Getting Started with oneAPI DPC++

The DPC++ Compiler compiles C++ and SYCL* source files with code for both CPU and a wide range of compute accelerators such as GPU and FPGA.

Table of contents

Prerequisites

- `git` - Download
- `cmake` version 3.14 or later - Download
- `python` - Download
- `ninja` - Download
- C++ compiler
  - Linux: `GCC` version 7.1.0 or later (including libstdc++) - Download
  - Windows: `Visual Studio` version 15.7 preview 4 or later - Download

Create DPC++ workspace

Throughout this document `DPCPP_HOME` denotes the path to the local directory created as DPC++ workspace. It might be useful to create an environment variable with the same name.

Linux:

```
export DPCPP_HOME=~/sycl_workspace
mkdir $DPCPP_HOME
cd $DPCPP_HOME

git clone https://github.com/intel/llvm -b sycl
```

Windows (64-bit):

Open a developer command prompt using one of two methods:

- Click start menu and search for "**x64** Native Tools Command Prompt for VS XXXX", where XXXX is a version of installed Visual Studio.
- Ctrl-R, write "cmd", click enter, then run `"C:\Program Files (x86)\Microsoft Visual Studio\2017\Community\VC\Auxiliary\Build\vcvarsall.bat" x64`

```
set DPCPP_HOME=%USERPROFILE%\sycl_workspace
mkdir %DPCPP_HOME%
cd %DPCPP_HOME%

git clone --config core.autocrlf=false https://github.com/intel/llvm -b
sycl
```

Build DPC++ toolchain

The easiest way to get started is to use the buildbot configure and compile scripts.

In case you want to configure CMake manually the up-to-date reference for variables is in these files.

Linux:

```
python $DPCPP_HOME/llvm/buildbot/configure.py
python $DPCPP_HOME/llvm/buildbot/compile.py
```

Windows (64-bit):

```
python %DPCPP_HOME%\llvm\buildbot\configure.py
python %DPCPP_HOME%\llvm\buildbot\compile.py
```

You can use the following flags with `configure.py` (full list of available flags can be found by launching the script with `--help`):

- `--system-ocl` -> Don't download OpenCL headers and library via CMake but use the system ones
- `--no-werror` -> Don't treat warnings as errors when compiling llvm
- `--cuda` -> use the cuda backend (see Nvidia CUDA)
- `--rocm` -> use the rocm backend (see AMD ROCm)
- `--shared-libs` -> Build shared libraries
- `-t` -> Build type (debug or release)
- `-o` -> Path to build directory
- `--cmake-gen` -> Set build system type (e.g. `--cmake-gen "Unix Makefiles"`)

Please note that no data about flags is being shared between `configure.py` and `compile.py` scripts, which means that if you configured your build to be placed in non-default directory using `-o` flag, you must also specify this flag and the same path in `compile.py` options. This allows you, for example, to configure several different builds and then build just one of them which is needed at the moment.

Build DPC++ toolchain with libc++ library

There is experimental support for building and linking DPC++ runtime with libc++ library instead of libstdc++. To enable it the following CMake options should be used.

Linux:

```
-DSYCL_USE_LIBCXX=ON \
-DSYCL_LIBCXX_INCLUDE_PATH=<path to libc++ headers> \
-DSYCL_LIBCXX_LIBRARY_PATH=<path to libc++ and libc++abi libraries>
```

You can also use configure script to enable:

```
python %DPCPP_HOME%\llvm\buildbot\configure.py --use-libcxx \
--libcxx-include <path to libc++ headers> \
--libcxx-library <path to libc++ and libc++ abi libraries>
python %DPCPP_HOME%\llvm\buildbot\compile.py
```

Build DPC++ toolchain with support for NVIDIA CUDA

There is experimental support for DPC++ for CUDA devices.

To enable support for CUDA devices, follow the instructions for the Linux DPC++ toolchain, but add the `--cuda` flag to `configure.py`

Enabling this flag requires an installation of CUDA 10.2 on the system, refer to NVIDIA CUDA Installation Guide for Linux.

Currently, the only combination tested is Ubuntu 18.04 with CUDA 10.2 using a Titan RTX GPU (SM 71), but it should work on any GPU compatible with SM 50 or above. The default SM for the NVIDIA CUDA backend is 5.0. Users can specify lower values, but some features may not be supported.

Build DPC++ toolchain with support for AMD ROCm

There is experimental support for DPC++ for ROCm devices.

To enable support for ROCm devices, follow the instructions for the Linux DPC++ toolchain, but add the `--rocm` flag to `configure.py`

Enabling this flag requires an installation of ROCm 4.1.0 on the system, refer to AMD ROCm Installation Guide for Linux.

Currently, the only combination tested is Ubuntu 18.04 with ROCm 4.1.0 using a Vega20 gfx906.

LLD is necessary for the AMD GPU compilation chain. The AMDGPU backend generates a standard ELF [ELF] relocatable code object that can be linked by lld to produce a standard ELF shared code object which can be loaded and executed on an AMDGPU target. So if you want to support AMD ROCm, you should also build the lld project. LLD Build Guide

Build Doxygen documentation

Building Doxygen documentation is similar to building the product itself. First, the following tools need to be installed:

- doxygen
- graphviz

Then you'll need to add the following options to your CMake configuration command:

```
-DLLVM_ENABLE_DOXYGEN=ON
```

After CMake cache is generated, build the documentation with `doxygen-sycl` target. It will be put to `$DPCPP_HOME/llvm/build/tools/sycl/doc/html` directory.

Deployment

TODO: add instructions how to deploy built DPC++ toolchain.

Use DPC++ toolchain

Install low level runtime

To run DPC++ applications on OpenCL devices, OpenCL implementation(s) must be present in the system.

To run DPC++ applications on Level Zero devices, Level Zero implementation(s) must be present in the system. You can find the link to the Level Zero spec in the following section Find More.

The Level Zero RT for `GPU`, OpenCL RT for `GPU`, OpenCL RT for `CPU`, FPGA emulation RT and TBB runtime which are needed to run DPC++ application on Intel `GPU` or Intel `CPU` devices can be downloaded using links in the dependency configuration file and installed following the instructions below. The same versions are used in PR testing.

Linux:

1. Extract the archive. For example, for the archives `oclcpuexp_<cpu_version>.tar.gz` and `fpgaemu_<fpga_version>.tar.gz` you would run the following commands

   ```
   # Extract OpenCL FPGA emulation RT
   mkdir -p /opt/intel/oclfpgaemu_<fpga_version>
   cd /opt/intel/oclfpgaemu_<fpga_version>
   tar zxvf fpgaemu_<fpga_version>.tar.gz
   # Extract OpenCL CPU RT
   mkdir -p /opt/intel/oclcpuexp_<cpu_version>
   cd /opt/intel/oclcpuexp_<cpu_version>
   tar -zxvf oclcpu_rt_<cpu_version>.tar.gz
   ```

2. Create ICD file pointing to the new runtime (requires root access)

   ```
   # OpenCL FPGA emulation RT
   echo  /opt/intel/oclfpgaemu_<fpga_version>/x64/libintelocl_emu.so >
     /etc/OpenCL/vendors/intel_fpgaemu.icd
   # OpenCL CPU RT
   echo /opt/intel/oclcpuexp_<cpu_version>/x64/libintelocl.so >
     /etc/OpenCL/vendors/intel_expcpu.icd
   ```

3. Extract or build TBB libraries using links in the dependency configuration file. For example, for the archive oneapi-tbb-<tbb_version>-lin.tgz:

   ```
   mkdir -p /opt/intel
   cd /opt/intel
   tar -zxvf oneapi-tbb*lin.tgz
   ```

4. Copy files from or create symbolic links to TBB libraries in OpenCL RT folder:

   ```
   # OpenCL FPGA emulation RT
   ln -s /opt/intel/oneapi-tbb-<tbb_version>/lib/intel64/gcc4.8/libtbb.so
     /opt/intel/oclfpgaemu_<fpga_version>/x64
   ln -s /opt/intel/oneapi-tbb-
   <tbb_version>/lib/intel64/gcc4.8/libtbbmalloc.so
     /opt/intel/oclfpgaemu_<fpga_version>/x64
   ln -s /opt/intel/oneapi-tbb-
   <tbb_version>/lib/intel64/gcc4.8/libtbb.so.12
     /opt/intel/oclfpgaemu_<fpga_version>/x64
   ln -s /opt/intel/oneapi-tbb-
   <tbb_version>/lib/intel64/gcc4.8/libtbbmalloc.so.2
   ```

```
  /opt/intel/oclfpgaemu_<fpga_version>/x64
# OpenCL CPU RT
ln -s /opt/intel/oneapi-tbb-<tbb_version>/lib/intel64/gcc4.8/libtbb.so
  /opt/intel/oclcpuexp_<cpu_version>/x64
ln -s /opt/intel/oneapi-tbb-
<tbb_version>/lib/intel64/gcc4.8/libtbbmalloc.so
  /opt/intel/oclcpuexp_<cpu_version>/x64
ln -s /opt/intel/oneapi-tbb-
<tbb_version>/lib/intel64/gcc4.8/libtbb.so.12
  /opt/intel/oclcpuexp_<cpu_version>/x64
ln -s /opt/intel/oneapi-tbb-
<tbb_version>/lib/intel64/gcc4.8/libtbbmalloc.so.2
  /opt/intel/oclcpuexp_<cpu_version>/x64
```

5. Configure library paths (requires root access)

```
echo /opt/intel/oclfpgaemu_<fpga_version>/x64 >
  /etc/ld.so.conf.d/libintelopenclexp.conf
echo /opt/intel/oclcpuexp_<cpu_version>/x64 >>
  /etc/ld.so.conf.d/libintelopenclexp.conf
ldconfig -f /etc/ld.so.conf.d/libintelopenclexp.conf
```

Windows (64-bit):

1. If you need OpenCL runtime for Intel `GPU` as well, then update/install it first. Do it **before** installing OpenCL runtime for Intel `CPU` runtime as OpenCL runtime for Intel `GPU` installer may re-write some important files or settings and make existing OpenCL runtime for Intel `CPU` runtime not working properly.

2. Extract the archive with OpenCL runtime for Intel `CPU` and/or for Intel `FPGA` emulation using links in the dependency configuration file. For example, to `c:\oclcpu_rt_<cpu_version>`.

3. Extract the archive with TBB runtime or build it from sources using links in the dependency configuration file. For example, to `c:\oneapi-tbb-<tbb_version>`.

4. Run `Command Prompt` as `Administrator`. To do that click `Start` button, type `Command Prompt`, click the Right mouse button on it, then click `Run As Administrator`, then click `Yes` to confirm.

5. In the opened windows run `install.bat` provided with the extracted files to install runtime to the system and setup environment variables. So, if the extracted files are in `c:\oclcpu_rt_<cpu_version>\` folder, then type the command:

```
# Install OpenCL FPGA emulation RT
# Answer N to clean previous OCL_ICD_FILENAMES configuration
c:\oclfpga_rt_<fpga_version>\install.bat c:\oneapi-tbb-
<tbb_version>\redist\intel64\vc14
# Install OpenCL CPU RT
# Answer Y to setup CPU RT side-bi-side with FPGA RT
```

```
c:\oclcpu_rt_<cpu_version>\install.bat c:\oneapi-tbb-
<tbb_version>\redist\intel64\vc14
```

Obtain prerequisites for ahead of time (AOT) compilation

Ahead of time compilation requires ahead of time compiler available in `PATH`. There is AOT compiler for each device type:

- `GPU`, Level Zero and OpenCL runtimes are supported,
- `CPU`, OpenCL runtime is supported,
- `Accelerator` (FPGA or FPGA emulation), OpenCL runtime is supported.

GPU

- Linux

  There are two ways how to obtain GPU AOT compiler `ocloc`:

  - (Ubuntu) Download and install intel-ocloc_***.deb package from intel/compute-runtime releases. This package should have the same version as Level Zero / OpenCL GPU runtimes installed on the system.
  - (other distros) `ocloc` is a part of Intel® software packages for general purpose GPU capabilities.

- Windows

  - GPU AOT compiler `ocloc` is a part of Intel® oneAPI Base Toolkit (Intel® oneAPI DPC++/C++ Compiler component).
    Make sure that the following path to `ocloc` binary is available in `PATH` environment variable:

    - `<oneAPI installation location>/compiler/<version>/windows/lib/ocloc`

CPU

- CPU AOT compiler `opencl-aot` is enabled by default. For more, see opencl-aot documentation.

Accelerator

- Accelerator AOT compiler `aoc` is a part of Intel® oneAPI Base Toolkit (Intel® oneAPI DPC++/C++ Compiler component).
  Make sure that these binaries are available in `PATH` environment variable:

  - `aoc` from `<oneAPI installation location>/compiler/<version>/<OS>/lib/oclfpga/bin`
  - `aocl-ioc64` from `<oneAPI installation location>/compiler/<version>/<OS>/bin`

Test DPC++ toolchain

Run in-tree LIT tests

To verify that built DPC++ toolchain is working correctly, run:

Linux:

```
python $DPCPP_HOME/llvm/buildbot/check.py
```

Windows (64-bit):

```
python %DPCPP_HOME%\llvm\buildbot\check.py
```

If no OpenCL GPU/CPU runtimes are available, the corresponding tests are skipped.

If CUDA support has been built, it is tested only if there are CUDA devices available.

Run DPC++ E2E test suite

Follow instructions from the link below to build and run tests: README

Run Khronos* SYCL* conformance test suite (optional)

Khronos* SYCL* conformance test suite (CTS) is intended to validate implementation conformance to Khronos* SYCL* specification. DPC++ compiler is expected to pass significant number of tests, and it keeps improving.

Follow Khronos* SYCL* CTS instructions from README file to obtain test sources and instructions how build and execute the tests.

To configure testing of DPC++ toochain set `SYCL_IMPLEMENTATION=Intel_SYCL` and `Intel_SYCL_ROOT=<path to the SYCL installation>` CMake variables.

Linux:

```
cmake -DIntel_SYCL_ROOT=$DPCPP_HOME/deploy -
DSYCL_IMPLEMENTATION=Intel_SYCL ...
```

Windows (64-bit):

```
cmake -DIntel_SYCL_ROOT=%DPCPP_HOME%\deploy -
DSYCL_IMPLEMENTATION=Intel_SYCL ...
```

Run simple DPC++ application

A simple DPC++ or SYCL* program consists of following parts:

1. Header section
2. Allocating buffer for data
3. Creating SYCL queue
4. Submitting command group to SYCL queue which includes the kernel
5. Wait for the queue to complete the work
6. Use buffer accessor to retrieve the result on the device and verify the data
7. The end

Creating a file `simple-sycl-app.cpp` with the following C++/SYCL code:

```
#include <CL/sycl.hpp>

int main() {
  // Creating buffer of 4 ints to be used inside the kernel code
  cl::sycl::buffer<cl::sycl::cl_int, 1> Buffer(4);

  // Creating SYCL queue
  cl::sycl::queue Queue;

  // Size of index space for kernel
  cl::sycl::range<1> NumOfWorkItems{Buffer.get_count()};

  // Submitting command group(work) to queue
  Queue.submit([&](cl::sycl::handler &cgh) {
    // Getting write only access to the buffer on a device
    auto Accessor = Buffer.get_access<cl::sycl::access::mode::write>(cgh);
    // Executing kernel
    cgh.parallel_for<class FillBuffer>(
        NumOfWorkItems, [=](cl::sycl::id<1> WIid) {
          // Fill buffer with indexes
          Accessor[WIid] = (cl::sycl::cl_int)WIid.get(0);
        });
  });

  // Getting read only access to the buffer on the host.
  // Implicit barrier waiting for queue to complete the work.
  const auto HostAccessor =
Buffer.get_access<cl::sycl::access::mode::read>();

  // Check the results
  bool MismatchFound = false;
  for (size_t I = 0; I < Buffer.get_count(); ++I) {
    if (HostAccessor[I] != I) {
      std::cout << "The result is incorrect for element: " << I
                << " , expected: " << I << " , got: " << HostAccessor[I]
                << std::endl;
      MismatchFound = true;
    }
  }
```

```
    if (!MismatchFound) {
      std::cout << "The results are correct!" << std::endl;
    }

    return MismatchFound;
  }
```

To build simple-sycl-app put `bin` and `lib` to PATHs:

Linux:

```
export PATH=$DPCPP_HOME/llvm/build/bin:$PATH
export LD_LIBRARY_PATH=$DPCPP_HOME/llvm/build/lib:$LD_LIBRARY_PATH
```

Windows (64-bit):

```
set PATH=%DPCPP_HOME%\llvm\build\bin;%PATH%
set LIB=%DPCPP_HOME%\llvm\build\lib;%LIB%
```

and run following command:

```
clang++ -fsycl simple-sycl-app.cpp -o simple-sycl-app.exe
```

When building for CUDA, use the CUDA target triple as follows:

```
clang++ -fsycl -fsycl-targets=nvptx64-nvidia-cuda-sycldevice \
  simple-sycl-app.cpp -o simple-sycl-app-cuda.exe
```

When building for ROCm, please note that the option `mcpu` must be specified, use the ROCm target triple as follows:

```
clang++ -fsycl -fsycl-targets=amdgcn-amd-amdhsa-sycldevice \
  -mcpu=gfx906 simple-sycl-app.cpp -o simple-sycl-app-cuda.exe
```

To build simple-sycl-app ahead of time for GPU, CPU or Accelerator devices, specify the target architecture:

`-fsycl-targets=spir64_gen-unknown-unknown-sycldevice` for GPU,
`-fsycl-targets=spir64_x86_64-unknown-unknown-sycldevice` for CPU,
`-fsycl-targets=spir64_fpga-unknown-unknown-sycldevice` for Accelerator.

Multiple target architectures are supported.

E.g., this command builds simple-sycl-app for GPU and CPU devices in ahead of time mode:

```
clang++ -fsycl -fsycl-targets=spir64_gen-unknown-unknown-
sycldevice,spir64_x86_64-unknown-unknown-sycldevice simple-sycl-app.cpp -o
simple-sycl-app-aot.exe
```

Additionally, user can pass specific options of AOT compiler to the DPC++ compiler using `-Xsycl-target-backend` option, see Device code formats for more. To find available options, execute:

`ocloc compile --help` for GPU, `opencl-aot --help` for CPU, `aoc -help -sycl` for Accelerator.

The `simple-sycl-app.exe` application doesn't specify SYCL device for execution, so SYCL runtime will use `default_selector` logic to select one of accelerators available in the system or SYCL host device. In this case, the behavior of the `default_selector` can be altered using the `SYCL_BE` environment variable, setting `PI_CUDA` forces the usage of the CUDA backend (if available), `PI_ROCM` forces the usage of the ROCm backend (if available), `PI_OPENCL` will force the usage of the OpenCL backend.

```
SYCL_BE=PI_CUDA ./simple-sycl-app-cuda.exe
```

The default is the OpenCL backend if available. If there are no OpenCL or CUDA devices available, the SYCL host device is used. The SYCL host device executes the SYCL application directly in the host, without using any low-level API.

NOTE: `nvptx64-nvidia-cuda-sycldevice` is usable with `-fsycl-targets` if clang was built with the cmake option `SYCL_BUILD_PI_CUDA=ON`.

Linux & Windows (64-bit):

```
./simple-sycl-app.exe
The results are correct!
```

NOTE: Currently, when the application has been built with the CUDA target, the CUDA backend must be selected at runtime using the `SYCL_BE` environment variable.

```
SYCL_BE=PI_CUDA ./simple-sycl-app-cuda.exe
```

NOTE: DPC++/SYCL developers can specify SYCL device for execution using device selectors (e.g. `cl::sycl::cpu_selector`, `cl::sycl::gpu_selector`, Intel FPGA selector(s)) as explained in following section Code the program for a specific GPU.

Code the program for a specific GPU

To specify OpenCL device SYCL provides the abstract `cl::sycl::device_selector` class which the can be used to define how the runtime should select the best device.

The method `cl::sycl::device_selector::operator()` of the SYCL `cl::sycl::device_selector` is an abstract member function which takes a reference to a SYCL device and returns an integer score. This abstract member function can be implemented in a derived class to provide a logic for selecting a SYCL device. SYCL runtime uses the device for with the highest score is returned. Such object can be passed to `cl::sycl::queue` and `cl::sycl::device` constructors.

The example below illustrates how to use `cl::sycl::device_selector` to create device and queue objects bound to Intel GPU device:

```
#include <CL/sycl.hpp>

int main() {
  class NEOGPUDeviceSelector : public cl::sycl::device_selector {
  public:
    int operator()(const cl::sycl::device &Device) const override {
      using namespace cl::sycl::info;

      const std::string DeviceName = Device.get_info<device::name>();
      const std::string DeviceVendor = Device.get_info<device::vendor>();

      return Device.is_gpu() && (DeviceName.find("HD Graphics NEO") !=
std::string::npos);
    }
  };

  NEOGPUDeviceSelector Selector;
  try {
    cl::sycl::queue Queue(Selector);
    cl::sycl::device Device(Selector);
  } catch (cl::sycl::invalid_parameter_error &E) {
    std::cout << E.what() << std::endl;
  }
}
```

The device selector below selects an NVIDIA device only, and won't execute if there is none.

```
class CUDASelector : public cl::sycl::device_selector {
  public:
    int operator()(const cl::sycl::device &Device) const override {
      using namespace cl::sycl::info;
      const std::string DriverVersion =
Device.get_info<device::driver_version>();

      if (Device.is_gpu() && (DriverVersion.find("CUDA") !=
std::string::npos)) {
        std::cout << " CUDA device found " << std::endl;
```

```
            return 1;
          };
          return -1;
        }
    };
```

Using the DPC++ toolchain on CUDA platforms

The DPC++ toolchain support on CUDA platforms is still in an experimental phase. Currently, the DPC++ toolchain relies on having a recent OpenCL implementation on the system in order to link applications to the DPC++ runtime. The OpenCL implementation is not used at runtime if only the CUDA backend is used in the application, but must be installed.

The OpenCL implementation provided by the CUDA SDK is OpenCL 1.2, which is too old to link with the DPC++ runtime and lacks some symbols.

We recommend installing the low level CPU runtime, following the instructions in the next section.

Instead of installing the low level CPU runtime, it is possible to build and install the Khronos ICD loader, which contains all the symbols required.

C++ standard

- DPC++ runtime and headers require C++17 at least.
- DPC++ compiler builds apps as C++17 apps by default. Higher versions of standard are supported as well.

Known Issues and Limitations

- DPC++ device compiler fails if the same kernel was used in different translation units.
- SYCL host device is not fully supported.
- SYCL 2020 support work is in progress.
- 32-bit host/target is not supported.
- DPC++ works only with OpenCL low level runtimes which support out-of-order queues.
- On Windows linking DPC++ applications with `/MTd` flag is known to cause crashes.

CUDA back-end limitations

- Backend is only supported on Linux
- The only combination tested is Ubuntu 18.04 with CUDA 10.2 using a Titan RTX GPU (SM 71), but it should work on any GPU compatible with SM 50 or above
- The NVIDIA OpenCL headers conflict with the OpenCL headers required for this project and may cause compilation issues on some platforms

ROCm back-end limitations

- For supported Operating Systems, please refer to the Supported Operating Systems
- The only combination tested is Ubuntu 18.04 with ROCm 4.1 using a Vega20 gfx906.

- Judging from the current test results, there is still a lot of room for improvement in ROCm back-end support. The current problems include three aspects. The first one is at compile time: the `barrier` and `atomic` keywords are not supported. The second is at runtime: when calling `hipMemcpyDtoHAsync` ROCm API, the program will cause an exception if the input data size is too large. The third is calculation accuracy: the ROCm backend has obvious errors in the calculation results of some float type operators

Find More

- DPC++ specification: https://spec.oneapi.com/versions/latest/elements/dpcpp/source/index.html
- SYCL* 1.2.1 specification: www.khronos.org/registry/SYCL/specs/sycl-1.2.1.pdf
- oneAPI Level Zero specification: https://spec.oneapi.com/versions/latest/oneL0/index.html