Intel® Open Image Denoise

High-Performance Denoising Library for Ray Tracing

Version 2.3.2
January 13, 2025

Contents

Chapter 1
Overview

Intel Open Image Denoise is an open source library of high-performance, high-quality denoising filters for images rendered with ray tracing. Intel Open Image Denoise is part of the Intel® Rendering Toolkit and is released under the permissive Apache 2.0 license.

The purpose of Intel Open Image Denoise is to provide an open, high-quality, efficient, and easy-to-use denoising library that allows one to significantly reduce rendering times in ray tracing based rendering applications. It filters out the Monte Carlo noise inherent to stochastic ray tracing methods like path tracing, reducing the amount of necessary samples per pixel by even multiple orders of magnitude (depending on the desired closeness to the ground truth). A simple but flexible C/C++ API ensures that the library can be easily integrated into most existing or new rendering solutions.

At the heart of the Intel Open Image Denoise library is a collection of efficient deep learning based denoising filters, which were trained to handle a wide range of samples per pixel (spp), from 1 spp to almost fully converged. Thus it is suitable for both preview and final-frame rendering. The filters can denoise images either using only the noisy color (*beauty*) buffer, or, to preserve as much detail as possible, can optionally utilize auxiliary feature buffers as well (e.g. albedo, normal). Such buffers are supported by most renderers as arbitrary output variables (AOVs) or can be usually implemented with little effort.

Although the library ships with a set of pre-trained filter models, it is not mandatory to use these. To optimize a filter for a specific renderer, sample count, content type, scene, etc., it is possible to train the model using the included training toolkit and user-provided image datasets.

Intel Open Image Denoise supports a wide variety of CPUs and GPUs from different vendors:

- Intel® 64 architecture compatible CPUs (with at least SSE4.1)
- ARM64 (AArch64) architecture CPUs (e.g. Apple silicon CPUs)
- Intel Xe, Xe2, and Xe3 architecture dedicated and integrated GPUs, including Intel® Arc™ B-Series Graphics, Intel® Arc™ A-Series Graphics, Intel® Arc™ Pro Series Graphics, Intel® Data Center GPU Flex Series, Intel® Data Center GPU Max Series, Intel® Iris® Xe Graphics, Intel® Core™ Ultra Processors with Intel® Arc™ Graphics, 11th-14th Gen Intel® Core™ processor graphics, and related Intel Pentium® and Celeron® processors (Xe-LP, Xe-LPG, Xe-LPG+, Xe-HPG, Xe-HPC, Xe2-LPG, Xe2-HPG, and Xe3-LPG microarchitectures)
- NVIDIA GPUs with Volta, Turing, Ampere, Ada Lovelace, and Hopper architectures

- AMD GPUs with RDNA2 (Navi 21 only) and RDNA3 (Navi 3x) architectures
- Apple silicon GPUs (M1 and newer)

It runs on most machines ranging from laptops to workstations and compute nodes in HPC systems. It is efficient enough to be suitable not only for offline rendering, but, depending on the hardware used, also for interactive or even real-time ray tracing.

Intel Open Image Denoise exploits modern instruction sets like SSE4, AVX2, AVX-512, and NEON on CPUs, Intel® Xe Matrix Extensions (Intel® XMX) on Intel GPUs, and tensor cores on NVIDIA GPUs to achieve high denoising performance.

1.1 System Requirements

You need an Intel® 64 (with SSE4.1) or ARM64 architecture compatible CPU to run Intel Open Image Denoise, and you need a 64-bit Windows, Linux, or macOS operating system as well.

For Intel GPU support, please also install the latest Intel graphics drivers:

- Windows: Intel® Graphics Driver 31.0.101.4953 or newer
- Linux: Intel® software for General Purpose GPU capabilities release 20230323 or newer

Using older driver versions is *not* supported and Intel Open Image Denoise might run with only limited capabilities, have suboptimal performance or might be unstable. Also, Resizable BAR *must* be enabled in the BIOS for Intel dedicated GPUs if running on Linux, and strongly recommended if running on Windows.

For NVIDIA GPU support, please also install the latest NVIDIA graphics drivers:

- Windows: Version 527.41 or newer
- Linux: Version 525.60.13 or newer

For AMD GPU support, please also install the latest AMD graphics drivers:

- Windows: AMD Software: Adrenalin Edition 24.10.1 or newer
- Linux: Radeon Software for Linux version 24.20.3 or newer

For Apple GPU support, macOS Ventura or newer is required.

1.2 Support and Contact

Intel Open Image Denoise is under active development, and though we do our best to guarantee stable release versions a certain number of bugs, as-yet-missing features, inconsistencies, or any other issues are still possible. Should you find any such issues please report them immediately via the Intel Open Image Denoise GitHub Issue Tracker (or, if you should happen to have a fix for it, you can also send us a pull request); for missing features please contact us via email at `openimagedenoise@googlegroups.com`.

Join our mailing list to receive release announcements and major news regarding Intel Open Image Denoise.

1.3 Citation

If you use Intel Open Image Denoise in a research publication, please cite the project using the following BibTeX entry:

```
@misc{OpenImageDenoise,
  author = {Attila T. {\'A}fra},
  title  = {{Intel\textsuperscript{\textregistered} Open Image Denoise}},
  year   = {2025},
  note   = {\url{https://www.openimagedenoise.org}}
}
```

1.4 Version History

Changes in v2.3.2:

- Improved performance for Intel Lunar Lake and Battlemage GPUs
- Added Intel Panther Lake GPU support
- Fixed compile error when building with OpenImageIO 3.x

Changes in v2.3.1:

- Fixed corrupted output when in-place denoising high-resolution (> 1080p) images where the input and output are stored in different shared buffer objects (created with `oidnNewSharedBuffer*`) that overlap in memory
- Fixed issues with cancellation through progress monitor callbacks:
  - Fixed cancellation requests not being fulfilled on CPU devices since `v2.3.0`
  - Fixed not calling the callback anymore after requesting cancellation, while the operation is still being executed
- Added support for creating shared buffers on Metal devices
- Enabled accessing system allocated memory for CUDA devices which support this feature (see `systemMemorySupported` device parameter)
- Added LUID support for HIP devices. Importing DX12 and Vulkan buffers is now functional when using recent AMD GPU drivers on Windows

Changes in v2.3.0:

- Significantly improved image quality of the `RT` filter in *high* quality mode for HDR denoising with prefiltering, i.e., the following combinations of input features and parameters: - HDR color + albedo + normal + `cleanAux` - albedo - normal In these cases a much more complex filter is used, which results in lower performance than before (about 2x). To revert to the previous performance behavior, please switch to the *balanced* quality mode.
- Added *fast* quality mode (`OIDN_QUALITY_FAST`) for even higher performance (about 1.5-2x) interactive/real-time previews and lower default memory usage at the cost of somewhat lower image quality. Currently this is implemented for the `RT` filter except prefiltering (albedo, normal). In other cases denoising implicitly falls back to *balanced* mode.
- Added Intel Arrow Lake, Lunar Lake, and Battlemage GPU support
- Execute `Async` functions asynchronously on CPU devices as well
- Load/initialize device modules lazily (improves stability)
- Added `oidnIsCPUDeviceSupported`, `oidnIsSYCLDeviceSupported`, `oidnIsCUDADeviceSupported`, `oidnIsHIPDeviceSupported`, and `oidnIsMetalDeviceSupported` API functions for checking whether a physical device of a particular type is supported

- Release the CUDA primary context when destroying the device object if using the CUDA driver API
- Added `OIDN_LIBRARY_NAME` CMake option for setting the base name of the Open Image Denoise library files
- Fixed device creation error with `oidnNewDevice` when the default device of the specified type (e.g. CUDA) is not supported but there are other supported non-default devices of that type in the system
- Fixed CMake error when building with Metal support using non-Apple Clang
- Fixed iOS build errors
- Added support for building with ROCm 6.x
- `oidnNewCUDADevice` and `oidnNewHIPDevice` no longer accept negative device IDs. If the goal is to use the current device, its actual ID needs to be passed.
- Upgraded to oneTBB 2021.12.0 in the official binaries
- Training:
  - Improved training performance on CUDA and MPS devices, added `--compile` option
  - Added `--quality` option (`high`, `balanced`, `fast`) for selecting the size of the model to train, changed the default from `balanced` to `high`
  - Added new models to the `--model` option (`unet_small`, `unet_large`, `unet_xl`)
  - Added support for training with prefiltered auxiliary features by passing `--aux_results` to `preprocess.py` and `train.py`
  - Added experimental support for depth (`z`)

Changes in v2.2.2:

- Fully fixed GPU memory leak when releasing SYCL, CUDA and HIP device objects
- Fixed CUDA context error in some cases when using the CUDA driver API
- Fixed crash on systems with an unsupported AMD Vega integrated GPU and old driver

Changes in v2.2.1:

- Fixed memory leak when releasing SYCL, CUDA and HIP device objects
- Fixed memory leak when initializing Metal filters

Changes in v2.2.0:

- Improved denoising quality (better fine detail reconstruction)
- Added Intel Meteor Lake GPU support (in Intel® Core™ Ultra Processors)
- Added Metal device for Apple silicon GPUs (requires macOS Ventura or newer)
- Added ARM64 (AArch64) CPU support on Windows and Linux (in addition to macOS)
- Improved CPU performance
- Significantly reduced overhead of committing filter changes
- Switched to the CUDA driver API by default, added the `OIDN_DEVICE_CUDA_API` CMake option for manually selecting between the driver and runtime APIs
- Fixed crash when releasing a buffer after releasing the device

Changes in v2.1.0:

- Added support for denoising 1-channel (e.g. alpha) and 2-channel images
- Added support for arbitrary combinations of input image data types (e.g. `OIDN_FORMAT_FLOAT3` for `color` but `OIDN_FORMAT_HALF3` for `albedo`)
- Improved performance for most dedicated GPU architectures
- Re-added `OIDN_STATIC_LIB` CMake option which enables building as a static (CPU support only) or a hybrid static/shared (GPU support as well) library
- Added `release()` method to C++ API objects (`DeviceRef`, `BufferRef`, `FilterRef`)
- Fixed possible crash when releasing GPU devices, buffers or filters
- Fixed possible crash at process exit for some SYCL runtime versions
- Fixed image quality inconsistency on Intel integrated GPUs, but at the cost of some performance loss
- Fixed future Windows driver compatibility for Intel integrated GPUs
- Fixed rare output corruption on AMD RDNA2 GPUs
- Fixed device detection on Windows when the path to the library has non-ANSI characters
- Added support for Intel® oneAPI DPC++/C++ Compiler 2024.0 and compatible open source compiler versions
- Upgraded to oneTBB 2021.10.0 in the official binaries
- Improved detection of old oneTBB versions

Changes in v2.0.1:

- Fixed performance issue for Intel integrated GPUs using recent Linux drivers
- Fixed crash on systems with both dedicated and integrated AMD GPUs
- Fixed importing `D3D12_RESOURCE`, `D3D11_RESOURCE`, `D3D11_RESOURCE_KMT`, `D3D11_TEXTURE` and `D3D11_TEXTURE_KMT` external memory types on CUDA and HIP devices
- Fixed the macOS deployment target of the official x86 binaries (lowered from 11.0 to 10.11)
- Minor improvements to verbose output

Changes in v2.0.0:

- Added SYCL device for Intel Xe architecture GPUs (Xe-LP, Xe-HPG and Xe-HPC)
- Added CUDA device for NVIDIA Volta, Turing, Ampere, Ada Lovelace and Hopper architecture GPUs
- Added HIP device for AMD RDNA2 (Navi 21 only) and RDNA3 (Navi 3x) architecture GPUs
- Added new buffer API functions for specifying the storage type (host, device or managed), copying data to/from the host, and importing external buffers from graphics APIs (e.g. Vulkan, Direct3D 12)
- Removed the `oidnMapBuffer` and `oidnUnmapBuffer` functions
- Added support for asynchronous execution (e.g. `oidnExecuteFilterAsync`, `oidnSyncDevice` functions)
- Added physical device API for querying the supported devices in the system
- Added functions for creating a device from a physical device ID, UUID, LUID or PCI address (e.g. `oidnNewDeviceByID`)
- Added SYCL, CUDA and HIP interoperability API functions (e.g. `oidnNewSYCLDevice`, `oidnExecuteSYCLFilterAsync`)
- Added `type` device parameter for querying the device type

- Added `systemMemorySupported` and `managedMemorySupported` device parameters for querying memory allocations supported by the device
- Added `externalMemoryTypes` device parameter for querying the supported external memory handle types
- Added `quality` filter parameter for setting the filtering quality mode (*high* or *balanced* quality)
- Minor API changes with backward compatibility:
  - Added `oidn(Get|Set)(Device|Filter)(Bool|Int|Float)` functions and deprecated `oidn(Get|Set)(Device|Filter)(1b|1i|1f)` functions
  - Added `oidnUnsetFilter(Image|Data)` functions and deprecated `oidnRemoveFilter(Image|Data)` functions
  - Renamed `alignment` and `overlap` filter parameters to `tileAlignment` and `tileOverlap` but the old names remain supported
- Removed `OIDN_STATIC_LIB` and `OIDN_STATIC_RUNTIME` CMake options due to technical limitations
- Fixed over-conservative buffer bounds checking for images with custom strides
- Upgraded to oneTBB 2021.9.0 in the official binaries

Changes in v1.4.3:

- Fixed hardcoded library paths in installed macOS binaries
- Disabled VTune profiling support of oneDNN kernels by default, can be enabled using CMake options if required (`DNNL_ENABLE_JIT_PROFILING` and `DNNL_ENABLE_ITT_TASKS`)
- Upgraded to oneTBB 2021.5.0 in the official binaries

Changes in v1.4.2:

- Added support for 16-bit half-precision floating-point images
- Added `oidnGetBufferData` and `oidnGetBufferSize` functions
- Fixed performance issue on x86 hybrid architecture CPUs (e.g. Alder Lake)
- Fixed build error when using OpenImageIO 2.3 or later
- Upgraded to oneTBB 2021.4.0 in the official binaries

Changes in v1.4.1:

- Fixed crash when in-place denoising images with certain unusual resolutions
- Fixed compile error when building for Apple Silicon using some unofficial builds of ISPC

Changes in v1.4.0:

- Improved fine detail preservation
- Added the `cleanAux` filter parameter for further improving quality when the auxiliary feature (albedo, normal) images are noise-free
- Added support for denoising auxiliary feature images, which can be used together with the new `cleanAux` parameter for improving quality when the auxiliary images are noisy (recommended for final frame denoising)
- Normals are expected to be in the [-1, 1] range (but still do not have to be normalized)
- Added the `oidnUpdateFilterData` function which must be called when the contents of an opaque data parameter bound to a filter (e.g. `weights`) has been changed after committing the filter

- Added the `oidnRemoveFilterImage` and `oidnRemoveFilterData` functions for removing previously set image and opaque data parameters of filters
- Reduced the overhead of `oidnCommitFilter` to zero in some cases (e.g. when changing already set image buffers/pointers or the `inputScale` parameter)
- Reduced filter memory consumption by about 35%
- Reduced total memory consumption significantly when using multiple filters that belong to the same device
- Reduced the default maximum memory consumption to 3000 MB
- Added the `OIDN_FILTER_RT` and `OIDN_FILTER_RTLIGHTMAP` CMake options for excluding the trained filter weights from the build to significantly decrease its size
- Fixed detection of static TBB builds on Windows
- Fixed compile error when using future glibc versions
- Added `oidnBenchmark` option for setting custom resolutions
- Upgraded to oneTBB 2021.2.0 in the official binaries

Changes in v1.3.0:

- Improved denoising quality
  - Improved sharpness of fine details / less blurriness
  - Fewer noisy artifacts
- Slightly improved performance and lowered memory consumption
- Added directional (e.g. spherical harmonics) lightmap denoising to the `RTLightmap` filter
- Added `inputScale` filter parameter which generalizes the existing (and thus now deprecated) `hdrScale` parameter for non-HDR images
- Added native support for Apple Silicon and the BNNS library on macOS (currently requires rebuilding from source)
- Added `OIDN_NEURAL_RUNTIME` CMake option for setting the neural network runtime library
- Reduced the size of the library binary
- Fixed compile error on some older macOS versions
- Upgraded release builds to use oneTBB 2021.1.1
- Removed tbbmalloc dependency
- Appended the library version to the name of the directory containing the installed CMake files
- Training:
  - Faster training performance
  - Added mixed precision training (enabled by default)
  - Added efficient data-parallel training on multiple GPUs
  - Enabled preprocessing datasets multiple times with possibly different options
  - Minor bugfixes

Changes in v1.2.4:

- Added OIDN_API_NAMESPACE CMake option that allows to put all API functions inside a user-defined namespace
- Fixed bug when TBB_USE_GLIBCXX_VERSION is defined
- Fixed compile error when using an old compiler which does not support OpenMP SIMD
- Added compatibility with oneTBB 2021
- Export only necessary symbols on Linux and macOS

Changes in v1.2.3:

- Fixed incorrect detection of AVX-512 on macOS (sometimes causing a crash)
- Fixed inconsistent performance and costly initialization for AVX-512
- Fixed JIT'ed AVX-512 kernels not showing up correctly in VTune

Changes in v1.2.2:

- Fixed unhandled exception when canceling filter execution from the progress monitor callback function

Changes in v1.2.1:

- Fixed tiling artifacts when in-place denoising (using one of the input images as the output) high-resolution (> 1080p) images
- Fixed ghosting/color bleeding artifacts in black regions when using albedo/normal buffers
- Fixed error when building as a static library (`OIDN_STATIC_LIB` option)
- Fixed compile error for ISPC 1.13 and later
- Fixed minor TBB detection issues
- Fixed crash on pre-SSE4 CPUs when using some recent compilers (e.g. GCC 10)
- Link C/C++ runtime library dynamically on Windows too by default
- Renamed example apps (`oidnDenoise`, `oidnTest`)
- Added benchmark app (`oidnBenchmark`)
- Fixed random data augmentation seeding in training
- Fixed training warning with PyTorch 1.5 and later

Changes in v1.2.0:

- Added neural network training code
- Added support for specifying user-trained models at runtime
- Slightly improved denoising quality (e.g. less ringing artifacts, less blurriness in some cases)
- Improved denoising speed by about 7-38% (mostly depending on the compiler)
- Added `OIDN_STATIC_RUNTIME` CMake option (for Windows only)
- Added support for OpenImageIO to the example apps (disabled by default)
- Added check for minimum supported TBB version
- Find debug versions of TBB
- Added testing

Changes in v1.1.0:

- Added `RTLightmap` filter optimized for lightmaps
- Added `hdrScale` filter parameter for manually specifying the mapping of HDR color values to luminance levels

Changes in v1.0.0:

- Improved denoising quality
  - More details preserved
  - Less artifacts (e.g. noisy spots, color bleeding with albedo/normal)
- Added `maxMemoryMB` filter parameter for limiting the maximum memory consumption regardless of the image resolution, potentially at the cost of

lower denoising speed. This is internally implemented by denoising the image in tiles
- Significantly reduced memory consumption (but slightly lower performance) for high resolutions (> 2K) by default: limited to about 6 GB
- Added `alignment` and `overlap` filter parameters that can be queried for manual tiled denoising
- Added `verbose` device parameter for setting the verbosity of the console output, and disabled all console output by default
- Fixed crash for zero-sized images

Changes in v0.9.0:

- Reduced memory consumption by about 38%
- Added support for progress monitor callback functions
- Enabled fully concurrent execution when using multiple devices
- Clamp LDR input and output colors to 1
- Fixed issue where some memory allocation errors were not reported

Changes in v0.8.2:

- Fixed wrong HDR output when the input contains infinities/NaNs
- Fixed wrong output when multiple filters were executed concurrently on separate devices with AVX-512 support. Currently the filter executions are serialized as a temporary workaround, and a full fix will be included in a future release.
- Added `OIDN_STATIC_LIB` CMake option for building as a static library (requires CMake 3.13.0 or later)
- Fixed CMake error when adding the library with add_subdirectory() to a project

Changes in v0.8.1:

- Fixed wrong path to TBB in the generated CMake configs
- Fixed wrong rpath in the binaries
- Fixed compile error on some macOS systems
- Fixed minor compile issues with Visual Studio
- Lowered the CPU requirement to SSE4.1
- Minor example update

Changes in v0.8.0:

- Initial beta release

Chapter 2
Compilation

The latest Intel Open Image Denoise sources are always available at the Intel Open Image Denoise GitHub repository. The default `master` branch should always point to the latest tested bugfix release.

2.1 Prerequisites

You can clone the latest Intel Open Image Denoise sources using Git with the Git Large File Storage (LFS) extension installed:

```
git clone --recursive https://github.com/OpenImageDenoise/oidn.git
```

Please note that installing the Git LFS extension is *required* to correctly clone the repository. Cloning without Git LFS will seemingly succeed but actually some of the files will be invalid and thus compilation will fail.

Intel Open Image Denoise currently supports 64-bit Linux, Windows, and macOS operating systems. Before you can build Intel Open Image Denoise you need the following basic prerequisites:

- CMake 3.15 or newer
- A C++11 compiler (we recommend using a Clang-based compiler but also support GCC and Microsoft Visual Studio 2015 and newer)
- Python 3

To build support for different types of CPUs and GPUs, the following additional prerequisites are needed:

CPU device:

- Intel® SPMD Program Compiler (ISPC) 1.21.0 or newer. Please obtain a release of ISPC from the ISPC downloads page. The build system looks for ISPC in the `PATH` and in the directory right "next to" the checked-out Intel Open Image Denoise sources. For example, if Intel Open Image Denoise is in `~/Projects/oidn`, ISPC will also be searched in `~/Projects/ispc-v1.21.0-linux`. Alternatively set the CMake variable `ISPC_EXECUTABLE` to the location of the ISPC compiler.
- Intel® Threading Building Blocks (TBB) 2017 or newer

SYCL device for Intel GPUs:

- oneAPI DPC++ Compiler, one of the following versions (other versions are *not* supported):
  - oneAPI DPC++ Compiler 2023-10-26. This is the open source version of the compiler, which is more up-to-date but less stable, so we *strongly* recommend to use this exact version. On Linux we also recommend to rebuild it from source with the `--disable-fusion` flag to minimize the size of the SYCL runtime.
  - oneAPI DPC++ Compiler 2022-12. *Must* be rebuilt from source.
  - Intel® oneAPI DPC++/C++ Compiler 2024.1 or newer
- Intel® Graphics Offline Compiler for OpenCL™ Code (OCLOC)
  - Windows: Version 2025.0.0 / 32.0.101.6129 or newer as a standalone component of Intel® oneAPI Toolkits, which must be extracted and its contents added to the `PATH`. Also included with Intel® oneAPI Base Toolkit.
  - Linux: Included with Intel® software for General Purpose GPU capabilities release 2441.19 or newer (install at least `intel-opencl-icd` on Ubuntu, `intel-ocloc` on RHEL or SLES). Also available with Intel® Graphics Compute Runtime for oneAPI Level Zero and OpenCL™ Driver.
- If using Intel® oneAPI DPC++/C++ Compiler: CMake 3.25.2 or newer
- Ninja or Make as the CMake generator. The Visual Studio generator is *not* supported.

CUDA device for NVIDIA GPUs:

- CMake 3.18 or newer
- NVIDIA CUDA Toolkit 11.8 or newer

HIP device for AMD GPUs:

- CMake 3.21 or newer
- Ninja or Make as the CMake generator. The Visual Studio generator is *not* supported.
- AMD ROCm (HIP SDK) v6.1.2 or newer.
- Perl (e.g. Strawberry Perl on Windows)

Metal device for Apple GPUs:

- CMake 3.21 or newer
- Xcode 15.0 or newer

Depending on your operating system, you can install some required dependencies (e.g., TBB) using `yum` or `apt-get` on Linux, Homebrew or MacPorts on macOS, and `vcpkg` on Windows. For the other dependencies please download the necessary packages or installers and follow the included instructions.

2.2 Compiling on Linux/macOS

If you are building with SYCL support on Linux, make sure that the DPC++ compiler is properly set up. The open source oneAPI DPC++ Compiler can be downloaded and simply extracted. However, before using the compiler, the environment must be set up as well with the following command:

```
source ./dpcpp_compiler/startup.sh
```

The `startup.sh` script will put `clang` and `clang++` from the oneAPI DPC++ Compiler into your `PATH`.

Alternatively, if you have installed Intel® oneAPI DPC++/C++ Compiler instead, you can set up the compiler by sourcing the `vars.sh` script in the `env` directory of the compiler install directory, for example,

```
source /opt/intel/oneAPI/compiler/latest/env/vars.sh
```

This script will put the `icx` and `icpx` compiler executables from the Intel(R) oneAPI DPC++/C++ Compiler in your `PATH`.

- Create a build directory, and go into it using a command prompt

  ```
  mkdir oidn/build
  cd oidn/build
  ```

  (We do recommend having separate build directories for different configurations such as release, debug, etc.).

- CMake will use the default compiler, which on most Linux machines is `gcc`, but it can be switched to `clang` by executing the following:

  ```
  cmake -G Ninja -DCMAKE_C_COMPILER=clang -DCMAKE_CXX_COMPILER=clang++ ..
  ```

  If you are building with SYCL support, you must set the DPC++ compiler (`clang`/`clang++` or `icx`/`icpx`) as the C/C++ compiler here. Note that the compiler variables cannot be changed after the first `cmake` or `ccmake` run.

- Open the CMake configuration dialog

  ```
  ccmake ..
  ```

- Make sure to properly set the build mode and enable the components and options you need. By default only CPU support is built, so SYCL and other device support must be enabled manually (e.g. with the `OIDN_DEVICE_SYCL` option). Then type 'c'onfigure and 'g'enerate. When back on the command prompt, build the library using

  ```
  ninja
  ```

2.3 Compiling on Windows

If you are building with SYCL support, make sure that the DPC++ compiler is properly set up. The open source oneAPI DPC++ Compiler can be downloaded and simply extracted. However, before using the compiler, the environment must be set up. To achieve this, open the "x64 Native Tools Command Prompt for VS" that ships with Visual Studio and execute the following commands:

```
set "DPCPP_DIR=path_to_dpcpp_compiler"
set "PATH=%DPCPP_DIR%\bin;%PATH%"
set "PATH=%DPCPP_DIR%\lib;%PATH%"
set "CPATH=%DPCPP_DIR%\include;%CPATH%"
set "INCLUDE=%DPCPP_DIR%\include;%INCLUDE%"
set "LIB=%DPCPP_DIR%\lib;%LIB%"
```

The `path_to_dpcpp_compiler` should point to the unpacked oneAPI DPC++ Compiler.

Alternatively, if you have installed Intel® oneAPI DPC++/C++ Compiler instead, you can either open a regular "Command Prompt" and execute the `vars.bat` script in the `env` directory of the compiler install directory, for example

```
C:\Program Files (x86)\Intel\oneAPI\compiler\latest\env\vars.bat
```

or simply open the installed "Intel oneAPI command prompt for Intel 64 for Visual Studio". Either way, the `icx` compiler executable from the Intel® oneAPI DPC++/C++ Compiler will be added to your `PATH`.

On Windows we highly recommend to use Ninja as the CMake generator because not all devices can be built using the Visual Studio generator (e.g. SYCL).

- Create a build directory, and go into it using a Visual Studio command prompt

  ```
  mkdir oidn/build
  cd oidn/build
  ```

  (We do recommend having separate build directories for different configurations such as release, debug, etc.).

- CMake will use the default compiler, which on most Windows machines is MSVC, but it can be switched to `clang` by executing the following:

  ```
  cmake -G Ninja -DCMAKE_C_COMPILER=clang -DCMAKE_CXX_COMPILER=clang++ ..
  ```

  If you are building with SYCL support, you must set the DPC++ compiler (`clang`/`clang++` or `icx`) as the C/C++ compiler here. Note that the compiler variables cannot be changed after the first `cmake` or `cmake-gui` run.

- Open the CMake GUI (`cmake-gui.exe`)

  ```
  cmake-gui ..
  ```

- Make sure to properly set the build mode and enable the components and options you need. By default only CPU support is built, so SYCL and other device support must be enabled manually (e.g. `OIDN_DEVICE_SYCL` option). Then click on Configure and Generate. When back on the command prompt, build the library using

  ```
  ninja
  ```

2.4 CMake Configuration

The following list describes the options that can be configured in CMake:

- `CMAKE_BUILD_TYPE`: Can be used to switch between Debug mode (Debug), Release mode (Release) (default), and Release mode with enabled assertions and debug symbols (RelWithDebInfo).
- `OIDN_STATIC_LIB`: Build Open Image Denoise as a static (if only CPU support is enabled) or a hybrid static/shared (if GPU support is enabled as well) library.
- `OIDN_LIBRARY_NAME`: Specifies the base name of the Open Image Denoise library files (`OpenImageDenoise` by default).
- `OIDN_API_NAMESPACE`: Specifies a namespace to put all Open Image Denoise API symbols inside. This is also added as an outer namespace for the C++ wrapper API. By default no namespace is used and plain C symbols are exported.
- `OIDN_DEVICE_CPU`: Enable CPU device support (ON by default).
- `OIDN_DEVICE_SYCL`: Enable SYCL device support for Intel GPUs (OFF by default).
- `OIDN_DEVICE_SYCL_AOT`: Enable ahead-of-time (AOT) compilation for SYCL kernels (ON by default). Turning this off removes dependency on OCLOC at build time and decreases binary size but significantly increases initialization time at runtime, so it is recommended only for development.
- `OIDN_DEVICE_CUDA`: Enable CUDA device support for NVIDIA GPUs (OFF by default).
- `OIDN_DEVICE_CUDA_API`: Use the CUDA driver API (`Driver`, default), the static CUDA runtime library (`RuntimeStatic`), or the shared CUDA runtime library (`RuntimeShared`).
- `OIDN_DEVICE_HIP`: Enable HIP device support for AMD GPUs (OFF by default).
- `OIDN_DEVICE_METAL`: Enable Metal device support for Apple GPUs (OFF by default).
- `OIDN_FILTER_RT`: Include the trained weights of the `RT` filter in the build (ON by default). Turning this OFF significantly decreases the size of the library binary, while the filter remains functional if the weights are set by the user at runtime.
- `OIDN_FILTER_RTLIGHTMAP`: Include the trained weights of the `RTLightmap` filter in the build (ON by default).
- `OIDN_APPS`: Enable building example and test applications (ON by default).
- `OIDN_APPS_OPENIMAGEIO`: Enable OpenImageIO support in the example and test applications to be able to load/save OpenEXR, PNG, and other image file formats (OFF by default).
- `OIDN_INSTALL_DEPENDENCIES`: Enable installing the dependencies (e.g. TBB, SYCL runtime) as well.
- `TBB_ROOT`: The path to the TBB installation (autodetected by default).

- `ROCM_PATH`: The path to the ROCm installation (autodetected by default).
- `OpenImageIO_ROOT`: The path to the OpenImageIO installation (autodetected by default).

Chapter 3

Open Image Denoise API

Open Image Denoise provides a C99 API (also compatible with C++) and a C++11 wrapper API as well. For simplicity, this document mostly refers to the C99 version of the API.

The API is designed in an object-oriented manner, e.g. it contains device objects (`OIDNDevice` type), buffer objects (`OIDNBuffer` type), and filter objects (`OIDNFilter` type). All objects are reference-counted, and handles can be released by calling the appropriate release function (e.g. `oidnReleaseDevice`) or retained by incrementing the reference count (e.g. `oidnRetainDevice`).

An important aspect of objects is that setting their parameters do not have an immediate effect (with a few exceptions). Instead, objects with updated parameters are in an unusable state until the parameters get explicitly committed to a given object. The commit semantic allows for batching up multiple small changes, and specifies exactly when changes to objects will occur.

All API calls are thread-safe, but operations that use the same device will be serialized, so the amount of API calls from different threads should be minimized.

3.1 Examples

To have a quick overview of the C99 and C++11 APIs, see the following simple example code snippets.

3.1.1 Basic Denoising (C99 API)

```
#include <OpenImageDenoise/oidn.h>
...

// Create an Open Image Denoise device
OIDNDevice device = oidnNewDevice(OIDN_DEVICE_TYPE_DEFAULT); // CPU or GPU if available
// OIDNDevice device = oidnNewDevice(OIDN_DEVICE_TYPE_CPU);
oidnCommitDevice(device);

// Create buffers for input/output images accessible by both host (CPU) and device (CPU/GPU)
OIDNBuffer colorBuf  = oidnNewBuffer(device, width * height * 3 * sizeof(float));
OIDNBuffer albedoBuf = ...

// Create a filter for denoising a beauty (color) image using optional auxiliary images too
// This can be an expensive operation, so try not to create a new filter for every image!
OIDNFilter filter = oidnNewFilter(device, "RT"); // generic ray tracing filter
oidnSetFilterImage(filter, "color",  colorBuf,
                   OIDN_FORMAT_FLOAT3, width, height, 0, 0, 0); // beauty
oidnSetFilterImage(filter, "albedo", albedoBuf,
```

```
                    OIDN_FORMAT_FLOAT3, width, height, 0, 0, 0); // auxiliary
oidnSetFilterImage(filter, "normal", normalBuf,
                    OIDN_FORMAT_FLOAT3, width, height, 0, 0, 0); // auxiliary
oidnSetFilterImage(filter, "output", colorBuf,
                    OIDN_FORMAT_FLOAT3, width, height, 0, 0, 0); // denoised beauty
oidnSetFilterBool(filter, "hdr", true); // beauty image is HDR
oidnCommitFilter(filter);

// Fill the input image buffers
float* colorPtr = (float*)oidnGetBufferData(colorBuf);
...

// Filter the beauty image
oidnExecuteFilter(filter);

// Check for errors
const char* errorMessage;
if (oidnGetDeviceError(device, &errorMessage) != OIDN_ERROR_NONE)
  printf("Error: %s\n", errorMessage);

// Cleanup
oidnReleaseBuffer(colorBuf);
...
oidnReleaseFilter(filter);
oidnReleaseDevice(device);
```

3.1.2 Basic Denoising (C++11 API)

```
#include <OpenImageDenoise/oidn.hpp>
...

// Create an Open Image Denoise device
oidn::DeviceRef device = oidn::newDevice(); // CPU or GPU if available
// oidn::DeviceRef device = oidn::newDevice(oidn::DeviceType::CPU);
device.commit();

// Create buffers for input/output images accessible by both host (CPU) and device (CPU/GPU)
oidn::BufferRef colorBuf  = device.newBuffer(width * height * 3 * sizeof(float));
oidn::BufferRef albedoBuf = ...

// Create a filter for denoising a beauty (color) image using optional auxiliary images too
// This can be an expensive operation, so try no to create a new filter for every image!
oidn::FilterRef filter = device.newFilter("RT"); // generic ray tracing filter
filter.setImage("color",  colorBuf,  oidn::Format::Float3, width, height); // beauty
filter.setImage("albedo", albedoBuf, oidn::Format::Float3, width, height); // auxiliary
filter.setImage("normal", normalBuf, oidn::Format::Float3, width, height); // auxiliary
filter.setImage("output", colorBuf,  oidn::Format::Float3, width, height); // denoised beauty
filter.set("hdr", true); // beauty image is HDR
filter.commit();

// Fill the input image buffers
float* colorPtr = (float*)colorBuf.getData();
...

// Filter the beauty image
filter.execute();
```

```
// Check for errors
const char* errorMessage;
if (device.getError(errorMessage) != oidn::Error::None)
  std::cout << "Error: " << errorMessage << std::endl;
```

3.1.3 Denoising with Prefiltering (C++11 API)

```
// Create a filter for denoising a beauty (color) image using prefiltered auxiliary images too
oidn::FilterRef filter = device.newFilter("RT"); // generic ray tracing filter
filter.setImage("color",  colorBuf,  oidn::Format::Float3, width, height); // beauty
filter.setImage("albedo", albedoBuf, oidn::Format::Float3, width, height); // auxiliary
filter.setImage("normal", normalBuf, oidn::Format::Float3, width, height); // auxiliary
filter.setImage("output", outputBuf, oidn::Format::Float3, width, height); // denoised beauty
filter.set("hdr", true); // beauty image is HDR
filter.set("cleanAux", true); // auxiliary images will be prefiltered
filter.commit();

// Create a separate filter for denoising an auxiliary albedo image (in-place)
oidn::FilterRef albedoFilter = device.newFilter("RT"); // same filter type as for beauty
albedoFilter.setImage("albedo", albedoBuf, oidn::Format::Float3, width, height);
albedoFilter.setImage("output", albedoBuf, oidn::Format::Float3, width, height);
albedoFilter.commit();

// Create a separate filter for denoising an auxiliary normal image (in-place)
oidn::FilterRef normalFilter = device.newFilter("RT"); // same filter type as for beauty
normalFilter.setImage("normal", normalBuf, oidn::Format::Float3, width, height);
normalFilter.setImage("output", normalBuf, oidn::Format::Float3, width, height);
normalFilter.commit();

// Prefilter the auxiliary images
albedoFilter.execute();
normalFilter.execute();

// Filter the beauty image
filter.execute();
```

3.2 Upgrading from Open Image Denoise 1.x

Open Image Denoise 2 introduces GPU support, which requires implementing some minor changes in applications. There are also small API changes, additions and improvements in this new version. In this section we summarize the necessary code modifications and also briefly mention the new features that users might find useful when upgrading to version 2.x. For a full description of the changes and new functionality, please see the API reference.

3.2.1 Buffers

The most important required change is related to how data is passed to Open Image Denoise. If the application is explicitly using only the CPU (by specifying `OIDN_DEVICE_TYPE_CPU`), no changes should be necessary. But if it wants to support GPUs as well, passing pointers to memory allocated with the system allocator (e.g. `malloc`) would raise an error because GPUs cannot access such memory in almost all cases.

To ensure compatibility with any kind of device, including GPUs, the application should use `OIDNBuffer` objects to store all image data passed to the library.

Memory allocated using buffers is by default accessible by both the host (CPU) and the device (CPU or GPU).

Ideally, the application should directly read and write image data to/from such buffers to avoid redundant and inefficient data copying. If this cannot be implemented, the application should try to minimize the overhead of copying as much as possible:

- Data should be copied to/from buffers only if the data in system memory indeed cannot be accessed by the device. This can be determined by simply querying the `systemMemorySupported` device parameter. If system allocated memory is accessible by the device, no buffers are necessary and filter image parameters can be set with `oidnSetSharedFilterImage`.
- If the image data cannot be accessed by the device, buffers must be created and the data must be copied to/from these buffers. These buffers should be directly passed to filters as image parameters instead of the original pointers using `oidnSetFilterImage`.
- Data should be copied asynchronously using using the new `oidnReadBufferAsync` and `oidnWriteBufferAsync` functions, which may achieve higher performance than plain `memcpy`.
- If image data must be copied, using the default buffer allocation may not be the most efficient method. If the device memory is not physically shared with the host memory (e.g. for dedicated GPUs), higher performance may be achieved by creating the buffers with device storage (`OIDN_STORAGE_DEVICE`) using the new `oidnNewBufferWithStorage` function. This way, the buffer data cannot be directly accessed by the host anymore but this should not matter because the data must be copied from some other memory location anyway. However, this ensures that the data is stored only in high-performance device memory, and the user has full control over when and how the data is transferred between host and device.

The `oidnMapBuffer` and `oidnUnmapBuffer` functions have been removed from the API due to these not being supported by any of the device backends. Please use `oidnReadBuffer(Async)` and `oidnWriteBuffer(Async)` instead.

3.2.2 Interop with Compute (SYCL, CUDA, HIP) and Graphics (DX, Vulkan, Metal) APIs

If the application is explicitly using a particular device type which supports unified memory allocations, e.g. SYCL or CUDA, it may directly pass pointers allocated using the native allocator of the respective compute API (e.g. `sycl::malloc_device`, `cudaMalloc`) instead of using buffers. This way, it is the responsibility of the user to correctly allocate the memory for the device.

In such cases, it often necessary to have more control over the device creation as well, to ensure that filtering is running on the intended device and command queues or streams from the application can be shared to improve performance. If the application is using the same compute or graphics API as the Open Image Denoise device, this can be achieved by creating devices with `oidnNewSYCLDevice`, `oidnNewCUDADevice`, etc. For some APIs there are additional interoperability functions as well, e.g. `oidnExecuteSYCLFilterAsync`.

If the application is using a graphics API which does not support unified memory allocations, e.g. DX12 or Vulkan, it may be still possible to share memory between the application and Open Image Denoise using buffers, avoiding expensive copying through host memory. External buffers can be imported from graphics APIs with the new `oidnNewSharedBufferFromFD` and `oidnNewSharedBufferFromWin32Handle` functions. To use this feature, buffers must be exported in the

graphics API and must be imported in Open Image Denoise using the same kind of handle. Care must be taken to select an external memory handle type which is supported by both APIs. The external memory types supported by an Open Image Denoise device can be queried using the `externalMemoryTypes` device parameter. Note that some devices do not support importing external memory at all (e.g. CPUs, and on GPUs it primarily depends on the installed drivers), so the application should always implement a fallback too, which copies the data through the host if there is no other supported way. Metal buffers can be used directly with the `oidnNewSharedBufferFromMetal` function.

Sharing textures is currently not supported natively but it is still possible avoid copying texture data by using a linear texture layout (e.g. `VK_IMAGE_TILING_LINEAR` in Vulkan) and sharing the buffer that backs this data. In this case, you should ensure that the row stride of the linear texture data is correctly set.

Importing external synchronization primitives (e.g. semaphores) from graphics APIs is not yet supported either but it is planned for a future release. Meanwhile, synchronizing access to shared memory should be done on the host using `oidnSyncDevice` and the used graphics API.

When importing external memory, the application also needs to make sure that the Open Image Denoise device is running on the same *physical* device as the graphics API. This can be easily achieved by using the new physical device feature, described in the next section.

3.2.3 Physical Devices

Although it is possible to explicitly create devices of a particular type (with, e.g., `OIDN_DEVICE_TYPE_SYCL`), this is often insufficient, especially if the system has multiple devices of the same type, and with GPU support it is very common that there are multiple different types of supported devices in the system (e.g. a CPU and one or more GPUs).

Open Image Denoise 2 introduces a simple *physical device* API, which enables the application to query the list of supported physical devices in the system, including their name, type, UUID, LUID, PCI address, etc. (see `oidnGetNumPhysicalDevices`, `oidnGetPhysicalDeviceString`, etc.). New logical device (i.e. `OIDNDevice`) creation functions for have been also introduced, which enable creating a logical device on a specific physical device: `oidnNewDeviceByID`, `oidnNewDeviceByUUID`, etc.

Creating a logical device on a physical device having a particular UUID, LUID or PCI address is particularly important when importing external memory from graphics APIs. However, not all device types support all types of IDs, and some graphics drivers may even report mismatching UUIDs or LUIDs for the same physical device, so applications should try to implement multiple identification methods, or at least assume that identification might fail.

3.2.4 Asynchronous Execution

It is now possible to execute some operations asynchronously, most importantly filtering (`oidnExecuteFilterAsync`, `oidnExecuteSYCLFilterAsync`) and copying data (the already mentioned `oidnReadBufferAsync` and `oidnWriteBufferAsync`).

When using any asynchronous function it is the responsibility of the application to handle correct synchronization using `oidnSyncDevice`.

3.2.5 Filter Quality

Open Image Denoise still delivers the same high image quality on all device types as before, including on GPUs. But often filtering performance is more important

than having the highest possible image quality, so it is now possible to switch between multiple filter quality modes. Filters have a new parameter called `quality`, which defaults to the existing *high* image quality (`OIDN_QUALITY_HIGH`) but *balanced* (`OIDN_QUALITY_BALANCED`) and *fast* (`OIDN_QUALITY_FAST`) quality modes have been added as well for even higher performance. We recommend using *balanced* or *fast* quality for interactive and real-time use cases.

3.2.6 Small API Changes

A few existing API functions have been renamed to improve clarity (e.g. `oidnSetFilter1i` to `oidnSetFilterInt`) but the old function names are still available as deprecated functions. When compiling legacy code, warnings will be emitted for these deprecated functions. To upgrade to the new API, please simply follow the instructions in the warnings.

Some filter parameters have been also renamed (`alignment` to `tileAlignment`, `overlap` to `tileOverlap`). When using the old names, warnings will be emitted at runtime.

3.2.7 Building as a Static Library

The support to build Open Image Denoise as a static library (`OIDN_STATIC_LIB` CMake option) has been limited to CPU-only builds due to switching to a modular library design that was necessary for adding multi-vendor GPU support. If the library is built with GPU support as well, the `OIDN_STATIC_LIB` option is still available but enabling it results in a hybrid static/shared library.

If the main reason for building as a static library would be is the ability to use multiple versions of Open Image Denoise in the same process, please use the existing `OIDN_API_NAMESPACE` CMake option instead. With this feature all symbols of the library will be put into a custom namespace, which can prevent symbol clashes.

3.3 Physical Devices

Systems often have multiple different types of devices supported by Open Image Denoise (CPUs and GPUs). The application can get the list of supported *physical devices* and select which of these to use for denoising.

The number of supported physical devices can be queried with

```
int oidnGetNumPhysicalDevices();
```

The physical devices can be identified using IDs between 0 and (`oidnGetNumPhysicalDevices()` − 1), and are ordered *approximately* from fastest to slowest (e.g., ID of 0 corresponds to the likely fastest physical device). Note that the reported number and order of physical devices may change between application runs, so no assumptions should be made about this list.

Parameters of these physical devices can be queried using

```
bool         oidnGetPhysicalDeviceBool  (int physicalDeviceID, const char* name);
int          oidnGetPhysicalDeviceInt   (int physicalDeviceID, const char* name);
unsigned int oidnGetPhysicalDeviceUInt  (int physicalDeviceID, const char* name);
const char*  oidnGetPhysicalDeviceString(int physicalDeviceID, const char* name);
const void*  oidnGetPhysicalDeviceData  (int physicalDeviceID, const char* name,
                                         size_t* byteSize);
```

where `name` is the name of the parameter, and `byteSize` is the number of returned bytes for data parameters. The following parameters can be queried:

It is also possible to directly query whether a physical device of a particular type is supported, without iterating over all supported physical devices:

Table 3.1 – Constant parameters supported by physical devices.

Type	Name	Description
`Int`	`type`	device type as an `OIDNDeviceType` value
`String`	`name`	name string
`Bool`	`uuidSupported`	device supports universally unique identifier (UUID)
`Data`	`uuid`	opaque UUID (`OIDN_UUID_SIZE` bytes, exists only if `uuidSupported` is `true`)
`Bool`	`luidSupported`	device supports locally unique identifier (UUID)
`Data`	`luid`	opaque LUID (`OIDN_LUID_SIZE` bytes, exists only if `luidSupported` is `true`)
`UInt`	`nodeMask`	bitfield identifying the node within a linked device adapter corresponding to the device (exists only if `luidSupported` is `true`)
`Bool`	`pciAddressSupported`	device supports PCI address
`Int`	`pciDomain`	PCI domain (exists only if `pciAddressSupported` is `true`)
`Int`	`pciBus`	PCI bus (exists only if `pciAddressSupported` is `true`)
`Int`	`pciDevice`	PCI device (exists only if `pciAddressSupported` is `true`)
`Int`	`pciFunction`	PCI function (exists only if `pciAddressSupported` is `true`)

```
bool oidnIsCPUDeviceSupported();
bool oidnIsSYCLDeviceSupported(const sycl::device* device);
bool oidnIsCUDADeviceSupported(int deviceID);
bool oidnIsHIPDeviceSupported(int deviceID);
bool oidnIsMetalDeviceSupported(MTLDevice_id device);
```

3.4 Devices

Open Image Denoise has a *logical* device concept as well, or simply referred to as *device*, which allows different components of the application to use the Open Image Denoise API without interfering with each other. Each physical device may be associated with one ore more logical devices. A basic way to create a device is by calling

```
OIDNDevice oidnNewDevice(OIDNDeviceType type);
```

where the `type` enumeration maps to a specific device implementation, which can be one of the following:

Table 3.2 – Supported device types, i.e., valid constants of type `OIDNDeviceType`.

Name	Description
`OIDN_DEVICE_TYPE_DEFAULT`	select the likely fastest device (same as physical device with ID 0)
`OIDN_DEVICE_TYPE_CPU`	CPU device
`OIDN_DEVICE_TYPE_SYCL`	SYCL device (requires a supported Intel GPU)
`OIDN_DEVICE_TYPE_CUDA`	CUDA device (requires a supported NVIDIA GPU)
`OIDN_DEVICE_TYPE_HIP`	HIP device (requires a supported AMD GPU)
`OIDN_DEVICE_TYPE_METAL`	Metal device (requires a supported Apple GPU)

If there are multiple supported devices of the specified type, an implementation-dependent default will be selected.

A device can be created by specifying a physical device ID as well using

```
OIDNDevice oidnNewDeviceByID(int physicalDeviceID);
```

Applications can manually iterate over the list of physical devices and select from them based on their properties but there are also some built-in helper functions as well, which make creating a device by a particular physical device property easier:

```
OIDNDevice oidnNewDeviceByUUID(const void* uuid);
OIDNDevice oidnNewDeviceByLUID(const void* luid);
OIDNDevice oidnNewDeviceByPCIAddress(int pciDomain, int pciBus, int pciDevice,
                                     int pciFunction);
```

These functions are particularly useful when the application needs interoperability with a graphics API (e.g. DX12, Vulkan). However, not all of these properties may be supported by the intended physical device (or drivers might even report inconsistent identifiers), so it is recommended to select by more than one property, if possible.

If the application requires interoperability with a particular compute or graphics API (SYCL, CUDA, HIP, Metal), it is recommended to use one of the following dedicated functions instead:

```
OIDNDevice oidnNewSYCLDevice(const sycl::queue* queues, int numQueues);
OIDNDevice oidnNewCUDADevice(const int* deviceIDs, const cudaStream_t* streams,
                             int numPairs);
OIDNDevice oidnNewHIPDevice(const int* deviceIDs, const hipStream_t* streams,
                            int numPairs);
OIDNDevice oidnNewMetalDevice(const MTLCommandQueue_id* commandQueues,
                              int numQueues);
```

For SYCL, it is possible to pass one or more SYCL queues which will be used by Open Image Denoise for all device operations. This is useful when the application wants to use the same queues for both denoising and its own operations (e.g. rendering). Passing multiple queues is not intended to be used for different physical devices but just for a single SYCL root-device which consists of multiple sub-devices (e.g. Intel® Data Center GPU Max Series having multiple Xe-Stacks/tiles). The only supported SYCL backend is oneAPI Level Zero.

For CUDA and HIP, pairs of CUDA/HIP device IDs and corresponding streams can be specified but the current implementation supports only one pair. A `NULL` stream corresponds to the default stream on the corresponding device. Open Image Denoise automatically sets and restores the current CUDA/HIP device/context on the calling thread when necessary, thus the current device does not have to be changed manually by the application.

For Metal, a single command queue is supported.

Once a device is created, you can call

```
bool oidnGetDeviceBool(OIDNDevice device, const char* name);
void oidnSetDeviceBool(OIDNDevice device, const char* name, bool value);
int  oidnGetDeviceInt (OIDNDevice device, const char* name);
void oidnSetDeviceInt (OIDNDevice device, const char* name, int  value);
int  oidnGetDeviceUInt(OIDNDevice device, const char* name);
void oidnSetDeviceUInt(OIDNDevice device, const char* name, unsigned int value);
```

to set and get parameter values on the device. Note that some parameters are constants, thus trying to set them is an error. See the tables below for the parameters supported by devices.

Note that the CPU device heavily relies on setting the thread affinities to achieve optimal performance, so it is highly recommended to leave this option enabled. However, this may interfere with the application if that also sets the

Table 3.3 – Parameters supported by all devices.

Type	Name	Default	Description
`Int`	`type`	*constant*	device type as an `OIDNDeviceType` value
`Int`	`version`	*constant*	combined version number (major.minor.patch) with two decimal digits per component
`Int`	`versionMajor`	*constant*	major version number
`Int`	`versionMinor`	*constant*	minor version number
`Int`	`versionPatch`	*constant*	patch version number
`Bool`	`systemMemorySupported`	*constant*	device can directly access memory allocated with the system allocator (e.g. `malloc`)
`Bool`	`managedMemorySupported`	*constant*	device supports buffers created with managed storage (`OIDN_STORAGE_MANAGED`)
`Int`	`externalMemoryTypes`	*constant*	bitfield of `OIDNExternalMemoryTypeFlag` values representing the external memory types supported by the device
`Int`	`verbose`	0	verbosity level of the console output between 0–4; when set to 0, no output is printed, when set to a higher level more output is printed

Table 3.4 – Additional parameters supported only by CPU devices.

Type	Name	Default	Description
`Int`	`numThreads`	0	maximum number of threads which the library should use; 0 will set it automatically to get the best performance
`Bool`	`setAffinity`	`true`	enables thread affinitization (pinning software threads to hardware threads) if it is necessary for achieving optimal performance

thread affinities, potentially causing performance degradation. In such cases, the recommended solution is to either disable setting the affinities in the application or in Open Image Denoise, or to always set/reset the affinities before/after each parallel region in the application (e.g., if using TBB, with `tbb::task_arena` and `tbb::task_scheduler_observer`).

Once parameters are set on the created device, the device must be committed with

```
void oidnCommitDevice(OIDNDevice device);
```

This device can then be used to construct further objects, such as buffers and filters. Note that a device can be committed only once during its lifetime.

Some functions may execute asynchronously with respect to the host. The names of these functions are suffixed with `Async`. Asynchronous operations are executed *in order* on the device but may not block on the host. Eventually, it is necessary to wait for all asynchronous operations to complete, which can be done by calling

```
void oidnSyncDevice(OIDNDevice device);
```

If any errors have occurred during asynchronous operations (e.g., cancellation through a progress monitor callback), those will be reported only when synchronization is triggered explicitly with `oidnSyncDevice` or implicitly with some other API call (e.g., `oidnExecuteFilter`, `oidnCommitFilter`).

Before the application exits, it should release all devices by invoking

```
void oidnReleaseDevice(OIDNDevice device);
```

Note that Open Image Denoise uses reference counting for all object types, so this function decreases the reference count of the device, and if the count reaches 0 the device will automatically get deleted. It is also possible to increase the reference count by calling

```
void oidnRetainDevice(OIDNDevice device);
```

An application should typically create only a single device object per physical device (one for *all* CPUs or one per GPU) as creation can be very expensive and additional device objects may incur a significant memory overhead. If required differently, it should only use a small number of device objects at any given time.

3.4.1 Error Handling

Each user thread has its own error code per device. If an error occurs when calling an API function, this error code is set to the occurred error if it stores no previous error. The currently stored error can be queried by the application via

```
OIDNError oidnGetDeviceError(OIDNDevice device, const char** outMessage);
```

where `outMessage` can be a pointer to a C string which will be set to a more descriptive error message, or it can be `NULL`. This function also clears the error code, which assures that the returned error code is always the first error occurred since the last invocation of `oidnGetDeviceError` on the current thread. Note that the optionally returned error message string is valid only until the next invocation of the function.

Alternatively, the application can also register a callback function of type

```
typedef void (*OIDNErrorFunction)(void* userPtr, OIDNError code, const char* message);
```

via

```
void oidnSetDeviceErrorFunction(OIDNDevice device, OIDNErrorFunction func, void* userPtr);
```

to get notified when errors occur. Only a single callback function can be registered per device, and further invocations overwrite the previously set callback function, which do *not* require also calling the `oidnCommitDevice` function. Passing `NULL` as function pointer disables the registered callback function. When the registered callback function is invoked, it gets passed the user-defined payload (`userPtr` argument as specified at registration time), the error code (`code` argument) of the occurred error, as well as a string (`message` argument) that further describes the error. The error code is always set even if an error callback function is registered. It is recommended to always set a error callback function, to detect all errors.

When the device construction fails, `oidnNewDevice` returns `NULL` as device. To detect the error code of a such failed device construction, pass `NULL` as device to the `oidnGetDeviceError` function. For all other invocations of `oidnGetDeviceError`, a proper device handle must be specified.

The following errors are currently used by Open Image Denoise:

3.4.2 Environment Variables

Open Image Denoise supports environment variables for overriding certain settings at runtime, which can be useful for debugging and development:

Table 3.5 – Possible error codes, i.e., valid constants of type `OIDNError`.

Name	Description
`OIDN_ERROR_NONE`	no error occurred
`OIDN_ERROR_UNKNOWN`	an unknown error occurred
`OIDN_ERROR_INVALID_ARGUMENT`	an invalid argument was specified
`OIDN_ERROR_INVALID_OPERATION`	the operation is not allowed
`OIDN_ERROR_OUT_OF_MEMORY`	not enough memory to execute the operation
`OIDN_ERROR_UNSUPPORTED_HARDWARE`	the hardware (CPU/GPU) is not supported
`OIDN_ERROR_CANCELLED`	the operation was cancelled by the user

Table 3.6 – Environment variables supported by Open Image Denoise.

Name	Description
`OIDN_DEFAULT_DEVICE`	overrides what physical device to use with `OIDN_DEVICE_TYPE_DEFAULT`; can be `cpu`, `sycl`, `cuda`, `hip`, or a physical device ID
`OIDN_DEVICE_CPU`	value of 0 disables CPU device support
`OIDN_DEVICE_SYCL`	value of 0 disables SYCL device support
`OIDN_DEVICE_CUDA`	value of 0 disables CUDA device support
`OIDN_DEVICE_HIP`	value of 0 disables HIP device support
`OIDN_DEVICE_METAL`	value of 0 disables Metal device support
`OIDN_NUM_THREADS`	overrides `numThreads` device parameter
`OIDN_SET_AFFINITY`	overrides `setAffinity` device parameter
`OIDN_NUM_SUBDEVICES`	overrides number of SYCL sub-devices to use (e.g. for Intel® Data Center GPU Max Series)
`OIDN_VERBOSE`	overrides `verbose` device parameter

3.5 Buffers

Image data can be passed to Open Image Denoise either via pointers to memory allocated and managed by the user or by creating buffer objects. Regardless of which method is used, the data must be allocated in a way that it is accessible by the device (either CPU or GPU). Using buffers is typically the preferred approach because this ensures that the allocation requirements are fulfilled regardless of device type. To create a new data buffer with memory allocated and owned by the device, use

```
OIDNBuffer oidnNewBuffer(OIDNDevice device, size_t byteSize);
```

The created buffer is bound to the specified device (`device` argument). The specified number of bytes (`byteSize`) are allocated at buffer construction time and deallocated when the buffer is destroyed. The memory is by default allocated as managed memory automatically migrated between host and device, if supported, or as pinned host memory otherwise.

If this default buffer allocation is not suitable, a buffer can be created with a manually specified storage mode as well:

```
OIDNBuffer oidnNewBufferWithStorage(OIDNDevice device, size_t byteSize, OIDNStorage storage);
```

The supported storage modes are the following:

Table 3.7 – Supported storage modes for buffers, i.e., valid constants of type `OIDNStorage`.

Name	Description
`OIDN_STORAGE_UNDEFINED`	undefined storage mode
`OIDN_STORAGE_HOST`	pinned host memory, accessible by both host and device
`OIDN_STORAGE_DEVICE`	device memory, *not* accessible by the host
`OIDN_STORAGE_MANAGED`	automatically migrated between host and device, accessible by both (*not* supported by all devices, `managedMemorySupported` device parameter must be checked before use)

Note that the host and device storage modes are supported by all devices but managed storage is an optional feature. Before using managed storage, the `managedMemorySupported` device parameter should be queried.

It is also possible to create a "shared" data buffer with memory allocated and managed by the user with

```
OIDNBuffer oidnNewSharedBuffer(OIDNDevice device, void* devPtr, size_t byteSize);
```

where `devPtr` points to user-managed device-accessible memory and `byteSize` is its size in bytes. At buffer construction time no buffer data is allocated, but the buffer data provided by the user is used. The buffer data must remain valid for as long as the buffer may be used, and the user is responsible to free the buffer data when no longer required. The user must also ensure that the memory is accessible to the device by using a supported allocation function (e.g., `sycl::malloc_device`, `cudaMalloc`, `hipMalloc`) and alignment (e.g., Metal requires the allocation to be page-aligned).

Buffers can be also imported from graphics APIs as external memory, to avoid expensive copying of data through host memory. Different types of external memory can be imported from either POSIX file descriptors or Win32 handles using

```
OIDNBuffer oidnNewSharedBufferFromFD(OIDNDevice device,
                                     OIDNExternalMemoryTypeFlag fdType,
                                     int fd, size_t byteSize);

OIDNBuffer oidnNewSharedBufferFromWin32Handle(OIDNDevice device,
                                              OIDNExternalMemoryTypeFlag handleType,
                                              void* handle, const void* name, size_t byteSize);
```

Before exporting memory from the graphics API, the application should find a handle type which is supported by both the Open Image Denoise device (see `externalMemoryTypes` device parameter) and the graphics API. Note that different GPU vendors may support different handle types. To ensure compatibility with all device types, applications should support at least `OIDN_EXTERNAL_MEMORY_TYPE_FLAG_OPAQUE_WIN32` on Windows and both `OIDN_EXTERNAL_MEMORY_TYPE_FLAG_OPAQUE_FD` and `OIDN_EXTERNAL_MEMORY_TYPE_FLAG_DMA_BUF` on Linux. All possible external memory types are listed in the table below.

Metal buffers can be imported directly with

```
OIDNBuffer oidnNewSharedBufferFromMetal(OIDNDevice device, MTLBuffer_id buffer);
```

Note that if a buffer with an `MTLStorageModeManaged` storage mode is imported, it is the responsibility of the user to synchronize the contents of the buffer between the host and the device.

Similar to device objects, buffer objects are also reference-counted and can be retained and released by calling the following functions:

Table 3.8 – Supported external memory type flags, i.e., valid constants of type `OIDNExternalMemoryTypeFlag`.

Name	Description
`OIDN_EXTERNAL_MEMORY_TYPE_FLAG_NONE`	
`OIDN_EXTERNAL_MEMORY_TYPE_FLAG_OPAQUE_FD`	opaque POSIX file descriptor handle (recommended on Linux)
`OIDN_EXTERNAL_MEMORY_TYPE_FLAG_DMA_BUF`	file descriptor handle for a Linux dma_buf (recommended on Linux)
`OIDN_EXTERNAL_MEMORY_TYPE_FLAG_OPAQUE_WIN32`	NT handle (recommended on Windows)
`OIDN_EXTERNAL_MEMORY_TYPE_FLAG_OPAQUE_WIN32_KMT`	global share (KMT) handle
`OIDN_EXTERNAL_MEMORY_TYPE_FLAG_D3D11_TEXTURE`	NT handle returned by `IDXGIResource1::CreateSharedHandle` referring to a Direct3D 11 texture resource
`OIDN_EXTERNAL_MEMORY_TYPE_FLAG_D3D11_TEXTURE_KMT`	global share (KMT) handle returned by `IDXGIResource::GetSharedHandle` referring to a Direct3D 11 texture resource
`OIDN_EXTERNAL_MEMORY_TYPE_FLAG_D3D11_RESOURCE`	NT handle returned by `IDXGIResource1::CreateSharedHandle` referring to a Direct3D 11 resource
`OIDN_EXTERNAL_MEMORY_TYPE_FLAG_D3D11_RESOURCE_KMT`	global share (KMT) handle returned by `IDXGIResource::GetSharedHandle` referring to a Direct3D 11 resource
`OIDN_EXTERNAL_MEMORY_TYPE_FLAG_D3D12_HEAP`	NT handle returned by `ID3D12Device::CreateSharedHandle` referring to a Direct3D 12 heap resource
`OIDN_EXTERNAL_MEMORY_TYPE_FLAG_D3D12_RESOURCE`	NT handle returned by `ID3D12Device::CreateSharedHandle` referring to a Direct3D 12 committed resource

```
void oidnRetainBuffer (OIDNBuffer buffer);
void oidnReleaseBuffer(OIDNBuffer buffer);
```

The size of in bytes and storage mode of the buffer can be queried using

```
size_t      oidnGetBufferSize   (OIDNBuffer buffer);
OIDNStorage oidnGetBufferStorage(OIDNBuffer buffer);
```

It is possible to get a pointer directly to the buffer data, which is usually the preferred way to access the data stored in the buffer:

```
void* oidnGetBufferData(OIDNBuffer buffer);
```

Accessing the data on the host through this pointer is possible *only* if the buffer was created with `OIDN_STORAGE_HOST` or `OIDN_STORAGE_MANAGED`. Note that a `NULL` pointer may be returned if the buffer is empty.

In some cases better performance can be achieved by using device storage for buffers. Such data can be accessed on the host by copying to/from host memory (including pageable system memory) using the following functions:

```
void oidnReadBuffer(OIDNBuffer buffer,
                    size_t byteOffset, size_t byteSize, void* dstHostPtr);
```

```
void oidnWriteBuffer(OIDNBuffer buffer,
                     size_t byteOffset, size_t byteSize, const void* srcHostPtr);
```

These functions will always block until the read/write operation has been completed, which is often suboptimal. The following functions execute these operations asynchronously:

```
void oidnReadBufferAsync(OIDNBuffer buffer,
                         size_t byteOffset, size_t byteSize, void* dstHostPtr);

void oidnWriteBufferAsync(OIDNBuffer buffer,
                          size_t byteOffset, size_t byteSize, const void* srcHostPtr);
```

When copying asynchronously, the user must ensure correct synchronization with the device by calling `oidnSyncDevice` before accessing the copied data or releasing the buffer. Failure to do so will result in undefined behavior.

3.5.1 Data Format

Buffers store opaque data and thus have no information about the type and format of the data. Other objects, e.g. filters, typically require specifying the format of the data stored in buffers or shared via pointers. This can be done using the `OIDNFormat` enumeration type:

Name	Description
`OIDN_FORMAT_UNDEFINED`	undefined format
`OIDN_FORMAT_FLOAT`	32-bit floating-point scalar
`OIDN_FORMAT_FLOAT[234]`	32-bit floating-point [234]-element vector
`OIDN_FORMAT_HALF`	16-bit floating-point scalar
`OIDN_FORMAT_HALF[234]`	16-bit floating-point [234]-element vector

Table 3.9 – Supported data formats, i.e., valid constants of type `OIDNFormat`.

3.6 Filters

Filters are the main objects in Open Image Denoise that are responsible for the actual denoising. The library ships with a collection of filters which are optimized for different types of images and use cases. To create a filter object, call

```
OIDNFilter oidnNewFilter(OIDNDevice device, const char* type);
```

where `type` is the name of the filter type to create. The supported filter types are documented later in this section.

Creating filter objects can be very expensive, therefore it is *strongly* recommended to reuse the same filter for denoising as many images as possible, as long as the these images have the same same size, format, and features (i.e., only the memory locations and pixel values may be different). Otherwise (e.g. for images with different resolutions), reusing the same filter would not have any benefits.

Once created, filter objects can be retained and released with

```
void oidnRetainFilter (OIDNFilter filter);
void oidnReleaseFilter(OIDNFilter filter);
```

After creating a filter, it needs to be set up by specifying the input and output images, and potentially setting other parameter values as well.

To set image parameters of a filter, you can use one of the following functions:

```
void oidnSetFilterImage(OIDNFilter filter, const char* name,
                        OIDNBuffer buffer, OIDNFormat format,
                        size_t width, size_t height,
                        size_t byteOffset,
                        size_t pixelByteStride, size_t rowByteStride);

void oidnSetSharedFilterImage(OIDNFilter filter, const char* name,
                              void* devPtr, OIDNFormat format,
                              size_t width, size_t height,
                              size_t byteOffset,
                              size_t pixelByteStride, size_t rowByteStride);
```

It is possible to specify either a data buffer object (`buffer` argument) with the `oidnSetFilterImage` function, or directly a pointer to user-managed device-accessible data (`devPtr` argument) with the `oidnSetSharedFilterImage` function. Regardless of whether a buffer or a pointer is specified, the data *must* be accessible to the device. The easiest way to guarantee this regardless of the device type (CPU or GPU) is using buffer objects.

In both cases, you must also specify the name of the image parameter to set (`name` argument, e.g. `"color"`, `"output"`), the pixel format (`format` argument), the width and height of the image in number of pixels (`width` and `height` arguments), the starting offset of the image data (`byteOffset` argument), the pixel stride (`pixelByteStride` argument) and the row stride (`rowByteStride` argument), in number of bytes.

If the pixels and/or rows are stored contiguously (tightly packed without any gaps), you can set `pixelByteStride` and/or `rowByteStride` to 0 to let the library compute the actual strides automatically, as a convenience.

Images support only `FLOAT` and `HALF` pixel formats with up to 3 channels. Custom image layouts with extra channels (e.g. alpha channel) or other data are supported as well by specifying a non-zero pixel stride. This way, expensive image layout conversion and copying can be avoided but the extra channels will be ignored by the filter. If these channels also need to be denoised, separate filters can be used.

To unset a previously set image parameter, returning it to a state as if it had not been set, call

```
void oidnRemoveFilterImage(OIDNFilter filter, const char* name);
```

Some special data used by filters are opaque/untyped (e.g. trained model weights blobs), which can be specified with the `oidnSetSharedFilterData` function:

```
void oidnSetSharedFilterData(OIDNFilter filter, const char* name,
                             void* hostPtr, size_t byteSize);
```

This data (`hostPtr`) must be accessible to the *host*, therefore system memory allocation is suitable (i.e., there is no reason to use buffer objects for allocation).

Modifying the contents of an opaque data parameter after setting it as a filter parameter is allowed but the filter needs to be notified that the data has been updated by calling

```
void oidnUpdateFilterData(OIDNFilter filter, const char* name);
```

Unsetting an opaque data parameter can be performed with

```
void oidnRemoveFilterData(OIDNFilter filter, const char* name);
```

Filters may have parameters other than buffers as well, which you can set and get using the following functions:

```
bool  oidnGetFilterBool (OIDNFilter filter, const char* name);
void  oidnSetFilterBool (OIDNFilter filter, const char* name, bool  value);
int   oidnGetFilterInt  (OIDNFilter filter, const char* name);
void  oidnSetFilterInt  (OIDNFilter filter, const char* name, int   value);
float oidnGetFilterFloat(OIDNFilter filter, const char* name);
void  oidnSetFilterFloat(OIDNFilter filter, const char* name, float value);
```

Filters support a progress monitor callback mechanism that can be used to report progress of filter operations and to cancel them as well. Calling `oidnSetFilterProgressMonitorFunction` registers a progress monitor callback function (`func` argument) with payload (`userPtr` argument) for the specified filter (`filter` argument):

```
typedef bool (*OIDNProgressMonitorFunction)(void* userPtr, double n);

void oidnSetFilterProgressMonitorFunction(OIDNFilter filter,
                                          OIDNProgressMonitorFunction func,
                                          void* userPtr);
```

Only a single callback function can be registered per filter, and further invocations overwrite the previously set callback function. Passing `NULL` as function pointer disables the registered callback function. Once registered, Open Image Denoise will invoke the callback function multiple times during filter operations, by passing the payload as set at registration time (`userPtr` argument), and a `double` in the range $[0, 1]$ which estimates the progress of the operation (`n` argument). When returning `true` from the callback function, Open Image Denoise will continue the filter operation normally. When returning `false`, the library will attempt to cancel the filter operation as soon as possible, and if that is fulfilled, it will raise an `OIDN_ERROR_CANCELLED` error. Note that cancellation is not guaranteed.

Using a progress monitor callback function introduces some overhead, which may be significant on GPU devices, hurting performance. Therefore we strongly recommend progress monitoring only for offline denoising, when denoising an image is expected to take several seconds.

After setting all necessary parameters for the filter, the changes must be committed by calling

```
void oidnCommitFilter(OIDNFilter filter);
```

The parameters can be updated after committing the filter, but it must be recommitted for any new changes to take effect. Committing major changes to the filter (e.g. setting new image parameters, changing the image resolution) can be expensive, and thus should not be done frequently (e.g. per frame).

Finally, an image can be filtered by executing the filter with

```
void oidnExecuteFilter(OIDNFilter filter);
```

which will read the input image data from the specified buffers and produce the denoised output image.

This function will always block until the filtering operation has been completed. The following function executes the operation asynchronously:

```
void oidnExecuteFilterAsync(OIDNFilter filter);
```

For filters created on a SYCL device it is also possible to specify dependent SYCL events (`depEvents` and `numDepEvents` arguments, may be `NULL`/0) and get a completion event as well (`doneEvent` argument, may be `NULL`):

```
void oidnExecuteSYCLFilterAsync(OIDNFilter filter,
                                const sycl::event* depEvents, int numDepEvents,
                                sycl::event* doneEvent);
```

When filtering asynchronously, the user must ensure correct synchronization with the device by calling `oidnSyncDevice` before accessing the output image data or releasing the filter. Failure to do so will result in undefined behavior.

In the following we describe the different filters that are currently implemented in Open Image Denoise.

3.6.1 RT

The `RT` (**r**ay **t**racing) filter is a generic ray tracing denoising filter which is suitable for denoising images rendered with Monte Carlo ray tracing methods like unidirectional and bidirectional path tracing. It supports depth of field and motion blur as well, but it is *not* temporally stable. The filter is based on a convolutional neural network (CNN) and comes with a set of pre-trained models that work well with a wide range of ray tracing based renderers and noise levels.

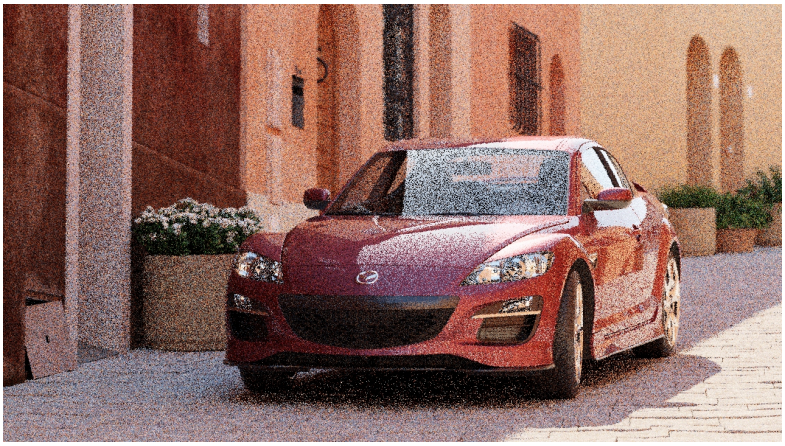

Figure 3.1 – Example noisy beauty image rendered using unidirectional path tracing (4 samples per pixel). *Scene by Evermotion.*

Figure 3.2 – Example output beauty image denoised using prefiltered auxiliary feature images (albedo and normal) too.

For denoising *beauty* images, it accepts either a low dynamic range (LDR) or high dynamic range (HDR) image (`color`) as the main input image. In addition to this, it also accepts *auxiliary feature* images, `albedo` and `normal`, which are optional inputs that usually improve the denoising quality significantly, preserving more details.

It is possible to denoise auxiliary images as well, in which case only the respective auxiliary image has to be specified as input, instead of the beauty image.

This can be done as a *prefiltering* step to further improve the quality of the denoised beauty image.

The `RT` filter has certain limitations regarding the supported input images. Most notably, it cannot denoise images that were not rendered with ray tracing. Another important limitation is related to anti-aliasing filters. Most renderers use a high-quality pixel reconstruction filter instead of a trivial box filter to minimize aliasing artifacts (e.g. Gaussian, Blackman-Harris). The `RT` filter does support such pixel filters but only if implemented with importance sampling. Weighted pixel sampling (sometimes called *splatting*) introduces correlation between neighboring pixels, which causes the denoising to fail (the noise will not be filtered), thus it is not supported.

The filter can be created by passing `"RT"` to the `oidnNewFilter` function as the filter type. The filter supports the parameters listed in the table below. All specified images must have the same dimensions. The output image can be one of the input images (i.e. in-place denoising is supported). See section Examples for simple code snippets that demonstrate the usage of the filter.

Using auxiliary feature images like albedo and normal helps preserving fine details and textures in the image thus can significantly improve denoising quality. These images should typically contain feature values for the first hit (i.e. the surface which is directly visible) per pixel. This works well for most surfaces but does not provide any benefits for reflections and objects visible through transparent surfaces (compared to just using the color as input). However, this issue can be usually fixed by storing feature values for a subsequent hit (i.e. the reflection and/or refraction) instead of the first hit. For example, it usually works well to follow perfect specular (*delta*) paths and store features for the first diffuse or glossy surface hit instead (e.g. for perfect specular dielectrics and mirrors). This can greatly improve the quality of reflections and transmission. We will describe this approach in more detail in the following subsections.

The auxiliary feature images should be as noise-free as possible. It is not a strict requirement but too much noise in the feature images may cause residual noise in the output. Ideally, these should be completely noise-free. If this is the case, this should be hinted to the filter using the `cleanAux` parameter to ensure the highest possible image quality. But this parameter should be used with care: if enabled, any noise present in the auxiliary images will end up in the denoised image as well, as residual noise. Thus, `cleanAux` should be enabled only if the auxiliary images are guaranteed to be noise-free.

Usually it is difficult to provide clean feature images, and some residual noise might be present in the output even with `cleanAux` being disabled. To eliminate this noise and to even improve the sharpness of texture details, the auxiliary images should be first denoised in a prefiltering step, as mentioned earlier. Then, these denoised auxiliary images could be used for denoising the beauty image. Since these are now noise-free, the `cleanAux` parameter should be enabled. See section Denoising with prefiltering (C++11 API) for a simple code example. Prefiltering makes denoising much more expensive but if there are multiple color AOVs to denoise, the prefiltered auxiliary images can be reused for denoising multiple AOVs, amortizing the cost of the prefiltering step.

Thus, for final-frame denoising, where the best possible image quality is required, it is recommended to prefilter the auxiliary features if they are noisy and enable the `cleanAux` parameter. Denoising with noisy auxiliary features should be reserved for previews and interactive rendering.

All auxiliary images should use the same pixel reconstruction filter as the beauty image. Using a properly anti-aliased beauty image but aliased albedo or normal images will likely introduce artifacts around edges.

Table 3.10 – Parameters supported by the RT filter.

Type	Name	Default	Description
Image	color	*optional*	input beauty image (1–3 channels, LDR values in [0, 1] or HDR values in [0, $+\infty$), values being interpreted such that, after scaling with the inputScale parameter, a value of 1 corresponds to a luminance level of 100 cd/m^2)
Image	albedo	*optional*	input auxiliary image containing the albedo per pixel (1–3 channels, values in [0, 1])
Image	normal	*optional*	input auxiliary image containing the shading normal per pixel (1–3 channels, world-space or view-space vectors with arbitrary length, values in [-1, 1])
Image	output	*required*	output image (1–3 channels); can be one of the input images
Bool	hdr	false	the main input image is HDR
Bool	srgb	false	the main input image is encoded with the sRGB (or 2.2 gamma) curve (LDR only) or is linear; the output will be encoded with the same curve
Float	inputScale	NaN	scales values in the main input image before filtering, without scaling the output too, which can be used to map color or auxiliary feature values to the expected range, e.g. for mapping HDR values to physical units (which affects the quality of the output but *not* the range of the output values); if set to NaN, the scale is computed implicitly for HDR images or set to 1 otherwise
Bool	cleanAux	false	the auxiliary feature (albedo, normal) images are noise-free; recommended for highest quality but should *not* be enabled for noisy auxiliary images to avoid residual noise
Int	quality	high	image quality mode as an OIDNQuality value
Data	weights	*optional*	trained model weights blob
Int	maxMemoryMB	-1	if set to >= 0, a request is made to limit the memory usage below the specified amount in megabytes at the potential cost of slower performance, but actual memory usage may be higher (the target may not be achievable or there may be additional allocations beyond the control of the library); otherwise, memory usage will be limited to an unspecified device-dependent amount; in both cases, filters on the same device share almost all of their allocated memory to minimize total memory usage
Int	tileAlignment	*constant*	when manually denoising in tiles, the tile size and offsets should be multiples of this amount of pixels to avoid artifacts; when denoising HDR images inputScale *must* be set by the user to avoid seam artifacts
Int	tileOverlap	*constant*	when manually denoising in tiles, the tiles should overlap by this amount of pixels

3.6.1.1 Albedos

The albedo image is the feature image that usually provides the biggest quality improvement. It should contain the approximate color of the surfaces independent of illumination and viewing angle.

For simple matte surfaces this means using the diffuse color/texture as the albedo. For other, more complex surfaces it is not always obvious what is the best way to compute the albedo, but the denoising filter is flexible to a certain extent and works well with differently computed albedos. Thus it is not necessary to compute the strict, exact albedo values but must be always between 0 and 1.

For metallic surfaces the albedo should be either the reflectivity at normal incidence (e.g. from the artist friendly metallic Fresnel model) or the average reflectivity; or if these are constant (not textured) or unknown, the albedo can be simply 1 as well.

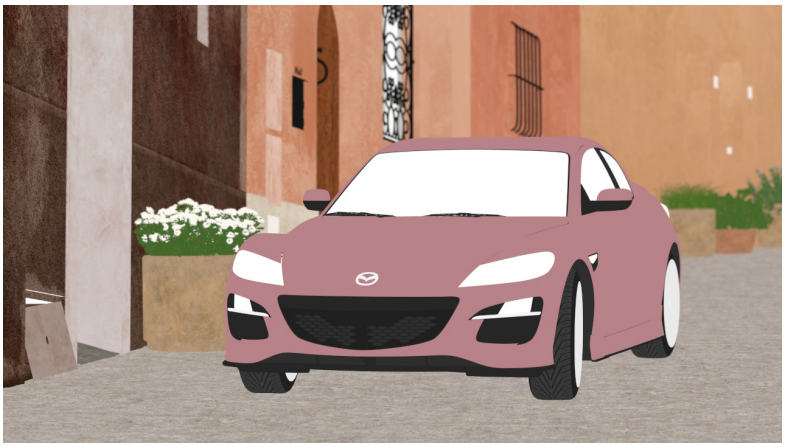

Figure 3.3 – Example albedo image obtained using the first hit. Note that the albedos of all transparent surfaces are 1.

Figure 3.4 – Example albedo image obtained using the first diffuse or glossy (non-delta) hit. Note that the albedos of perfect specular (delta) transparent surfaces are computed as the Fresnel blend of the reflected and transmitted albedos.

The albedo for dielectric surfaces (e.g. glass) should be either 1 or, if the surface is perfect specular (i.e. has a delta BSDF), the Fresnel blend of the reflected and transmitted albedos. The latter usually works better but only if it does not introduce too much noise or the albedo is prefiltered. If noise is an issue, we recommend to split the path into a reflected and a transmitted path at the first hit, and perhaps fall back to an albedo of 1 for subsequent dielectric hits. The reflected albedo in itself can be used for mirror-like surfaces as well.

The albedo for layered surfaces can be computed as the weighted sum of the albedos of the individual layers. Non-absorbing clear coat layers can be simply ignored (or the albedo of the perfect specular reflection can be used as well) but absorption should be taken into account.

3.6.1.2 Normals

The normal image should contain the shading normals of the surfaces either in world-space or view-space. It is recommended to include normal maps to preserve as much detail as possible.

Just like any other input image, the normal image should be anti-aliased (i.e. by accumulating the normalized normals per pixel). The final accumulated normals do not have to be normalized but must be in the [-1, 1] range (i.e. normals mapped to [0, 1] are *not* acceptable and must be remapped to [−1, 1]).

Similar to the albedo, the normal can be stored for either the first or a subsequent hit (if the first hit has a perfect specular/delta BSDF).

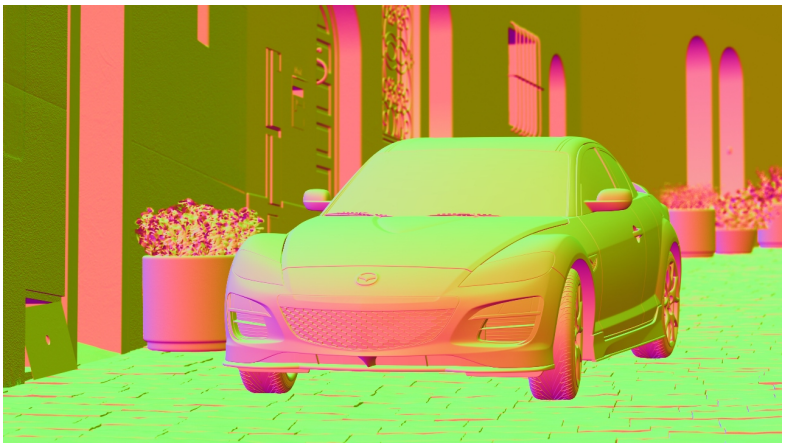

Figure 3.5 – Example normal image obtained using the first hit (the values are actually in [−1, 1] but were mapped to [0, 1] for illustration purposes).

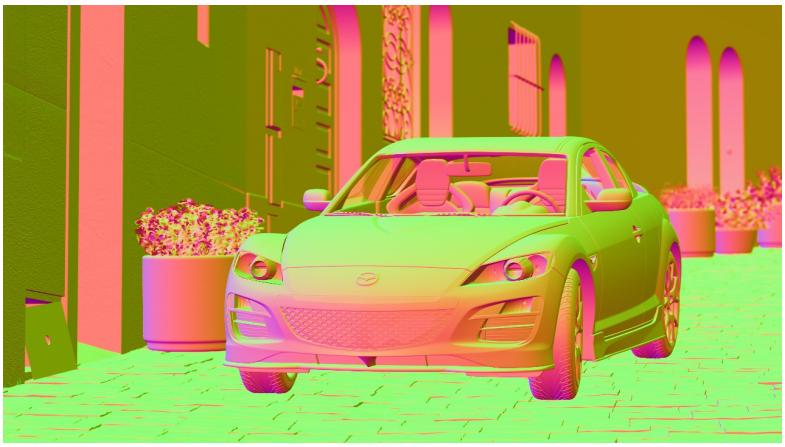

Figure 3.6 – Example normal image obtained using the first diffuse or glossy (non-delta) hit. Note that the normals of perfect specular (delta) transparent surfaces are computed as the Fresnel blend of the reflected and transmitted normals.

3.6.1.3 Quality

The filter supports setting an image quality mode, which determines whether to favor quality, performance, or have a balanced solution between the two. The supported quality modes are listed in the following table.

Name	Description
`OIDN_QUALITY_DEFAULT`	default quality
`OIDN_QUALITY_FAST`	high performance (for interactive/real-time preview rendering)
`OIDN_QUALITY_BALANCED`	balanced quality/performance (for interactive/real-time rendering)
`OIDN_QUALITY_HIGH`	high quality (for final-frame rendering); *default*

Table 3.11 – Supported image quality modes, i.e., valid constants of type `OIDNQuality`.

By default, filtering is performed in *high* quality mode, which is recommended for final-frame rendering. Using this setting the results have the same high quality regardless of what kind of device (CPU or GPU) is used. However, due to significant hardware architecture differences between devices, there might be small numerical differences between the produced outputs.

The *balanced* quality mode may provide somewhat lower image quality but higher performance and lower default memory usage, and is thus recommended for interactive and real-time rendering. For even higher performance and lower memory usage, a *fast* quality mode is also available but has noticeably lower image quality, making it suitable mainly for fast previews. Note that in the *bal-*

anced and *fast* quality modes larger numerical differences should be expected across devices compared to the *high* quality mode.

The difference in quality and performance between quality modes depends on the combination of input features, parameters (e.g. `cleanAux`), and the device architecture. In some cases the difference may be small or even none.

3.6.1.4 Weights

Instead of using the built-in trained models for filtering, it is also possible to specify user-trained models at runtime. This can be achieved by passing the model *weights* blob corresponding to the specified set of features and other filter parameters, produced by the included training tool. See Section Training for details.

3.6.2 RTLightmap

The `RTLightmap` filter is a variant of the `RT` filter optimized for denoising HDR and normalized directional (e.g. spherical harmonics) lightmaps. It does not support LDR images.

The filter can be created by passing `"RTLightmap"` to the `oidnNewFilter` function as the filter type. The filter supports the following parameters:

Table 3.12 – Parameters supported by the `RTLightmap` filter.

Type	Name	Default	Description
`Image`	`color`	*required*	input beauty image (1–3 channels, HDR values in [0, $+\infty$), interpreted such that, after scaling with the `inputScale` parameter, a value of 1 corresponds to a luminance level of 100 cd/m^2; directional values in [-1, 1])
`Image`	`output`	*required*	output image (1–3 channels); can be one of the input images
`Bool`	`directional`	`false`	whether the input contains normalized coefficients (in [-1, 1]) of a directional lightmap (e.g. normalized L1 or higher spherical harmonics band with the L0 band divided out); if the range of the coefficients is different from [-1, 1], the `inputScale` parameter can be used to adjust the range without changing the stored values
`Float`	`inputScale`	NaN	scales input color values before filtering, without scaling the output too, which can be used to map color values to the expected range, e.g. for mapping HDR values to physical units (which affects the quality of the output but *not* the range of the output values); if set to NaN, the scale is computed implicitly for HDR images or set to 1 otherwise
`Int`	`quality`	high	image quality mode as an `OIDNQuality` value
`Data`	`weights`	*optional*	trained model weights blob
`Int`	`maxMemoryMB`	-1	if set to >= 0, a request is made to limit the memory usage below the specified amount in megabytes at the potential cost of slower performance, but actual memory usage may be higher (the target may not be achievable or there may be additional allocations beyond the control of the library); otherwise, memory usage will be limited to an unspecified device-dependent amount; in both cases, filters on the same device share almost all of their allocated memory to minimize total memory usage
`Int`	`tileAlignment`	*constant*	when manually denoising in tiles, the tile size and offsets should be multiples of this amount of pixels to avoid artifacts; when denoising HDR images `inputScale` *must* be set by the user to avoid seam artifacts
`Int`	`tileOverlap`	*constant*	when manually denoising in tiles, the tiles should overlap by this amount of pixels

Chapter 4
Examples

Intel Open Image Denoise ships with a couple of simple example applications.

4.1 oidnDenoise

`oidnDenoise` is a minimal working example demonstrating how to use Intel Open Image Denoise, which can be found at `apps/oidnDenoise.cpp`. It uses the C++11 convenience wrappers of the C99 API.

This example is a simple command-line application that denoises the provided image, which can optionally have auxiliary feature images as well (e.g. albedo and normal). By default the images must be stored in the Portable FloatMap (PFM) format, and the color values must be encoded in little-endian format. To enable other image formats (e.g. OpenEXR, PNG) as well, the project has to be rebuilt with OpenImageIO support enabled.

Running `oidnDenoise` without any arguments or the `-h` argument will bring up a list of command-line options.

4.2 oidnBenchmark

`oidnBenchmark` is a basic command-line benchmarking application for measuring denoising speed, which can be found at `apps/oidnBenchmark.cpp`.

Running `oidnBenchmark` with the `-h` argument will bring up a list of command-line options.

Chapter 5
Training

The Intel Open Image Denoise source distribution includes a Python-based neural network training toolkit (located in the `training` directory), which can be used to train the denoising filter models with image datasets provided by the user. This is an advanced feature of the library which usage requires some background knowledge of machine learning and basic familiarity with deep learning frameworks and toolkits (e.g. PyTorch or TensorFlow, TensorBoard).

The training toolkit consists of the following command-line scripts:

- `preprocess.py`: Preprocesses training and validation datasets.
- `train.py`: Trains a model using preprocessed datasets.
- `infer.py`: Performs inference on a dataset using the specified training result.
- `export.py`: Exports a training result to the runtime model weights format.
- `find_lr.py`: Tool for finding the optimal minimum and maximum learning rates.
- `visualize.py`: Invokes TensorBoard for visualizing statistics of a training result.
- `split_exr.py`: Splits a multi-channel EXR image into multiple feature images.
- `convert_image.py`: Converts a feature image to a different image format.
- `compare_image.py`: Compares two feature images using the specified quality metrics.

5.1 Prerequisites

Before you can run the training toolkit you need the following prerequisites:

- Linux (other operating systems are currently not supported)
- Python 3.7 or later
- PyTorch 2.4 or later
- NumPy 1.19 or later
- OpenImageIO 2.1 or later
- TensorBoard 2.4 or later

5.2 Devices

Most scripts in the training toolkit support selecting what kind of device (e.g. CPU, GPU) to use for the computations (`--device` or `-d` option). If multiple devices of the same kind are available (e.g. multiple GPUs), the user can specify which one of these to use (`--device_id` or `-k` option). Additionally, some scripts, like `train.py`, support data-parallel execution on multiple devices for faster performance (`--num_devices` or `-n` option).

5.3 Datasets

A dataset should consist of a collection of noisy and corresponding noise-free reference images. It is possible to have more than one noisy version of the same image in the dataset, e.g. rendered at different samples per pixel and/or using different seeds.

The training toolkit expects to have all datasets (e.g. training, validation) in the same parent directory (e.g. `data`). Each dataset is stored in its own subdirectory (e.g. `train`, `valid`), which can have an arbitrary name.

The images must be stored in OpenEXR format (`.exr` files), and the filenames must have a specific format but the files can be stored in an arbitrary directory structure inside the dataset directory. The only restriction is that all versions of an image (noisy images and the reference image) must be located in the same subdirectory. Each feature of an image (e.g. color, albedo) must be stored in a separate image file, i.e. multi-channel EXR image files are not supported. If you have multi-channel EXRs, you can split them into separate images per feature using the included `split_exr.py` tool.

An image filename must consist of a base name, a suffix with the number of samples per pixel or whether it is the reference image (e.g. `_0128spp`, `_ref`), the feature type extension (e.g. `.hdr`, `.alb`), and the image format extension (`.exr`). The exact filename format as a regular expression is the following:

```
.+_([0-9]+(spp)?|ref|reference|gt|target)\.(hdr|ldr|sh1[xyz]|alb|nrm)\.exr
```

The number of samples per pixel should be padded with leading zeros to have a fixed number of digits. If the reference image is not explicitly named as such (i.e. has the number of samples instead), the image with the most samples per pixel will be considered the reference.

The following image features are supported:

Feature	Description	Channels	File extension
`hdr`	color (HDR)	3	`.hdr.exr`
`ldr`	color (LDR)	3	`.ldr.exr`
`sh1`	color (normalized L1 spherical harmonics)	3 × 3 images	`.sh1x.exr`, `.sh1y.exr`, `.sh1z.exr`
`alb`	albedo	3	`.alb.exr`
`nrm`	normal	3	`.nrm.exr`

Table 5.1 – Image features supported by the training toolkit.

The following directory tree demonstrates an example root dataset directory (`data`) containing one dataset (`rt_train`) with HDR color and albedo feature images:

```
data
`-- rt_train
    |-- scene1
```

```
|   |-- view1_0001.alb.exr
|   |-- view1_0001.hdr.exr
|   |-- view1_0004.alb.exr
|   |-- view1_0004.hdr.exr
|   |-- view1_8192.alb.exr
|   |-- view1_8192.hdr.exr
|   |-- view2_0001.alb.exr
|   |-- view2_0001.hdr.exr
|   |-- view2_8192.alb.exr
|   `-- view2_8192.hdr.exr
|-- scene2_000008spp.alb.exr
|-- scene2_000008spp.hdr.exr
|-- scene2_000064spp.alb.exr
|-- scene2_000064spp.hdr.exr
|-- scene2_reference.alb.exr
`-- scene2_reference.hdr.exr
```

5.4 Preprocessing (preprocess.py)

Training and validation datasets can be used only after preprocessing them using the `preprocess.py` script. This will convert the specified training (`--train_data` or `-t` option) and validation datasets (`--valid_data` or `-v` option) located in the root dataset directory (`--data_dir` or `-D` option) to a format that can be loaded more efficiently during training. All preprocessed datasets will be stored in a root preprocessed dataset directory (`--preproc_dir` or `-P` option).

The preprocessing script requires the set of image features to include in the preprocessed dataset as command-line arguments. Only these specified features will be available for training but it is not required to use all of them at the same time. Thus, a single preprocessed dataset can be reused for training multiple models with different combinations of the preprocessed features.

By default, all input features are assumed to be noisy, including the auxiliary features (e.g. albedo, normal), each having versions at different samples per pixel. It is also possible to train with noise-free auxiliary features, in which case the reference auxiliary features are used instead of the various noisy ones (`--clean_aux` option). This improves quality significantly if the auxiliary features used for inference will be either originally noise-free or prefiltered with separately trained auxiliary feature denoising models. If inference will be done only with prefiltered features, even higher quality can be achieved by training with prefiltered features instead of the reference onces. This can be achieved by first training the auxiliary feature models and then specifying the list of these results when preprocessing the dataset for the main feature (`--aux_results` or `-a` option).

Preprocessing also depends on the filter that will be trained (e.g. determines which HDR/LDR transfer function has to be used), which should be also specified (`--filter` or `-f` option). The alternative is to manually specify the transfer function (`--transfer` or `-x` option) and other filter-specific parameters, which could be useful for training custom filters.

For example, to preprocess the training and validation datasets (`rt_train` and `rt_valid`) with HDR color, albedo, and normal image features, for training the `RT` filter, the following command can be used:

```
./preprocess.py hdr alb nrm --filter RT --train_data rt_train --valid_data rt_valid
```

It is possible to preprocess the same dataset multiple times, with possibly different combinations of features and options. The training script will use the most suitable and most recent preprocessed version depending on the training parameters.

For more details about using the preprocessing script, including other options, please have a look at the help message:

```
./preprocess.py -h
```

5.5 Training (train.py)

The filters require separate trained models for each supported combination of input features. Thus, depending on which combinations of features the user wants to support for a particular filter, one or more models have to be trained.

After preprocessing the datasets, it is possible to start training a model using the `train.py` script. Similar to the preprocessing script, the input features must be specified (could be a subset of the preprocessed features), and the dataset names, directory paths, and the filter can be also passed. If the `--clean_aux` or `--aux_results` options were specified for preprocessing, these must be passed identically to the training script as well.

Open Image Denoise uses models of different sizes for different quality modes (high, balanced, fast). Specifying the quality mode (`--quality` or `-q` option) will cause the model to be implicitly selected, or the model can be specified explicitly as well (`--model` or `-m` option).

The tool will produce a training *result*, the name of which can be either specified (`--result` or `-r` option) or automatically generated (by default). Each result is stored in its own subdirectory, and these are located in a common parent directory (`--results_dir` or `-R` option). If a training result already exists, the tool will resume training that result from the latest checkpoint.

The default training hyperparameters should work reasonably well in general, but some adjustments might be necessary for certain datasets to attain optimal performance, most importantly: the number of epochs (`--num_epochs` or `-e` option), the global mini-batch size (`--batch_size` or `-b` option), and the learning rate. The training tool uses a one-cycle learning rate schedule with cosine annealing, which can be configured by setting the base learning rate (`--learning_rate` or `--lr` option), the maximum learning rate (`--max_learning_rate` or `--max_lr` option), and the percentage of the cycle spent increasing the learning rate (`--learning_rate_warmup` or `--lr_warmup` option).

Example usage:

```
./train.py hdr alb --filter RT --train_data rt_train --valid_data rt_valid --result rt_hdr_alb
```

For finding the optimal learning rate range, we recommend using the included `find_lr.py` script, which trains one epoch using an increasing learning rate and logs the resulting losses in a comma-separated values (CSV) file. Plotting the loss curve can show when the model starts to learn (the base learning rate) and when it starts to diverge (the maximum learning rate).

The model is evaluated with the validation dataset at regular intervals (`--num_valid_epochs` option), and checkpoints are also regularly created (`--num_save_epochs` option) to save training progress. Also, some statistics are logged (e.g. training and validation losses, learning rate) per epoch, which can be later visualized with TensorBoard by running the `visualize.py` script, e.g.:

```
./visualize.py --result rt_hdr_alb
```

Training is performed with mixed precision (FP16 and FP32) by default, if it supported by the hardware, which makes training faster and use less memory. However, in some rare cases this might cause some convergence issues. The training precision can be manually set to FP32 if necessary (`--precision` or `-p` option).

5.6 Inference (infer.py)

A training result can be tested by performing inference on an image dataset (`--input_data` or `-i` option) using the `infer.py` script. The dataset does *not* have to be preprocessed. In addition to the result to use, it is possible to specify which checkpoint to load as well (`-e` or `--num_epochs` option). By default the latest checkpoint is loaded.

The tool saves the output images in a separate directory (`--output_dir` or `-O` option) in the requested formats (`--format` or `-F` option). It also evaluates a set of image quality metrics (`--metric` or `-M` option), e.g. PSNR, SSIM, for images that have reference images available. All metrics are computed in tonemapped non-linear sRGB space. Thus, HDR images are first tonemapped (with Naughty Dog's Filmic Tonemapper from John Hable's *Uncharted 2: HDR Lighting* presentation) and converted to sRGB before evaluating the metrics.

Example usage:

```
./infer.py --result rt_hdr_alb --input_data rt_test --format exr png --metric ssim
```

The inference tool supports prefiltering of auxiliary features as well, which can be performed by specifying the list of training results for each feature to prefilter (`--aux_results` or `-a` option).

5.7 Exporting Results (export.py)

The training result produced by the `train.py` script cannot be immediately used by the main library. It has to be first exported to the runtime model weights format, a *Tensor Archive* (TZA) file. Running the `export.py` script for a training result (and optionally a checkpoint epoch) will create a binary `.tza` file in the directory of the result, which can be either used at runtime through the API or it can be included in the library build by replacing one of the built-in weights files.

Example usage:

```
./export.py --result rt_hdr_alb
```

5.8 Image Conversion and Comparison

In addition to the already mentioned `split_exr.py` script, the toolkit contains a few other image utilities as well.

`convert_image.py` converts a feature image to a different image format (and/or a different feature, e.g. HDR color to LDR), performing tonemapping and other transforms as well if needed. For HDR images the exposure can be adjusted by passing a linear exposure scale (`--exposure` or `-E` option). Example usage:

```
./convert_image.py view1_0004.hdr.exr view1_0004.png --exposure 2.5
```

The `compare_image.py` script compares two feature images (preferably having the dataset filename format to correctly detect the feature) using the specified image quality metrics, similar to the `infer.py` tool. Example usage:

```
./compare_image.py view1_0004.hdr.exr view1_8192.hdr.exr --exposure 2.5 --metric mse ssim
```