hipSYCL in 2021
Peculiarities, Unique Features and SYCL 2020

Aksel Alpay
Heidelberg University
Introduction to hipSYCL

hipSYCL
A generic, multi-backend SYCL implementation with emphasis on aggregating existing toolchains.

- Source compatible with vendor-specific programming models
- Unique extensions, e.g. full buffer-USM interoperability
hipSYCL: Multiple toolchains in one

```
> syclcc -03 --hipsycl-targets="omp;cuda:sm_70;hip:gfx906" test.cpp
> CMake integration available: find_package(hipSYCL), add_sycl_to_target()
```
hipSYCL runtime architecture

- 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)
## SYCL 2020 in hipSYCL

<table>
<thead>
<tr>
<th>Feature</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accessor simplifications</td>
<td>✔</td>
</tr>
<tr>
<td>USM: Memory management functions</td>
<td>✔</td>
</tr>
<tr>
<td>USM: Queue shortcuts</td>
<td>✔</td>
</tr>
<tr>
<td>USM: Prefetch</td>
<td>✔</td>
</tr>
<tr>
<td>USM: mem_advise</td>
<td>✗</td>
</tr>
<tr>
<td>USM: memcpy</td>
<td>✔</td>
</tr>
<tr>
<td>USM: memset/align</td>
<td>✔</td>
</tr>
<tr>
<td>host tasks</td>
<td>✗</td>
</tr>
<tr>
<td>Optional lambda naming</td>
<td>✔</td>
</tr>
<tr>
<td>Subgroups</td>
<td>✔</td>
</tr>
<tr>
<td>In-order queues</td>
<td>✔</td>
</tr>
<tr>
<td>Explicit dependencies</td>
<td>✔</td>
</tr>
<tr>
<td>Backend Interop API</td>
<td>✔</td>
</tr>
<tr>
<td>Reductions</td>
<td>✔</td>
</tr>
<tr>
<td>Group algorithms</td>
<td>✔</td>
</tr>
<tr>
<td>New device selector API</td>
<td>✗</td>
</tr>
<tr>
<td>Aspect API</td>
<td>✗</td>
</tr>
<tr>
<td>Deduction guides</td>
<td>✔</td>
</tr>
<tr>
<td>atomic_ref</td>
<td>✗</td>
</tr>
<tr>
<td>array</td>
<td>✗</td>
</tr>
<tr>
<td>New: SYCL/sycl_i.hpp header</td>
<td>✔</td>
</tr>
<tr>
<td>C++17 by default</td>
<td>✔</td>
</tr>
</tbody>
</table>

### Buffers Changes
- ctz(), clz()
- Remove `*` class types
- const return type for read accessor `operator[]`
- Remove buffer API for `unique_ptr`
- Replace `program class with module` with
- Add `kernel_handler`
- Explicit queue, context constructors
- Only require C++ trivially copyable for shared data
- Update group class with new types/member functions
- Remove `nd_item::barrier()`
- Replace `num_Fence with atomic_fence`
- Add `vec::operator[]` and `::operator[]`
- `static constexpr get_size() / get_count()`
- Buffer, local accessor are C++ contiguous container
- Replace `image with sampled_image`.
- `unsampled_image`
- All accessors are placeholders
- Use single exception type derived from `std::exception`
- Default asynchronous handler should terminate program

### Kernel Invocation APIs
- `take const reference to kernels, kernels must be immutable`
- `Queue constructor accepting both device and context`
- Simplified `parallel_for` API
- `Clarified names for device specific info queues`
- `Address space changes, generic address spaces`
- Updated `multi_ptr` interface
- Remove OpenGL types, `cl_int`, etc

[https://github.com/hipSYCL/featuresupport](https://github.com/hipSYCL/featuresupport)
A `sycl::queue` is not a queue

A `sycl::queue` in hipSYCL is

- an interface to SYCL task graph
- an interface to a collection of task graph nodes for synchronization
- decoupled from backend objects
Automatic distribution of work across backend queues

Works independently of how many `sycl::queue` objects submit work!

- `blocked_transform` benchmark from SYCL-Bench\(^1\)

```cpp
1  data = array(size, input_values)
2  for each block B of fixed block size in data:
3      parallel_for operating on each element i in B:
4          data[i] = work(data[i]);
5  wait for all blocks to finish
6  Copy data back to host
```

- USM version: Use explicit USM `memcpy` and `parallel_for` for each block
- Buffer version: Use buffers and ranged accessors

---

blocked_transform: USM version

1 block/single kernel: 0.62 seconds
128 blocks: 0.44 seconds.\(^2\) \(\approx 50\%\) more perf, more if we exclude final copy back to host

hipSYCL tried to even overlap kernels, but hardware did not accept the offer.

\(^2\)134217728 input elements, 8 work iterations inside kernels, GeForce GTX 1080
blocked_transform: Buffer version

- 1 block/single kernel: 0.63 seconds
- 128 blocks: 0.62 seconds

What happened?

- SYCL memory model: Data dependencies are tracked globally per entire buffer. Accessor access ranges don’t matter.
- Kernels for individual blocks are not independent operations!
- Need to create subbuffers - cumbersome, requires contiguous memory (not well-suited for 2D/3D buffers)
hipSYCL buffer model

hipSYCL specifies its own buffer model\(^3\) that extends the SYCL model:

- Buffers are divided into 3D chunks of data – “pages” (inspired by, but not related to OS pages)
- Data state and dependencies are tracked on page granularity
- Kernels operating on different pages of the same buffer are independent
- `hipSYCL_page_size` buffer property can be used to set page size (default is one page for entire buffer)

Works in 1D/2D/3D and more intuitive than subbuffers.

\(^3\)https://github.com/illuhad/hipSYCL/blob/develop/doc/runtime-spec.md
// Construct a buffer consisting of four pages in total
sycl::buffer<int, 2> buff{sycl::range{512, 512},
  sycl::property::buffer::hipSYCL_page_size<2>{
    sycl::range{256, 256}};

q.submit([&](sycl::handler &cgh) {
  // accesses page (0, 0) and (0, 1)
  sycl::range range{256, 512};
  sycl::id offset{0, 0};

  sycl::accessor<int, 2> acc{buff, cgh, range, offset};
  cgh.parallel_for(...);
});

q.submit([&](sycl::handler &cgh) {
  // accesses page (1, 0) and (1, 1)
  sycl::range range{256, 512};
  sycl::id offset{256, 0};

  sycl::accessor<int, 2> acc{buff, cgh, range, offset};
  cgh.parallel_for(...);
});
**blocked_transform: Buffers + pages**

Use 128 pages, such that each kernel operates on a different buffer page.

- 1 block/single kernel: 0.62 seconds
- 128 blocks: 0.47 seconds
- Recovers behavior of USM version
Summary: Backend queue scheduling

Lessons learned

- Try splitting up work into smaller chunks instead of one large kernel
- Particularly effective when data transfer time $\approx$ kernel time
- If the hipSYCL scheduling is not optimal, use `hipSYCL_prefer_execution_lane(id)`
- Don’t make chunks too small (latencies, scheduling overheads)
- For buffers: use `hipSYCL_page_size` to make chunk accesses independent
- Expose parallelism to hipSYCL!

---

https://github.com/illuhad/hipSYCL/blob/develop/doc/extensions.md#hipsycl_ext_cg_property_prefer_execution_lane
Transparencyly distribute work across multiple devices with a single queue

```cpp
void my_async_sycl_library(sycl::queue& q){
    std::vector<sycl::device> devs = ...;
    q.submit(sycl::property::command_group::hipSYCL_retarget{devs[0]},
             [&] (sycl::handler& cgh){
                cgh.parallel_for<class Kernel1>(...);
             });
    q.submit(sycl::property::command_group::hipSYCL_retarget{devs[1]},
             [&] (sycl::handler& cgh){
                cgh.parallel_for<class Kernel2>(...);
             });
}
...

sycl::queue q;
my_async_sycl_library(q)
q.wait();
```
Efficient backend interoperability

Meaningful backend interoperability usually requires access to `sycl::buffer` memory. This is possible with SYCL 2020 host tasks:

```cpp
q.submit([&](sycl::handler &cgh) {
    auto acc = buff.get_access<sycl::access::mode::read>(cgh);
    cgh.host_task([&](sycl::interop_handle &h) {
        void *native_mem = h.get_native_mem<sycl::backend::cuda>(acc);
        auto stream = h.get_native_queue<sycl::backend::cuda>();
        execute_native_work(native_mem, stream);
    });
});
```

- Executed as part of the SYCL task graph
- Very flexible, but if the user only wishes to enqueue additional backend work (common!), performance suffers from delayed work submission at task graph execution time
Efficient backend interoperability

`hipSYCL_enqueue_custom_operation` as optimized solution for *enqueueing* backend work

- Executed at task graph submission time
- **Only** when additional work should be enqueued

```cpp
q.submit([&](sycl::handler &cgh) {
    auto acc = buff.get_access<sycl::access::mode::read>(cgh);
    cgh.hipSYCL_enqueue_custom_operation([=](sycl::interop_handle &h) {
        void *native_mem = h.get_native_mem<sycl::backend::cuda>(acc);
        auto stream = h.get_native_queue<sycl::backend::cuda>();
        execute_native_work(native_mem, stream);
    });
});
```

**Case study:** Submitting 256 native CUDA saxpy kernels with backend interoperability mechanisms (simulates native CUDA library)
SYCL 2020 host tasks (prototype)

- Host task: Submit
- Native operation

hipSYCL custom operations extension

- Custom Op submit
- Native operation

CPU (12)
- Processes (4)
- CUDA HW (GeForce GTX 1080)
  - 97.4% Context 1
  - [All Streams]
hipSYCL custom operations

Lessons learned

- Prefer hipSYCL custom operations over host tasks to enqueue additional work (cuBLAS, native kernels, …)
- Especially true for short-running work!

Efficient backend interoperability II

**hipSYCL buffer-USM interoperability**

```cpp
1  using namespace sycl::
2       buffer_allocation;
3  sycl::buffer<T> buff{
4      view(usm_ptr1, device1),
5      empty_view(usm_ptr2, device2, take_ownership),
6      sycl::range{size}};
```

- Make sure to understand the hipSYCL buffer model
- USM pointers have lower overhead compared to buffer-accessor model. Useful when bound by SYCL runtime performance.

Performance portable execution models

$\text{nd\_range parallel for is not performance portable if barriers are used.}$

- Need independent forward progress guarantees for each work item
  - OpenMP backend: Need to run each work item in its own fiber
  - No vectorization across work items
  - Context switch overhead

This is a fundamental issue of nd\_range parallel for – all library-only implementations are affected!
hipSYCL scoped parallelism

New execution model: Scoped parallelism

```cpp
q.parallel(num_groups, group_size,
           [=](sycl::group<1> grp, sycl::physical_item<1> p){
             // Implementation/backend defined number of threads
             grp.distribute_for( [&](sycl::sub_group sg, sycl::logical_item<1> idx){
                 // Executed for each user-requested work item
                 const int id = idx.get_global_id(0);
                 c[id] = a[id] + b[id];
             }); // Implicit barrier
           });
```

- API might change slightly to better fit SYCL 2020 patterns
- Can express everything that `nd_range` parallel for can
- `distribute_for` can be mapped to vectorized loop on CPU

---

Scoped parallelism

Lessons learned

- Avoid `nd_range` parallel for if possible
- For barriers and local memory, prefer hipSYCL scoped parallelism

Scoped parallelism is...

- easier to implement and better performance guarantees than SYCL 1.2.1 hierarchical parallelism
- Can also be used to allow work group sizes not natively supported by hardware
- provides similar guarantees as nested parallelism in other library-based solutions, e.g. Kokkos
- Can also be implemented on top of existing `nd_range` or hierarchical parallel for
Conclusion

- hipSYCL is a highly **flexible multi-backend SYCL implementation** that aggregates multiple toolchains (clang CUDA, clang HIP, clang SYCL, OpenMP) into one
- Rapidly moving towards **SYCL 2020**
- hipSYCL will **automatically distribute work across backend queues**
- Unique extensions:
  - hipSYCL pages (allow multiple kernels operate simultaneously on one buffer)
  - `hipSYCL_enqueue_custom_operation` (efficiently enqueue backend interop tasks)
  - Asynchronous buffers/Explicit buffer behaviors
  - Scoped parallelism (performance portability)
  - buffer-USM interoperability
Conclusion

▶ All mentioned features are publicly available:
  https://github.com/illuhad/hipSYCL
▶ Get in touch: aksel.alpay@uni-heidelberg.de
Buffer destruction antipattern

Buffer destruction blocks, which can introduce unnecessary/undesired synchronization.

```cpp
{
    sycl::buffer<T> b1{ptr1, size};
    sycl::buffer<T> b2{ptr2, size};
    sycl::buffer<T> b3{ptr3, size};

    // Kernels

} // Destructors issue write-back

1. b3.buffer() → submit writeback → wait
2. b2.buffer() → submit writeback → wait
3. b1.buffer() → submit writeback → wait
```
hipSYCL explicit buffer behaviors

Enforce conscious decision from user whether ...

<table>
<thead>
<tr>
<th>Destructor blocks?</th>
<th>Writes back?</th>
<th>Operates on input pointer?</th>
</tr>
</thead>
<tbody>
<tr>
<td>yes</td>
<td>sync_writeback_view</td>
<td>yes</td>
</tr>
<tr>
<td>no</td>
<td>async_buffer</td>
<td>no</td>
</tr>
</tbody>
</table>

Clarify intent, reduce errors, improve performance

```cpp
sycl::queue q;
{
    auto b1 = sycl::make_async_writeback_view(ptr1, size, q);
    auto b2 = sycl::make_async_writeback_view(ptr2, size, q);
    auto b3 = sycl::make_async_writeback_view(ptr3, size, q);
    // Kernels
} // Non-blocking buffer destructors
q.wait(); // Single wait for all writebacks
```

https://github.com/illuhad/hipSYCL/blob/develop/doc/explicit-buffer-policies.md