

Abstract

SYCL is a single-source C++ DSL targeting a large variety of accelerators in a unified way by using different back-ends.

We present an experimental SYCL implementation targeting Xilinx Alveo FPGA cards by merging 2 different open-source implementations, Intel's oneAPI DPC++ with some LLVM passes from triSYCL.

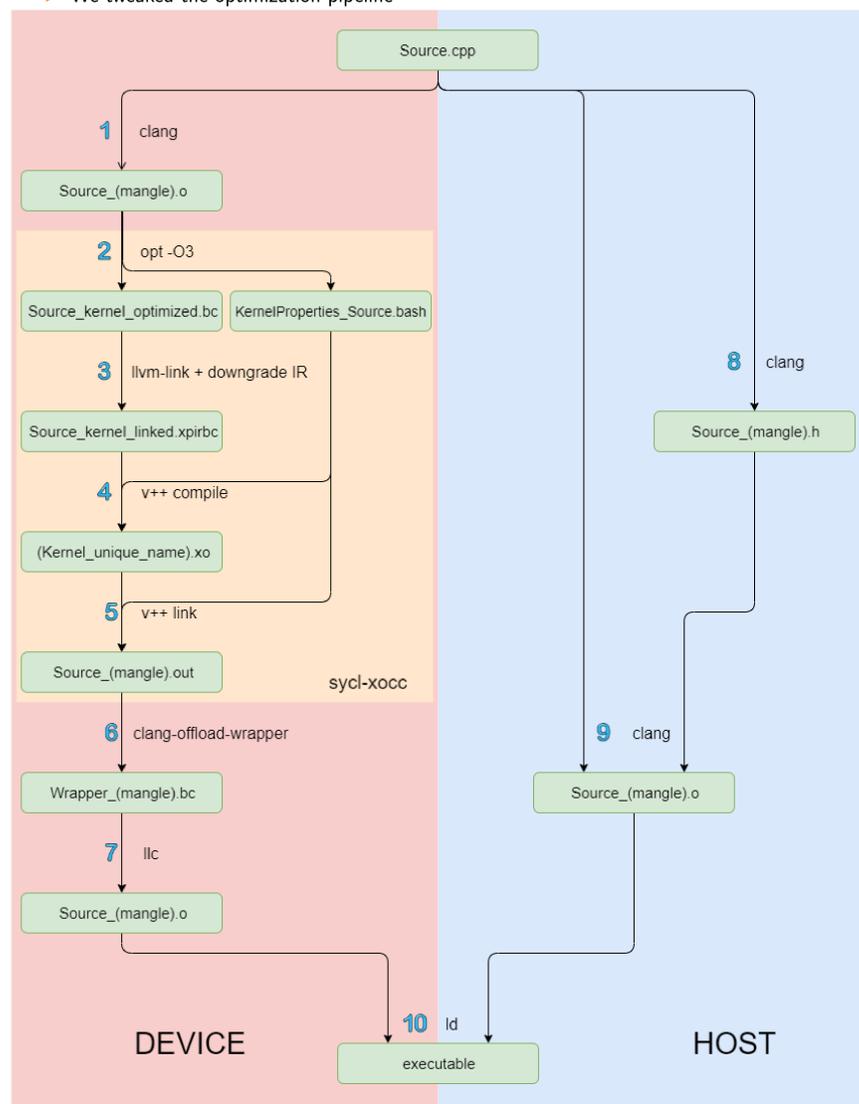
The FPGA device configuration is generated by Xilinx Vitis 2020.2 fed with LLVM IR SPIR and Xilinx XRT is used as a host OpenCL API to control the device.

Motivation

- ▶ FPGA are hard to program
 - ▶ HLS (High-Level Synthesis) has made it better
 - ▶ Tools use lots of non-standard language extensions
 - ▶ Tools are usually split source
- ▶ SYCL for FPGA is trying to make it simpler
 - ▶ Single-source
 - ▶ Uses the usual compiling process of C++
 - ▶ Pure modern C++, can be implemented as a normal library for host only execution
 - ▶ Standard

Implementation

- ▶ Based on Intel's oneAPI DPC++ because:
 - ▶ Open-source
 - ▶ Using OpenCL
 - ▶ Based on LLVM latest ToT
- ▶ We use Xilinx's OpenCL runtime from XRT
- ▶ The compilation flow required changes
 - ▶ Using Vitis's v++ as backend compiler
 - ▶ Needs downgrading from LLVM ToT to LLVM 6.x
 - ▶ Needs converting SPIR-V builtin to "SPIR-df (*de facto*)"
 - ▶ We tweaked the optimization pipeline



1. Device front-end, will only emit device code
2. ▶ Run optimizations
 - ▶ Convert SPIR-V builtins to "SPIR-df (*de facto*)"
 - ▶ Generate configuration for the backend
3. ▶ Link the device code with the device runtime
 - ▶ Run optimizations
 - ▶ Downgrade the IR to LLVM 6.x
4. ▶ Compile each kernel with the backend
5. ▶ Link all kernels
6. ▶ Package the device image as data for the host
7. ▶ Assemble the packaged device image into a .o
8. ▶ Generate the inclusion header
9. ▶ Compile the host code
10. ▶ Link everything together

Targets

We support 3 emulation targets and 1 for real hardware execution

- ▶ Library only
 - ▶ SYCL runtime is used as normal C++ library and does not use Vitis at all
 - ▶ Any C++ compiler can be used to compile
 - ▶ Fastest to compile and execute on CPU
 - ▶ Kernel code will be executed natively on the host
 - ▶ Can use runtime checkers like: sanitizers, valgrind... and usual debuggers
 - ▶ But its the furthest from hardware
- ▶ Software emulation
 - ▶ Needs to use our custom compiler and use Vitis compiler
 - ▶ Kernel code is run natively on the host with the Vitis software emulator
 - ▶ Kernel code is isolated from host code
- ▶ Hardware emulation
 - ▶ Needs to use our custom compiler and Vitis RTL synthesis
 - ▶ Generate reports about expected resource usage and timings
 - ▶ Kernel code is executed in Vitis RTL simulator
 - ▶ Kernel code is isolated from host code
- ▶ Hardware execution on FPGA
 - ▶ Generates reports about the real resource usage and timings
 - ▶ Kernel code is executed on the FPGA
 - ▶ Slowest to compile including Vitis RTL synthesis and FPGA place & route

FPGA-specific extensions to the SYCL standard

Allow better control on the design and performances

- ▶ Pipeline annotations


```
for (...)
```

```
  xilinx::pipeline([&] { ... });
```

The for loop will get each stage pipelined in hardware and this will speedup the loop at the cost of using a little bit more hardware and latency.
- ▶ DDR bank accessor property


```
sycl::accessor Accessor
```

```
  { Buffer, cgh, sycl::ONEAPI::accessor_property_list{sycl::xilinx::ddr_bank<1>}};
```

The memory accessed by Accessor will be placed in the DDR memory bank 1
- ▶ Kernel backend compilation option for each kernel


```
cgh.single_task(xilinx::kernel_param("--kernel_frequency 300"_cstr, [=] {
```

```
  ...
```

```
});
```

The backend compiler invocation for this kernel will receive `--kernel_frequency 300` forcing the generation with a kernel clock of 300 MHz
- ▶ Partition array


```
xilinx::partition_array<int, 12, xilinx::partition::complete<1>> arr{ ... };
```

is similar:

```
std::array<int, 12> arr{ ... };
```

but arr will be fully partitioned as registers making access faster by using more resources
- ▶ And more...

Using multiple devices in the same single-source application

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

int main() {
  sycl::buffer<int> v { 10 };

  // Implement a generic heterogeneous "executor"
  auto run = [&] (auto sel, auto work) {
    sycl::queue { sel }.submit([&] (auto& h) {
      auto a = sycl::accessor { v, h };
      h.parallel_for(a.get_count(), [=] (auto i) { work(i, a); });
    });
  };

  run(sycl::host_selector {}, [] (auto i, auto a) { a[i] = i; }); // CPU
  run(sycl::accelerator_selector {}, [] (auto i, auto a) { a[i] = 2*a[i]; }); // FPGA
  run(sycl::gpu_selector {}, [] (auto i, auto a) { a[i] = a[i] + 3; }); // GPU

  sycl::host_accessor acc { v };
  for (int i = 0; i != v.get_count(); ++i)
    std::cout << acc[i] << ", ";
  std::cout << std::endl;
}
```

- ▶ The above example is running code on a CPU, a Xilinx FPGA and an Nvidia GPU within a single application
- ▶ Type-safe generic and functional programming with C++20 without using template or typename keyword
- ▶ Compiled with 1 single command: `clang++ -std=c++20 -fsycl -fsycl-unnamed-lambda src.cpp -fsycl-targets=nvptx64-nvidia-cuda-sycldevice,fpga64_hw`

Future Work

- ▶ Focus more on performance
 - ▶ Expose more hardware details
 - ▶ Give more control over HLS to the user
 - ▶ Better adapt the optimizations to FPGA
- ▶ Usability
 - ▶ Fix more compatibility issues between the SYCL toolchain and Vitis v++
 - ▶ Improve support of the SYCL standard
 - ▶ Add support for more Xilinx hardware
- ▶ Test the implementation on more and bigger applications