oneAPI Technical Advisory Board Meeting: Toward function pointers in DPC++

September.23.2020

Sergey Kozhukhov

Rules of the Road

- DO NOT share any confidential information or trade secrets with the group
- DO keep the discussion at a High Level
  - Focus on the specific Agenda topics
  - We are asking for feedback on features for the oneAPI specification (e.g. requirements for functionality and performance)
  - We are NOT asking for feedback on any implementation details
- Please submit any implementation feedback in writing on Github in accordance with the Contribution Guidelines at spec.oneapi.com. This will allow Intel to further upstream your feedback to other standards bodies, including The Khronos Group SYCL* specification.

Notices and Disclaimers

oneAPI

The content of this oneAPI Specification is licensed under the Creative Commons Attribution 4.0 International License . Unless stated otherwise, the sample code examples in this document are released to you under the MIT license.

This specification is a continuation of Intel's decades-long history of working with standards groups and industry/academia initiatives such as The Khronos Group*, to create and define specifications in an open and fair process to achieve interoperability and interchangeability. oneAPI is intended to be an open specification and we encourage you to help us make it better. Your feedback is optional, but to enable Intel to incorporate any feedback you may provide to this specification, and to further upstream your feedback to other standards bodies, including The Khronos Group SYCL* specification, please submit your feedback under the terms and conditions below. Any contribution of your feedback to the oneAPI Specification does not prohibit you from also contributing your feedback directly to The Khronos Group or other standard bodies under their respective submission policies.

By opening an issue, providing feedback, or otherwise contributing to the specification, *you agree that Intel will be free to use, disclose, reproduce, modify, license, or otherwise distribute your feedback in its sole discretion without any obligations or restrictions of any kind, including without limitation, intellectual property rights or licensing obligations.* For complete contribution policies and guidelines, see Contribution Guidelines on www.spec.oneapi.com.

This document contains information on products, services and/or processes in development. All information provided here is subject to change without notice.

Agenda

1. Overview and motivation
2. Direction for implementation
3. Call for input

Overview

- In SYCL, function pointers are not allowed in device code
  - Aligned with restricted accelerator hardware capabilities
- DPC++ direction: Extension to relax this restriction
  - Stepping stone to virtual functions
  - In some applications function pointers substantially simplify the code
  - Expected in the modern C++ world
- Can be implemented on most hardware with manageable cost
  - Should be optional at a device granularity

One motivating example

- Embree is a library for ray tracing
- When intersecting rays, user shaders (defined as part of the user application) are executed by Embree. Production rendering can have 1000s of user shaders
- Callback pattern - app invokes lib and passes in functions for lib to call (shaders)

```
struct aclass {
    int (*method)(aclass*,int);
};
. . .
init_m (aclass* obj) {
     int(*m)(aclass*, int) = my_m;
     obj->method = m;
}
```

```
. . .
foo(aclass* obj, int A[N]) {
  for (i=0;i<n;i++){
    A[i] = (*obj->method)(obj, i);
  }
}
```

Creating/using a function pointer

Two options to represent function pointers:

1. **Implicit:** Usual C/C++ function pointers
   - As described later, must point to multiple versions of a function

```
int foo(int i) {return i+1;}
auto test() {
    int (*foo_simd) (int)  = foo;
    return foo_simd(5);
}
```

OR

```
class A {
    virtual int func(int);
};
```

2. **Explicit:** Use a special wrapper around the pointer(s)
   - Typically to encapsulate additional information, such as a table of pointers for different execution contexts (e.g., SIMD widths)

Reality - Sets of function variants / pointers

- Different calling contexts may require different versions of the callee
  - On SIMD devices, natural to vectorize kernels to subgroup size

```
float bar(float*, int);

int foo() [[intel::reqd_sub_group_size(8)]] {
   float *A =…;    int i = …;
   return bar(A, i); }
```

```
// another call, with different sg-size

int foo() [[intel::reqd_sub_group_size(16)]] {
   float *A =…;    int k = …;
   return bar(A, k); }
```

- Many other possible variants that a compiler may optimize for

```
float r = bar(A, i);  // unmasked context
if (r < 0) r += bar(A+2, i);  // masked context
```

- To store variants function pointers are table-like – dependent on target

Reality - Sets of function variants/pointers (2)

- Not a new problem. E.g. OpenMP `declare simd` and vector variants
- Need to be able to specify variants for performance, including:
  - Parameter styles (e.g., linear, uniform, varying)
  - Vector factor, masking

- Many options for language and implementation
  - Attributes vs wrappers
  - Variant encoding part of function pointer type vs not

- Intel has experience with vector function pointers in icc, described here
  - OpenMP vector attributes enabled on fptr but not added to language type system
  - Caused many challenges
    - See “SIMD-Enabled Function Pointers and the C++ Type System” in the link above

Option 1: Use C/C++ Function Pointers

- Every pointer defined this way is created with default set of variants
  - Set of variants defined by device compiler
  - If multiple translation units, might require LTO customization
  - Avoids challenges with type system – variants consistent / set by compiler
- Can cause unneeded/unused function variant creation
- Variants less specialized than if user-defined
  - To avoid combinatorial explosion in automatically-defined variants

DPC++ Proposed Direction:

- Use this mechanism to support **virtual functions**
- Also suitable for function pointers in SPMD compilations

Option 2: Create Wrapper for Function Ptrs

- Wrapper type provides interface to define and use function pointers containing addresses of different specific function variants
  - Interface includes conversions to different pointers for calling
  - Definition of specific variants in wrapper implies creation of those variants
- Wrapper avoids challenges with C++ type system while creating variants of the function
  - E.g., do we otherwise use complex attribute lists to mark function details?

DPC++ Proposed Direction:

- Wrapper type to support **SIMD variants** particularly when **user wants control**

Common to Both Options

In both cases, special intrinsics + annotations used in implementation

1. In place of indirect calls
2. On functions to indicate their address is taken and to provide correct initialization of function pointers (which are now tables)

More Details of WIP Wrapper

```
template <auto F, int S, class... T> // function to construct one vector variant
constexpr auto make_function_ref_tuned_impl() noexcept {
return std::array { __builtin_generate_SIMD_variant(F, S, std::add_pointer_t<T>())...};
}

template <typename Ret, int... S, class... Args, typename... T>
class function_ref_tuned<Ret(Args...), int_list<S...>, T...> {
  using F = Ret(Args...);
public:
// Call operator
  Ret operator()(Args &&... args) const {
    return __builtin_call_SIMD_variant(detail::variant_list<T...>(),  int_list<S...>(),
 ptrs.data(), std::forward<decltype(args)>(args)...);
  }
}
```

Work-in-progress Wrapper – Example Use

```
int foo(int i) {return A[i]+1;}  // A is a global array

using MyPtr = function_ref_tuned<int(int), int_list<4, 8>, unmasked(linear), masked(varying)>;


auto test(MyPtr* buf, cl::sycl::nd_item<1> item) [[intel::reqd_sub_group_size(8)]] {
  auto foo_simd = make_function_ref_tuned<foo, int_list<4,8>, unmasked(linear), masked(varying)>();

  // The wrapper-object can be put into a buffer for transfer
  *buf = foo_simd;

  // Or it can be just called.
  return foo_simd(5)                    // masked(varying) variant will be called here
    + foo_simd(item.get_local_id(0));   // unmasked(linear) variant will be called here
```

Looking for TAB Input

1. What use cases do you need enabled via function pointers?
2. Input on option 1 vs 2 from your experiences?
3. Do you have device considerations to factor into language design?
4. Any additions / corrections on what has been presented?
5. Would your code bases use functions pointers, and if so, what variant parameterizations would you require?]
6. Thoughts on proposing function pointers for the SYCL spec?