPU to feed the GPU devices. However, each GPU requires quite a lot of power (up to 250W each), and that requires a decent PSU.

- **Speed boosts** - A lot of render setups are suited for GPU hardware and software. As GPUs are designed for massively parallel tasks, they can provide better performance per watt, and better performance in general. For some scenes, using CPU vs GPU hardware, where the cost of each system is considered approximately the same, users can see speed-ups that are 2 to 15x faster than a CPU.

- **More gains from different processors** - Many GPGPU applications, such as V-Ray are heterogeneous, which means these systems can gain performance or energy efficiency not just by adding the same type of processors, but by adding dissimilar processors. Because of the way some GPGPU APIs such as OpenCL are designed, the same code can be run on a system's GPU and the CPU, using all available hardware for calculations.



*multiple GPUs can be **stacked** within a PC case*

{fig 11}

Drawbacks:

- **Memory** -The GPU is a mini-computer with its own memory –usually GDDR3, GDDR5 or HBM. The GPU can perform at its best as long as the resources it needs can fit within that memory. When that threshold is surpassed, either the performance, quality or quantity of the resources will decrease.

- **Communication bottlenecks** - GPUs have to communicate with the main system memory and hard drives through the CPU; they don't have direct access. The latency of these communications is high, and the wait for such distant and long reads can decrease performance. Therefore, each GPU has to contain the entire scene and textures in its own memory to render. So if you have a mixture of video cards, say a card with 4GB of memory and one with 12GB, the entire scene must fit in the 4GB to render properly. In order to use more than 4GB you would have to disable the 4GB card for rendering and only use the 12GB card.

- **Driver dependencies** - Writing code that works on the GPU is dependent on that particular system's drivers. This extra level of dependence on the driver can be a challenge. GPU drivers change often, and are sometimes updated on a monthly basis, which is much more frequent than an operating system. While this update can help development move forward, it also presents the chance that things can break. When this happens, developers need to report the bug, wait for a fix, and inform their users to try different drivers until the issue is addressed.

- **Frequent hardware updates** - In the CPU world, where there are few hardware architectures that do not change frequently (these days they are x86-64 and ARM). In the GPU world this is not the case. Hardware coming from the same vendors have new architectures every couple of years. This means that software either has to be recompiled in order to run on the new hardware, or it has to be shipped in a form that is suitable for the new drivers to transform to the new architectures. And while GPGPU can remain somewhat agnostic to its hardware, it is still dependent on elements like driver updates and features to make sure that it will remain compatible with the new hardware.

- **Calculation flaws** - As GPUs are designed for games, general purpose calculations tend to not work for all setups in an optimal fashion.

Remember: *Because of space limitations, early GPUs were valued for small render tasks like look development. In recent years, based on bigger cards and smarter software, many of those limitations have gone away.*

Picking the right GPU

Professional vs Gamer's GPUs

A professional artist could use either option for their work. That said, NVIDIA and AMD have at least two separate lines of GPUs, one for professional tasks, such as ray tracing or video editing, and one for gaming. The professional ones (NVIDIA's Quadro, AMD's FirePro) usually have additional features, more memory, better drivers and support, and these features are useful for some applications. Some professional cards also have double precision, which can provide a boost for certain simulations and models..

GPU rendering in V-Ray, however, is implemented in a way which doesn't need any exclusive professional-GPU features. Therefore, gaming cards (NVIDIA's GeForce, AMD's Radeon) work just as well. Depending on the card, they can even outperform professional cards at a fraction of the cost.

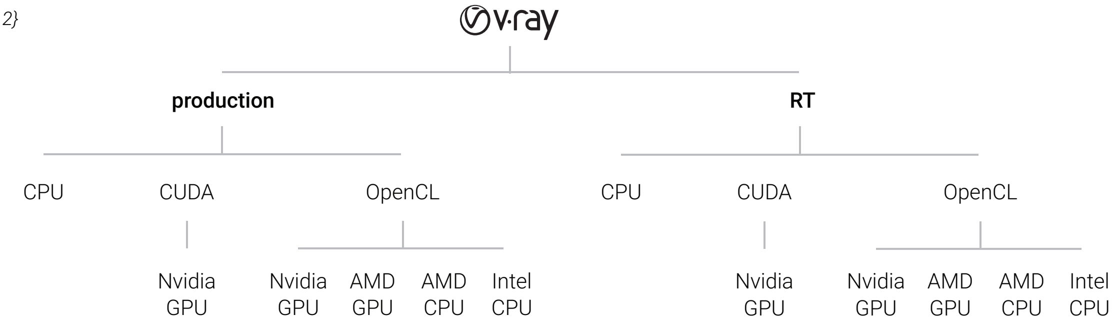
Understanding the different APIs choices for GPU rendering

Different GPU manufacturers favor different APIs for their cards. While all cards will work with both open or proprietary APIs, each card works best with a specific one. For the moment, for NVIDIA, that means CUDA. For AMD, that means OpenCL.

There are several different APIs that can be used to compile V-Ray on the GPU, including CUDA, OpenCL, Metal, Renderscript, DirectCompute and VulkanCompute. NVIDIA's CUDA and the open-source OpenCL offer the most benefits to V-Ray users, so we will concentrate on these next. ***{fig 12}***

V-Ray production / RT *using different APIs*

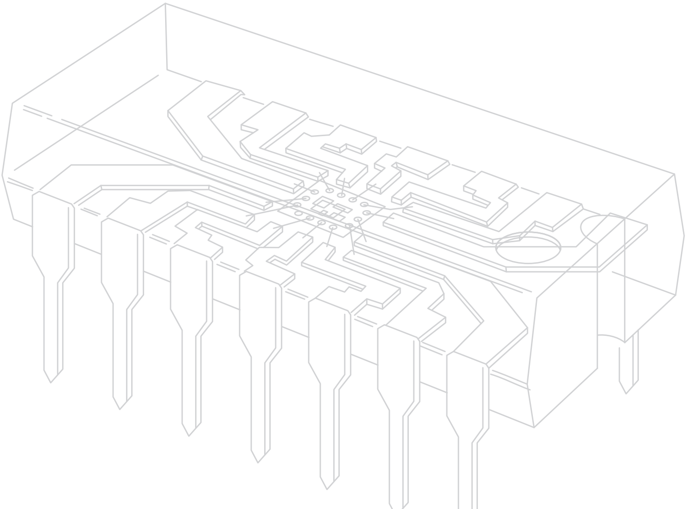
{fig 12}



CUDA

The first popular API was CUDA, created by NVIDIA. At the time of writing, CUDA is proprietary and only runs on NVIDIA hardware. But its specification is public, and AMD aims to implement CUDA at some point. It should be noted that just because certain hardware supports an API does not mean it will be optimized for it. As previously stated, NVIDIA GPUs will almost always run faster with CUDA and AMDs will most likely run faster in OpenCL.

Thanks to CUDA's proprietary nature, it's evolved faster than other committee-driven APIs such as OpenCL, since decisions are much easier when you have complete control. NVIDIA's CUDA drivers are known to be very good– CUDA applications run fast, there are rarely any driver-related issues. Its API is easy to work with, and backed up with developer tools to help debug and profile CUDA applications. GPGPU apps that have both CUDA and OpenCL versions such as V-Ray on the GPU should always be run with their CUDA versions on NVIDIA hardware, since this allows better performance and more features.



OpenCL

OpenCL is the "open" equivalent of CUDA – which means that it is a specification and standard that is being written by a committee made up of many hardware and software companies. Everybody is free to implement the standard, and quite a few vendors have.

For each individual piece of hardware and operating system, a different implementation is required, and there are versions of OpenCL from AMD and NVIDIA as part of their video drivers. There are implementations for x86 processors from Intel and AMD as well. This means that one can install an implementation (also called runtime) and run OpenCL programs on a CPU, as well as the GPU. This works because the OpenCL abstracts all the hardware into something called a "device". The software should not know how a device is implemented, or if it is a CPU or GPU. Because OpenCL is heterogeneous, some time is needed for the implementations and runtimes to translate the OpenCL program to something that the hardware understands. This process is called "compilation" and usually its result is cached, so it only has to happen once.

In an OpenCL setup, different devices might be mixed, so you could potentially have an AMD GPU, an NVIDIA GPU, and an Intel CPU all working together.

V-Ray RT and Production V-Ray

What is V-Ray RT?

V-Ray RT is an interactive ray trace engine which monitors the changes in the host application, and updates to reflect any render setup changes. GPU rendering was first introduced in V-Ray as an option of V-Ray RT. Because of this, many people assume that V-Ray RT means GPU.

V-Ray RT is actually offered in three flavors: CPU, CUDA, and OpenCL. RT CPU in Maya, MODO and others use the same code base and match all the features of the advanced renderer. V-Ray for Max has specialized advanced render mode which matches results from the RT CPU engine almost perfectly.

Using V-Ray RT in 3ds Max

V-Ray RT is best launched in 3ds Max through an Active Shade window.

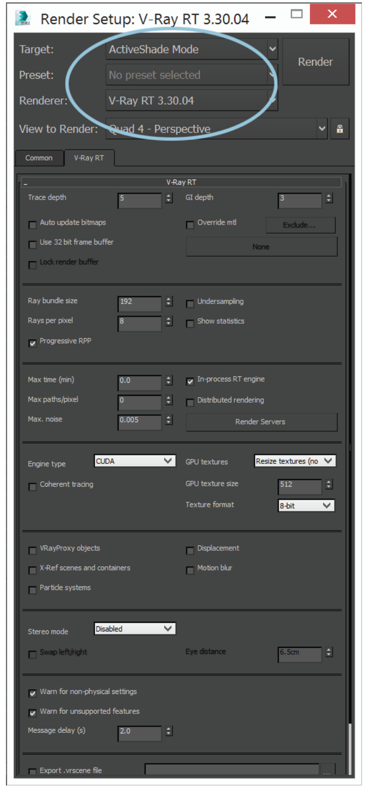
1. Open the Render dialogue.

2. At the top of the menu under Target, select ActiveShade Mode.

3. Under Renderer, select V-Ray RT.

4. Select either CUDA or OpenCL based on your GPU needs.

5. When you open an ActiveShade window, V-Ray RT will launch.



Render Setup: V-Ray RT 3.30.04

Target: ActiveShade Mode
Preset: No preset selected
Renderer: V-Ray RT 3.30.04
View to Render: Quad 4 - Perspective

V-Ray RT

Trace depth 5
GI depth 3
□ Auto update bitmaps
□ Override mtl
□ Use 32 bit frame buffer
□ Lock render buffer

Ray bundle size 192
Rays per pixel 8
□ Undersampling
□ Show statistics
☑ Progressive RPP

Max time (min) 0.0
Max paths/pixel 0
Max. noise 0.005
☑ In-process RT engine
□ Distributed rendering

Engine type CUDA
□ Coherent tracing
GPU textures Resize textures (no
GPU texture size 512
Texture format 8-bit

□ VRayProxy objects
□ X-Ref scenes and containers
□ Particle systems
□ Displacement
□ Motion blur

Stereo mode Disabled
□ Swap left/right
Eye distance 6.5cm

☑ Warn for non-physical settings
☑ Warn for unsupported features
Message delay (s) 2.0

□ Export .vrscene file

Using V-Ray RT in Maya

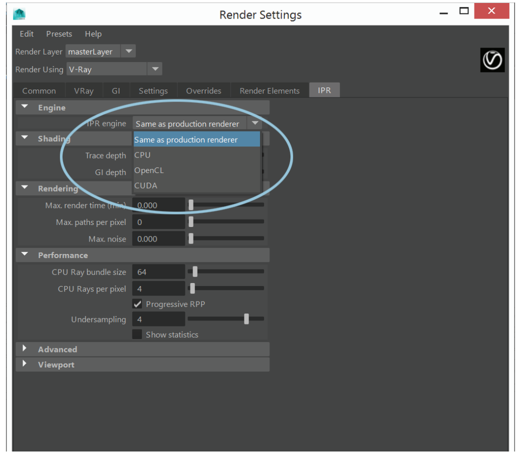
V-Ray RT is used as part of Maya's IPR rendering.

1. Open the Maya Render dialogue.

2. Go to the IPR tab.

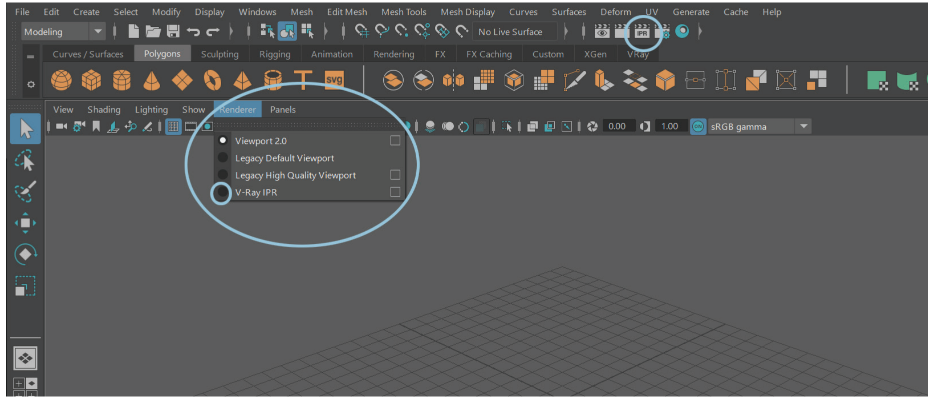
3. In there, you will be given a choice. By default, V-Ray RT will use the same hardware that you have selected in your production rendering.

4. If you have already selected CUDA or OpenCL there, you will be all set, otherwise under IPR engine, select CUDA or OpenCL based on your needs.



Render Layer: masterLayer
Render Using: V-Ray
IPR engine: Same as production renderer
Same as production renderer
CPU
OpenCL
CUDA
Trace depth
GI depth
Max. render time (min): 0.000
Max. paths per pixel: 0
Max. noise: 0.000
CPU Ray bundle size: 64
CPU Rays per pixel: 4
☑ Progressive RPP
Undersampling: 4
□ Show statistics

5. Once that is selected, you can launch V-Ray either by selecting the IPR render button, or within a viewport, by clicking on the Renderer option and selecting V-Ray IPR.



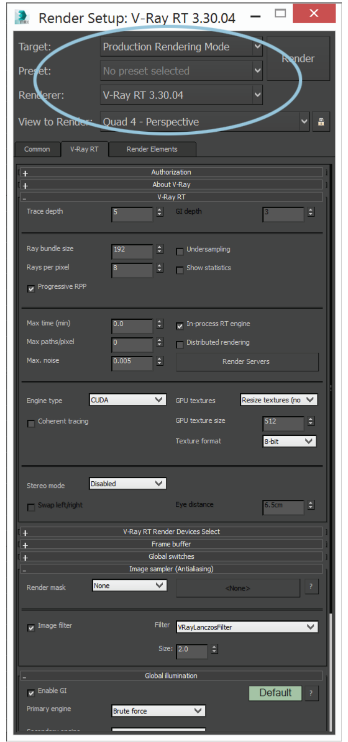
● Viewport 2.0 □
● Legacy Default Viewport
● Legacy High Quality Viewport □
● V-Ray IPR □

How to use GPU rendering for production rendering

Since GPU rendering has now evolved to a point where the software and hardware are production-ready, it has become desirable for many artists to render final production frames with V-Ray on the GPU. This also makes it easier to use GPU rendering on a render farm.

3ds Max

1. In the render dialogue, first select the Production Renderer under Target.
2. Select V-Ray RT as your renderer.
3. This will give you the same options you had on the ActiveShade dialogue.
4. Make your selections, and you're done.



Render Setup: V-Ray RT 3.30.04

Target: Production Rendering Mode
Preset: No preset selected
Renderer: V-Ray RT 3.30.04
View to Render: Quad 4 - Perspective

Trace depth 5
GI depth 3
Ray bundle size 192
☐ Undersampling
Rays per pixel 8
☐ Show statistics
☑ Progressive RPP
Max time (min) 0.0
☑ In-process RT engine
Max paths/pixel 0
☐ Distributed rendering
Max. noise 0.005

Engine type CUDA
GPU textures Resize textures (no
☐ Coherent tracing
GPU texture size 512
Texture format 8-bit
Stereo mode Disabled
☐ Swap left/right
Eye distance 6.5cm

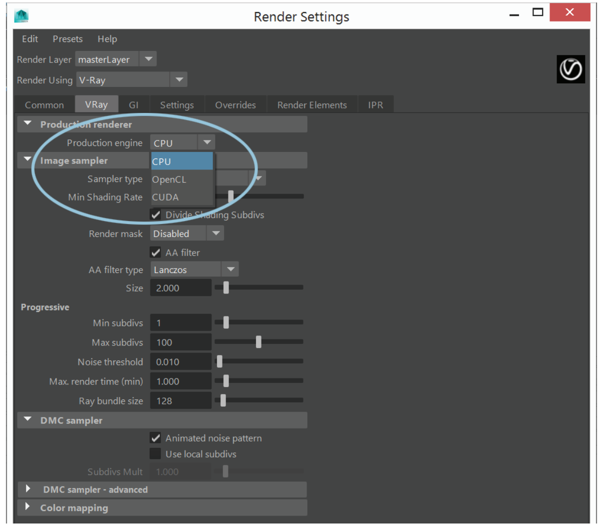
+ V-Ray RT Render Devices Select
+ Frame buffer
+ Global switches
- Image sampler (Antialiasing)

Render mask None <None>
☑ Image filter
Filter VRayLanczosFilter
Size: 2.0

☑ Enable GI
Primary engine Brute force

Maya

1. Open the Render dialogue.
2. Make sure that “Render using” is set to V-Ray.
3. At the top select CUDA or OpenCL under Production renderer.



Render Settings

Render Layer masterLayer
Render Using V-Ray

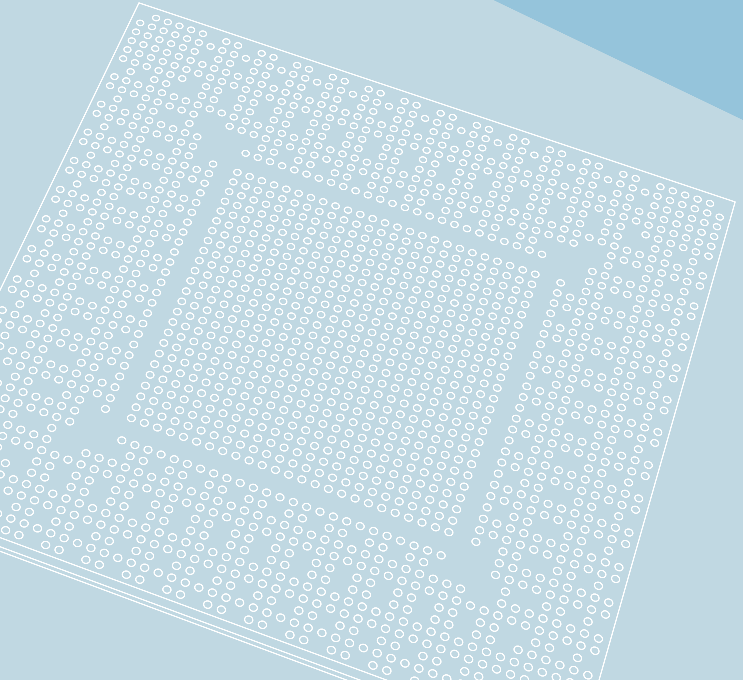
Production engine CPU
CPU
OpenCL
CUDA

Sampler type
Min Shading Rate
☑ Divide Shading Subdivs
Render mask Disabled
☑ AA filter
AA filter type Lanczos
Size 2.000

Progressive
Min subdivs 1
Max subdivs 100
Noise threshold 0.010
Max. render time (min) 1.000
Ray bundle size 128

☑ Animated noise pattern
☐ Use local subdivs
Subdivs Mult 1.000

Best practices when rendering on the GPU



Start your project with the GPU in mind

As we have learned, V-Ray on the GPU is actually a completely different renderer compared to the CPU version. While many users have the ability to jump from one to the other, it is not always best to do so as scenes get more complex. As we know, not all scenes can fit in the memory footprint of your GPU, and a few features may not be available on the GPU depending on your V-Ray version. When shaders get very complex, it is hard to find where those differences exist. For these reasons, if you plan on using the GPU, it is a good idea to start your project with the GPU in mind. Make sure that your shaders work as expected, and that you are still below your memory ceiling.

Optimizing your scene

Geometry

One of the biggest drawbacks of the GPU is the memory footprint. While the current top of the line GPUs have 24GB of memory from NVIDIA and 32GB from AMD, that still falls short of the 64GB or 128GB that many workstations have in system ram. In terms of memory, there is a lot of room for geometry. One million polygons needs approximately 220MB of memory. So 4GB of memory can hold approximately 18 million polygons. Nonetheless, as memory is still a premium for other things, it is important to optimize your geometry.

- When possible, make sure and use instances of your geometry. Only a single instance of your geometry will be held in memory, with all others being a transform.

- When using displacement, be aware of your displacement settings as the geometry is created on the fly at rendertime. This can massively increase your geometry footprint if your settings are not optimized.

Textures

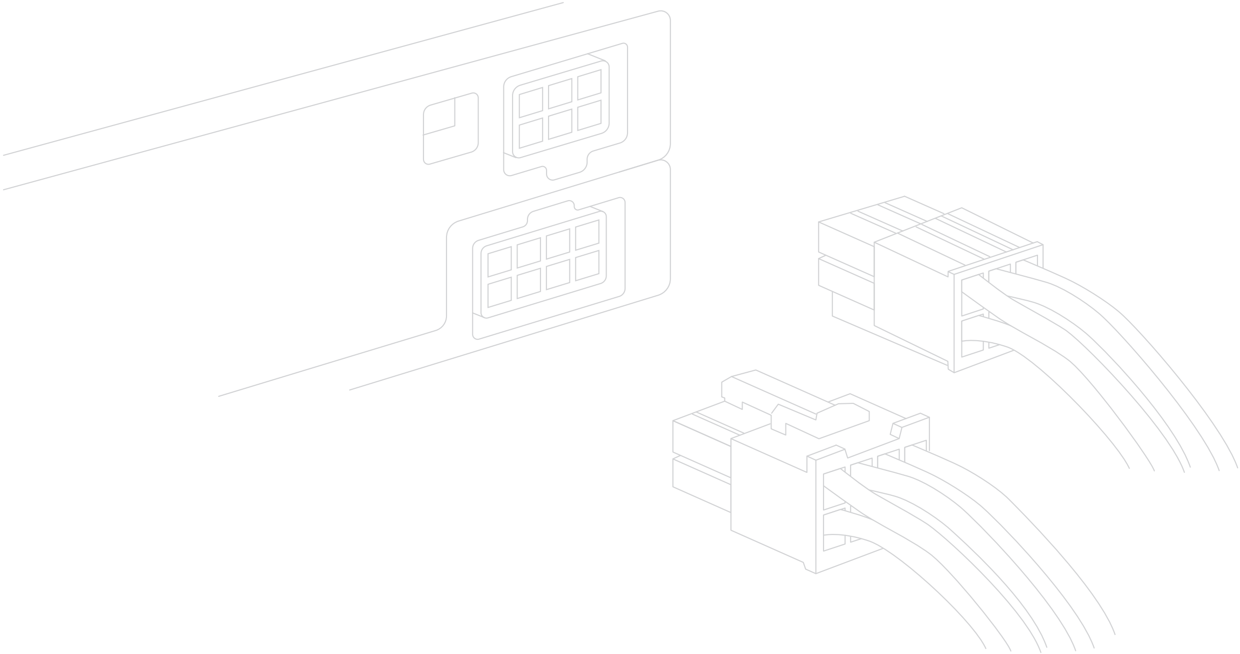
The texture process has evolved a great deal in the last few years and today this is the biggest footprint in your memory. Users tend to create textures for many different parameters of their shaders. Additionally, with programs like The Foundry's MARI, users are taking advantage of UV tiles, which means some assets may have hundreds of textures, compared to maybe 12 a few years ago. A few things should be kept in mind when creating textures and rendering on the GPU to minimize the memory footprint.

- For any texture, especially if you are using UV tiles, make sure that your texture space is maximized. Don't leave large areas of your texture blank or unused. Even if it is not used, it will occupy memory.

- Currently, mipmaps are not supported in V-Ray on the GPU, so the largest texture size will be loaded into the memory. Therefore, it is best if the user plans their texture size at multiple resolutions to find the best resolution that is needed. At GTC 2016, Chaos Group presented something under development that would allow V-Ray to automatically mipmap and pick the optimal resolution to use. This is also done under the hood so that the user will never have to worry if he or she is using the best resolution or wasting memory. Therefore, this problem will most likely go away and greatly reduce the memory footprint of textures.

- Only use 8-bit textures unless absolutely necessary. 16-bit textures use twice as much memory as 8-bit.

- Lastly, while not necessarily ideal for final images, if you are doing RT renders for look development or lighting tests, you can use the option in RT to automatically resize all the textures to a smaller resolution. It should be noted that V-Ray is smart enough to not resize textures or color depth of images used for image-based lighting as this would drastically effect the look.



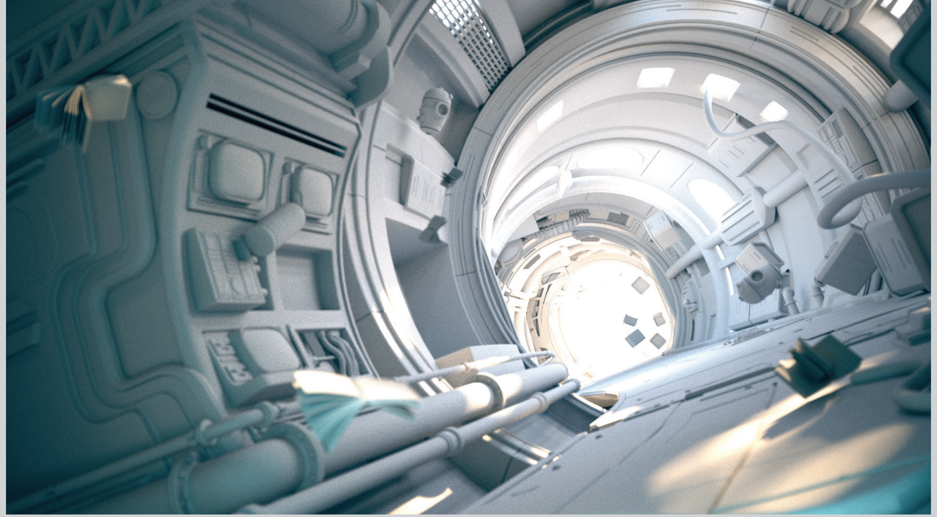
Other useful tips

Global Illumination using V-Ray on the GPU

V-Ray RT supports a few global illumination solutions, the first being Brute Force. By the nature of pathtracing on the GPU, Brute Force is generally very fast. However, those speeds can be much slower if you are dealing with an interior scene that requires a lot of light bounces. Additionally, a lot more samples need to be used in order get a clean render, and in some cases this method can be slower than traditional ways of doing global illumination on the CPU. ***{fig 13}***

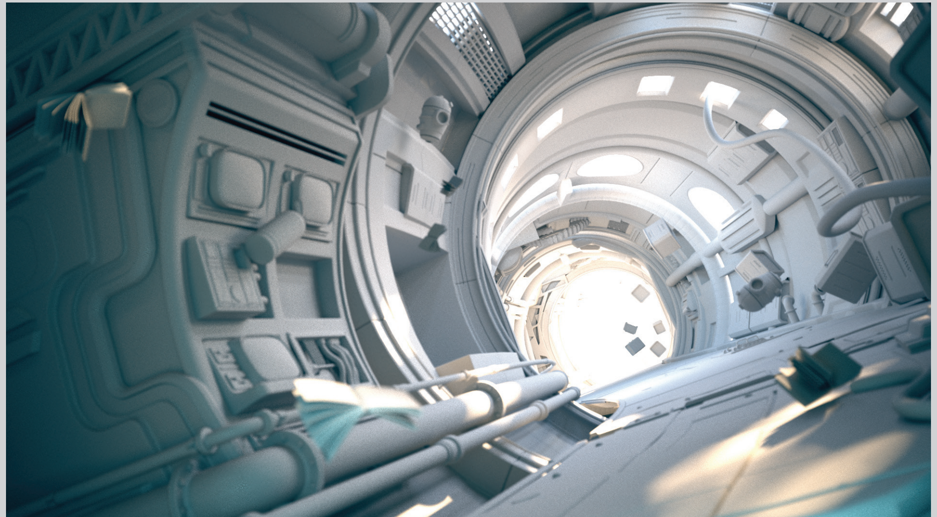
For this reason, V-Ray on the GPU supports Light Cache. This is an excellent way of getting a lot of illumination and bounces of light for interior scenes. ***{fig 14}*** In order to do this, under the Global Illumination section of the render dialogue, select Brute Force for the primary rays and Light Cache for the secondary rays. When the render launches, a prepass will be done for the Light Cache using the CPU. This pass is fairly quick, depending on your Light Cache settings. This data is stored in memory and passed on to the GPU and will give you a great globally illuminated solution which will be very clean.

However, it should be noted that when using V-Ray RT to do interactive rendering, such as Active Shade in 3dsmax or IPR in Maya, it will resort to the Brute Force method. This is because it would have the redo the Light Cache for every change, which can slow down interactivity.



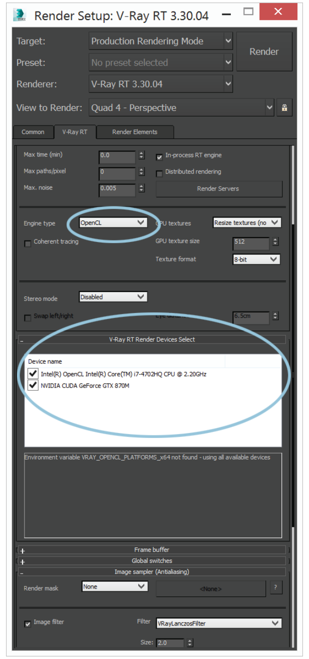
V-Ray on GPU (CUDA) ***Brute Force / Brute Force*** *render: 50 minutes 24 seconds*
{fig 13}

V-Ray on GPU (CUDA) ***Light Cache / Brute Force*** *render: 22 minutes 54 seconds*
{fig 14}



Selecting the available devices with CUDA or OpenCL

When you are rendering with V-Ray on the GPU you can select which devices are available for you to render on. If you select CUDA, a new section in the render dialogue will appear called “V-Ray RT Render Device Select.” It will list all the supported GPUs that are available to you. They will be selected by default. If you wish to exclude some of them, you can deselect them. When you select OpenCL, it will list all the compatible GPUs and CPUs.



Render Setup: V-Ray RT 3.30.04

Target: Production Rendering Mode
Preset: No preset selected
Renderer: V-Ray RT 3.30.04
View to Render: Quad 4 - Perspective

Max time (min) 0.0
Max paths/pixel 0
Max. noise 0.005
☑ In-process RT engine
□ Distributed rendering

Engine type OpenCL
□ Coherent tracing
GPU textures Resize textures (no
GPU texture size 512
Texture format 8-bit

Stereo mode Disabled
□ Swap left/right
Eye distance 6.5cm

V-Ray RT Render Devices Select

Device name
☑ Intel(R) OpenCL Intel(R) Core(TM) i7-4702HQ CPU @ 2.20GHz
☑ NVIDIA CUDA GeForce GTX 870M

Environment variable VRAY_OPENCL_PLATFORMS_x64 not found - using all available devices

Render mask None <None>
☑ Image filter
Filter VRayLanczosFilter
Size: 2.0

Dealing with a sluggish interface while rendering on the GPU

The GPU is not as good as the CPU at resource sharing. An operating system can run thousands of processes on the CPU at the same time. The OS will share the CPU time and memory between all these processes – even if one is very resource hungry.

If you run a very well-optimized GPU program, such as V-Ray on the GPU, it will utilize the GPU to its maximum potential. However, it is quite possible that the GPU will be so busy doing calculations that it will have no time for other resources, such as the OS, which needs the GPU to draw graphics on the screen. Therefore, the OS user interface may become sluggish if all the GPUs in a system are being used for V-Ray GPU rendering. You have a few options to deal with this.

- The best way is to reserve one of your GPUs to drive the interface and not have it used for rendering. A good choice for this would be to have a smaller cheaper card that is not used for rendering. You can then deselect it from list of available devices as stated above.
- If you don’t have a second card, or would rather not exclude one of your available devices, you can try decreasing the size of the chunks of work that are being given to the GPU. This will give it more chances to complete calculations needed for other apps. In V-Ray on the GPU this is done with the “Ray Bundle Size” and “Rays Per Pixel” parameters. Increasing them (to something like 256/32) utilizes more of the GPUs, giving final frames more quickly–but also uses all the GPU time. Reducing them (to something like 128/1) increases the interactivity of the render and the OS UI (if only one GPU is used) but may underutilize the GPU.

Optimizing GPU rendering for Distributed Rendering

GPU rendering supports distributed rendering. However, due to the speed of the rendering and the nature of progressive rendering, this can cause a great deal of network traffic. Increasing "Bundle Size" / "Rays Per Pixel" increases the size of the workload chunks that are being given to the DR nodes and helps reducing the communication between the client and the servers (you can try something like 192/32 or 256/64).

Checking V-Ray log and docs for supported features

RT GPU has a concise and very useful FAQ and Supported Feature list. These answer most common questions, and can save a lot of time and frustration. We make sure they're kept up-to-date, so they can be trusted. You can find them at *docs.chaosgroup.com*.

The V-Ray log also has important messages and should be checked if something goes wrong. Usually, warnings are printed in yellow and errors are red, so they are easy to see.

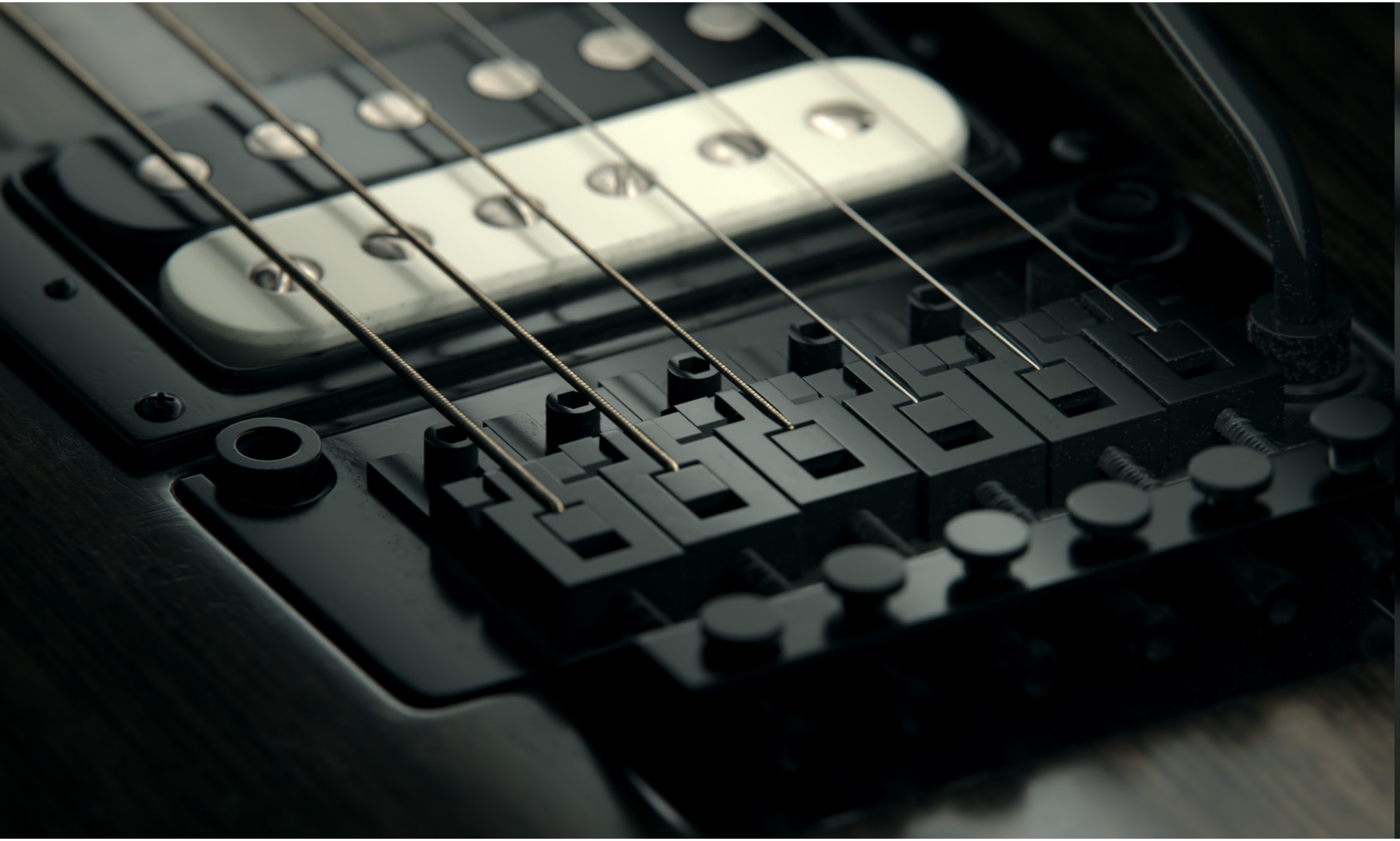
Conclusion

To the delight of professional artists, GPUs are becoming more viable after years of CPU dominance. Recent trends suggest that this will only increase in the future. However, even with recent advancements, GPUs still aren't the best solution for every situation. And for price-conscious or cloud-leaning firms, the answer might still be a CPU renderer most of the time.

Technologies like V-Ray that allow users to do production-quality rendering on either system will continue to be valuable since they offer flexibility. Certain tasks can still be accomplished faster on CPUs, which will have to be conquered if GPUs want to see wider use in mainstream rendering.

In the coming year, Chaos Group plans to make several GPU development announcements for V-Ray and V-Ray RT. We look forward to sharing those soon.

GPU render gallery - artwork by Dabarti Studio



Glossary

API (Application Programming Interface) – A set of tools used for creating software, and ensuring it interacts with other software and hardware properly.

BrookGPU – A compiler and runtime designed by Stanford University to enabled general purpose computing on GPUs.

Cache – A small amount of memory built into a processor, and used to temporarily store data before it's processed.

Clock rate – See *Frequency.*

Core – The individual processing units on a multi-core processor.

CrossFire – AMD's interface for linking multiple graphics cards in order to boost performance.

CUDA – NVIDIA's platform and API for programming on its GPGPUs

DirectX – Microsoft's realtime graphics API.

Die – The square of silicon which makes up a chip, and contains the transistors and the integrated circuit.

Distributed Rendering (DR) – A method of rendering a scene across multiple computers on a network.

Double precision – A computer number format that occupies 8 bytes (64 bits) of memory. The format is common in CPUs, but only tends to feature in professional graphics cards.

FPGA – A processor designed to be configured by a customer after manufacturing.

Frequency – The speed of a processor in clock cycles per second, usually represented in hertz (Hz).

GDDR (Graphics DDR SDRAM) – A type of memory developed for GPUs.

GPGPU (General Purpose Graphics Processing Unit) – A GPU which is capable of handling a broader range of computing tasks, as well as graphics-based ones.

GPU (Graphics Processing Unit) – A chip responsible for speeding up the delivery of images to a screen.

HBM (High Bandwidth Memory) – An memory interface created by AMD and Hynix. It stacks memory three-dimensionally so the GPU can access it more quickly.

Latency – The number of clock cycles it takes a processor to deal with a particular task.

Leakage – A quantum phenomenon in which mobile charge carriers escape through the insulated areas or junctions on a chip.

OpenCL – An open-source framework for writing programs which run across CPUs, GPUs and other processors.

OpenGL – An open-source realtime graphics API initially developed by Silicon Graphics.

OpenRL – A ray tracing API developed by Caustic Professional.

Mipmapping – The process of reducing the quality of an image when necessary to reduce memory usage.

Nanometer (nm) – A billionth of a meter, and the base unit of measurement in the microarchitecture of CPUs and GPUs. The smaller the number, the more components can be added to a processor.

NVLink – Nvidia's communication protocol for exchanging data between GPU and GPU, or GPU and CPU.

Parallelism – The distribution of a task among multiple processor cores, processors or computers, so it can be processed more quickly.

PCI (Peripheral Component Interconnect) – A computer expansion bus used to add graphics cards to mainboards. Its most recent version, PCI Express, is the most common.

SLI (Scalable Link Interface) - NVIDIA's interface for linking multiple graphics cards in order to boost performance.

Throughput – The number of tasks a processor can deal with at any one time.

Transistors – The tiny electronic switches which form a fundamental part of a GPU or CPU. A higher transistor count means a processor can potentially deal with more data at the same time.

Vendor agnostic – Software which will run on any compatible hardware, regardless of manufacturer.

Voltage – The difference in electrical energy between two parts of a circuit. The bigger the difference in energy, the bigger the voltage.

v·ray

by CHAOSGROUP