SYCL C++ and OpenCL interoperability experimentation with triSYCL

Anastasios Doumoulakis
Ronan Keryell
Kenneth O'Brien
anastasi@xilinx.com
Ronan.Keryell@xilinx.com
kennetho@xilinx.com
Xilinx Research Labs
2020 Bianconi Avenue
Dublin, Ireland D24 T683

Abstract

Heterogeneous computing is required in systems ranging from low-end embedded systems up to the high-end HPC systems to reach high-performance while keeping power consumption low. Having more and more accelerators and CPUs also creates challenges for the programmer, requiring even more expertise of them. Fortunately, new modern C++-based domain-specific languages, such as the SYCL open standard from Khronos Group, simplify the programming at the full system level while keeping high performance.

SYCL is a single-source programming model providing a task graph of heterogeneous kernels that can be run on various accelerators or even just the CPU. The memory heterogeneity is abstracted through buffer objects and the memory usage is abstracted with accessor objects. From these accessors, the task graph is implicitly constructed, the synchronizations and the data movements across the various physical memories are done automatically, by opposition to OpenCL or CUDA.

Sometimes, some applications or libraries already exist using the OpenCL standard or some OpenCL kernels are provided, either as OpenCL kernel source code or even as built-in OpenCL kernels written in RTL for extreme optimization on FPGA.

SYCL provides an OpenCL interoperability mode to reuse existing OpenCL code while keeping the higher level task graph programming model without needing explicit memory transfers.

We present some experiments on two applications on GPU and FPGA with the triSYCL open-source implementation to show the benefits of this OpenCL interoperability mode.

CCS Concepts • **Computing methodologies → Parallel programming languages**; *Massively parallel algorithms*; • **Software and its engineering → Parallel programming languages**; **Domain specific languages**; *Object oriented frameworks*; • **Computer systems organization →** *Heterogeneous (hybrid) systems*;

Keywords C++17, SYCL, DSeL, OpenCL, FPGA, GPU, triSYCL

ACM Reference format:
Anastasios Doumoulakis, Ronan Keryell, and Kenneth O'Brien. 2017. SYCL C++ and OpenCL interoperability experimentation with triSYCL. In *Proceedings of Distributed & Heterogeneous Programming in C/C++ Workshop 2017, Toronto, Canada, May 2017 (DHPCC'2017)*, 8 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

1 Introduction

Computing architectures nowadays are huge hybrid multiprocessor system-on-chips with different kind of processors, GPU, configurable specific accelerators (video CODEC...), reconfigurable programmable logic (FPGA), various hierarchies of memory and memory interfaces, configurable IO and network support, security support, and power control etc. High-performance applications may use a hierarchy of such systems scaling up towards utilizing a full-scale data-center.

So the programmer is facing a fractal architecture, demanding also more and more control for power efficiency. This tends to require a dense fractal set of skills and tools.

SYCL [12, 13] is a new open standard from Khronos Group aiming at solving some of the programming issues related to heterogeneous computing. This pure C++14 (for SYCL 1.2) or C++17 (for SYCL 2.2) domain-specific embedded language allows the programmer to write single-source C++ host code with accelerated code expressed as functors. The data accesses are described with accessor objects that implicitly define a task graph that can be asynchronously scheduled on a distributed-memory system including several CPU and accelerators.

This programming model is quite generic but provides also an interoperability mode with the OpenCL realm, another standard from Khronos Group aimed at heterogeneous computing with a C host API and separate language for the kernels (C, C++, SPIR and SPIR-V). This allows a SYCL C++ application to recycle existing OpenCL kernels into a higher level C++ programming model, relieving the programmer from explicitly defining the memory transfers.

In this article we present in Section 2 the SYCL standard, then in Section 3 SYCL's interoperability mode with OpenCL, in Section 4 some experiments with the triSYCL open source implementation of the SYCL standard and comparing to some related work in Section 5.

2 SYCL

SYCL [12, 13] (pronounced "sickle") is a royalty-free, cross-platform abstraction C++ programming model for OpenCL [8, 9]. SYCL builds on the underlying concepts, portability and efficiency of OpenCL

```
// Demonstrate the use of an asynchronous task graph of kernels to
// initialize and add 2 matrices.
#include <CL/sycl.hpp>
#include <iostream>

using namespace cl::sycl;

// Size of the matrices
constexpr size_t N = 2000;
constexpr size_t M = 3000;

int main() {
  // Create a queue to work on
  queue q;

  // Create some 2D buffers of N*M floats for our matrices
  buffer<float, 2> a { { N, M } };
  buffer<float, 2> b { { N, M } };
  buffer<float, 2> c { { N, M } };

  // Launch a first asynchronous kernel to initialize a
  q.submit([&] (handler &cgh) {
      // The kernel writes a, so get a write accessor on it
      auto A = a.get_access<access::mode::write>(cgh);

      // Enqueue a parallel kernel iterating on a N*M 2D iteration space
      cgh.parallel_for<class init_a>({ N, M },
                                     [=] (id<2> index) {
                                       A[index] = index[0]*2 + index[1];
                                     });
    });

  // Launch an asynchronous kernel to initialize b
  q.submit([&] (handler &cgh) {
      // The kernel writes b, so get a write accessor on it
      auto B = b.get_access<access::mode::write>(cgh);
      /* From the access pattern above, the SYCL runtime detects this
         command group is independent from the first one and can be
         scheduled independently */

      // Enqueue a parallel kernel iterating on a N*M 2D iteration space
      cgh.parallel_for<class init_b>({ N, M },
                                     [=] (id<2> index) {
                                       B[index] = index[0]*2014 + index[1]*42;
                                     });
    });

  // Launch an asynchronous kernel to compute matrix addition c = a + b
    q.submit([&] (handler &cgh) {
        // In the kernel a and b are read, but c is written
        auto A = a.get_access<access::mode::read>(cgh);
        auto B = b.get_access<access::mode::read>(cgh);
        auto C = c.get_access<access::mode::write>(cgh);
        // From these accessors, the SYCL runtime will ensure that when
        // this kernel is run, the kernels computing a and b completed

        // Enqueue a parallel kernel iterating on a N*M 2D iteration space
        cgh.parallel_for<class matrix_add>({ N, M },
                                           [=] (id<2> index) {
                                             C[index] = A[index] + B[index];
                                           });
      });

    /* Request an accessor to read c from the host-side. The SYCL runtime
       ensures that c is ready when the accessor is returned */
    auto C = c.get_access<access::mode::read>();
    std::cout << std::endl << "Result:" << std::endl;
    for (size_t i = 0; i < N; i++)
      for (size_t j = 0; j < M; j++)
        // Compare the result to the analytic value
        if (C[i][j] != i*(2 + 2014) + j*(1 + 42)) {
          std::cout << "Wrong␣value␣" << C[i][j] << "␣on␣element␣"
                   << i << '␣' << j << std::endl;
          exit(-1);
        }

  std::cout << "Accurate␣computation!" << std::endl;
  return 0;
}
```

Figure 1. Example of a SYCL C++ program producing and adding 2 matrices, coming from from `https://github.com/triSYCL/triSYCL/blob/master/tests/examples/demo_parallel_matrix_add.cpp`.

while adding much of the ease of use and flexibility of single-source C++.

Developers using SYCL are able to write standard C++14/C++17 code, with many of the techniques they are accustomed to, such as inheritance and templating. At the same time developers have access to the full range of capabilities of OpenCL both through the features of the SYCL libraries and, where necessary, through interoperation with code written directly to the OpenCL APIs [9].

SYCL implements a single-source multiple compiler-passes design which offers the power of source integration while allowing tool-chains to remain flexible. This design supports embedding of code intended to be compiled for an OpenCL device, for example a GPU or an FPGA, inline with host code. This embedding of code offers three primary benefits:

simplicity: for novice programmers, the separation of host and device source code in OpenCL can become complicated to deal with, particularly when similar kernel code is used for multiple different operations. A single compiler flow and integrated tool chain combined with libraries that perform a lot of simple tasks simplifies initial OpenCL programs to a minimum complexity. This reduces the learning curve for programmers new to OpenCL and allows them to concentrate on parallelization techniques rather than syntax;

reuse: C++'s type system allows for complex interactions between different code units and supports efficient abstract interface design and reuse of library code. For example, a C++ `std::transform` or `std::for_each` algorithm applied to an array of data may allow specialization on both the operation applied to each element of the array and on the type of the data;

efficiency: tight integration with the type system and reuse of library code enables a compiler to perform inlining of code and to produce efficient specialized device code based on decisions made in the host code without having to generate kernel source strings dynamically as done with other frameworks [5, 15].

SYCL is a pure single-source C++ DSeL (Domain-Specific Embedded Language) providing simpler abstractions for heterogeneous computing. In Figure 1 is presented a small application using SYCL concepts to create a graph of 3 asynchronous tasks to initialize 2 matrices and addition them before checking for the final result.

The main interesting features that SYCL brings are:

asynchronous task graphs to break an application in parallel pieces able to run on various accelerators or on the host, taking advantage of the CPU cores and accelerators of the platform;

hierarchical parallelism to take advantage inside a task of the common intrinsic parallelism found in accelerators as shown on Figure 2, that can be expressed either as in OpenCL with ND-ranges (multi-dimensional iteration spaces) or in a similar hierarchical way to how it is done in OpenMP [16], but with a nicer C++-friendly method based on lambda functions. The simpler hierarchical way relieves the programmer from using painful work-group synchronizations and is also more efficient on CPU and FPGA;

buffers defining location-independent storage, (no explicit move) usable as multi-dimensional arrays to be used from the various CPU cores and accelerators;

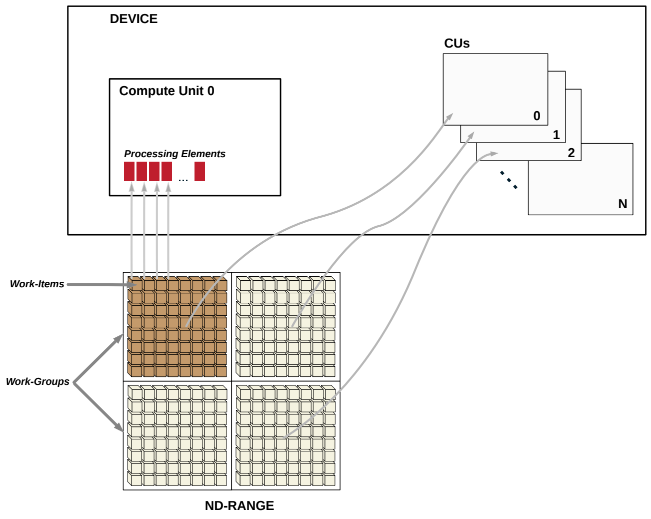

Figure 2. OpenCL execution model, with the parallel iteration space of work-groups of work-items mapped for execution onto independent compute units composed of processing elements.

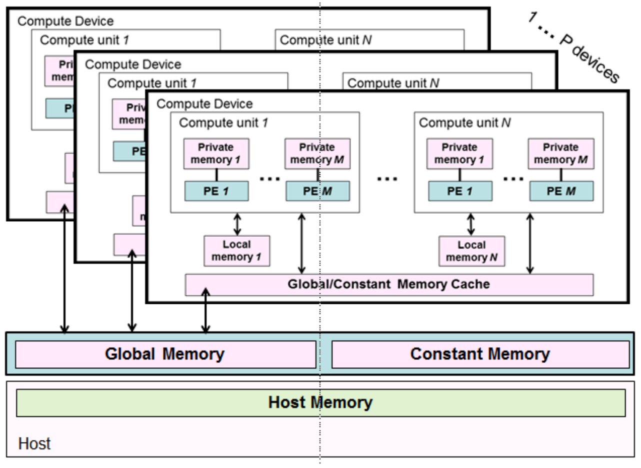

Figure 3. OpenCL memory model, with the 4 different explicit kinds of memory, besides the host memory.

accessors to express usage for buffers and other objects with some attributes such as read/write/..., the location or the kind of memory to use, allowing finer control on the complex memory hierarchy found on accelerators as shown on Figure 3 to reach the maximum power and compute efficiency;

implicit dependency graph construction is done with the separation in SYCL of the data access from data storage. By relying on the C++-style RAII (resource acquisition is initialization) idiom on accessors, the runtime library can capture data dependencies between device code blocks and construct the task graph implicitly with all the dependencies;

automatic data motion is then done by the runtime by tracking the data through the accessors ahead of time, making sure the data are available when needed by a kernel on a device or by the host, without the programmer requiring like in OpenCL or in CUDA to explicitly move the data;

overlapping kernels and communications is provided automatically by the SYCL scheduler by using the dependency graph between tasks without requiring the programmer to manage explicitly several command queues and synchronizing events for this;

single-source programming model, similar to OpenMP simplicity and type safety but in a pure C++-friendly world, without requiring `#pragma` that do not compose nicely with C++. This allows the writing of high-level programming and meta-programming. For example on the Figure 1, except in the buffer creations on lines 17–19, the data type does not appear anywhere in the code and is just inferred by the compiler even across the host-device boundary;

host fallback is a by-product of having a pure C++ DSeL. By just providing a C++ implementation of the SYCL runtime on the host. Thus the same code can work either on the host CPU or on the device, allowing more parallelism but also just to run even if a device is missing;

host debugging for free is a nice side-effect of the host implementation of SYCL. The development of heterogeneous applications is quite challenging but having the same code running on the host allows the use of the normal C++ development tools chains, from high-end static analysis tools, dynamic thread or memory checkers, debuggers, watchpoints, etc. down to inserting plain standard I/O messages in the code;

host emulation is also for free, which interesting in the case of the FPGA world where synthesizing the code for the device is an offline activity compared to running the code on CPU or even on GPU;

cross-platform support allows to have buffers used by different devices from different vendors in a seamless way, which is not possible directly in OpenCL.

SYCL retains the execution model, runtime feature set and device capabilities of the underlying OpenCL standard. This is why SYCL 1.2 [12] targets devices with OpenCL 1.2 [6] capabilities, while SYCL 2.2 [13] targets devices with OpenCL 2.2 [9] capabilities, adding for example the pipes and the shared virtual memory between the host and devices.

The OpenCL C specification imposes some limitations on the full range of C++ features that SYCL is able to support. This ensures portability of device code across as wide a range of devices as possible.

As a result, while the code can be written in standard C++ syntax with interoperability with standard C++ programs, the entire set of C++ features is not available in SYCL device code. In particular, SYCL device code, as defined by this specification, does not support virtual function calls, function pointers in general, exceptions, runtime type information or the full set of C++ libraries that may depend on these features or on features of a particular host compiler.

These features are not often used in high-performance code even in plain C++ in the hot-path because of performance issues. Fortunately, the use of C++ features such as templates and inheritance on top of the OpenCL execution model opens a wide scope for innovation in software design for heterogeneous systems, giving workarounds for some of the unsupported features.

Clean integration of device and host code within a single C++ type system enables the development of modern, templated libraries that build simple, yet efficient, interfaces to offer more developers access to OpenCL capabilities and devices. SYCL is intended to serve as a foundation for innovation in programming models for heterogeneous systems, that builds on an open and widely implemented standard foundation in the form of OpenCL.

This is why the OpenCL version of TensorFlow [17], the C++ machine learning framework from Google, is actually using SYCL instead of plain OpenCL.

SYCL is one of the candidates giving inputs on parallelism and heterogeneous computing to the C++ ISO/IEC JTC1/SC22/WG21 standardization committee [11, 18–20].

3 SYCL and OpenCL interoperability mode

SYCL is a very generic data-parallel task graph model implemented as a C++ DSeL that often relies on OpenCL and SPIR behind the hood to target accelerators, but it could use some other technology.

There is also in SYCL a specific OpenCL interoperability mode if needed, allowing direct interaction with the OpenCL world, and by transitivity to Vulkan/OpenGL/DirectX/... In this way it is possible to use existing programs or libraries with no overhead.

There are 2 main parts in the interoperability mode:

1. it is possible to construct SYCL objects from existing OpenCL objects to run SYCL single-source programs in relation with an existing OpenCL framework. For example a SYCL buffer can be constructed from an OpenCL `cl_mem` or SYCL queue from a `cl_command_queue`;
2. it is possible to get some OpenCL objects from higher-level SYCL objects to execute plain OpenCL code from the SYCL world, for example launching an OpenCL kernel that uses an implicit `cl_mem` associated to a SYCL accessor.

Whereas the interoperability mode was included in the SYCL standard to extend the applicability of SYCL on the OpenCL realm, it appears that this mode is useful by itself to do plain OpenCL programming in higher level C++, in the same way there already exists various C++ OpenCL wrapper.

While it does not take advantage from the single-source programming style of SYCL, it has some value for OpenCL programmers as it simplifies the boilerplate and housekeeping. For example, it allows to use the task graph model on top of OpenCL kernels and the synergy between buffers and accessors relieves the programmer from managing explicitly the buffer content transfers between host and devices.

In the following we present some use case of OpenCL interoperability using the triSYCL implementation.

4 Experimenting OpenCL interoperability mode of SYCL with triSYCL

triSYCL [10] is an open-source implementation based on C++17 and various Boost libraries [2]. The device compiler used to outline the kernels in single-source mode is based on Clang [3]/LLVM [14]. The OpenCL interoperability mode is based on Boost.Compute [15].

To even simplify further the programming style, triSYCL implements Boost.Compute interoperability mode too, as an extension to the SYCL standard.

We have used triSYCL to simplify the OpenCL host-side programming of our machine learning applications.

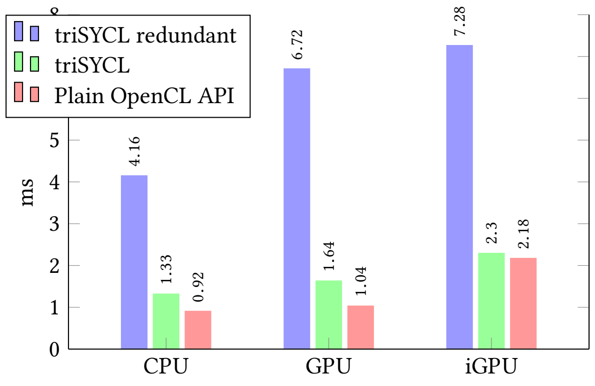

Figure 4. Average total execution time for digit recognition application per image running OpenCL kernels with different host API for 3 different accelerator architectures.

4.1 Handwritten digit recognition in gray images using L2 norm

To benchmark various languages and frameworks, we used an application for recognition of handwritten digits in 8-bit gray images of 28×28 pixels.

There are 500 images compared against a training reference set of labeled digit images. A kernel is launched to compare each image against all the reference images.

The examples are run with SYCL, SYCL with OpenCL interoperability running OpenCL kernel and a pure OpenCL implementations. The data sets can be found on `https://github.com/a-doumoulakis/triSYCL_knn` for further inspection.

The main part of the SYCL with OpenCL interoperability version is shown on Figure 5. Even if the kernel is described as OpenCL code with the help of Boost.Compute [15] for conciseness, the data are still stored in SYCL buffers and their usages are precised by some SYCL accessors. Compared to the pure SYCL version found in `https://github.com/a-doumoulakis/triSYCL_knn/blob/master/knn_trisycl.cpp`, we built a SYCL kernel from the OpenCL kernel and inside `q.submit()` the `cgh.set_args()` connects the SYCL world to the OpenCL kernel world, connecting the SYCL accessors to some OpenCL **`cl_mem`** provided to the OpenCL kernel during execution. The OpenCL kernel is still scheduled asynchronously by the SYCL runtime according to the task graph as would be a normal SYCL kernel.

Even if there is no explicit OpenCL buffer motion between host and device, the SYCL run-time uses the dependency graph constructed by the accessors to move or not to move the data for kernel execution.

The performance results comparing a dumb triSYCL version with redundant transfers between host and device as a reference[1], the one with optimized transfers and the pure OpenCL version are shown on Figure 4.

All the measurements were made on the same Linux machine (i7-6700HQ with 16 GB of RAM and an nVidia GTX 960M for the GPU) with 4.10.6-1 kernel and GCC 6.3.1. The OpenCL runtimes were nvidia-opencl 378.13-6 for the discrete nVidia GPU, beignet 1.3.1-1 for the integrated Intel GPU and intel-opencl-runtime 1:16.1.1-2 for the Intel CPU. The compiler options can be seen at the GitHub project page.

[1]The task graph is not used to optimize transfers or to limit OpenCL buffer creations, to serve as an implementation baseline.

Table 1. Performance Results (ms)

	Setup	*Memory*	*Build*	*Kernel*	*Total*
OpenCL	6490.22	.08	4598.1	4.011	11092.43
SYCL	6411.32	.002	4903.65	3.91	11325.57

Note that since current triSYCL [10] cannot target GPU yet and the community edition version of the major SYCL implementation, ComputeCpp [4], cannot run on our Linux distribution yet, we cannot compare the results against the normal pure SYCL code running on GPU.

4.2 Binary Neural Network for image classification on FPGA

In our second example, we present the performance results of a binary neural network based application, which can accelerate its computation on an FPGA using SYCL or OpenCL APIs. The SYCL implementation uses triSYCL with OpenCL interoperabilty mode or by using directly the Xilinx OpenCL implementation, SDAccel. In this particular application, image data is passed as an input to the accelerator for classification. The output is a label of what the image contains.

The kernel code of the application is written in High Level Synthesis C++, which, when synthesized for the FPGA, acts as a built-in OpenCL binary kernel that is loaded by either SYCL or OpenCL runtimes.

In Figures 6 and 7, we display the differences in kernel execution code section of the OpenCL and SYCL implementations. In the case of the SYCL implementation, when the *inputBuffer* is created, its data is loaded from the host to the device from *inputVector*. The code following this is contained within braces, ensuring the kernel has completed execution and the data from the output has been copied back to the host to the *outputVector* before executing subsequent code. The kernel arguments are also set with a single variadic function call. In the case of the OpenCL implementation, copies between host and device are explicit and kernel arguments must be set individually.

In Table 1, we report the performance of four critical sections our application and the total execution time for each. The results are an average of 10 executions. Measurements are taken from the wall clock using a high resolution timer. *Setup* accounts for the time to acquire an OpenCL platform, device, context and queue. *Memory* accounts of the buffer allocation time of the input and output. *Build* describes the time taken to read the binary kernel file from disk, build an OpenCL program, and, from it, instantiate the kernel. For both cases, this includes the reconfiguration of the FPGA. *Kernel* accounts for the execution of the kernel, including the setting of arguments and launch. Finally, *Total* accounts for the entirety of the accelerated section, from setup until the results are returned to the host.

Our findings with this example is that SYCL with OpenCL interoperability mode is competitive with directly using OpenCL, with the exception of a small one-time overhead during the build phase. We are currently working to resolve this issue. Encouragingly, the kernel performance between both are comparable.

```
  [...]

int search_image(buffer<int>& training, const Img& img, queue& q, const kernel& k) {
  int res[training_set_size];

  {
    buffer<int> A { std::begin(img.pixels), std::end(img.pixels) };
    buffer<int> B { res, training_set_size };
    // Compute the L2 distance between an image and each one from the
    // training set
    q.submit([&] (handler &cgh) {
        // Set the kernel arguments. The accessors lazily trigger data
        // transfers between host and device only if necessary. For
        // example "training" is only transfered the first time the
        // kernel is executed.
        cgh.set_args(training.get_access<access::mode::read>(cgh),
                     A.get_access<access::mode::read>(cgh),
                     B.get_access<access::mode::write>(cgh),
                     int { training_set_size }, int { pixel_number });
        // Launch the kernel with training_set_size work-items
        cgh.parallel_for(training_set_size, k);
      });
    // The destruction of B here waits for kernel execution and copy
    // back the data to res
  }

  // Find the image with the minimum distance
  auto min_image = std::min_element(std::begin(res), std::end(res));

  // Test if we found the correct digit
  return
    training_set[std::distance(std::begin(res), min_image)].label == img.label;
}

int main(int argc, char* argv[]) {
  int correct = 0;
  training_set = slurp_file("data/trainingsample.csv");
  validation_set =  slurp_file("data/validationsample.csv");
  buffer<int> training_buffer = get_buffer(training_set);

  // A SYCL queue to send the heterogeneous work-load to
  queue q { boost::compute::system::default_queue() };

  // Use real OpenCL program for the kernel
  auto program = boost::compute::program::create_with_source(R"(
␣␣␣␣__kernel␣void␣kernel_compute(__global␣const␣int*␣trainingSet,
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣__global␣const␣int*␣data,
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣__global␣int*␣res,␣int␣setSize,␣int␣dataSize)␣{
␣␣␣␣␣␣int␣diff,␣toAdd,␣computeId;
␣␣␣␣␣␣computeId␣=␣get_global_id(0);
␣␣␣␣␣␣if␣(computeId␣<␣setSize)␣{
␣␣␣␣␣␣␣␣diff␣=␣0;
␣␣␣␣␣␣␣␣for␣(int␣i␣=␣0;␣i␣<␣dataSize;␣i++)␣{
␣␣␣␣␣␣␣␣␣␣␣␣toAdd␣=␣data[i]␣-␣trainingSet[computeId*dataSize␣+␣i];
␣␣␣␣␣␣␣␣␣␣␣␣diff␣+=␣toAdd␣*␣toAdd;
␣␣␣␣␣␣␣␣}
␣␣␣␣␣␣␣␣res[computeId]␣=␣diff;
␣␣␣␣␣␣}
␣␣␣␣}
␣␣␣␣)", boost::compute::system::default_context());

  program.build();

  // Construct a SYCL kernel from OpenCL kernel to be used in
  // interoperability mode
  kernel k { boost::compute::kernel { program, "kernel_compute"} };

  // Match each image from the validation set against the images from
  // the training set
  for (auto const & img : validation_set)
    correct += search_image(training_buffer, img, q, k);

  [...]

  return 0;
}
```

Figure 5. Digit recognition application using SYCL with OpenCL interoperability mode.

```
[...]
  buffer<long> inputBuffer { inputVector, numInputElems };
  {
    buffer<long> outputBuffer { outputVector, numOutputElems };
    queue.submit([&] (handler &cgh) {
      cgh.setArgs(inputBuffer.get_access<access::mode::read>(cgh),
                  outputBuffer.get_access<access::mode::write>(cgh),
                  false, 0, 0, 0, 0L, count);
      cgh.single_task(kernel);
    });
  }
[...]
```

Figure 6. SYCL kernel launch example.

```
[...]
  queue.enqueueWriteBuffer(inputBuffer, CL_TRUE, 0, numInputElems, inputVector, NULL, &event);
  kernel.setArgs(0, inputBuffer);
  kernel.setArgs(1, outputBuffer);
  kernel.setArgs(2, false);
  kernel.setArgs(3, 0);
  kernel.setArgs(4, 0);
  kernel.setArgs(5, 0);
  kernel.setArgs(6, 0L);
  kernel.setArgs(7, count);
  queue.enqueueTask(kernel, NULL, &event);
  queue.enqueueReadBuffer(outputBuffer, CL_TRUE, 0, numOutputElems, outputVector, NULL, &event);
[...]
```

Figure 7. OpenCL kernel launch example.

5 Related work

The problems addressed here are really meaningful if we consider all the frameworks developed by a lot of people using heterogeneous computing. This also means it is impossible to be exhaustive, even by looking at the C++ frameworks only.

The natural candidate is the second version of the official C++ wrapper from Khronos, `cl2.hpp` [7]. It is a straightforward C++ mapping around the OpenCL C API, at least managing the lifetime of the OpenCL using RAII mechanism. There is a variadic C++ API to launch kernels, but it is not very high-level compared to plain OpenCL. Since it is a basic C++ API, this API is often used by other higher-level API.

For example Boost.Compute [15] is built on top of the basic previous one and offers also OpenCL API in a C++ mood. Boost.Compute is actually 2 different API. A basic layer is like `cl2.hpp` but in a more modern C++ and STL mind. This is why it is used by triSYCL. But there is also a higher level API, with parallel STL-like algorithms operating on device vectors. But there is no transparent motion of the data of these device vectors between host and device.

VexCL [5] is higher-level than Boost.Compute since it can target both OpenCL or CUDA. It provides parallel STL algorithms with vectors that can be spread across several devices. There is some support to generate kernels by using symbolic execution. But there is still some explicit copying required between device vectors and the host domain.

Google contributed the Acxxel library [1] inside the LLVM compiler runtime `parallel-libs`. It allows programmers to manage OpenCL and CUDA devices and to launch kernels on them, while unifying most of the similar concepts on the host side within a unique syntax. There is no dependency graph between tasks and thus the transfers between host and devices are still to be done by the programmer.

6 Conclusion

Heterogeneous computing in embedded and high-performance computing is here to stay because of physical constraints. This puts the pressure on the programmers to integrate a full system across the various accelerators. The SYCL standard C++ DSeL allows a single-source approach for both host and accelerators parts in type-safe way to simplify the process while interoperable with the ubiquitous C/C++ world. The SYCL runtime provides an implicit task graph managing asynchronicity and data transfers across the various memory spaces.

Besides this very general programming model, SYCL provides also interoperability with the OpenCL world, allowing to launch existing OpenCL kernels while taking advantage of the SYCL framework. While no longer a single-source programming model in that case, it still provides the implicit task graph with buffers and accessors, relieving the programmer to manage explicitly the buffers and memory transfers between the host and the devices.

This interoperability mode in the SYCL standard has some value by itself even if this role was not envisioned in the first place. It allows to use SYCL as a high-level framework to run existing OpenCL kernels (even built-in kernels controlling networking devices on FPGA) using a pure C++ approach without the need of a device compiler.

The open-source triSYCL implementation [10] we are working on provides also this interoperability mode, as shown with the application samples presented in this article and the performance comparisons with other frameworks.

While the performance is generally not better with SYCL than with plain OpenCL kernels using the OpenCL host API, SYCL increases the programmer's productivity with the single-source type-safe programming model for both host code and kernel code and the task graph model with implicit data transfers.

When interoperability with existing OpenCL code or OpenCL-compatible kernels is required, for example with FPGA kernels programmed directly in RTL or HLS C++, the OpenCL interoperability mode of SYCL provides a nicer host API, simpler to use than the usual OpenCL ones, without requiring a specific SYCL device compiler.

References

[1] Acxxel 2016. LLVM parallel-libs Subproject Charter — Acxxel. (Oct. 2016). `https://github.com/llvm-project/parallel-libs`

[2] Boost 2016. Boost C++ libraries 1.63. (Dec. 2016). `http://www.boost.org`

[3] Clang 2016. Clang: a C language family frontend for LLVM 4.0. (March 2016). `https://clang.llvm.org/`

[4] ComputeCpp 2017. ComputeCpp — Accelerate Complex C++ Applications on Heterogeneous Compute Systems using Open Standards. (March 2017). `https://www.codeplay.com/products/computesuite/computecpp`

[5] Denis Demidov. 2016. VexCL: a C++ vector expression template library for OpenCL/CUDA. (Oct. 2016). `https://github.com/ddemidov/vexcl`

[6] Khronos OpenCL Working Group. 2012. The OpenCL Specification, Version: 1.2, Document Revision: 19. (Nov. 2012). `https://www.khronos.org/registry/OpenCL/specs/opencl-1.2.pdf`

[7] Khronos OpenCL Working Group. 2016. OpenCL C++ Bindings. (July 2016). `https://github.com/KhronosGroup/OpenCL-CLHPP`

[8] Khronos OpenCL Working Group. 2016. The OpenCL C++ Specification, Version: 1.0, Document Revision: 22. (April 2016). `https://www.khronos.org/registry/OpenCL/specs/opencl-2.2-cplusplus.pdf`

[9] Khronos OpenCL Working Group. 2016. The OpenCL Specification, Version: 2.2, Document Revision: 06. (March 2016). `https://www.khronos.org/registry/OpenCL/specs/opencl-2.2.pdf`

[10] Ronan Keryell. 2014. triSYCL — An open source implementation of OpenCL SYCL from Khronos Group. (April 2014). `https://github.com/triSYCL/triSYCL`

[11] Ronan Keryell and Joël Falcou. 2016. Accessors — a C++ standard library class to qualify data accesses. (May 2016). `www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0367r0.pdf`

[12] SYCL subgroup Khronos OpenCL Working Group. 2015. SYCL Specification, SYCL integrates OpenCL devices with modern C++. (May 2015). `https://www.khronos.org/registry/SYCL/specs/sycl-1.2.pdf`

[13] SYCL subgroup Khronos OpenCL Working Group. 2016. SYCL Provisional Specification, SYCL integrates OpenCL devices with modern C++ using a single source design. (Feb. 2016). `https://www.khronos.org/registry/SYCL/specs/sycl-2.2.pdf`

[14] LLVM 2016. The LLVM Compiler Infrastructure 4.0. (March 2016). `http://llvm.org`

[15] Kyle Lutz. 2014. Boost.Compute. (Dec. 2014). `http://boostorg.github.io/compute/`

[16] All members of the OpenMP Language Working Group. 2016. OpenMP Technical Report 4: Version 5.0 Preview 1. (Nov. 2016). `http://www.openmp.org/wp-content/uploads/openmp-tr4.pdf`

[17] TensorFlow 2017. TensorFlow 1.0: An open-source software library for Machine Intelligence. (Feb. 2017). `https://www.tensorflow.org/`

[18] Michael Wong, Andrew Richards, Maria Rovatsou, and Ruyman Reyes. 2016. Khronos's OpenCL SYCL to support Heterogeneous Devices for C++. (Feb. 2016). `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0236r0.pdf`

[19] Michael Wong, Andrew Richards, Maria Rovatsou, Ruyman Reyes, Lee Howes, and Gordon Brown. 2016. Towards support for Heterogeneous Devices in C++ (Concurrency aspects). (May 2016). `www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0362r0.pdf`

[20] Michael Wong, Andrew Richards, Maria Rovatsou, Ruyman Reyes, Lee Howes, and Gordon Brown. 2016. Towards support for Heterogeneous Devices in C++ (Language aspects). (May 2016). `www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0363r0.pdf`