triSYCL for Xilinx FPGA

Gozillon, Andrew; Keryell, Ronan; Yu, Lin-Ya; Harnisch, Gauthier; Keir, Paul

Published in:
Proceedings of the 2020 International Conference on High Performance Computing & Simulation (HPCS)

Accepted/In press: 22/12/2020

Document Version
Peer reviewed version

triSYCL for Xilinx FPGA

Gozillon, Andrew; Keryell, Ronan; Yu, Lin-Ya; Harnisch, Gauthier; Keir, Paul

Published in:
Proceedings of the 2020 International Conference on High Performance Computing & Simulation (HPCS)

Accepted/In press: 22/12/2020

Document Version
Peer reviewed version

triSYCL for Xilinx FPGA

Andrew Gozillon*, Ronan Keryell†, Lin-Ya Yu†, Gauthier Harnisch†, Paul Keir*
*University of the West of Scotland, High St., Paisley PA1 2BE, Scotland, United Kingdom
{andrew.gozillon, paul.keir}@uws.ac.uk
†Xilinx, Inc, 2100 Logic Dr., San Jose CA 95124, California, United States
{rkeryell, linyay, gauthier}@xilinx.com

***Abstract*—Khronos SYCL is a C++ based open-source specification that aims to increase the programmability of heterogeneous architectures. Several SYCL implementations exist, with variations both in terms of conformance to the specification; as well as in the range of hardware they target. Intel recently contributed the first open-source feature-complete SYCL implementation to the LLVM compiler project. The triSYCL project is another open-source SYCL implementation which targets Xilinx FPGAs. We describe here initial work to combine components of the triSYCL implementation with Intel's SYCL implementation, and provide details of the resulting updated compiler infrastructure for targeting the FPGA. We also highlight what is currently possible with the new hybrid triSYCL implementation alongside some interesting extensions for Xilinx FPGAs.**

***Keywords*—SYCL; Reconfigurable Computing; Clang LLVM; FPGA; SPIR-V; HPC**

I. Introduction

Heterogeneous architectures proliferate the market; not only are everyday computers now heterogeneous by virtue of a dedicated or integrated GPU alongside the CPU, but increasingly exotic hardware is being developed and used for specialised tasks. To ease the programmability of these complex systems, programming models that can provide an abstraction over the complexity of the underlying hardware, while retaining access to components necessary to attain full performance are pivotal. There are several programming models that fit into this spectrum of research, with recent notable examples including RAJA [1], Kokkos [2], SYCL [3] and HPX [4].

SYCL is our focus, a programming model developed by the Khronos Group. It was originally built as a single-source modern C++ abstraction on top of OpenCL, allowing a more typesafe and approachable method for programming OpenCL capable devices. Several SYCL implementations exist today; namely ComputeCpp [5], triSYCL [6], hipSYCL [7], sycl-gtx [8] and most recently Intel's SYCL [9]. These projects allow SYCL to target a variety of different hardware architectures, encompassing Intel CPUs, GPUs and FPGAs; Xilinx FPGAs; AMD GPUs and even NVIDIA GPUs through CUDA.

The prominence of SYCL support within Intel's oneAPI, as Data Parallel C++ (DPC++) [10], is a noteworthy indication of SYCL's increasing adoption; and follows the open-sourcing of their LLVM-based compiler and runtime; which they intend to merge into the original Clang LLVM compiler.

This is significant as most existing SYCL compilers are built on Clang LLVM. Having one unified SYCL implementation everyone can make a concerted effort to contribute to; expand upon; and use, will help the SYCL ecosystem and hopefully encourage wider adoption of heterogeneous architectures.

In the research introduced here, we combine and extend the open-source triSYCL compiler for Xilinx FPGA [11], with Intel's open-source SYCL efforts. Our goal is to yield a more robust SYCL compiler for Xilinx FPGAs; to allow us to contribute to the discussion of the ongoing upstreaming effort of Intel SYCL to the Clang LLVM project; and to aid in the discussion of FPGA related extensions and modifications to the SYCL programming model.

In this paper we will introduce considerable developments within an open-source Xilinx FPGA-targeting SYCL compiler, incorporating relevant advances within Intel's open-source LLVM-based implementation, so creating a new hybrid compiler.

II. The triSYCL Project & oneAPI SYCL

The triSYCL project is an open-source SYCL implementation that consists of a runtime library and an experimental Clang LLVM compiler. The runtime without the compiler supports multi-threaded CPU execution. Alternatively, there is experimental support for GPUs and Xilinx FPGAs through OpenCL; this requires the use of the compiler and a complex compilation recipe implemented within a Makefile. The FPGA compilation flow also requires Xilinx's FPGA toolchain.

Intel's oneAPI contains a range of domain specific libraries, as well as their SYCL implementation: DPC++, which extends SYCL with extensions for Intel devices. As with triSYCL, DPC++ consists of a compiler and runtime. DPC++ currently supports Intel FPGAs, CPUs and GPUs and NVIDIA GPUs through just in time, and ahead of time compilation.

In our experiments merging DPC++ and triSYCL we replaced the existing triSYCL compiler with a modified variation of DPC++'s compiler. This merged compiler incorporates components of the previous triSYCL's Makefile compilation recipe into the compiler itself; while simultaneously leveraging DPC++'s driver and semantic analysis. Two useful benefits of this are that it makes diagnostics more comprehensive and allows more complex compilation capabilities through Clang LLVM's command line. Rather than use triSYCL's current runtime, we made use of DPC++'s runtime and incorporated some of triSYCL's existing SYCL extensions.

There are differences between triSYCL and DPC++ that make this transition interesting even though both compilers

have a Clang LLVM foundation. For example DPC++'s primary intermediate representation (IR) for consumption by Intel accelerators is SPIR-V, whereas triSYCL requires a variation of SPIR, as Xilinx FPGAs only support offline compilation from SPIR. Both are Khronos IRs, intended for consumption by compatible accelerators. SPIR-V is the modern dialect and a language unto itself whereas SPIR is based on an older version of LLVM IR.

Another noteworthy difference between the implementations is the way the SYCL kernels are *outlined* from the C++ source code and transformed to their equivalent IR for the target devices. In SYCL kernel *outlining* is the extraction of device segments of the program from host segments. In triSYCL *late outlining* is performed, where the kernels are extracted after semantic analysis and IR generation. This occurs through a series of optimisation passes that find the kernel IR within wrapper functions. In contrast, DPC++ performs *early outlining* at the semantic analysis stage, finding kernels through a compiler attribute. One of the main benefits of the transition from late to early outlining is that it is then possible to emit useful SYCL related diagnostics, as the semantic information required to emit these diagnostics is still available. However, a downside is that as the host and device divide is done early, cross host-device boundary optimisations are harder to implement or impossible; for example constant propogation.

III. Compilation Flow of oneAPI SYCL

This section gives a brief overview of the original DPC++ compilation process, covering relevant device compilation steps that occur when compiling the simple case of a single source file to a binary.

From a user's perspective, a SYCL program is often viewed as a single translation unit (TU) of C++ source code. Device and host code are represented as plain C++, augmented with SYCL constructs to abstract over scheduling; data transfer; and kernel invocation. The device components of this program are extracted from the host components and compiled into a representation consumable by the target accelerators, and then embedded into the final host binary alongside some other metadata maintained in compiler-generated C++ constructs.

The compiler processes the source-file in two separate phases: first for the device, and then for the host. In SYCL this is referred to as single-source multiple compiler-passes. The ordering of the compilation phases is notable in DPC++ as the host compilation has a dependency on the device compilation in the form of a C++ header generated by the device stage that is used by the SYCL host runtime. This file is named the *integration header* and contains important information about compiled kernels that the host side SYCL runtime needs to launch the kernels on devices.

After the Clang frontend has semantically parsed and code-generated the equivalent LLVM IR for each of the kernels, it is then passed unoptimised to the `SYCLToolChain` either for translation to SPIR-V; or for offline compilation to a binary for a specific accelerator through an independent backend compiler. The `SYCLToolChain` is a communication layer that the Clang driver uses to forward key tasks in the compilation pipeline to SYCL-related external tools in place of the regular Clang compilation tools. For example, in the case of compiling LLVM IR to SPIR-V the driver forwards to the `SYCLToolChain` which then diverts it to an external SPIRV-LLVM translator tool. The main addition that triSYCL adds to support Xilinx FPGA compilation is a new toolchain, named `XOCCToolChain`, that replaces this existing `SYCLToolChain`.

After the device side LLVM IR has been translated to its final state it is ready to be offloaded into the final fat binary alongside the host binary.

IV. Compiling SYCL For Xilinx FPGA

The `XOCCToolChain` is named after the Xilinx OpenCL Compiler (**xocc**) [12]. The **xocc** compiler can be used to generate Xilinx FPGA compatible binaries; usually from OpenCL C or High Level Synthesis (HLS) C++. These binaries can then be executed on a Xilinx FPGA device using Xilinx's runtime for FPGA (XRT) which supports a subset of OpenCL 1.2. As DPC++ uses OpenCL as one underlying communication API to Intel devices, we can use XRT's OpenCL implementation.

In the case of the `XOCCToolChain`, **xocc** is used to compile Xilinx FPGA binaries from SYCL kernels that have been compiled to a pseudo form of SPIR we call SPIR de facto (SPIR-df). SPIR was originally based upon an older version of LLVM IR (3.2) and as the Clang LLVM compiler has matured, so has the LLVM IR it produces. As Clang LLVM cannot generate older LLVM IR versions, we end up with SPIR that slightly deviates from the specification (SPIR-df).

In our case SPIR's dependency on LLVM IR leads to compatibility issues between what we produce and what **xocc** can consume. This is because **xocc** is based on an older version of Clang LLVM than the SYCL compiler. Consequently, the generated SPIR-df is downgraded to something compatible with **xocc**.

The `XOCCToolChain` performs two actions. Firstly, it locates **xocc**, and some of Xilinx's C++ library locations. Secondly, it creates a command to a bash script called **sycl-xocc**. This script acts as a wrapper and driver for **xocc**. It performs all of the steps required to compile a SPIR-df file containing multiple SYCL kernels to an Xilinx FPGA binary that is ready for offloading. The steps taken by the script can be seen in Fig. 1.

This script is used as an abstraction on top of **xocc**. SPIR-df compilation is not the intended use for the compiler so there are some idiosyncrasies that need worked around. An example is that **xocc** can only compile one kernel in a TU at a time and it needs the kernel's name to be passed to the compiler invocation. This means we need an extra step to retrieve a list of all the kernels in a TU, to compile each individually. This leads to several kernel object files that must then be linked.

Following the numbering from Fig. 1, in step 1 the SPIR-df file generated by the frontend is fed to the `XOCCToolChain`. The toolchain then constructs a call to **sycl-xocc**, passing the file alongside other arguments, such as the location of **xocc**.

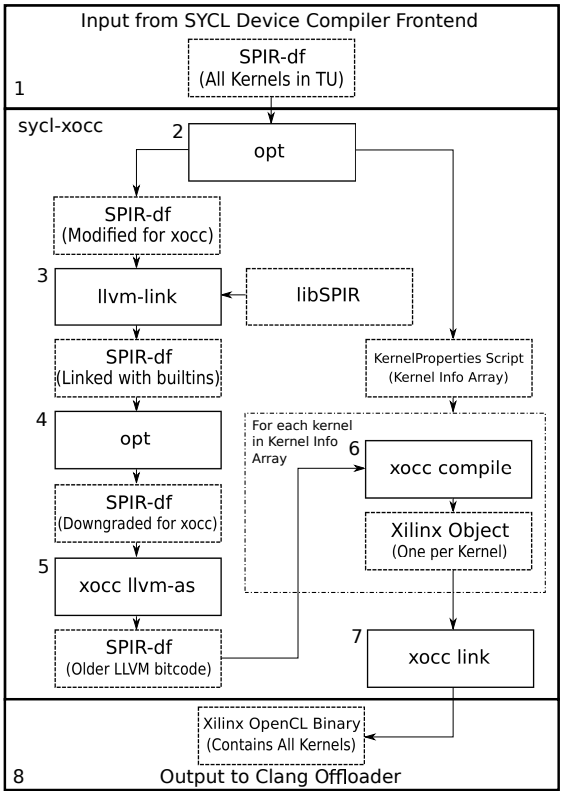

Figure 1. Compile flow of **sycl-xocc** driver script

This SPIR-df file can contain multiple kernels based on how many were contained in the current TU.

Step 2 occurs inside the script. Clang LLVM's optimiser **opt** is invoked on the file, using several optimisation passes. There are three key passes in this case. The first pass, `InferAddressSpace` [13], was originally developed elsewhere for CUDA. This pass is used to replace generic address spaces with concrete address spaces, by inferring the replacement from related uses of concrete address spaces.

In SYCL kernels are based on OpenCL and make use of OpenCL's address spaces. In OpenCL 2.0 a *generic* address space was introduced alongside OpenCL's concrete address spaces *private*, *local* and *constant*. Currently, in the case of DPC++ any address space that is not directly qualified or immediately deduceable ends up in the *generic* address space. However, this is a problem for devices that do not support OpenCL 2.0 and the *generic* address space, such as the Xilinx FPGA's we target. The `InferAddressSpace` pass helps us resolve this by replacing usages of the generic address space. However, the pass can fail and this will result in a failure to complete the compilation.

The second pass, `inSPIRation` in part deals with certain idiosyncrasies of **xocc**. For example, **xocc** requires that all function arguments are named; and as argument names are removed by default in LLVM IR, we must name them. We also have to pass kernel names to **xocc** when compiling, however it cannot handle certain characters that are contained in mangled function names, so we rename kernels.

The `inSPIRation` pass does two other tasks, the first is modifying SPIR and SPIR-V intrinsic names. In the former case we use SPIR intrinsics for retrieving information like work item position and range (e.g. `get_global_id`). However, in SPIR these intrinsics when mangled do not reside inside of a namespace or another abstraction. This means they will cause an error if a user defines a function with the same name. This is a problem in SYCL as these intrinsics are hidden, so a user will find this a surprise if they encounter this issue. To prevent this we prefix the intrinsics with `__spir_ocl_`, as a form of reserved name. The `inSPIRation` pass then removes the excess components of the mangled intrinsic. This returns it to its correct mangled name; allowing the intrinsic to be linked against its definition. User functions that conflict with the intrinsic are also renamed, preventing conflicting definitions.

In the latter case, as DPC++ targets SPIR-V, a lot of the SYCL runtime relies on SPIR-V intrinsics which we cannot make use of. To combat this issue the `inSPIRation` pass manipulates the mangled name of certain SPIR-V intrinsics with similarly named SPIR intrinsics to their counterparts.

The second task handles an experimental kernel property feature. We allow properties to be applied to SYCL kernels by applying template types to the kernel's name (in SYCL, kernel names are derived from types). This property then gets combined into the mangled name of the kernel and `inSPIRation` will detect and then perform the relevant modifications required. Currently we only support the `reqd_work_group_size` kernel attribute that is found in OpenCL as a property.

The third pass, named `KernelPropGen`, generates information from kernels for the **xocc** linker and compiler. This information is injected into the **sycl-xocc** script through a bash script named **KernelProperties** which the pass generates. Currently the main information it gathers is the kernel names for the compilation step; and the kernel arguments which must be assigned to DDR banks for the linker.

In step 3, we resolve the SPIR intrinsic declarations by linking against Xilinx's SPIR HLS library.

Step 4 then runs a final optimisation pass called the `XOCCIRDowngrader` on the SPIR-df. Which simply removes incompatibilities in the IR that **xocc** cannot consume due to the differing LLVM versions. It is necessary that this pass outputs the textual rather than the binary representation for the IR for step 5.

As in step 5 we use **xocc**'s assembler to translate the IR to its binary format, rather than the assembler from the SYCL frontend. This circumvents the incompatibility of the IR binaries generated by the SYCL frontend with **xocc**.

In step 6, we execute the **KernelProperties** bash script generated by the `KernelPropGen` pass. This exports a bash array of kernel names into **sycl-xocc** used to then invoke the **xocc** compilation command for each kernel in the TU. In step 7 all the compiled kernels are linked together using the **xocc** linker, generating the final binary. Step 8 simply forwards this binary to the next stage of compilation.

V. What's Currently Possible

The SYCL example in Fig. 2 is a simple edge detection example using a Sobel filter. For conciseness some postamble and preamble has been removed. Currently, this is the most complex example we have in our test suite. It is possible to compile and execute it for both hardware and software emulation, as well as some real hardware. For the moment we have only tested this example on an Xilinx Alveo U200 FPGA. Calculation results are functionally correct, though the program lacks performance. This is due to initially focusing on functionality, correctness and the compilation process.

Whilst we have not been focusing on performance we have still implemented several SYCL extensions for Xilinx FPGA which are analogous to existing constructs in Xilinx HLS C++.

The example in Fig. 2 makes use of the pipeline and partition array extensions discussed in [6]. These were part of the original triSYCL implementation and have been ported. The `xilinx::pipeline` extension indicates to perform loop pipelining; whereas `xilinx::partition_array` indicates how an array should be distributed across the FPGA hardware. The final Xilinx extension is the kernel property `xilinx::reqd_work_group_size`. This property is analogous to the attribute `reqd_work_group_size` in OpenCL. Using `reqd_work_group_size` specifies the required range the kernel must be executed with. In the case of Xilinx FPGAs it allows **xocc** to optimise the hardware resource usage of the kernel, rather than making a worst case assumption.

```
queue.submit([&](handler &cgh) {
  auto rb = ib.get_access<access::mode::read>(cgh);
  auto wb = ob.get_access<access::mode::write>(cgh);

  cgh.single_task<
    xilinx::reqd_work_group_size<1,1,1,sobel>>([=](){
    using ct = xilinx::partition::complete<0>;
    using pa = xilinx::partition_array<char, 9, ct>;
    auto gX = pa({-1, 0, 1, -2, 0, 2, -1,  0,  1 });
    auto gY = pa({ 1, 2, 1,  0, 0, 0, -1, -2, -1 });

    int magX, magY, gI, pIndex;
    for (size_t x = 1; x < width - 1; ++x) {
      for (size_t y = 1; y < height - 1; ++y) {
        magX = 0; magY = 0;
        xilinx::pipeline([&] {
          for(size_t k = 0; k < 3; ++k) {
            for(size_t m = 0; m < 3; ++m) {
              gI = k * 3 + m;
              pIndex = (x+k-1) + (y+m-1) * width;
              magX += gX[gI] * rb[pIndex];
              magY += gY[gI] * rb[pIndex];
            }
          }
        });

        wb[x + y * width] = cl::sycl::min(
                  cl::sycl::abs(magX)
                + cl::sycl::abs(magY), 0xFF);
      }
    }
  });
});
```

Figure 2. SYCL Sobel Filter for Xilinx FPGA

VI. Conclusion

SYCL simplifies programming heterogeneous architectures by providing a typesafe single-source environment to target multiple architectures. In this paper we outlined the compilation process for the triSYCL compiler based on the DPC++ compiler. Effort has been focused on the addition of a new Clang LLVM toolchain and driver script to the existing DPC++ infrastructure. The script makes extensive use of optimisation passes to gather information to drive the compilation flow and to massage the LLVM IR into a format (SPIR-df) which may be consumed by Xilinx's **xocc**. Such steps allow us to generate binaries for Xilinx FPGA, which are useable by the DPC++ runtime in conjunction with Xilinx's XRT. We have also given an example which showcases several extensions. This has been a useful first step in aligning triSYCL with DPC++, showing it is possible to target Xilinx FPGA by adding some minor alterations to the compiler infrastructure and runtime. While the current implementation lacks performance we do not believe it impossible to reach performance equal to languages like HLS C++. The task is passing sufficient information to **xocc** so that it can optimise appropriately.

References

[1] R. Hornung, H. Jones, J. Keasler, R. Neely, O. Pearce, S. Hammond, C. Trott, P. Lin, C. Vaughan, J. Cook *et al.*, "ASC Tri-lab Co-design Level 2 Milestone Report 2015," Lawrence Livermore National Lab (LLNL), Livermore, CA (United States), Tech. Rep., 2015.

[2] H. C. Edwards, C. R. Trott, and D. Sunderland, "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," *Journal of Parallel and Distributed Computing*, vol. 74, no. 12, pp. 3202 – 3216, 2014, domain-Specific Languages and High-Level Frameworks for High-Performance Computing. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0743731514001257

[3] T. K. Group, "SYCL 1.2.1 Specification," Nov. 2019. [Online]. Available: https://www.khronos.org/registry/SYCL/specs/sycl-1.2.1.pdf

[4] H. Kaiser, T. Heller, B. Adelstein-Lelbach, A. Serio, and D. Fey, "HPX: A task based programming model in a global address space," in *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*. ACM, 2014, p. 6.

[5] Codeplay - ComputeCpp. [Online]. Available: https://www.codeplay.com/products/computesuite/computecpp

[6] R. Keryell and L.-Y. Yu, "Early experiments using SYCL single-source modern C++ on Xilinx FPGA: Extended abstract of technical presentation," in *Proceedings of the International Workshop on OpenCL*, 2018, pp. 1–8.

[7] (2018, Jul.) hipSYCL, Implementation of SYCL 1.2.1 over AMD HIP/NVIDIA CUDA. [Online]. Available: https://github.com/illuhad/hipSYCL

[8] P. Žužek. (2014, Oct.) SYCL-GTX: Implementation of the SYCL specification. [Online]. Available: https://github.com/ProGTX/sycl-gtx

[9] (2019, Jan.) Home for Intel LLVM-based projects: SYCL* Compiler and Runtimes. [Online]. Available: https://github.com/intel/llvm/tree/sycl

[10] (2019, Aug.) Intel® oneAPI Data Parallel C++: A Standards-Based, Cross-Architecture Language. [Online]. Available: https://software.intel.com/en-us/oneapi/dpc-compiler

[11] (2019) Experimental fusion of triSYCL with Intel SYCL. [Online]. Available: https://github.com/triSYCL/sycl

[12] L. Wirbel, "Xilinx SDAccel: a unified development environment for tomorrow's data center," *The Linley Group*, Nov. 2014.

[13] G. Chakrabarti, V. Grover, B. Aarts, X. Kong, M. Kudlur, Y. Lin, J. Marathe, M. Murphy, and J.-Z. Wang, "CUDA: Compiling and optimizing for a GPU platform," *Procedia Computer Science*, vol. 9, pp. 1910–1919, 2012.