Leveraging MLIR for Better SYCL Compilation

Víctor Pérez, Ettore Tiotto, Whitney Tsang, Lukas Sommer, Victor Lomüller

Codeplay Software & Intel

EuroLLVM 2023 - 11th of May

What is SYCL?

- An open standard heterogenous programming API introduced by Khronos
- Provides single-source programming model for accelerator processors
- Using ISO standard C++ code
- Allow accessing both high-level and low-level code
- Multiple implementations
  - ComputeCPP
  - DPC++
  - hipSYCL

Vector Add in SYCL

```
class add;
int main() {
  std::vector<float> a{…}, b{…}, c{…};
  queue q;
  {
    buffer<float> bufA{a}, bufB{b}, bufC{c};
    q.submit([&](handler &cgh) {
      accessor accA{bufA, cgh, read_only};
      accessor accB{bufB, cgh, read_only};
      accessor out{bufC, cgh, write_only, no_init};
      cgh.parallel_for<add>(a.size(),
          [=](id<1> i) { out[i] = accA[i] + accB[i]; });
    });
  }
}
```

Create device work queue

Vector Add in SYCL

```
class add;
int main() {
  std::vector<float> a{…}, b{…}, c{…};
  queue q;
  {
    buffer<float> bufA{a}, bufB{b}, bufC{c};
    q.submit([&](handler &cgh) {
      accessor accA{bufA, cgh, read_only};
      accessor accB{bufB, cgh, read_only};
      accessor out{bufC, cgh, write_only, no_init};
      cgh.parallel_for<add>(a.size(),
          [=](id<1> i) { out[i] = accA[i] + accB[i]; });
    });
  }
}
```

Create device work queue

Create data buffers

Vector Add in SYCL

```
class add;
int main() {
  std::vector<float> a{…}, b{…}, c{…};
  queue q;
  {
    buffer<float> bufA{a}, bufB{b}, bufC{c};
    q.submit([&](handler &cgh) {
      accessor accA{bufA, cgh, read_only};
      accessor accB{bufB, cgh, read_only};
      accessor out{bufC, cgh, write_only, no_init};
      cgh.parallel_for<add>(a.size(),
          [=](id<1> i) { out[i] = accA[i] + accB[i]; });
    });
  }
}
```

Create device work queue

Create data buffers

Specify access and requirements

Vector Add in SYCL

```
class add;
int main() {
  std::vector<float> a{…}, b{…}, c{…};
  queue q;
  {
    buffer<float> bufA{a}, bufB{b}, bufC{c};
    q.submit([&](handler &cgh) {
      accessor accA{bufA, cgh, read_only};
      accessor accB{bufB, cgh, read_only};
      accessor out{bufC, cgh, write_only, no_init};
      cgh.parallel_for<add>(a.size(),
          [=](id<1> i) { out[i] = accA[i] + accB[i]; });
    });
  }
}
```

Create device work queue

Create data buffers

Specify access and requirements

Submit data-parallel device kernel

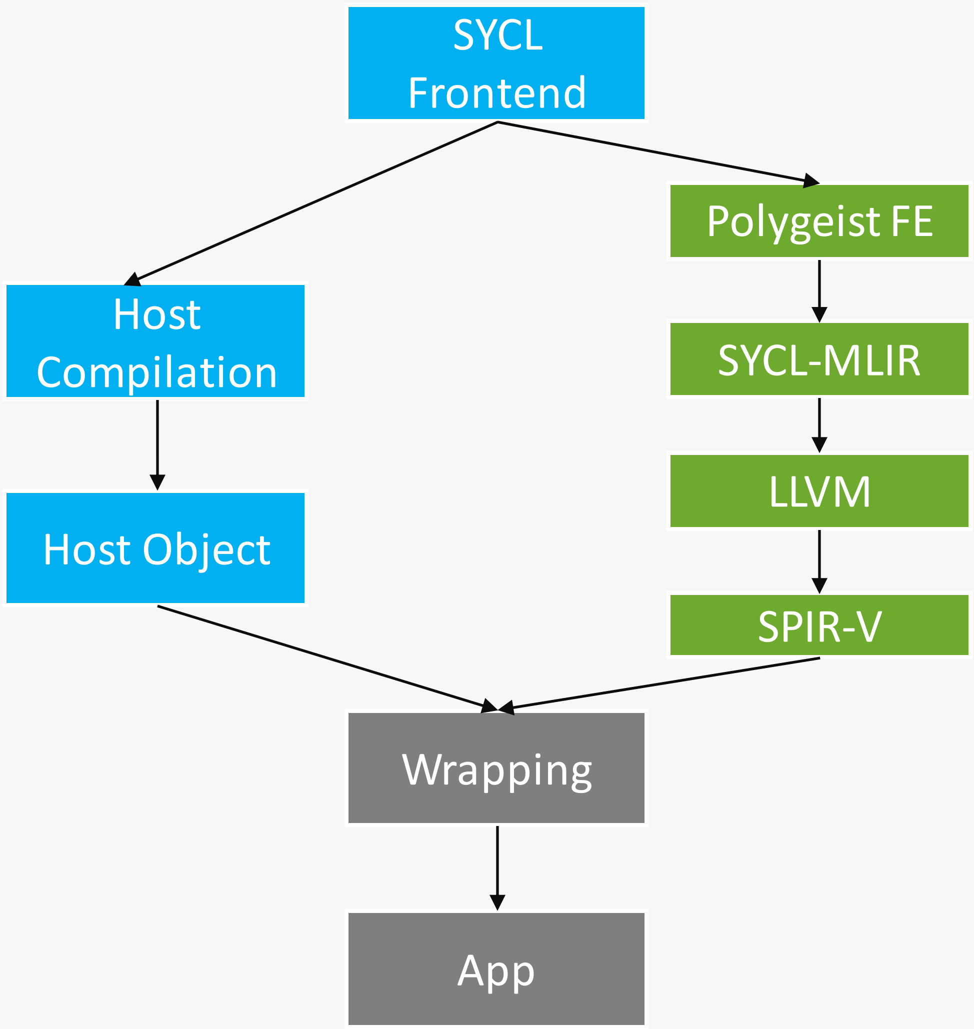
SYCL-MLIR Project Overview

- Aim: Better optimizations for SYCL compilers
  - Better optimization for device code
  - Optimization across the border between host and device code
- LLVM IR just not enough
  - Too low-level for some advanced optimizations
  - Currently no way of representing host and device code in one module
- MLIR better suited
  - Benefit from higher-level abstractions and gradual lowering
  - Ability to nest device code inside host

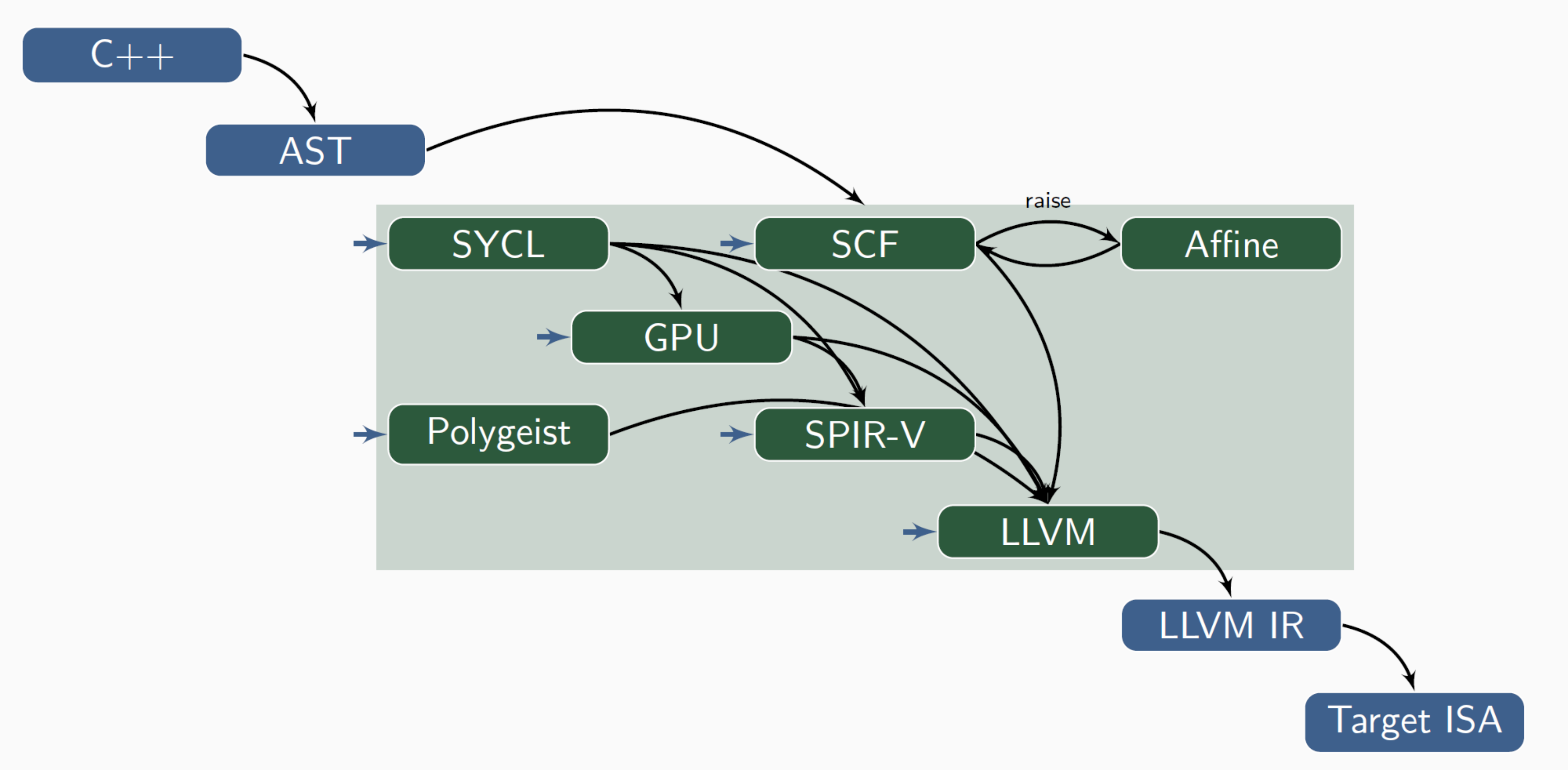
➡ Build an MLIR-based SYCL compiler

SYCL-MLIR – Current Status

- Based on DPC++
  - Currently, still two-pass compilation
- Use fork of Polygeist to handle C++ -> MLIR
  - Many fixes
  - Device code filtering
- Defined SYCL dialect
  - Represent types and operations defined by SYCL specification



SYCL-MLIR – Lowering Structure



Optimization example

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

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

- Goal: replace uses of `R[item]` by a reduction variable

Optimization example: LLVM sees function calls

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

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

```
  call spir_func void
@_ZN4sycl3_V12idILi1EEC2ILi1ELb1EEERNSt9enable_ifIXeqT_Li1EEKNS0_4itemILi1EXT0_EEEE4typeE(%"class.sycl::_V1::id"
addrspace(4)* noundef align 8 dereferenceable_or_null(8) %agg.tmp3.ascast, %"class.sycl::_V1::item" addrspace(4)*
noundef align 8 dereferenceable(24) %item.ascast) #11
  %agg.tmp3.ascast.ascast = addrspacecast %"class.sycl::_V1::id" addrspace(4)* %agg.tmp3.ascast to
%"class.sycl::_V1::id"*
  %call4 = call spir_func noundef align 4 dereferenceable(4) i32 addrspace(4)*
@_ZNK4sycl3_V18accessorIiLi1ELNS0_6access4modeE1026ELNS2_6targetE2014ELNS2_11placeholderE0ENS0_3ext6oneapi22accessor
_property_listIJEEEEixILi1EvEERiNS0_2idILi1EEE(%"class.sycl::_V1::accessor" addrspace(4)* noundef align 8
dereferenceable_or_null(32) %R2, %"class.sycl::_V1::id"* noundef byval(%"class.sycl::_V1::id") align 8
%agg.tmp3.ascast.ascast) #11
```

Optimization example: SYCL-MLIR encodes semantic

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

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

```
sycl.constructor @id(%id, %item)
%R_item_ptr = sycl.accessor.subscript %accessor[%id]
%R_item = affine.load %R_item_ptr[0] : memref<?xf32, 4>
%0 = arith.addf %R_item, <expr(k)> : f32
affine.store %0, %R_item_ptr[0] : memref<?xf32, 4>
```

If you want to know more - come see our poster!