Leveraging MLIR for Better SYCL Compilation

Víctor Pérez-Carrasco*, Ettore Tiotto†, Whitney Tsang†, Lukas Sommer*,
Victor Lomüller*, James Brodman†, Mehdi Goli*

Codeplay Software Ltd.* Intel Corporation†

MLIR dialect to represent SYCL API / semantic → Host-device code representation in MLIR → Enable high-level SYCL specific optimizations

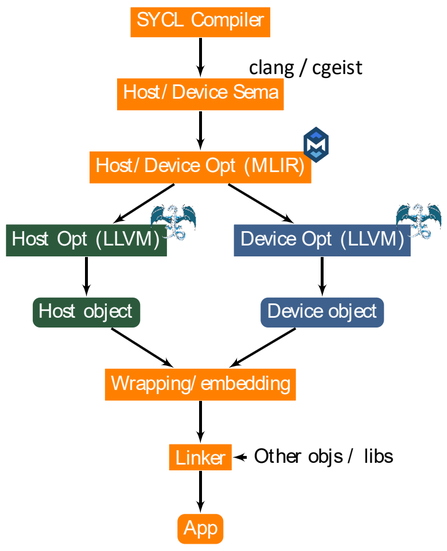
The SYCL Platform

- Single source, high-level, standard C++ programming model targeting a range of heterogeneous platforms
- Different implementations:
  - DPC++ (Intel)
  - ComputeCpp (Codeplay)
  - hipSYCL (Heidelberg University)
  - …

An MLIR-Based SYCL Compiler

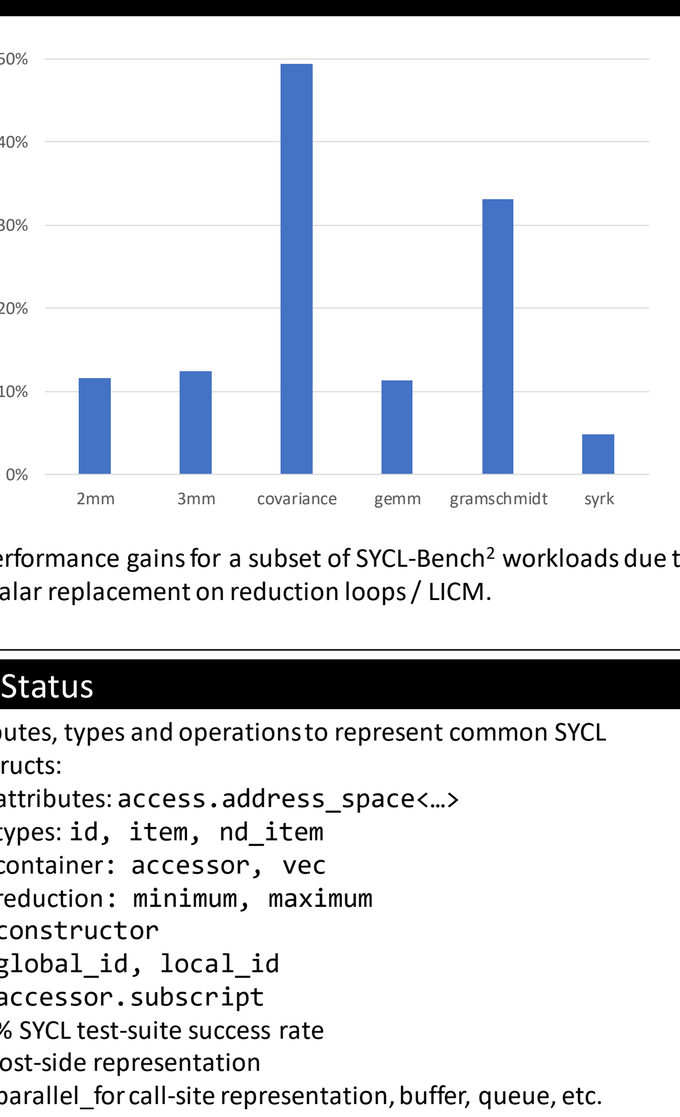
- Extensible framework to build compilers
- C/C++ lacks a proper MLIR frontend
  - Using polygeist (incomplete) for device code
  - Exploring raising from LLVM IR for host code
- Allows having code for different targets in a single module thanks to its nested structure
- Use higher-level abstractions for higher-level optimizations

Goal of Project: 1-pass MLIR-Based Compiler



- Keep host and device code as close as possible
  - Device code in GPU module
- Embed the SYCL API semantic into MLIR
- Optimize SYCL specific patterns
- Split into different modules when lowering to LLVM

Dialects: Capturing Higher-Level Semantics



SYCL to MLIR Translation

Reduction Loop in SYCL

```
#include <sycl/sycl.hpp>

void init() {
  std::vector <long> a{...};
  sycl::queue q;
  {
    sycl::buffer<long> bufA{a};
    q.submit([&](sycl::handler& cgh) {
      sycl::accessor accA{bufA, cgh, sycl::write_only,
                                     sycl::no_init};
      cgh.parallel_for(a.size(),
        [=](sycl::id<1> i) {
          for (size_t k=0; k<M; ++k)
            accA[i] += <expr(k)>;
        });
    });}}
```

```
gpu.func @caller(...) kernel {
  func.call @callee(%arg0)
    : (memref<?x!llvm.struct<(!sycl_accessor_1_f32_w_gb, !sycl_accessor_1_f32_r_gb)>>, index) -> ()
  gpu.return
}
func.func private @callee(%arg0: memref<?x!llvm.struct<(!sycl_accessor_1_f32_w_gb,
                                                        !sycl_accessor_1_f32_r_gb)>>) {
  %id_val = sycl.global_id : !sycl_id_1_
  memref.store %id_val, %id[%c0] : memref<?x!sycl_id_1_>
  affine.for %arg2 = 0 to %k {
      %A = polygeist.subindex(%arg0, %c0)
        : (memref<?x!llvm.struct<(!sycl_accessor_1_f32_w_gb, !sycl_accessor_1_f32_r_gb)>>, index)
        -> memref<?x!sycl_accessor_1_f32_w_gb>
      %A_item = sycl.accessor.subscript %A[%id]
        : (memref<?x!sycl_accessor_1_f32_w_gb>, memref<?x!sycl_id_1_>) -> memref<?xf32>
      %0 = affine.load %A_item[0] : memref<?xf32>
      %1 = arith.addf %0, <expr(k)> : f32
      affine.store %1, %A_item[0] : memref<?xf32>
  }
  return
}
```

SYCL-Bench Performance Gains[1]

Performance gains for a subset of SYCL-Bench[2] workloads due to scalar replacement on reduction loops / LICM.

Scalar Replacement on Reduction Loops

```
for(size_t k=0; k<M; ++k)
  R[i] += <expr(k)>;
```

→

```
DATA_TYPE R_reduction = R[i];
for(size_t k=0; k<M; ++k)
  R_reduction += <expr(k)>;
R[i] = R_reduction;
```

Calls to SYCL runtime functions obfuscate program semantics when the code is lowered to LLVM IR (making difficult to recognize the opportunity). In MLIR memory semantics of the dialect operations can be easily modeled.

Argument Promotion

```
gpu.func @caller(...) kernel {
  func.call @callee(%a, %b)
  gpu.return
}
func.func private @callee(%A: memref<?x!sycl_accessor_1_f32_w_gb> {llvm.noalias},
                          %B: memref<?x!sycl_accessor_1_f32_w_gb> {llvm.noalias}) {
  ...
  affine.for %arg2 = 0 to %k {
    %A_item = sycl.accessor.subscript %A[%id] -> memref<?xf32>
    %0 = affine.load %A_item[0]
    %1 = arith.addf %0, <expr(k)>
    affine.store %1, %A_item[0]
  }
  return
}
```

Kernel Specialization

```
gpu.func @caller(...) kernel {
  scf.if &a[a.get_range()] <= &b[sycl::id{}] || &a[sycl::id{}] >= &b[b.get_range()]
    func.call @callee.specialized(%0, %1)
  else
    func.call @callee(%0, %1)
  gpu.return
}
func.func private @callee.specialized(
    %A: memref<?x!sycl_accessor_1_f32_w_gb> {llvm.noalias, sycl.inner.disjoint},
    %B: memref<?x!sycl_accessor_1_f32_w_gb> {llvm.noalias, sycl.inner.disjoint}) {
  ...
}
```

Array Reduction

```
func.func private @callee.specialized(
    %A: memref<?x!sycl_accessor_1_f32_w_gb> {llvm.noalias, sycl.inner.disjoint},
    %B: memref<?x!sycl_accessor_1_f32_w_gb> {llvm.noalias, sycl.inner.disjoint}) {
  ...
  affine.if %k > 0 {
    %A_item = sycl.accessor.subscript %A[%id] -> memref<?xf32>
    %0 = affine.load %A_item[0]
    %r = affine.for %arg2 = 0 to %k iter_args(%arg = %0) -> (f32) {
      %1 = arith.addf %arg, <expr(k)>
      affine.yield %1
    }
    affine.store %r, %A_item[0]
  }
  return
}
```

LICM

```
func.func private @callee.specialized(
    %A: memref<?x!sycl_accessor_1_f32_w_gb> {llvm.noalias, sycl.inner.disjoint},
    %B: memref<?x!sycl_accessor_1_f32_w_gb> {llvm.noalias, sycl.inner.disjoint}) {
  ...
  affine.if %k > 0 { // Loop Guard
    %A_item = sycl.accessor.subscript %A[%id] -> memref<?xf32>
    affine.for %arg2 = 0 to %k {
      %0 = affine.load %A_item[0]
      %1 = arith.addf %0, <expr(k)>
      affine.store %1, %A_item[0]
    }
  }
  return
}
```

Project Status

- Attributes, types and operations to represent common SYCL constructs:
  - attributes: `access.address_space<…>`
  - types: `id, item, nd_item`
  - container: `accessor, vec`
  - reduction: `minimum, maximum`
  - `constructor`
  - `global_id, local_id`
  - `accessor.subscript`
- > 60 % SYCL test-suite success rate
- No host-side representation
  - parallel_for call-site representation, buffer, queue, etc.
- Host-side lowering is an open question:
  - In lack of a capable C++ frontend, current approach involves raising from LLVM IR

Next Steps

- Work on host-side code generation
  - Proper C++ compiler? Looking into some projects:
    - ClangIR/CIR
    - polygeist (currently used for device code)
  - Raising from LLVM IR (current approach for host code)
  - Connection with other dialects
- Enabling the optimizations we envision
  - Some require host-side handling to some extent
    - Not looking into generating runnable code for now
- Extend and improve the SYCL dialect:
  - parallel_for, buffer, queue, etc.
  - Support lowering to different targets using `sycl` dialect abstractions:
    - SPIR-V
    - NVPTX…

[1] Experiments performed on 28/04/2023 by Intel, with Intel® Iris® Xe Graphics 1.3, CPU Model: 11th Gen Intel(R) Core(TM) i7-1165G7 @ 2.80GHz, DPC++ (sycl-mlir branch), OS: Debian_bookworm/sid_x86_64, Memory: 63 GB

[2] https://github.com/bcosenza/sycl-bench