Backend interoperability in SYCL 2020

AMD: Luc Forget, Gauthier Harnisch, Ronan Keryell
ANL: Thomas Applencourt, Nevin Liber
Codeplay: Gordon Brown
Heidelberg University: Aksel Alpay
Intel: Greg Lueck

SYCLcon 2022
SYCL Backend Model

SYCL Application

SYCL Runtime Library

- OpenCL
  - h/w device
- Level Zero
  - h/w device
- ... (omitted)
- HIP
  - h/w device
- CUDA
  - h/w device
  - h/w device

SYCL API is exposed here

Compiler and/or runtime translate SYCL constructs into parallelism on some backend (typically)

Some typical backends

NO guarantee that a SYCL implementation supports these (or any) backends!
What is SYCL Backend Interoperability?

I know my SYCL implementation runs on top of OpenCL (e.g.) …

… so I want to call OpenCL APIs here intermixed with SYCL APIs
Reasons for Backend Interoperability

- **Porting existing code**
  - Code already based on OpenCL (or whatever backend)
  - Want to change just part of application to use SYCL

- **Incorporating a backend module into a SYCL application**
  - Application based on SYCL
  - Want to call some OpenCL library (or whatever backend)

- **Take advantage of backend-specific features**

- **Disadvantage: Reduces portability!**
  - Not all implementations may support your backend
Three Main Types of Backend Interoperability
Type 1: SYCL Object from Backend Object

void MyFunc(cl_device_id clDev) {
    sycl::device dev = sycl::make_device<sycl::backend::opencl>(clDev);
    sycl::queue q{dev};
    q.submit([&](sycl::handler &cgh) {
        cgh.parallel_for(/* ... */);
    }).wait();
}

Typical usage: Adding SYCL functionality to an existing backend-specific application.

Construct a SYCL device from an OpenCL device ID. Similar functions for most SYCL objects.
void MyFunc(sycl::device dev) {
    #ifdef SYCL_BACKEND_OPENCL
        cl_device_id clDev = sycl::get_native<sycl::backend::opencl>(dev);
        char builtins[SIZE];
        size_t sz;
        clGetDeviceInfo(clDev, CL_DEVICE_BUILT_IN_KERNELS, SIZE, builtins, &sz);
        /* Use OpenCL builtin */
    #else
        /* fallback if no OpenCL backend */
    #endif
}

This macro only defined if implementation supports OpenCL

Get underlying OpenCL device ID from SYCL device.

Typical usage: Incorporate a backend-specific library into a SYCL application or take advantage of a backend-specific feature.
Type 3: Schedule a Backend Specific Command

```c
void MyFunc(sycl::queue q, sycl::buffer<int> buf) {
    #ifdef SYCL_BACKEND_OPENCL
        q.submit([&](sycl::handler &cgh) {
            sycl::accessor acc{buf, cgh};
            cgh.host_task([&](const sycl::interop_handle &ih) {
                cl_mem clMem = ih.get_native_mem<sycl::backend::opencl>({acc}[0];
                /* use OpenCL APIs with clMem */
            });
        });
    #endif
}
```

This command scheduled according to dependency graph on accesses to buffer “buf”.

Get underlying cl_mem for a buffer accessor.

Typical usage: Same as previous.
Host Task

- The **host_task** schedule arbitrary C++ code within the SYCL data-dependency graph
- Work performed in a **host_task** must complete before the function returns

```cpp
gpuQueue.submit([&](const sycl::handler &cgh){
    sycl::accessor acc{buf, cgh, sycl::read_write_host_task};
    cgh.host_task([&]{
        auto ptr = acc.get_pointer();
        some_cpu_library(ptr);
    });
});
```
Host Task Interoperability

- An optional `interop_handle` parameter can be used to perform backend interoperability within a `host_task`

- The `interop_handle` is used to retrieve the native queue, event or memory objects

```cpp
gpuQueue.submit([&](const sycl::handler &cgh){
    sycl::accessor acc{buf, cgh, sycl::read_write_host_task};
    cgh.host_task([&](const sycl::interop_handle &ih){
        auto nativeStream = ih.get_native_queue<sycl::backend::cuda>();
        auto nativePtr = ih.get_native_mem<sycl::backend::cuda>(acc);
        some_native_library(nativeStream, nativePtr);
    });
});
```
Host Task interoperability in oneMKL and oneDNN

- The oneAPI libraries oneMKL and oneDNN provide an abstraction for BLAS, RAND and DNN operations across different SYCL backends
  - https://github.com/oneapi-src/oneDNN
  - https://github.com/oneapi-src/oneMKL

- For the Nvidia backend oneMKL and oneDNN map to the CUDA libraries cuBLAS, cuRAND, cuSolver and cuDNN

- Each oneMKL or oneDNN operation maps to a host_task executing the equivalent library function

- Each operation supports both buffer/accessor and USM memory management models

- As the host_task is executed within the SYCL data dependency graph these operations are automatically composable
Convolution Example from oneDNN

- Example of host_task in oneDNN for Nvidia backend

```cpp
... compat::host_task(cgh, [=](const compat::interop_handle &ih) {
    auto &sycl_engine = *utils::downcast<sycl_cuda_engine_t *>(cuda_stream->engine());
    auto sc = cuda_sycl_scoped_context_handler_t(sycl_engine);
    auto handle = cuda_stream->get_cudnn_handle();

    std::vector<void *> args;
    args.push_back(arg_src.get_native_pointer(ih));
    args.push_back(arg_weights.get_native_pointer(ih));
    args.push_back(arg_dst.get_native_pointer(ih));
    args.push_back(arg_bias.get_native_pointer(ih));
    args.push_back(arg_scratch.get_native_pointer(ih));
    args.push_back(arg_filter_scratch.get_native_pointer(ih));
    args.push_back(temp_dst.get_native_pointer(ih));
    args.push_back(temp_reorder.get_native_pointer(ih));

    pd()->impl_->execute(handle, args);
});
...```
Host Task Interoperability Conclusion

- The `host_task` interoperability is useful in cases where you have a SYCL application and you wish to call native backend APIs or libraries.
- This can be any backend; OpenCL, OpenMP, CUDA, HIP, Level Zero, etc.
- The backend interoperability within a `host_task` provides access to the native queue, event and memory objects.
- The `host_task` is enqueued within the data dependency graphs synchronizes with other SYCL kernels.
OpenMP ↔ SYCL: An HPC Story

- “Most” HPC applications are written in OpenMP
- But they may want to be interfaced with SYCL:
  - Some SYCL API are more flexible than the OpenMP counterpart
    - oneMKL provide both an OpenMP and SYCL API,
      but SYCL API give access to more function (batched DGEMM for example)
  - Some Library only provide a SYCL API
    - For example oneDPL (Intel oneAPI thrust)
- Interoperability to the rescue!

```c
#pragma omp target enter data map(to: data[0:N])
T* data_gpu;
#pragma omp target data use_device_ptr(data) { data_gpu = data }
std::sort(oneapi::dpl::execution::make_device_policy(q), data_gpu, data_gpu + N);
```
OpenMP → Backend → SYCL

- Use `#pragma omp interop` to get Native Handler (*OpenMP 5.1*)
- Use those handlers to create SYCL Object
- POC: Implementation using L0 API and ICPX:
  - Ensure that SYCL and OpenMP use the same context

- **Code example:**

```cpp
omp_interop_t o;
#pragma omp interop init(targetsync: o)
auto hDevice = static_cast<ze_device_handle_t>(
    omp_get_interop_ptr(o, omp_ipr_device, &err));
#pragma omp interop destroy(o)
const sycl::device sycl_device =
    sycl::make_device<sycl::backend::ext_oneapi_level_zero>(hDevice);
```
OpenMP ↔ SYCL Work

• Pro: It work!

```cpp
sycl::queue Q = get_interop_queue(); // Where the magic happens
T *ompMem = (T*) malloc(N*sizeof(T));
T *syclMem = sycl::malloc_device<T>(N,Q);
```

• OpenMP using SYCL memory

```cpp
#pragma omp target is_device_ptr(syclMem) map(from:ompMem[0:N])
for (size_t i=0 ; i < N; i++)
    ompMem[i] = syclMem[i];
```

• SYCL using OpenMP memory

```cpp
T* ompMem_gpu;
#pragma omp target enter data map(to:ompMem[0:N])
#pragma omp target data use_device_ptr(ompMem) { ompMem_gpu = ompMem }
Q.copy<T>(cpuMem, ompMem_gpu, N).wait();
```

• Con:

- Backend specific (need to cast pointer | specialize the templated API)
- Only tested on ICPX & need to use non-standard interoperability API to workaround some bugs

Collaboration are welcome to tests/implements support for more compiler / backend!
SYCL as a higher-level API for a native API?

- **Provide**
  - Modern C++
  - Simpler buffer allocation
  - Automatic data transfers with accessors
  - Embed existing API in SYCL task graph
  - Automatic compute & communication overlap between kernels
  - Asynchronous execution
  - ...
  - Can mix and match OpenCL/CUDA/Level 0/HIP/XRT...

- **Yes, native APIs like OpenCL have already plenty of C++ wrappers!**
  - OpenCL.hpp from Khronos [https://github.com/KhronosGroup/OpenCL-CLHPP/blob/main/include/CL/opencl.hpp](https://github.com/KhronosGroup/OpenCL-CLHPP/blob/main/include/CL/opencl.hpp)
  - Boost.Compute [https://github.com/boostorg/compute](https://github.com/boostorg/compute)
  - ...

- **SYCL: 1 high-level wrapper to rule them all, in the same application**
  - Just one to learn!

- **Important note: this is no longer normal single-source SYCL!**
#include <iostream>
#include <string>
#include <vector>

#include <CL/opencl.h>

/** Transform the value of a given symbol to a string. Since we expect a macro symbol, use a double evaluation... */
#define __STRING(s) #s

#define THROW_ERROR(message)
    /** Throw a nicer error message in the code by adding the file name and the position */
    __throw_stringyf(s, _stringyf(s))
    __throw_stringyf(s, _stringyf(s))
    __throw_stringyf(s, _stringyf(s))

#include <CL/opencl.h>
#include <string>

/** Throw a nicer error message in the code by adding the file name and the position */
#define __STRING(s) #s

#define __throw_stringyf(s, _stringyf(s))
    /** Throw a nicer error message in the code by adding the file name and the position */
    __throw_stringyf(s, _stringyf(s))
    __throw_stringyf(s, _stringyf(s))

Vector addition in OpenCL C + OpenCL host C API

This work is licensed under a Creative Commons Attribution 4.0 International License

142 lines https://github.com/keryell/heterogeneous_examples/blob/main/vector_add/OpenCL/opencl_vector_add.cpp
SYCL as a high-level host API to run OpenCL kernels

```cpp
#include <cassert>
#include <cstdlib>
#include <sycl/sycl.hpp>
#include <CL/opencl.h>

constexpr int size = 4;

auto check_error(auto&& function) {
  cl_int err;
  auto ret = function(&err);
  if (err != CL_SUCCESS)
    std::exit(err);
  return ret;
};

int main() {
  sycl::buffer<int> a { size };  
  sycl::buffer<int> b { size };  
  sycl::buffer<int> c { size };  

  sycl::host_accessor a_a { a };  
  sycl::host_accessor a_b { b };  
  for (int i = 0; i < size; ++i) {  
    a_a[i] = i;  
    a_b[i] = i + 42;  
  }

  sycl::queue q;

  std::array kernel_source { R"(
    __kernel void vector_add(const __global float *a,
    const __global float *b,
    __global float *c) {
      c[get_global_id(0)] = a[get_global_id(0)] + b[get_global_id(0)];
    }"")
  ];

  cl_context oc = sycl::get_native<sycl::backend::opencl>(q.get_context());
  auto program = check_error([&](auto err) {
    return clCreateProgramWithSource(oc, kernel_source.size(),
                                       kernel_source.data(), nullptr, err);
  });
  check_error([&](auto err) {
    return (*err = clBuildProgram(program, 0, nullptr, nullptr, nullptr));
  });
  sycl::kernel k = sycl::make_kernel<sycl::backend::opencl>(
    check_error([&](auto err) {
      return clCreateKernel(program, "vector_add", err);
    }), q.get_context());

  q.submit([&](sycl::handler& cgh) {  
    cgh.set_args(sycl::accessor { a, cgh, sycl::read_only },
                  sycl::accessor { b, cgh, sycl::read_only },
                  sycl::accessor { c, cgh, sycl::write_only, sycl::no_init });
    cgh.parallel_for(size, k);
  });

  {  
    sycl::host_accessor a_a { a };  
    sycl::host_accessor a_b { b };  
    sycl::host_accessor a_c { c };  
    for (int i = 0; i < size; ++i)  
      assert(a_c[i] == a_a[i] + a_b[i]);  
  }
};
```

66 lines


This work is licensed under a Creative Commons Attribution 4.0 International License
FPGA kernels in HLS C++ + XRT host application

https://github.com/Xilinx/Vitis_Accel_Examples/tree/master/host_xrt/hello_world_xrt

- Kernel code `src/vadd.cpp` 64 lines of HLS C++
- Host code `src/host.cpp` 81 lines of C++
- Both are rather high-level

```c
extern "C" {
    void vadd(unsigned int* in1, unsigned int* in2, unsigned int* out, int size) {
        static hls::stream<unsigned int> inStream1("input_stream_1");
        static hls::stream<unsigned int> inStream2("input_stream_2");
        static hls::stream<unsigned int> outStream("output_stream");

        #pragma HLS INTERFACE m_axi port = in1 bundle = gmem0
        #pragma HLS INTERFACE m_axi port = in2 bundle = gmem1
        #pragma HLS INTERFACE m_axi port = out bundle = gmem0

        #pragma HLS dataflow
        // dataflow pragma instruct compiler to run following three APIs in parallel
        read_input(in1, inStream1, size);
        read_input(in2, inStream2, size);
        compute_add(inStream1, inStream2, outStream, size);
        write_result(out, outStream, size);
    }
}
```

Input Vector 1 from Global Memory
```plaintext
load_input
__________
|          |<------ Input Vector 1 from Global Memory
|__________|
load_input
__________
|          |<------ Input Vector 1 from Global Memory
|__________|
load_input
__________
|          |<------ Input Vector 1 from Global Memory
|__________|
load_input
__________
|          |<------ Input Vector 1 from Global Memory
|__________|
load_input
__________
|          |<------ Input Vector 1 from Global Memory
|__________|
load_input
__________
|          |<------ Input Vector 1 from Global Memory
|__________|
load_input
__________
|          |<------ Input Vector 1 from Global Memory
|__________|
load_input
__________
|          |<------ Input Vector 1 from Global Memory
|__________|
load_input
__________
|          |<------ Input Vector 1 from Global Memory
|__________|
load_input
__________
|          |<------ Input Vector 1 from Global Memory
|__________|
load_input
__________
|          |<------ Input Vector 1 from Global Memory
|__________|
load_input
__________
|          |<------ Input Vector 1 from Global Memory
|__________|
load_input
__________
|          |<------ Input Vector 1 from Global Memory
|__________|
load_input
__________
|          |<------ Input Vector 1 from Global Memory
|__________|
load_input
__________
|          |<------ Input Vector 1 from Global Memory
|__________|
load_input
__________
|          |<------ Input Vector 1 from Global Memory
|__________|
load_input
__________
|          |<------ Input Vector 1 from Global Memory
|__________|
load_input
__________
|          |<------ Input Vector 1 from Global Memory
|__________|
load_input
__________
|          |<------ Input Vector 1 from Global Memory
|__________|
load_input
__________
|          |<------ Input Vector 1 from Global Memory
|__________|
load_input
__________
|          |<------ Input Vector 1 from Global Memory
|__________|
load_input
__________
|          |<------ Input Vector 1 from Global Memory
|__________|
load_input
__________
|          |<------ Input Vector 1 from Global Memory
|__________|
load_input
__________
|          |<------ Input Vector 1 from Global Memory
|__________|
load_input
__________
|          |<------ Input Vector 1 from Global Memory
|__________|
load_input
__________
|          |<------ Input Vector 1 from Global Memory
|__________|
load_input
__________
|          |<------ Input Vector 1 from Global Memory
|__________|
load_input
__________
|          |<------ Input Vector 1 from Global Memory
|__________|
load_input
__________
|          |<------ Input Vector 1 from Global Memory
|__________|
load_input
__________
|          |<------ Input Vector 1 from Global Memory
|__________|
load_input
__________
|          |<------ Input Vector 1 from Global Memory
|__________|
load_input
__________
|          |<------ Input Vector 1 from Global Memory
|__________|
load_input
__________
|          |<------ Input Vector 1 from Global Memory
|__________|
load_input
__________
|          |<------ Input Vector 1 from Global Memory
|__________|
load_input
__________
|          |<------ Input Vector 1 from Global Memory
|__________|
load_input
__________
|          |<------ Input Vector 1 from Global Memory
|__________|
load_input
__________
|          |<------ Input Vector 1 from Global Memory
|__________|
load_input
__________
|          |<------ Input Vector 1 from Global Memory
|__________|
load_input
__________
|          |<------ Input Vector 1 from Global Memory
|__________|
load_input
__________
|          |<------ Input Vector 1 from Global Memory
|__________|
load_input
__________
|          |<------ Input Vector 1 from Global Memory
|__________|
load_input
__________
|          |<------ Input Vector 1 from Global Memory
|__________|
load_input
__________
|          |<------ Input Vector 1 from Global Memory
|__________|
load_input
__________
|          |<------ Input Vector 1 from Global Memory
|__________|
load_input
__________
|          |<------ Input Vector 1 from Global Memory
|__________|
load_input
__________
|          |<------ Input Vector 1 from Global Memory
|__________|
load_input
__________
|          |<------ Input Vector 1 from Global Memory
|__________|
load_input
__________
|          |<------ Input Vector 1 from Global Memory
|__________|
load_input
__________
|          |<------ Input Vector 1 from Global Memory
|__________|
load_input
__________
|          |<------ Input Vector 1 from Global Memory
|__________|
load_input
__________
|          |<------ Input Vector 1 from Global Memory
|__________|
load_input
__________
|          |<------ Input Vector 1 from Global Memory
|__________|
load_input
__________
|          |<------ Input Vector 1 from Global Memory
|__________|
load_input
__________
|          |<------ Input Vector 1 from Global Memory
|__________|
load_input
__________
|          |<------ Input Vector 1 from Global Memory
|__________|
load_input
__________
|          |<------ Input Vector 1 from Global Memory
|__________|
load_input
__________
|          |<------ Input Vector 1 from Global Memory
|__________|
load_input
__________
|          |<------ Input Vector 1 from Global Memory
|__________|
load_input
__________
|          |<------ Input Vector 1 from Global Memory
|__________|
load_input
__________
|          |<------ Input Vector 1 from Global Memory
|__________|
load_input
__________
|          |<------ Input Vector 1 from Global Memory
|__________|
load_input
__________
|          |<------ Input Vector 1 from Global Memory
|__________|
load_input
__________
|          |<------ Input Vector 1 from Global Memory
|__________|
load_input
__________
|          |<------ Input Vector 1 from Global Memory
|__________|
load_input
__________
|          |<------ Input Vector 1 from Global Memory
|__________|
load_input
__________
|          |<------ Input Vector 1 from Global Memory
|__________|
load_input
__________
|          |<------ Input Vector 1 from Global Memory
|__________|
load_input
__________
|          |<------ Input Vector 1 from Global Memory
|__________|
load_input
__________
|          |<------ Input Vector 1 from Global Memory
|__________|
load_input
__________
|          |<------ Input Vector 1 from Global Memory
|__________|
load_input
__________
|          |<------ Input Vector 1 from Global Memory
|__________|
load_input
__________
|          |<------ Input Vector 1 from Global Memory
|__________|
load_input
__________
|          |<------ Input Vector 1 from Global Memory
|__________|
load_input
__________
|          |<------ Input Vector 1 from Global Memory
|__________|
load_input
__________
|          |<------ Input Vector 1 from Global Memory
|__________|
load_input
__________
|          |<------ Input Vector 1 from Global Memory
|__________|
load_input
__________
|          |<------ Input Vector 1 from Global Memory
|__________|
load_input
__________
|          |<------ Input Vector 1 from Global Memory
|__________|
load_input
__________
|          |<------ Input Vector 1 from Global Memory
|__________|
load_input
__________
|          |<------ Input Vector 1 from Global Memory
|__________|
load_input
__________
|          |<------ Input Vector 1 from Global Memory
|__________|
load_input
__________
|          |<------ Input Vector 1 from Global Memory
|__________|
load_input
__________
|          |<------ Input Vector 1 from Global Memory
|__________|
load_input
__________
|          |<------ Input Vector 1 from Global Memory
|__________|
load_input
__________
|          |<------ Input Vector 1 from Global Memory
|__________|
load_input
__________
|          |<------ Input Vector 1 from Global Memory
|__________|
load_input
__________
|          |<------ Input Vector 1 from Global Memory
|__________|
load_input
......
```

Input Vector 2 from Global Memory
```plaintext
load_input
__________
|          |
|__________|
load_input
__________
|          |
|__________|
load_input
__________
|          |
|__________|
load_input
__________
|          |
|__________|
load_input
__________
|          |
|__________|
load_input
__________
|          |
|__________|
load_input
__________
|          |
|__________|
load_input
__________
|          |
|__________|
load_input
.....
```
"Higher-level XRT" for FPGA in 43 lines with SYCL

```cpp
#include <cassert>
#include <sycl/sycl.hpp>
#include <sycl/ext/xilinx/xrt.hpp>
#include <xrt.h>
#include <xrt/xrt_kernel.h>

constexpr int size = 4;

int main()
{
    sycl::buffer<int> a { size };
    sycl::buffer<int> b { size };
    sycl::buffer<int> c { size };

    {
        sycl::host_accessor a_a { a };
        sycl::host_accessor a_b { b };
        for (int i = 0; i < size; ++i)
        {
            a_a[i] = i;
            a_b[i] = i + 42;
        }
    }

    sycl::queue q;
    xrt::device xdev = sycl::get_native<sycl::backend::xrt>(q.get_device());
    xrt::kernel xk { xdev, xdev.load_xclbin("vadd.hw_emu.xclbin"), "vadd" };
    sycl::kernel k { sycl::make_kernel<sycl::backend::xrt>(xk, q.get_context()) };

    q.submit(async([&](sycl::handler& cgh)
    {
        cgh.set_args(sycl::accessor { a, cgh, sycl::read_only },
                     sycl::accessor { b, cgh, sycl::read_only },
                     sycl::accessor { c, cgh, sycl::write_only, sycl::no_init },
                     size);
        cgh.single_task(k);
    }));

    {
        sycl::host_accessor a_a { a };
        sycl::host_accessor a_b { b };
        sycl::host_accessor a_c { c };
        for (int i = 0; i < size; ++i)
        {
            assert(a_c[i] == a_a[i] + a_b[i]);
        }
    }
}
```

Conclusion

- **SYCL**: open-standard for high-level heterogeneous accelerator programming

- Really targets broad range of architectures, back-ends and vendors

- “Normal” SYCL is single-source C++ and targets various native APIs
  - Different backends are used behind the scene transparently
  - No need for interoperability mode for usual cases

- Explicit interoperability features of SYCL are key for openness of the standard
  - SYCL can use specific features of different back-ends at the same time
  - SYCL can be used from other heterogeneous programming frameworks
  - SYCL can be used as single unifying high-level C++ wrapper for lower-level APIs
    - Asynchronous execution, task graph, automatic data transfers & computation overlap...