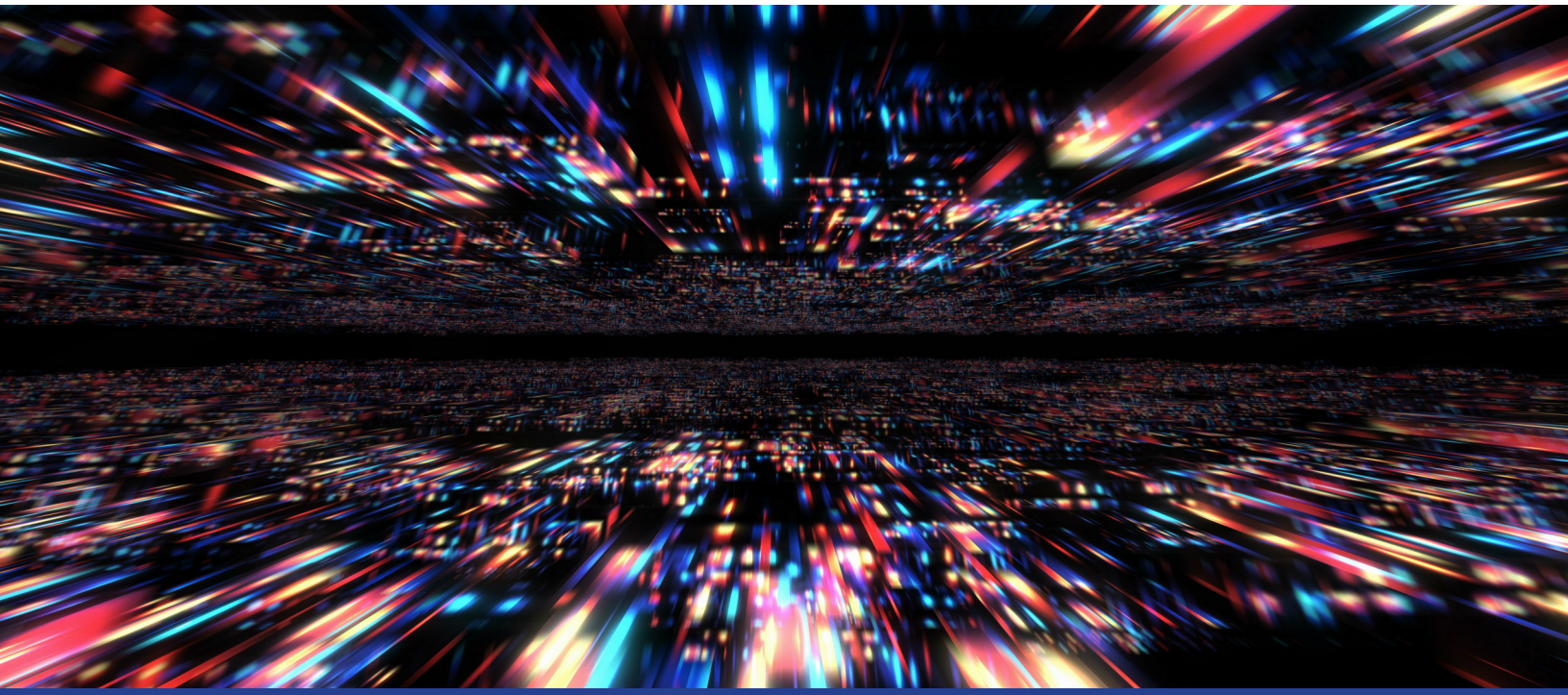


TURBO-CHARGED OPEN SHADING LANGUAGE* ON INTEL® XEON® PROCESSORS WITH INTEL® ADVANCED VECTOR EXTENSIONS 512

Up to 2x Faster Full Renders Speed Digital Content Creation

Steena Monteiro, Software Engineer, and Alex M. Wells, Principal Engineer, Intel Corporation

Oscar*-winning Open Shading Language* (OSL*)[1] is the *de facto* open-source standard for digital content creation. OSL has been adopted industry-wide, used in renderers such as Pixar's RenderMan* and Sony ImageWorks' Arnold*, and in more than 100 movies.[2]

Intel has been leading the rearchitecture of OSL to add single instruction multiple data (SIMD) to leverage **Intel® Advanced Vector Extensions 512 (Intel® AVX-512)** in modern Intel® processors. SIMD OSL uses single program multiple data (SPMD) with existing OSL shaders and OpenMP* explicit vectorization of OSL library functions. This effort can be broadly summarized in two steps:

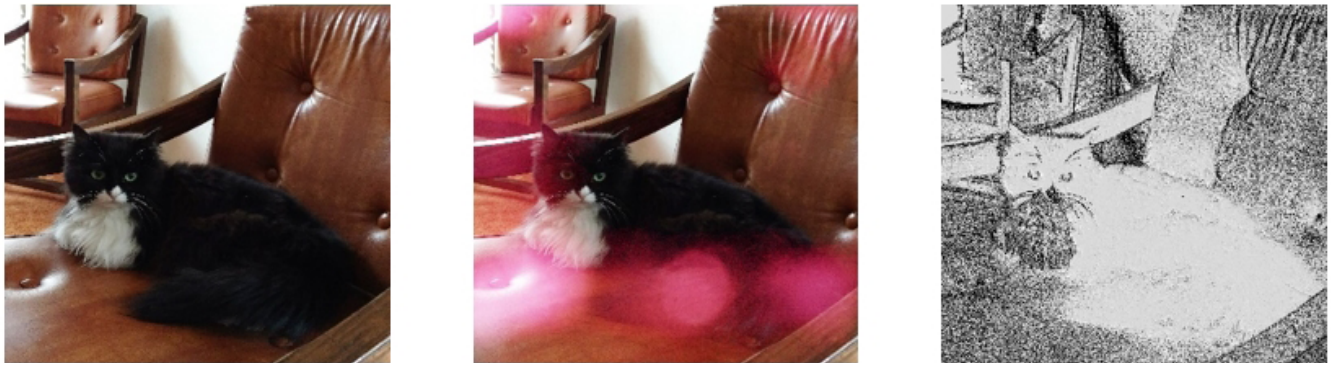
1. **Introducing vector LLVM IR generation** for just-in-time (JIT) compilation during render time optimization
2. **Adding a batched interface** to the OSL runtime

Since its start at SIGGRAPH 2016, SIMD OSL has been improved to natively support AVX*, AVX2*, and AVX-512* and include enhanced library features, debugging support, and an extensive test framework. SIMD OSL enables parallel execution of an entire shading network on Intel SIMD hardware.[4] It dynamically schedules concurrent operations over 8 or 16 data points in a single CPU instruction based on the CPU capabilities. According to Pixar, the company's RenderMan 22.5 now contains "SIMD hardware-accelerated OSL–up to 2x faster full renders, and 15% average speedups using **Intel® Xeon® Scalable processors** with Intel Advanced Vector Extensions."[3,5]

This article explores Intel's efforts in leading the rearchitecture of OSL to leverage the power of Intel AVX-512 on Intel SIMD hardware. We specifically discuss software engineering techniques used in SIMD OSL including strategic memory layout for OSL datatypes, masking for divergent control flows, and addition of an LLVM backend for vector code generation.

Shading and Its Role in Rendering Software

Shading in physically-based renderers implies providing surface description for objects in a 3D scene. Surface descriptions include color values, lighting values (specular, diffused, spot), and textures such as metal, ceramic, and marble (**Figure 1**). Shading in large scenes is done using several individual shader nodes, where each node represents a specific shading behavior. Individual shaders can be connected through directed acyclic graphs to procedurally create complex shading effects. Production shaders can grow to several hundreds and thousands of shaders, representing a multitude of shading behaviors. Renderers in production can spend up to 80% of their time shading via OSL. Due to the complexity of shading networks, shading in a render consumes millions of CPU-hours on render farms, both on-site and in the cloud.



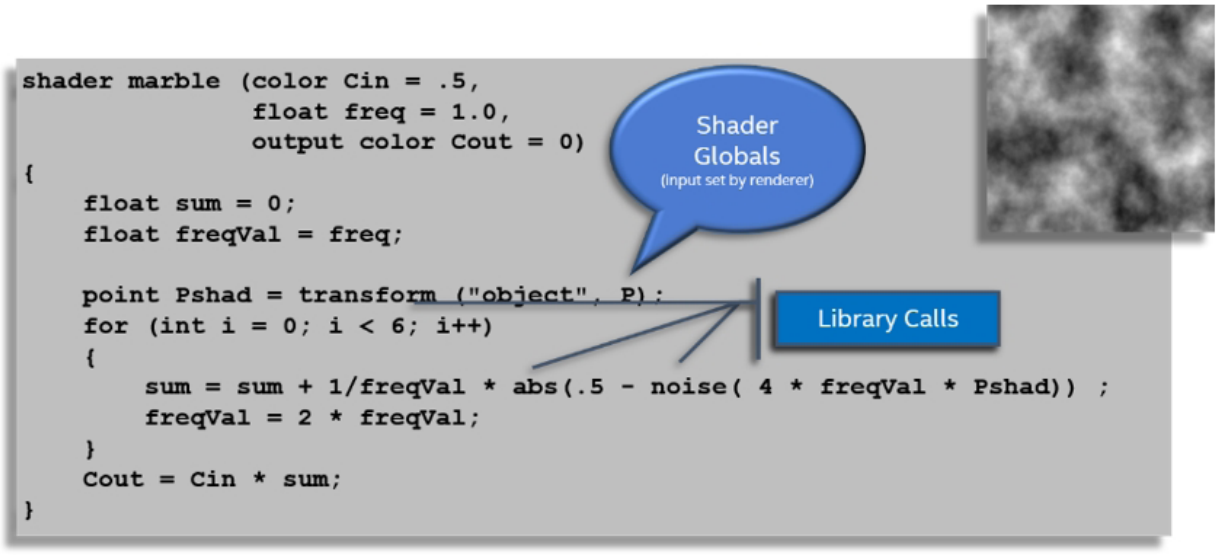
1 **Example of lighting and texture**

Shaders in shading networks can be written in C++. However, using C++ presents a host of challenges. These C++ shaders lack relevant shading information such as values of input parameters and geometry of the scene being shaded at compile time. The shaders don't know the mode of shading required and remain oblivious about the state of the surrounding shading network. Writing, maintaining, and optimizing performance of shaders written in C++ is challenging—primarily, because these shaders lack portability and necessitate tests with complex and nested control flows (branchy testing). Artists who design shading networks need extreme expertise in optimizing C++ to achieve high performance.

OSL, designed by Sony Pictures Imageworks[6], makes writing performance-compliant shading networks a little easier. Structured in C style, OSL is a domain-specific language designed for writing shaders. Designed primarily for physically-based rendering, OSL is restricted to shading and doesn't include raytracing, sampling, integrations, and tight loops (which reside in the renderer). Under the hood, OSL maximizes performance by JIT-ing machine code with extensive runtime optimization and yields shading networks with lazy evaluations.

Open Shading Language

Shaders in OSL are programs with inputs and outputs that perform a specific task when rendering a scene[6]. **Figure 2** shows a simple OSL shader representing a marble texture.[7] The important elements in this shader are the shader global P and the OSL library functions `abs()` and `noise()`. Shader globals are variables such as position, surface normals, and ray directions, provided by a renderer, that are consumed by the shader.



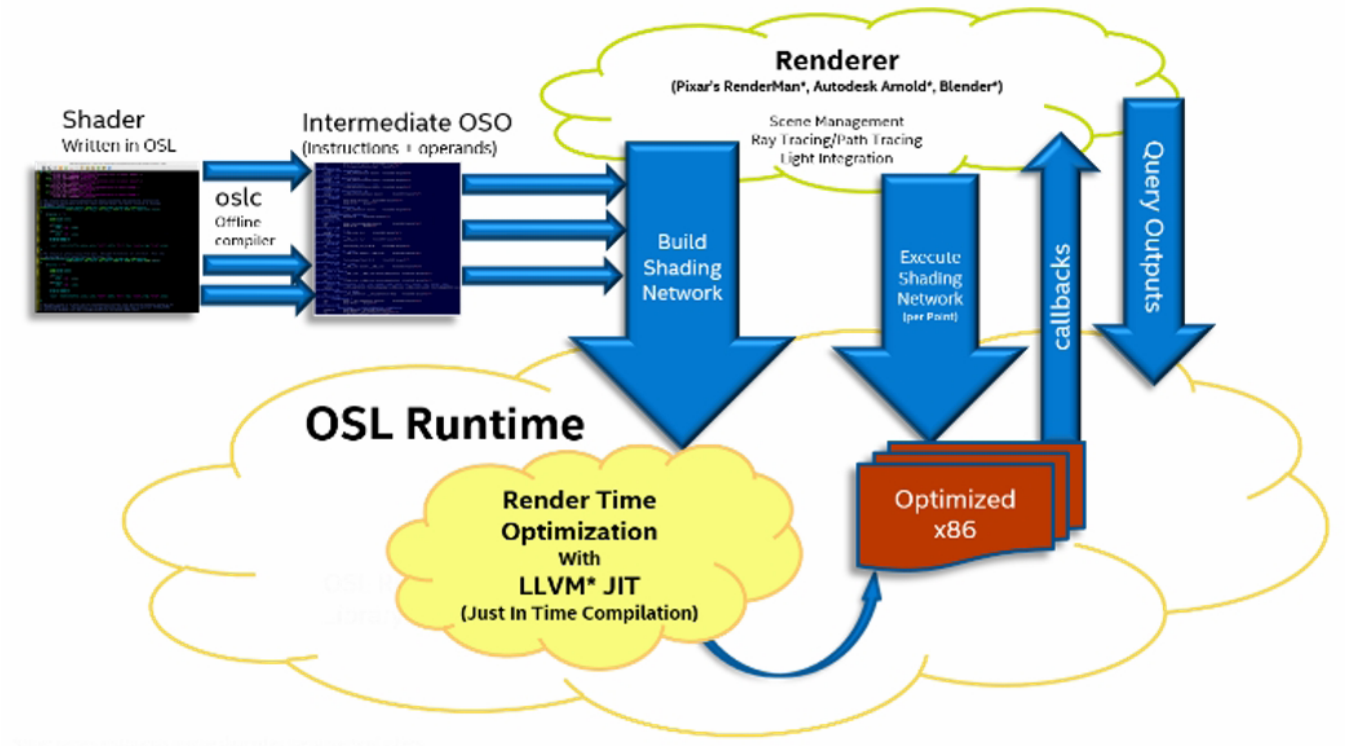
2 **A shader written in OSL (`marble.osl`[7])**

Benefit of OSL Shaders Over C++ Shaders

Shaders written in OSL are compiled by the OSLC (OSL compiler) into intermediate oso files that contain a mix of operands and instructions representing shader operations. Multiple shaders are compiled to build a large shading network. The OSL runtime employs LLVM to generate an intermediate representation (IR) of the shader, optimize it, and finally produce optimized x86 code, as demonstrated in **Figure 3**. Because of render time optimization, production scenes have benefitted from an orders-of-magnitude reduction in the number of operations, symbols, and empty shader instances. Scenes have demonstrated a:

- **99% reduction in operations** (from 280 million to 2.68 million operations)
- **98.8% reduction in symbols** (from 161 million to 1.9 million symbols)
- **63% optimization** by eliminating empty shader instances[8]

Because of its ability to leverage LLVM and JIT for render-time optimization, OSL can outperform precompiled C++ shaders.



3 **OSL framework from OSL shaders to optimized x86 code**

Introducing SIMD OSL

Even with all its advantages over C++ shaders, OSL in its original form lacks opportunities to leverage newer Intel SIMD instructions. Its block vectorization from using Intel® Streaming SIMD Extensions (Intel® SSE) uses only four lanes and offers limited support for complex noise, math, string, and texturing functions, among others. SIMD OSL uses the SPMD model to create a batched interface (process multiple points in the shading network) to the renderer. Features of SIMD OSL include:

- **Retaining OSL language specifications:** SIMD OSL does not change the way users interface with the OSL library (i.e., the original OSL shaders remain unchanged).
- **A new batched interface** enables the renderer to process batches of points from the shading network. Even so, it retains the original single-point interface, where a single point from a shading network is processed by the renderer.
- **Generating SIMD code via a wide backend:** SIMD OSL uses LLVM* vector data types `<16 * float>` for datatypes in vectorized intermediate representation (IR).
- **A new wide library:** Through its rearchitecture, SIMD OSL provides a wide interface to OSL functions (function families such as math, noise, etc.) using OpenMP-explicit vectorization.
- **Creating a comprehensive test framework** to test OSL library functions over combinations of uniform, varying, and constant operands.

Architecture of SIMD OSL

The rearchitecture of SIMD OSL introduces three major structural changes in existing single-point OSL

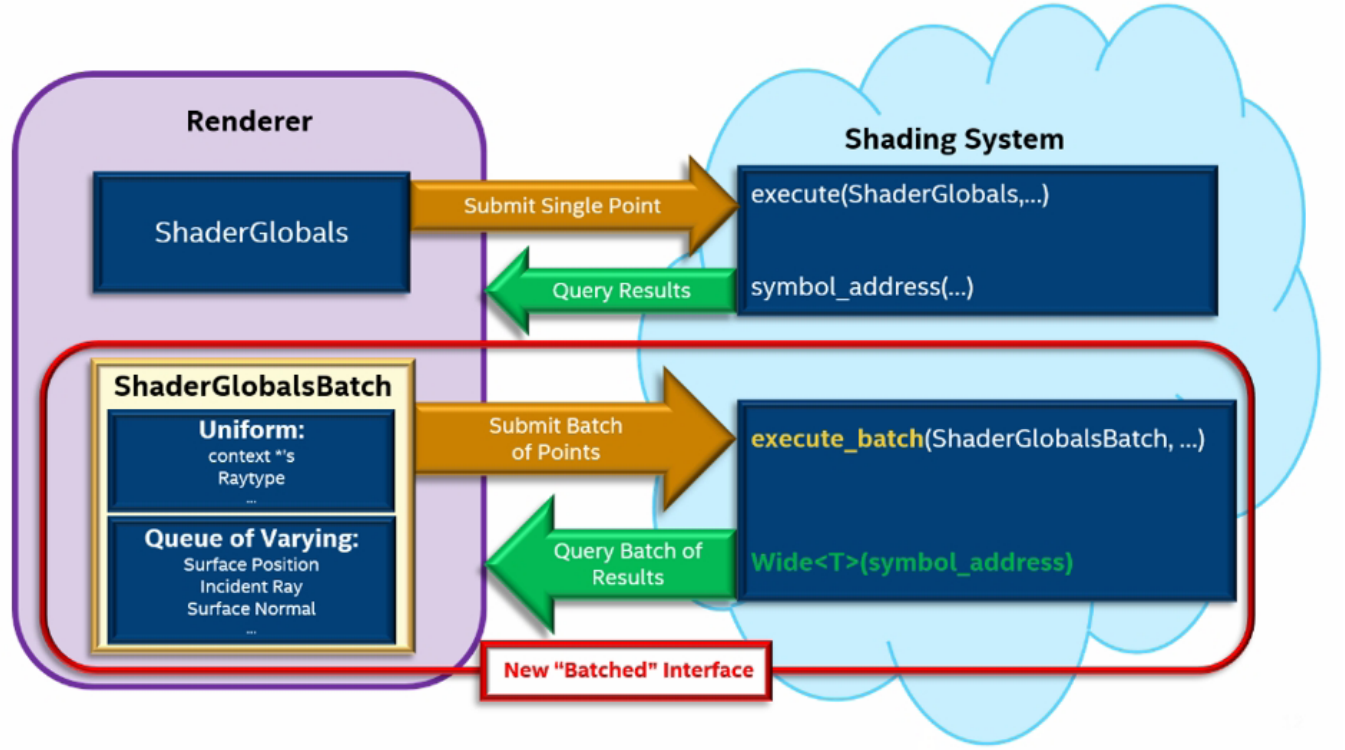
- **Providing** a batched interface (**Figure 4**)
- **Adding** wide accessors to represent and access varying data types
- **Handling** divergence among values across different SIMD lanes

Batched Interface in SIMD OSL

One of the changes in SIMD OSL is storing shader globals differently in the batched subsystem, depending on whether they are uniform or varying. The ability to use batches of points means the renderer can submit sets of points to the shading system, and the shading system can, in turn, query a set of results from the queue of varying globals in the renderer. For a renderer to be able to use SIMD OSL, it's important that it support the new wide interfaces while also accommodating for wide callbacks.

Datatypes in SIMD OSL

All variables are considered uniform unless they can be proven to be varying. A varying variable is one whose dependence can be traced to shader globals, which are known to be varying. For instance, shade globals like surface position, incident ray, and surface normal are always varying.



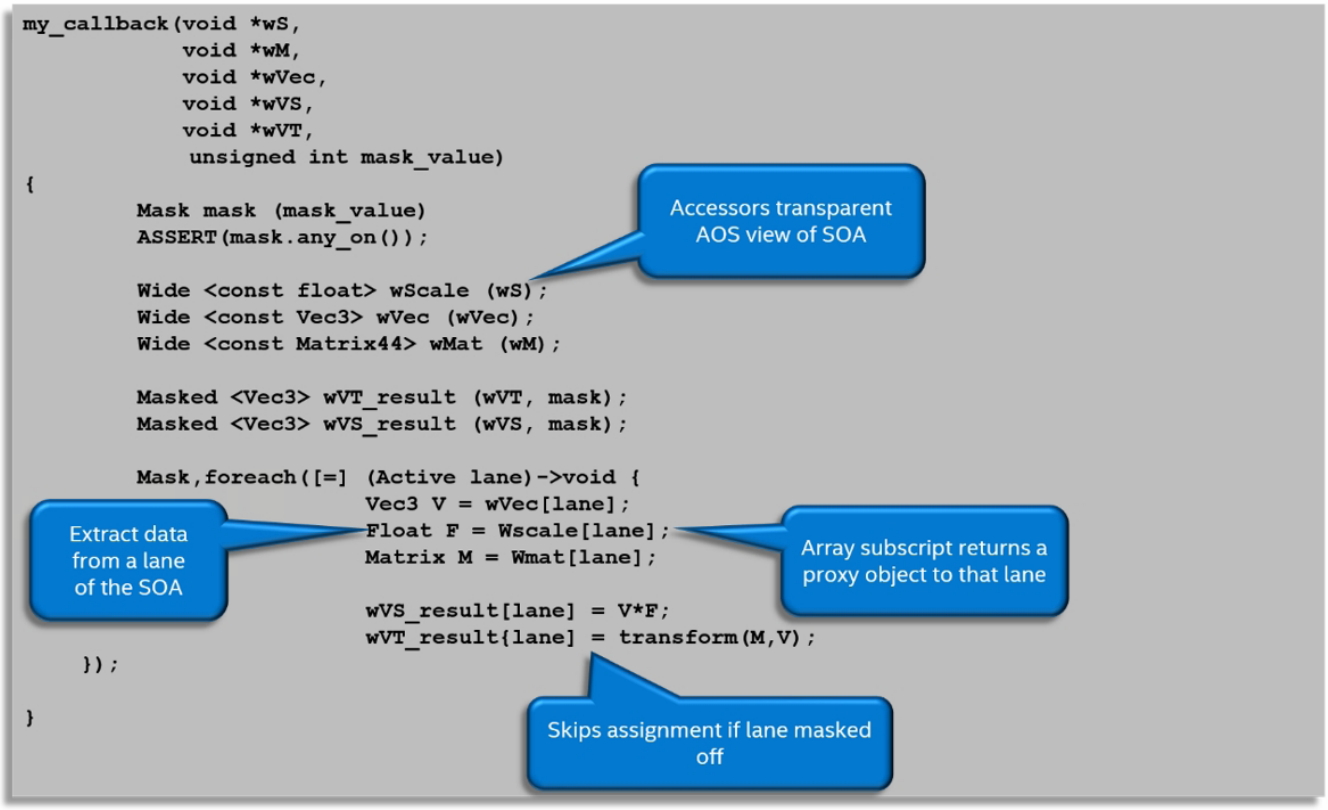
4 **Batched interface in SIMD OSL**

OSL contains a few aggregate datatypes—such as vector, color, point, and matrix—that are tuples. When used in arrays, data is stored in an array of structures (AOS) format. The overhead of looping through AOS values for storing into vector registers hinders vectorization. The Intel SIMD data layout template (SDLT)[9] uses containers with SIMD-friendly internal memory layout. Traditionally, datatypes of this form are represented as arrays of values in memory. SDLT containers provider accessor objects to import and export primitive datatypes between underlying memory layout and the original representation of the object.[9] Inspired by the SDLT library, the SIMD OSL Library provides wide accessors to support varying datatypes and callbacks through the renderer. Wide accessors resemble arrays of the datatype across each SIMD lane (**Figure 5**) and abstract the underlying structure of arrays (SOA) layout. Under the hood, masked accessors will skip inactive data lanes via a mask.

Masking Algorithm in SIMD OSL to Track Nested Control Flows

SIMD OSL uses a masking strategy to keep track of lanes that diverge at an if-condition. A mask tracks points that will execute on either path. Both code paths are then executed with the mask activated for each appropriate lane in each branch. However, this technique becomes complicated with nested control

flows, because performance would be bottlenecked by tracking points and their masks across various lanes along different code paths. To overcome this and still track the right points executing on the right control flow path, SIMD OSL uses a stack to keep track of masks at each conditional statement (**Figure 6**).

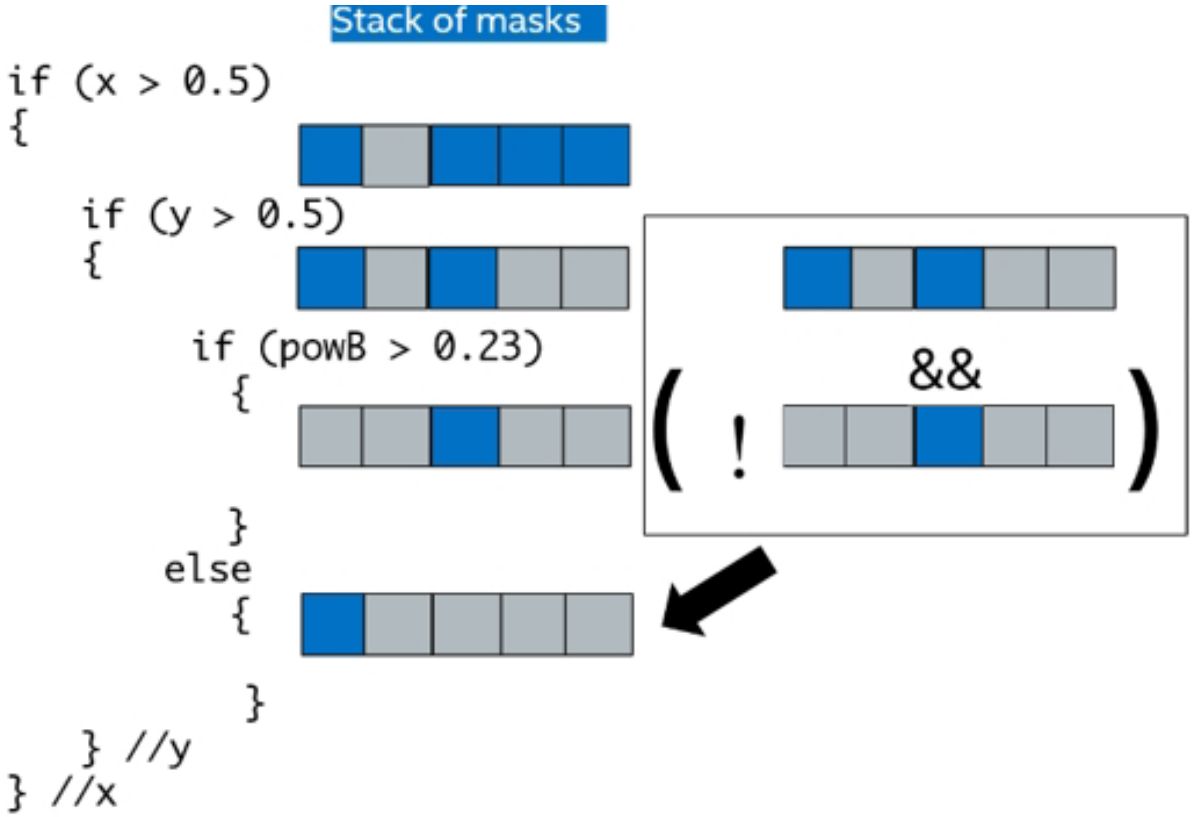


5 **Accessing varying data in SIMD OSL**

SIMD OSL's LLVM Backend

SIMD OSL uses LLVM to JIT target-specific code. For precompiled library functions, SIMD OSL generates different shared libraries for each supported platform—AVX, AVX2, and AVX-512. The appropriate library is loaded at run time to link addresses of each OSL precompiled library function with the JIT code.

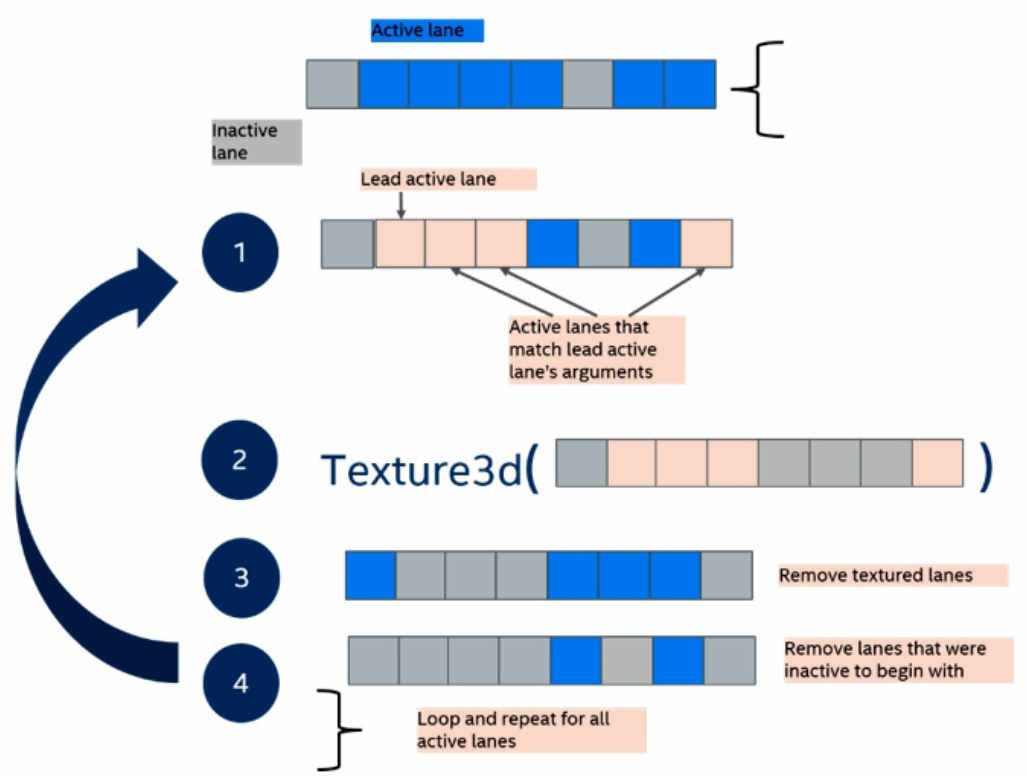
OSL, in its original form, contains an LLVM backend to support all families of functions. Intel rearchitected this backend to support our wide datatypes and masking controls when dealing with varying operands and control flows. In OSL's `andor` function, we use a four-step process to add SIMD support:



6 **Stack of masks to track divergence in control flow in SIMD OSL**

1. **Check** if the operand and result are uniform
2. **Load** operand values while accommodating for their type (uniform or varying)
3. **Emit** IR to either perform the operation or to call the appropriate precompiled library function
4. **Widen** result prior to storage if the result is varying

Because the `andor` function is simple, with only one operand that's required to be uniform, its support in the SIMD LLVM backend is uncomplicated. However, functions such as `texture3d()`, which contain multiple operands, require more complex LLVM backend support. The `texture3d()` function performs a 3D lookup of a volume texture, indexed by 3D coordinate `p`.[6] When we call `texture3d()`, the function expects a set of options, some of which are varying (`blur`, `width`, and the texture coordinate `p`), while some are expected to only be uniform (e.g., `wrap`). We first scan for lanes with the same argument settings so that they can execute together. The remaining lanes that don't match are turned off. The control flow is described in **Figure 7**.



7 **LLVM lane masking and filtering in `texture3d()` in SIMD OSL**

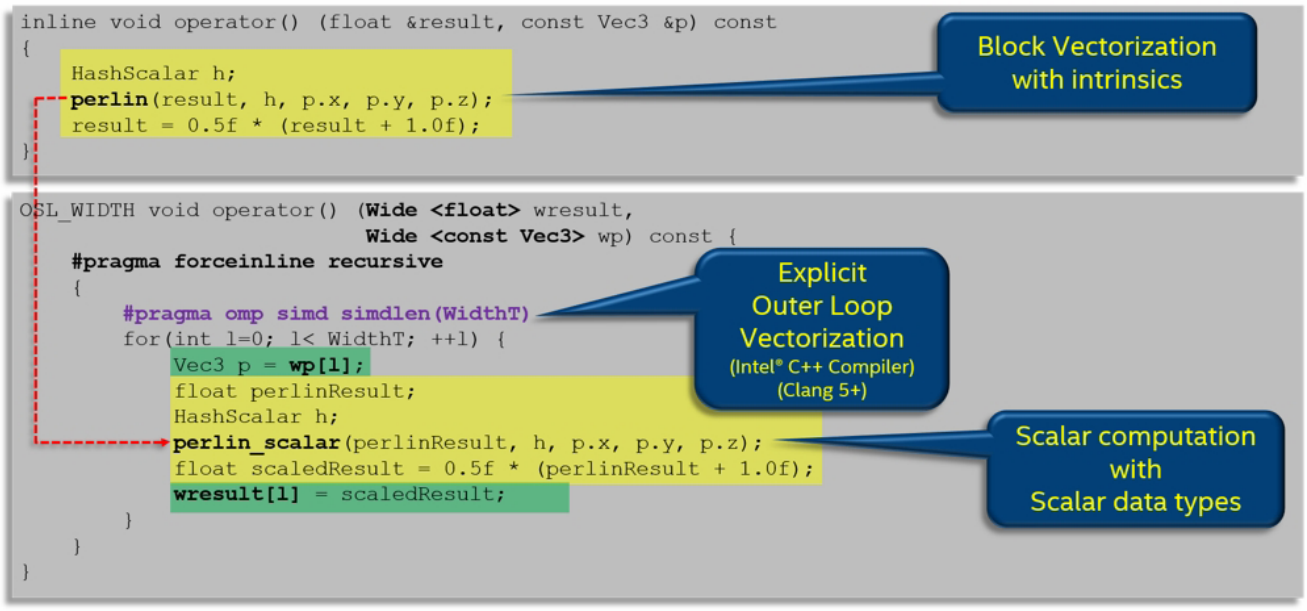
LLVM in SIMD OSL Loops and Control Flow

In OSL library functions, we can detect active lanes and implement a body of function calls in different ways, depending on lane utilization. For instance, in the Perlin* noise function (described below in greater detail), we use the default block vectorized Perlin noise implementation when the number of active lanes is less than four. We can also process each lane individually and vectorize it. In summary, leveraging LLVM for the SIMD OSL backend gives us the ability to change directions and vary the scope of vectorization, both inside and across lanes.

Perlin* Noise in SIMD OSL

The original version of Perlin noise in non-SIMD OSL is optimized to perform block vectorization within the algorithm using Intel SSE intrinsics (**Figure 8**). To enable outer loop vectorization while retaining performance of the original Perlin noise, we eliminate SSE intrinsics and revert to the original C++ version of the algorithm by creating a `perlin_scalar` helper. We then leverage the wide accessors to import and export the data type out of the underlying SOA data layout. The outer loop vectorization is implemented by OpenMP #pragma and specifying the SIMD width. Inside the loop, we export the data for the current lane, perform scalar computation via `perlin_scalar`, and then import results for the lane. Note that the

actual `perlin_scalar` computation is oblivious to our data layout and our outer SIMD loop. Once this is all inlined, the compiler can produce ideal code for multiple target ISAs (SSE2, AVX, AVX2, AVX-512, etc.). To ensure proper inlining on the **Intel® C++ Compiler**, we judiciously use `#pragma forceinline recursive`.



8 **Enabling SIMD in Perlin noise in OSL**

Performance of SIMD OSL on Intel® Xeon® Processors

We evaluate performance of OSL benchmarks and individual OSL shaders on two Intel® Xeon® processors:

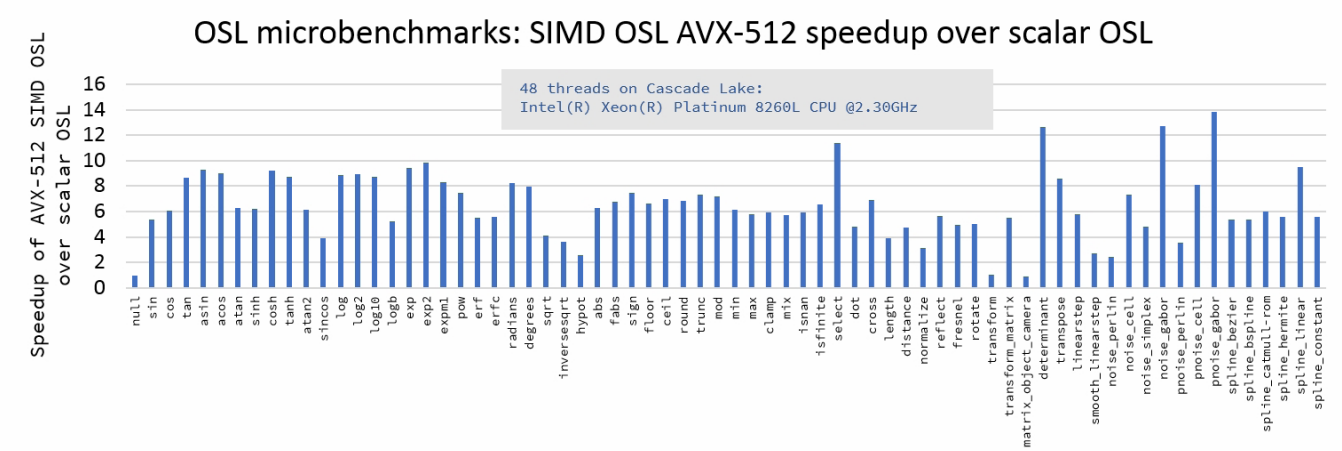
- A two-socket, 40-core Intel Xeon Gold 6248 processor @2.50GHz
- A two-socket, 48-core Intel Xeon Platinum 8260L processor @2.30GHz with hyperthreading turned off

OSL is run via testshade, a test harness that exercises shaders and shader groups. Testshade can also be viewed as a substitute for a shading module in a renderer. Testshade can be executed as single- or multi-threaded. We showcase the superior performance of SIMD OSL via:

- **A suite of microbenchmarks** comprising important OSL functions
- **A set of individual shaders** that represent different textures and patterns

Performance of SIMD OSL Microbenchmarks on Intel® Xeon® Platinum 8260L Processor

The OSL microbenchmark suite includes individual OSL functions from OSL function families—string, noise, math, trigonometry, logical operations, binary operations, spline, and others. **Figure 9** shows the speedup of AVX-512 SIMD OSL over scalar OSL from trials using 48 threads with batch size of 16.

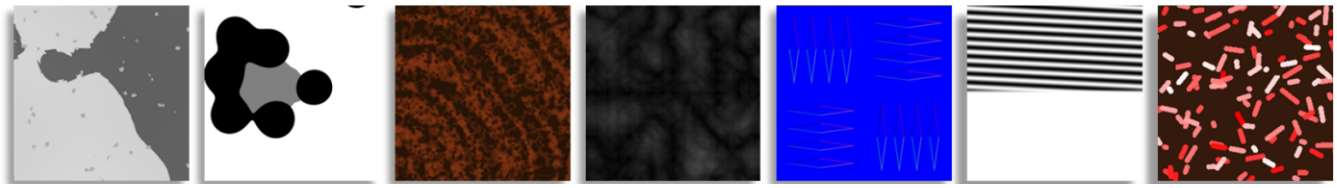


9 **SIMD OSL microbenchmarks on an Intel Xeon Platinum 8260L processor @2.30GHz**

The speedup across the 67 functions in the microbenchmark averages 7x, with a maximum speedup of 13.8x (Gabor noise).

Speedup of SIMD OSL over Single-Point Scalar OSL on Individual Shaders on Intel® Xeon® Gold 6248 Processor

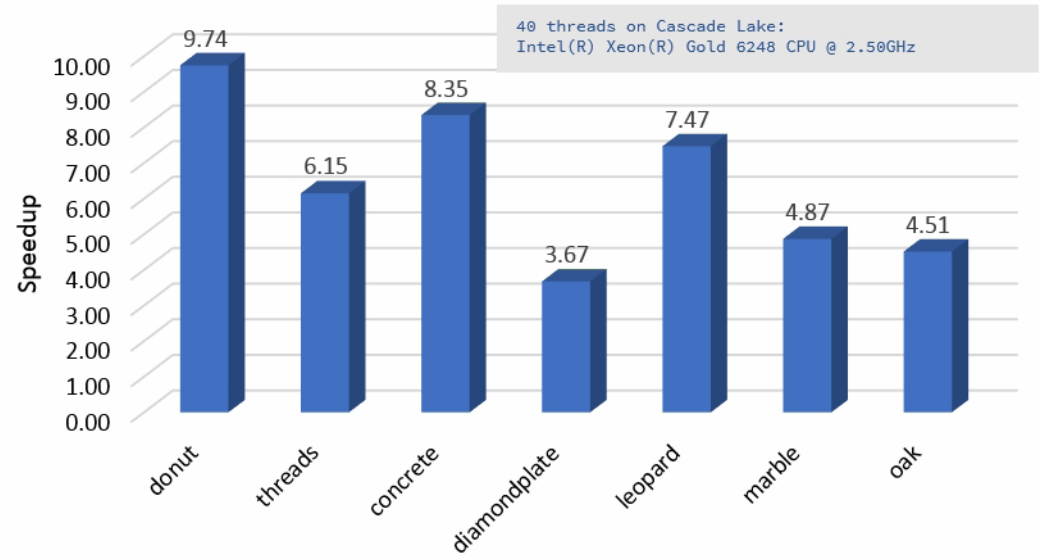
We evaluated a set of open source shaders using AVX-512 SIMD OSL, AVX2 SIMD OSL, and scalar OSL on an Intel Xeon Gold 6248 processor @2.5GHz. Each shader—marble[7], concrete[10], diamond plate[11], donut[12], leopard[13], oak[14], threads[15]—represents a distinct texture, as shown in **Figure 10**. The shaders differ in their complexity and the types and quantity of OSL functions they employ. For instance, the thread, marble, and oak shaders have a relatively simple control flow with only one or no branches, a single input, and a single output. On the other hand, shaders like concrete and leopard have a more complicated and divergent control flow.



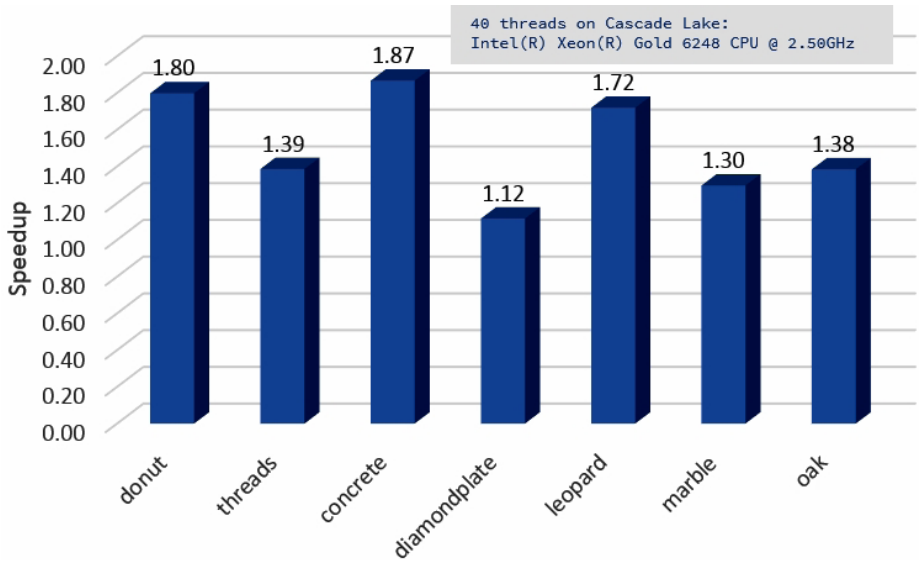
10 **OSL shaders. From left to right: concrete, leopard, oak, marble, diamondplate, threads, and donut**

First, we evaluate the speedup of the OSL shaders using SIMD OSL and compare performance against scalar single-point OSL (**Figure 11**). Note the concrete shader with its noise function calls enjoys a performance benefit of 9.7x. All shaders show a speedup between 3.7x to 8.4x.

We next evaluate the speedup of AVX-512 SIMD OSL over AVX2 SIMD OSL (**Figure 12**). All the shaders showed a benefit using the wider batch size that AVX-512 provides.



11 **Speedup of SIMD OSL over scalar OSL on an individual OSL shaders on Intel Xeon Gold 6248 processor @2.50GHz**



12 **Speedup of AVX-512 SIMD OSL over AVX2 SIMD OSL on individual OSL shaders on an Intel Xeon Gold 6248 processor @2.50GHz**

Turbo-Charged Open Shading Language

Intel has been leading the rearchitecture of SIMD OSL since 2016. This rearchitecture can be broadly summarized in two steps:

- **Introducing vector LLVM IR generation** (for JIT) during render-time optimization
- **Adding a batched interface** to the default single-point interface in OSL

SIMD OSL produced considerable benefit in physically-based renderers such as Pixar's RenderMan. The recently released RenderMan 22.5 with SIMD OSL has seen up to 2x faster full renders and a 15% average speedup using Intel Xeon Scalable processors with Intel AVX-512.[3]

References

1. **Open Shading Language Sci Tech Award in 2017**
2. **Open Shading Language Repository**
3. **Pixar Animation Studios Releases RenderMan 22.5**
4. **RenderMan: What's New**
5. **FMX 2019**
6. **Open Shading Language Specification**
7. **Marble Shader**
8. **OSL Talk at SIGGRAPH 2018**
9. **Intel® SIMD Data Layout Templates (SDLT)**
10. **Concrete.osl**
11. **DiamondPlate.osl**
12. **TheDonutShader.osl**
13. **Leopard.osl**
14. **Oak.osl**
15. **Threads.osl**