SYCL beyond OpenCL:
The architecture, current state and future direction of hipSYCL

IWOCL/SYCLcon 2020

Speaker: Aksel Alpay
(Heidelberg University)

April 10, 2020
Figure: SYCL implementations. (Figure is part of the hipSYCL project: https://github.com/illuhad/hipSYCL)
About the project

▶ Open source SYCL implementation led by Heidelberg University
▶ https://github.com/illuhad/hipSYCL
▶ Started in summer 2018 as hobby project, since 2019 funded by Heidelberg University.
▶ Used in the real-world on machines of all scales
▶ Thanks to the community – users and contributors!
▶ https://github.com/illuhad/hipSYCL/graphs/contributors
Motivation

- For maximum performance
  - Need vendor-optimized libraries, including template-libraries (e.g. rocPRIM, CUB)
  - Need access to latest hardware features (e.g. via intrinsics)
- For performance tuning and debugging, need access to vendor tools
- Not all hardware vendors have adopted SYCL yet ⇒ Need to make SYCL resilient against vendor adoption friction

⇒ Implement SYCL directly on top of programming models best supported by vendors

In hipSYCL:
- Build directly on top of HIP instead of OpenCL when targeting GPU
- Additional CPU backend using OpenMP available

AMD HIP:
- (almost) zero-cost abstraction layer for NVIDIA CUDA and AMD ROCm GPUs
- CUDA programming model, slightly renamed runtime functions (e.g. `hipMalloc` instead of `cudaMalloc`)
the ecosystem for SYCL on CUDA

Several solutions are available for “SYCL+CUDA”, with different characteristics:

- **CUDA Hardware Interop.**
  - SYCL compiler frontend
  - built on OpenCL API
  - PTX IR
  - more OpenCL interoperability

- **CUDA Platform Interop.**
  - SYCL compiler frontend
  - built on CUDA API
  - PTX IR

- **CUDA source code Interop.**
  - compiler understands CUDA and SYCL
  - built on CUDA API
  - PTX IR
  - more CUDA interoperability

⇒ SYCL is flexible, implementations cover the entire continuum of OpenCL/CUDA interoperability!
SYCL implementation components

- Compiler
  - kernel outlining, SYCL-specific semantics, ...

- SYCL Interface
  - Classes functions in `sycl::` namespace

- SYCL Runtime
  - Data management, Scheduling, interaction with backend, ...

- Glue
Overview:

- Use clang CUDA/HIP frontend, augmented with clang plugin to add support for SYCL constructs
- resulting compiler can compile **both** SYCL and HIP/CUDA code (in the same source file)
- allows for direct interoperability between CUDA and SYCL code in same source file
- Kernel performance generally on the level of clang-compiled CUDA/HIP **by design**
- Let existing compiler toolchain do heavy lifting
  - We are ambitious, not crazy!

When targeting CPU:

- No special compiler needed, SYCL is pure C++
- implementation of SYCL as a pure library with OpenMP parallelization
At the moment only AST transformations:

- Main obstacle to get a HIP/CUDA compiler to accept SYCL code:
  - Make sure all functions are CodeGen’d for host and device when required by kernels
  - ⇒ identify kernels
  - ⇒ determine kernel call graph
  - Important sidenote: hipSYCL initially parses all kernels as host code!

- Implement correct semantics for parallel hierarchical for (variables in work group scope must be allocated in local memory)

- **Future**: SYCL-specific diagnostics
Compiler: syclcc

- Compiler wrapper written in python (also known as syclcc-clang)
- hides all the nasty details – invoking compiler correctly, linking against hipSYCL, include paths, ...

```
1  syclcc --hipsycl-platform=cuda --hipsycl-gpu-arch=sm_60 -o test -O2 test.cpp
2  syclcc --hipsycl-platform=rocm --hipsycl-gpu-arch=gfx906 -o test -O2 test.cpp
3  syclcc --hipsycl-platform=cpu --hipsycl-cpu-cxx=clang++ -o test -O2 test.cpp
```

also possible with environment variables:

```
1  HIPSYCL_PLATFORM=cuda HIPSYCL_GPU_ARCH=sm_60 syclcc -o test -O2 test.cpp
2  HIPSYCL_PLATFORM=rocm HIPSYCL_GPU_ARCH=gfx906 syclcc -o test -O2 test.cpp
3  HIPSYCL_PLATFORM=cpu HIPSYCL_CPU_CXX=clang++ syclcc -o test -O2 test.cpp
```

- If no arguments are given, uses default settings from json configuration file
SYCL runtime

- Implement SYCL operations on top of HIP API
- SYCL needs dynamic out-of-order processing of tasks
- ⇒ currently implemented with callbacks and HIP events after operations

SYCL interface

- Implements SYCL classes and functions using functionality of HIP and hipSYCL runtime
- Invoking kernels requires a bit of trickery
Invoking kernels

General idea:

```cpp
template<class Kernel, int Dim>
__global__ void dispatch(Kernel k) {
    sycl::item<Dim> idx = ...; k(idx);
}

void parallel_for(Kernel k) {
    dispatch<<</*grid size, group size*/>>>(k);
}
```

But: All code is initially parsed as host code! k is host lambda, cannot be invoked!

```cpp
// __sycl_kernel attribute will be replaced by __global__ once initial
// parsing is complete
template<class Kernel, int Dim>
__sycl_kernel void dispatch(Kernel k) {
    sycl::item<Dim> idx = ...; k(idx);
}

template<class Kernel, int Dim>
void parallel_for(Kernel k) {
    // Or __cudaPushCallConfiguration(...) 
    hipConfigureCall(grid_size, group_size); dispatch(k);
}
```
Memory Performance

**Figure:** hipSYCL BabelStream results [Deakin T, Price J, Martineau M, McIntosh-Smith S (2016)]
Jehferson Mello’s toyBrot Mandelbrot renderer for many programming models: https://gitlab.com/VileLasagna/toyBrot
HIP/CUDA interoperability at source level

```c
#ifdef __HIPSYCL_PLATFORM_XYZ // XYZ == CUDA, ROCM
__host__ __device__
void optimized_function()
{
    #if defined(SYCL_DEVICE_ONLY)
    // Can call arbitrary CUDA/ROCm device code
    // e.g. intrinsics, vendor-optimized libraries, ...
    #endif
}
#endif
```

```c
...```

```c
q.submit([&](sycl::handler& cgh){
    cgh.parallel_for<class kernel_name>(sycl::range<1>{...},
        [=](sycl::id<1> idx){
            #ifdef __HIPSYCL_PLATFORM_XYZ
            optimized_function();
            #else
            regular_sycl_function();
            #endif
        });
});
```
Group reduction with rocPRIM

```cpp
#ifdef HIPSYCL_PLATFORM_ROCM
template<class T, int group_size>
__host__ __device__
T rocprim_local_reduction(T value)
{
    T output{};
    #ifdef SYCL_DEVICE_ONLY
    rocprim::block_reduce<T,group_size>{}
        .reduce(value, output);
    #endif
    return output;
} #endif

... cgh.parallel_for<KernelName<T>>(..., 
    [=](sycl::nd_item<1> item) {
        const int lid = item.get_local_id(0);
        const auto gid = item.get_global_id();
        #ifdef HIPSYCL_PLATFORM_ROCM
        T result =
            rocprim_local_reduction<T, GROUP_SIZE>(
                acc[ gid]);
        if(lid == 0) acc[ gid] = result;
        #else
        scratch[ lid] = acc [ gid]; // use local mem
        for(int i = GROUP_SIZE/2; i > 0; i /= 2){
            item.barrier();
            if(lid < i)
                scratch[ lid] += scratch[ lid + i];
        }
        if(lid == 0) acc[ gid] = scratch[0];
        #endif
    });
```
Group reduction with rocPRIM

Figure: Results on Radeon VII with group size of 256
For more hipSYCL benchmarking, please see the talk *SYCL-Bench: A Versatile Single-Source Benchmark Suite for Heterogeneous Computing*
Current limitations

- hipSYCL is pre-conformance.
  - largest missing features: images and (by design) OpenCL interoperability
  - Already usable for real-world applications
- Static compilation model requires decision which hardware to target at compile-time
  - Submitting work to both CPU and GPU is not yet possible – requires abstraction layer between hipSYCL and HIP
- ndrange parallel for on the CPU backend is very slow
CPU backend ndrange kernel performance

- ndrange parallel for *cannot* be implemented efficiently in pure-library SYCL implementations (not hipSYCL specific issue)
- ndrange parallel for allows explicit barriers
- → we need to launch as many threads as work items → inefficient
- Efficient pattern: Multithreading across groups + loop over work items
- Requires compiler support:

  ```
  for each work group in parallel
    for each work item
      A(); item.barrier(); B()
  ```

must be transformed into

  ```
  for each work group in parallel
    for each work item
      A()
      for each work item
        B()
  ```
Main focus of current development effort: New runtime component, rewritten from scratch

- Allow for arbitrary backends (all active simultaneously) not just HIP, with different execution models (task graph, in order queues, ...)
  - Remove restrictions of static compilation model CPU/GPU
  - Allow for additional backends, e.g. TBB, OpenAcc, OpenMP 5, CUDA graphs, ...
- Transition from 1:1 mapping of SYCL queue to backend queue to N:M mapping → better hardware utilization
- Strict separation between runtime and SYCL interface
- SYCL interface attaches hints/instructions to operations for the runtime
  - Pave way for novel scheduling mechanisms - automatic distribution of work across devices, execution priority, in-order low-latency mode, ...
- Allow for modifying scheduling behavior at sub-queue granularity
- Transition to batched submission model for higher task throughput
New submission mechanism
Getting hipSYCL

- deb, rpm, Arch package repositories at http://repo. urz.uni-heidelberg.de/sycl. We provide packages both for hipSYCL and matching LLVM and ROCm distributions.
- Community provided packages: spack (only CPU, CUDA backends), ongoing packaging efforts on Gentoo, NixOS, Arch Linux AUR
- Source code: https://github.com/illuhad/hipSYCL
- hipSYCL is an implementation of SYCL for CPUs, NVIDIA GPUs, AMD GPUs
- built directly on top of low-level vendor APIs HIP/CUDA for full interoperability at source code level
- targeted at HPC and use cases that require access to latest low-level hardware optimizations or vendor-optimized libraries
- Kernel performance can be expected to be on par with regular HIP/CUDA
- Open source project – always looking for contributors 😊
- https://github.com/illuhad/hipSYCL
- Get in touch with me: aksel.alpay@uni-heidelberg.de