Build from Source

Contents

- Prerequisites
- Step 1. Download the Source Code
- Step 2. Build the Library
- Step 3. (Optional) Validate the Build
- Step 4. (Optional) Build Documentation
- Step 5. Install the Library

You can build and install the oneDNN library using the source distribution.

Prerequisites

Ensure that all software dependencies are in place and have at least the minimal supported version.

Step 1. Download the Source Code

Download oneDNN source code or clone the repository.

```
git clone https://github.com/uxlfoundation/oneDNN.git
cd oneDNN
```

Step 2. Build the Library

You can quickly get started with building the library.

The general steps involved in building the library are as follows:

1. Set up the environment for the compiler

   Configure your operating system's environment variables to point to the compiler's location.

2. Generate the build system

   The oneDNN build system is based on CMake. Use the following command to generate a build system:

   ```
   mkdir -p build ; cd build
   cmake .. [<options>]
   ```

   Note

   You can use `cmake -B <path-to-build> [-S <path-to-source>] [<options>]` to specify the following:

   - `-B <path-to-build>` : Specify the path where the build files will be generated.
   - `-S <path-to-source>` : Specify the path to the source directory containing the source files, dependencies, compiler options etc.

   The following are a few useful options defined by CMake:

   - `-G <generator-name>` to specify build system generator (e.g. `"Visual Studio 17 2022"`, `Ninja`, `"Unix Makefiles"`).
   - `-DCMAKE_INSTALL_PREFIX=<path>` to control the library installation location.
   - `-DCMAKE_BUILD_TYPE=<build-type>` to select between build type (`Release`, `Debug`, `RelWithDebInfo`).
   - `-DCMAKE_PREFIX_PATH=<path>` to specify directories to be searched for the dependencies located at non-standard locations.

   See Use Build Options for detailed description of build-time configuration options defined by oneDNN.

3. Build the library

   CMake provides a unified method for building a project, independent of the generator or operating system used.

Multi-threaded compilation is recommended for a faster build process. Use the `--parallel` option to specify the number of parallel jobs.

```
cmake --build <path-to-build> --parallel <jobs> [<options>]
```

Full list of options can be found here.

You can build the library on Linux, macOS, or Windows using the compiler of your choice.

Build on Linux and macOS

Use GCC, Clang, or Intel oneAPI DPC++/C++ Compiler

1. Set up the environment for the compiler

```
# Uncomment the following lines to build with GCC
# export CC=gcc
# export CXX=g++

# Uncomment the following lines to build with Clang
# export CC=clang
# export CXX=clang++

# Uncomment the following lines to build with Intel oneAPI DPC++/C++ Compiler (x64
# export CC=icx
# export CXX=icpx
```

2. Generate the build system

```
mkdir -p build ; cd build
cmake ..
```

3. Build the library

   For Linux:

```
cmake --build . --parallel $(nproc)
```

   For macOS:

```
cmake --build . --parallel $(sysctl -n hw.ncpu)
```

Use Intel oneAPI DPC++/C++ Compiler with SYCL runtime

1. Set up the environment for the compiler

   Intel oneAPI DPC++/C++ Compiler uses the `setvars.sh` script to set all the required variables. The command below assumes you installed the compiler to the default folder. If you customized the installation folder, `setvars.sh` (Linux/macOS) is in your custom folder.

```
source /opt/intel/oneapi/setvars.sh

# Set Intel oneAPI DPC++/C++ Compiler as default C and C++ compilers
export CC=icx
export CXX=icpx
```

2. Generate the build system

```
mkdir -p build ; cd build
cmake .. -DONEDNN_CPU_RUNTIME=SYCL \
         -DONEDNN_GPU_RUNTIME=SYCL
```

> **Note**
>
> Open-source version of oneAPI DPC++ Compiler does not have the icx driver, use clang/clang++ instead. Open-source version of oneAPI DPC++ Compiler may not contain OpenCL runtime. In this case, you can use `OPENCLROOT` CMake option or environment variable of the same name to specify path to the OpenCL runtime if it is installed in a custom location.

3. Build the library

   For Linux:

```
cmake --build . --parallel $(nproc)
```

   For macOS:

```
cmake --build . --parallel $(sysctl -n hw.ncpu)
```

Use GCC targeting AArch64 on x64 host

1. Set up the environment for the compiler

```
export CC=aarch64-linux-gnu-gcc
export CXX=aarch64-linux-gnu-g++
```

2. Generate the build system

```
mkdir -p build ; cd build
cmake .. -DCMAKE_SYSTEM_NAME=Linux \
      -DCMAKE_SYSTEM_PROCESSOR=AARCH64 \
      -DCMAKE_LIBRARY_PATH=/usr/aarch64-linux-gnu/lib
```

3. Build the library

   For Linux:

```
cmake --build . --parallel $(nproc)
```

   For macOS:

```
cmake --build . --parallel $(sysctl -n hw.ncpu)
```

Use GCC with Arm Compute Library (ACL) on AArch64 host

1. Set up the environment for the compiler

   Download Arm Compute Library or build it from source and set `ACL_ROOT_DIR` to directory where it is installed.

```
export ACL_ROOT_DIR=<path/to/ComputeLibrary>
export CC=gcc
export CXX=g++
```

2. Generate the build system

```
mkdir -p build ; cd build
cmake .. -DONEDNN_AARCH64_USE_ACL=ON
```

3. Build the library

   For Linux:

```
cmake --build . --parallel $(nproc)
```

For macOS:

```
cmake --build . --parallel $(sysctl -n hw.ncpu)
```

Build on Windows

Use Microsoft Visual C++ Compiler

1. Set up the environment for the compiler

   Microsoft Visual Studio uses the `VsDevCmd.bat` script to set all required variables. The command below assumes you installed to the default folder. If you customized the installation folder, `VsDevCmd.bat` is in your custom folder.

   ```
   "C:\Program Files\Microsoft Visual Studio\2022\Professional\ ^
    Common7\Tools\VsDevCmd.bat" ^
   -startdir=none ^
   -arch=x64 ^
   -host_arch=x64
   ```

   or open `x64 Native Tools Command Prompt` from start menu instead.

2. Generate the build system

   ```
   mkdir build
   cd build
   cmake .. -G "Visual Studio 17 2022"
   ```

3. Build the library

   ```
   cmake --build . --config=Release --parallel %NUMBER_OF_PROCESSORS%
   ```

> **Note**
>
> Currently, the oneDNN build system has limited support for multi-config generators. Build configuration is based on the `CMAKE_BUILD_TYPE` option (`Release` by default), and CMake must be rerun from scratch every time the build type changes to apply the new build configuration. You can choose a specific build type with the `--config` option (the solution file supports both `Debug` and `Release` builds), but it must refer to the same build type (`Release`, `Debug`, etc.) as selected with the `CMAKE_BUILD_TYPE` option.

> **Note**
>
> Alternatively, you can open `oneDNN.sln` to build the project from the Microsoft Visual Studio IDE.

Use Intel oneAPI DPC++/C++ Compiler with SYCL Runtime

1. Set up the environment for the compiler

   Intel oneAPI DPC++/C++ Compiler uses the `setvars.bat` script to set all required variables. The command below assumes you installed to the default folder. If you customized the installation folder, `setvars.bat` is in your custom folder.

   ```
   "C:\Program Files (x86)\Intel\oneAPI\setvars.bat"
   :: Set Intel oneAPI DPC++/C++ Compiler as default C and C++ compilers
   set CC=icx
   set CXX=icx
   ```

   or open `Intel oneAPI Command Prompt` from start menu instead.

2. Generate the build system

   ```
   mkdir build
   cd build

   cmake .. -G Ninja ^
         -DONEDNN_CPU_RUNTIME=SYCL ^
         -DONEDNN_GPU_RUNTIME=SYCL
   ```

> **Warning**
>
> Intel oneAPI DPC++/C++ Compiler on Windows requires CMake v3.23 or later.

> **Warning**
>
> Intel oneAPI DPC++/C++ Compiler does not support CMake's Microsoft Visual Studio generator.

> **Note**
>
> Open-source version of oneAPI DPC++ Compiler does not have the icx driver, use clang/clang++ instead. Open-source version of oneAPI DPC++ Compiler may not contain OpenCL runtime. In this case, you can use `OPENCLROOT` CMake option or environment variable of the same name to specify path to the OpenCL runtime if it is installed in a custom location.

3. Build the library

```
cmake --build . --parallel %NUMBER_OF_PROCESSORS%
```

Step 3. (Optional) Validate the Build

After building the library, you can run a predefined test set using:

```
ctest
```

The https://uxlfoundation.github.io/oneDNN/dev_guide_build_options.html#onednn-test-set build option set during the build configuration determines the scope and depth of the test set. Useful values are `SMOKE` (smallest set), `CI` (default), and `NIGHTLY` (most comprehensive). The test set can be reconfigured after the entire project has been built, and only the missing tests will be compiled.

```
cmake .. -DONEDNN_TEST_SET=NIGHTLY
cmake --build .
ctest
```

ctest supports filtering the test set by using the `-R` option. For example, to run only the GPU tests, use:

```
ctest -R gpu
```

Another useful option is `--output-on-failure`, which will print verbose output in case a test fails. Full set of options can be found here.

> **Warning**
>
> When using the `/opt/intel/oneapi/setvars.sh` script from the Intel oneAPI Base Toolkit, the `LD_LIBRARY_PATH` environment variable is set to include the oneDNN library path. Make sure that the correct oneDNN library is present in `LD_LIBRARY_PATH` by setting it explicitly, if needed.

Step 4. (Optional) Build Documentation

1. Install the requirements

   ```
   conda env create -f ../doc/environment.yml
   conda activate onednn-doc
   ```

2. Build the documentation

   ```
   cmake --build . --target doc
   ```

Step 5. Install the Library

Install the library, headers, and documentation.

The install directory is specified by the CMAKE_INSTALL_PREFIX CMake variable. When installing in the default directory, you need to run the following command with administrative privileges using `sudo` on Linux/macOS or a command prompt run as administrator on Windows.

```
cmake --build . --target install
```