Pushing the boundaries of SYCL with hipSYCL

Aksel Alpay
Heidelberg University

IWOCL ’22
About SYCL
SYCL origins

SYCL started its life as higher-level model for OpenCL!

- In SYCL 1.2.1, there’s a 1:1 mapping from SYCL objects to OpenCL objects
  - `sycl::queue` wraps OpenCL command queue
  - `sycl::device` wraps OpenCL device
  - ...

- SYCL task graph mostly handled by OpenCL out-of-order queues and dependencies

- Host compilation pass compiles kernels as fallback using pure C++
  - Generally not intended for performance.

- An additional device compiler pass extracts kernels and generates SPIR/SPIR-V

- SYCL runtime passes SPIR/SPIR-V to OpenCL.

Even though there are now other backends apart from OpenCL, most implementations have this design in their DNA.
Enter hipSYCL

- hipSYCL has always been independent from traditional SYCL interpretations
- Never had OpenCL backend…

**hipSYCL has always been about exploring other interpretations of SYCL.**
(subject of this talk)
Introduction to hipSYCL
- Multi-backend architecture
- Aggregates multiple toolchains
  - OpenMP / clang CUDA / clang HIP / clang SYCL / nvc++

https://github.com/hipSYCL/featuresupport

https://github.com/illuhad/hipSYCL

- Open source
- Used in production by large projects
- Extensions such as buffer-USM interoperability
- Supported by SYCL libraries, e.g. oneMKL
- Supports most of SYCL 2020
Multi-backend runtime

User application

Device code

libhipSYCL-rt

Modular runtime backend plugins

librt-backend-cuda
CUDA

librt-backend-omp
OpenMP

librt-backend-hip
HIP

librt-backend-ze
oneAPI Level Zero

hipSYCL runtime library
Supported compilation flows

- **Pure library flow:**
  - `syclcc --hipsycl-targets="omp" --hipsycl-cpu-cxx=...`

- **clang flow:**
  - `syclcc --hipsycl-targets="omp;cuda:<archs>;hip:<archs>;spirv"`
Supported compilation flows

syclcc --hipsycl-targets="cuda-nvcxx"
Moving beyond OpenCL
…Towards a multi-backend interpretation of SYCL
(now part of SYCL 2020)

See also this talk on SYCL 2020 backend interoperability:

Using Interoperability Mode in SYCL 2020

Speaker: Aksel Alpay (Heidelberg University)
Co-Authors: Thomas Applencourt (Argonne National Laboratory), Gordon Brown (Codeplay Software), Ronan Keryell (AMD) and Greg Lueck (Intel)
Moving beyond OpenCL

- hipSYCL has pioneered SYCL beyond OpenCL using HIP and CUDA backends already in 2018 (this idea is now in SYCL 2020).

- Focus on integration with what is best supported by hardware vendors (performance, debuggers, profilers, …)

- Make SYCL independent from direct vendor support

- It is a myth that SYCL support from HW vendors is needed for stability/performance/… !
  - Don’t confuse SYCL with OpenCL, where HW vendors generally need to provide an implementation.
  - Everybody can build a SYCL implementation that compiles to some IR, which is then optimized by HW vendor compute stacks!
hipSYCL goes one step further: Integration with vendor toolchains and programming models enables mix-and-match of programming models inside kernels!

```cpp
HIPSYCL_KERNEL_TARGET void cuda_optimized() {
  __shared__ int cuda_shared_mem[16];
  // some work here
  __syncthreads();
}

void host_optimized() {
  #ifdef _OPENMP
    #pragma omp simd
    for(int i = 0; i < 16; ++i) {
      // Some work
    }
  #else
    // Use CPU vector intrinsics, or call external libraries
  #endif
}

q.parallel_for(range, [=](auto idx){
  __hipsycl_if_target_host(host_optimized());
  __hipsycl_if_target_cuda(cuda_optimized());
  __hipsycl_if_target_hip(...);
  __hipsycl_if_target_spirv(regular_sycl_version());
});
```
Backend interoperability at source level

This enables:

- Gradual transition from e.g. CUDA code to SYCL code (can keep kernel code in CUDA during transition)
- Optimized code paths for backends, including support for language extensions
- Use optimized libraries from vendor-specific ecosystems in kernels (e.g. AMD rocPRIM, NVIDIA CUB)
- When usage is guarded by appropriate macros (e.g. `__HIPSYCL__`), code can remain portable across SYCL implementations.
Making the SYCL ecosystem robust
…by riding on top of vendor-supported compilers!
Use vendor-supported compilers

Mature support:
- HIP-clang
- Vendor-provided OpenMP compilers

Experimental:
- DPC++
- CUDA using nvc++

*hipSYCL can ride on top of vendor-supported compilers from AMD/NVIDIA/Intel*

- Day 1 hardware support
- Leverage vendor hardware expertise
- Kernel performance on par with vendor programming models
Library-only backends

**Library-only backend/implementation:** Implementing SYCL as a library for a third-party compiler (explicitly allowed in the SYCL 2020 specification)

- Important pillar to allow SYCL on vendor-supported compilers!
- Can be important for portability! (hipSYCL OpenMP backend runs on practically any CPU)
- SYCL 2020 specification: Mainly intended to run on the host; not primarily for performance
  - hipSYCL is pushing the idea of a library-only host backend for **performance**
    - OpenMP backend can deliver competitive performance for many applications!
  - hipSYCL is pushing the idea of library-only backends for accelerators.
Library-only backends for accelerators

- No reason why library-only backends should have to remain limited to the host!
- A compiler does not need a lot to be able to support SYCL
  - Pure C++ in kernels (no attributes like CUDA __device__)
  - Heterogeneous execution model reasonably similar to SYCL/OpenCL/CUDA
- hipSYCL’s library-only NVC++ CUDA backend is the first library-only device backend in a major SYCL implementation

How Much SYCL Does a Compiler Need? Experiences from the Implementation of SYCL as a Library for nvc++

Speaker: Aksel Alpay (Heidelberg University)
Co-Authors: Vincent Heuveline (Heidelberg University)
The flexibility of SYCL

What is SYCL? What does it want to be?

► A full blown compiler and toolchain in itself? (Common interpretation)
  ► More control (is it needed?)
  ► Requires more effort to develop
  ► Requires more effort and time for widespread adoption/upstreaming
► A portability library layer for third-party compilers? (similarly to e.g. Kokkos)
  ► Easy to deploy and develop
  ► Dependency on quality and features exposed by other models/compilers
► Something in between? (hipSYCL has characteristics from both)

All of those are possible (and allowed by the specification)!

hipSYCL is actively exploring this.
The SYCL 2020 specification contradicts itself!

- Explicitly allows library-only implementations
- A couple of features are not/not well implementable for library-only implementations (attributes, kernel introspection)

Most noticable:

- `parallel_for(range)` model is efficiently implementable everywhere 😊
- The `parallel_for(nd_range)` model is notoriously difficult to implement for library-only host implementations.
- The SYCL 1.2.1 hierarchical `parallel_for` model (discouraged in SYCL 2020) is notoriously difficult to implement on GPUs, and might be impossible to implement on GPUs for library-only implementations.
hipSYCL’s scoped parallelism
The case for a new programming model in SYCL
Why do we need a new model?

- We need a model that exposes the functionality of `parallel_for(nd_range)`, but works well for all implementation choices on all hardware!
- We need a model that is flexible enough to adapt to all hardware architectures
  - Different levels of parallelism on different backends/hardware (e.g. on CPU: NUMA nodes, cores, SIMD units)
  - Towards flexible group hierarchies as in CUDA cooperative groups
- Backends need to be able to expose hardware-specific hierarchies of parallelism
  ⇒ Scoped parallelism - available in hipSYCL.
hipSYCL scoped parallelism

https://github.com/illuhad/hipSYCL/blob/develop/doc/scoped-parallelism.md

```cpp
sycl::queue{}.parallel(num_work_groups, logical_group_size,
[=](auto group){
    // Note that the group argument is of generic auto type;
    // this allows the implementation to provide arbitrary group
    // types that are optimized for the backend.
    sycl::distribute_groups(group, [&](auto subgroup){
        sycl::distribute_groups(subgroup, [&](auto subsubgroup){
            sycl::distribute_groups(subsubgroup, [&](auto subsubsubgroup){
                sycl::distribute_items(subsubsubgroup, [&](sycl::s_item<1> logical_idx){
                    ...
                });
            });
        });
    });
});
```
q.submit([&](sycl::handler& cgh){
    sycl::accessor data{buff, cgh};
    cgh.parallel(input_size / Group_size, Group_size,
                 [=](auto grp){
          sycl::local_memory_environment<int[Group_size]>(grp,
                 [&](auto& scratch){
                sycl::distribute_items(grp, [&](sycl::s_item<1> idx){
                    scratch[idx.get_local_id(grp, 0)] = data[idx.get_global_id(0)];
                });
          sycl::group_barrier(grp);

          for(int i = Group_size / 2; i > 0; i /= 2){
            sycl::distribute_items_and_wait(grp,
                   [&](sycl::s_item<1> idx){
              size_t lid = idx.get_innermost_local_id(0);
              if(lid < i)
                scratch[lid] += scratch[lid+i];
            });
          }
          sycl::single_item(grp, [&](){
            data[grp.get_group_id(0)*Group_size] = scratch[0];
          }); }); }); });});
Getting performance on CPUs without OpenCL

...if you have to use `parallel_for(nd_range)`.
How do other compiler-based SYCL implementations target CPU?

Problem is offloaded to OpenCL: OpenCL gets SPMD-style IR, and then performs required compiler transformations.

- Requires OpenCL CPU implementation which may be a portability issue
- More difficult to deploy due to OpenCL dependency
- Locks into using OpenCL runtime. What if we want to use TBB, or OpenMP, ...?
New accelerated CPU support in hipSYCL

Idea: Pull compiler transformations directly into the SYCL compiler

- Leverage existing hipSYCL LLVM clang plugin and add IR transformations during the host pass
- No dependency on OpenCL – works wherever LLVM works
- Just looks for specific attributes that mark functions that need to be considered as kernel entrypoints. Can be used with any C++ CPU runtime (TBB, OpenMP, ...)
- Retain many advantages of library-only implementations
See the poster for details!

**Exploring Compiler-aided nd-range Parallel-for Implementations on CPU in hipSYCL**

View Abstract

Speaker: Joachim Meyer (Saarland University)
Co-Authors: Aksel Alpay, Holger Fröning and Vincent Heuveline (Heidelberg University)
hipSYCL compiler-
accelerated CPU performance

▶ Tested on AMD, Intel, ARM (ThunderX2, A64fx)
▶ Competitive performance compared to OpenCL (pocl)
▶ Comes with any hipSYCL 0.9.2+
▶ New in SYCL ecosystem: Run SYCL kernels efficiently on any CPU supported by LLVM!
A modern SYCL runtime
Stepping back from the traditional OpenCL-style mappings
Queue design

- Traditionally, one SYCL queue is mapped to one backend queue
- hipSYCL decouples SYCL queues from backend objects!
- Backends maintain queue pool (if queue-based)
- Scheduler distributes work from all queues across backend resources
Consequences of queue decoupling

- Performance and concurrency of operations is independent of the number of user queues → consistent performance
  - Why should the user have to worry about the number of queues they construct in a high-level model like SYCL?
- Scheduler can make stronger assumptions about execution behavior (number of backend queues can be tied to hardware capabilities)

See hipSYCL extracting concurrency in action!
What is a queue?

- In hipSYCL, a queue is a light-weight object that does not represent actual backend execution resources.
- ...instead, is a mechanism to append work to the global SYCL task graph, and synchronize groups of tasks using queue::wait().
- Better name might be task_collection...
- This has substantial consequences!
A queue does not have to be tied to a device!

```cpp
sycl::queue q{some_device, sycl::property::queue::in_order{}};
q.parallel_for(/* runs on some_device */);
q.submit({sycl::property::command_group::hipSYCL_retarget{other_device}},
    [&](sycl::handler& cgh){
        cgh.parallel_for(/* runs on other_device*/);
    });
q.wait();
```

- Convenient if most operations on a queue should go to a specific device, with some exceptions.
- Single `queue::wait()` can synchronize operations distributed across multiple devices
- In-order queue can enforce in-order behavior across multiple devices
Multi-device queues

Have hipSYCL distribute a task graph automatically across the system!

```cpp
1 // User can also specify the list of devices to schedule to.
2 sycl::queue q{sycl::system_selector_v};
3 //kernels may be executed on different devices
4 q.parallel_for(...);
5 q.parallel_for(...);
6 q.parallel_for(...);
```

- Works, but don’t expect good performance yet from the scheduling 😊
- Generalization of extracting concurrency from a single device
- Remark: Reinterpreting a `sycl::queue` as a task collection also makes it apparent that SYCL graphs can be implemented with minimal additions to the queue interface.
Context

- `sycl::context` is similarly decoupled from backend contexts
- Prevents performance bugs (`sycl::queue()` constructing new context)
- Unclear what a `sycl::context` should be...
Subbuffers are an unnecessary OpenCL concept

- Needed to allow the runtime to execute kernels concurrently that use the same data
  - Disjoint accessor ranges is not enough per the specification
- …but it is in hipSYCL!
- hipSYCL tracks buffer data state below buffer granularity
- Fundamental difference in how buffer support in the runtime is designed
```cpp
sycl::buffer<int, 2> buff{sycl::range{size, size},
  sycl::property::buffer::hipSYCL_page_size<2>{{
  sycl::range{page_size, page_size}}};

// hipSYCL runtime will attempt to execute concurrently
q.submit([&](sycl::handler& cgh){
  sycl::accessor<int, 2> acc{buff, cgh,
    range{page_size, page_size}, id{0,0}};
  cgh.parallel_for(...);
});
q.submit([&](sycl::handler& cgh){
  sycl::accessor<int, 2> acc{buff, cgh,
    range{page_size, 10*page_size}, id{page_size,0}};
  cgh.parallel_for(...);
});

▶ Kernels may run concurrently if their accessors access different pages
```
Conclusion

- It is important to rethink SYCL independently of its history as OpenCL abstraction layer!
- From its inception, hipSYCL has been exploring new ways of designing SYCL implementations
- …non-OpenCL backends
- …Riding on top of vendor-supported compilers
- …device library-only backends, and the idea of aggregating multiple toolchains
- …New programming models like scoped parallelism
- …CPU acceleration of kernels without OpenCL
- …Decoupling backend objects from SYCL objects (like queue) leading to multi-device queues

And there is more!
More features

```cpp
int* input = ...; sycl::queue q;
// Asynchronous buffers & factory functions
auto b = sycl::make_async_view(input, size, q);
auto c = sycl::make_sync_buffer(size);
q.submit([&](sycl::handler& h){
    sycl::raw_accessor r{b, cgh}; // Light-weight accessors
cgh.parallel_for(...);
}); q.wait();
// Buffer-USM interop
void* data = b.get_pointer(q.get_device());
```

- Plus many standard SYCL 2020 features
- Wide-range of supported hardware
- Support for oneAPI components like oneMKL

Exploring the Possibility of a hipSYCL-based Implementation of oneAPI

Speaker: Aksel Alpay (Heidelberg University)
Co-Authors: Bálint Soproni, Holger Wünsche and Vincent Heuveline (Heidelberg University)
...and more to come!

- Single compilation pass for host and all targeted devices
- Integrated profiling functionality for SYCL task graphs
- ...

All features are available on github!
https://github.com/illuhad/hipSYCL