oneAPI Technical Advisory Board Meeting: invoke_simd

05/26/2021

John Pennycook, Roland Schulz, Jason Sewall

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.

Motivation

- On CPU and GPU: DPC++ maps SYCL sub-groups to hardware threads, and the work-items within each sub-group to SIMD lanes.
- User has limited control over SPMD-to-SIMD optimizations:
  - Compiler can deduce control flow/data sharing from certain operations (e.g. `any_of_group`, `all_of_group`, `group_broadcast`)
  - Sub-group functions expose some, but not all, "horizontal" operations (e.g. `reduce_over_group`)
- Direct control over SIMD behavior would be useful for optimization.

Design Goals

1. **Composability / Fine-grained Specialization**
   Enable very small (single instruction) SIMD regions.

2. **Portability**
   Support wide range of implementations for different architectures.
   No language changes to SYCL or C++.

3. **Interoperability**
   Allow the use of device-specific intrinsics where appropriate.

uniform<T>

- When mapping from SPMD to SIMD, compiler marks variables as:
  - ***Varying*** (i.e. the values on each work-item are assumed to be different)
  - ***Uniform*** (i.e. the values on each work-item can be proven to be the same)

- uniform<T> allows developers to override compiler analysis and declare that a variable is uniform.

```
template <class T>
class uniform {
  explicit uniform(T x) noexcept;
  operator T() const;
};
```


https://github.com/intel/llvm/blob/sycl/sycl/doc/extensions/Uniform/Uniform.asciidoc

Example Usage of `uniform<T>`

```
// If ptr is assumed non-uniform, use atomics to update it
template <typename T>
void update(sycl::sub_group sg, T* ptr, T x) {
  sycl::atomic_ref<T,
                   sycl::memory_order::relaxed,
                   sycl::memory_scope::device>(ptr) += x;
}

// If ptr is asserted to be uniform, use a sub-group reduce first
template <typename T>
void update(sycl::sub_group sg, uniform<T*> ptr, T x) {
  T sum = sycl::reduce_over_group(sg, x, std::plus<>());
  if (sg.leader()) {
    sycl::atomic_ref<T,
                     sycl::memory_order::relaxed,
                     sycl::memory_scope::device>(ptr) += sum;
  }
}
```

uniform<T> vs Intel® Implicit SPMD Program Compiler uniform

uniform<T>

- Variable is asserted to have same value
- Storage is implementation-defined
- Operations are performed by each work-item in convergent control flow:

```
uniform<int> x = 0;
if (sycl::any_of_group(sg, cond)) y = x + 1;
// y is 1 if cond was true for any lane
```

- Asserting uniformity for a varying value is user error and undefined behavior

uniform Keyword

- Variable is declared to have same address
- Storage is scalar (i.e. once per sub-group)
- Operations are performed once per sub-group without control flow requirements:

```
uniform int x = 0;
if (cond) y = x + 1;
// y is 1 if cond was true for any lane
```

- Assigning a varying (vector) value to a uniform is a compile-time error

Takeaway: uniform<T> is designed to convey semantic information with minimal implementation burden. Usages shown so far are optimization hints that can be ignored by an implementation.

invoke_simd

```
// SIMD function written using class based on Parallelism TS2
simd<float, 8> scale(simd<float, 8> x, float n) {
  return x * n;
}

q.parallel_for(..., sycl::nd_item<1> it) [[sycl::reqd_sub_group_size(8)]] {
  sycl::sub_group sg = it.get_sub_group();

  float x = ...;
  float n = ...;

  // invoke SIMD function from converged control flow
  // x values from each work-item are combined into a simd<float, 8>
  float y = invoke_simd(sg, scale, x, uniform(n));
});
```


https://github.com/intel/llvm/blob/sycl/sycl/doc/extensions/InvokeSIMD/InvokeSIMD.asciidoc

invoke_simd (with Mask)

```
// SIMD function written using class based on Parallelism TS2
simd<float, 8> masked_scale(simd<float, 8> x, float n, simd_mask<bool, 8> m) {
  where(mask, x) *= n;
  return x;
}

q.parallel_for(..., sycl::nd_item<1> it) [[sycl::reqd_sub_group_size(8)]] {
  sycl::sub_group sg = it.get_sub_group();

  float x = ...;
  float n = ...;

  // invoke SIMD function from converged control flow
  // x values from each work-item are combined into a simd<float, 8>
  // bool conditions from each work-item are combined into a simd_mask<bool, 8>
  float y = invoke_simd(sg, scale, x, uniform(n), it.get_local_id(0) % 2);
});
```

Argument Mapping

- When arguments are passed to `invoke_simd`, they are mapped to different types before being passed to the SIMD function:
  - `bool` ⇒ `simd_mask`
  - Arithmetic type T ⇒ `simd<T>`
  - `uniform<T>` ⇒ `T`
  - `std::tuple<bool, T, uniform<T>>` ⇒ `std::tuple<simd_mask, simd<T>, T>`

- Return values from the SIMD function undergo the reversed mapping.

Determining Sub-group Size

- To avoid specifying the sub-group size when calling **`invoke_simd`**, the call can only be valid for a single sub-group size

```
// Overload set with different sub-group sizes
struct foo {
  simd<float, 8> operator()(simd<float, 4> x, float n);           // (1)
  simd<float, 16> operator()(simd<float, 16> x, float n);         // (2)
};

// Overload set with different sub-group sizes and arguments
struct bar {
  simd<float, 8> operator()(simd<float, 8> x, float n);           // (3)
  simd<float, 16> operator()(simd<float, 16> x, simd<float, 16> y);  // (4)
};

invoke_simd(sg, foo{}, float(), uniform(float()));                // ambiguous: (1) or (2)
invoke_simd(sg, foo{}, uniform(simd<float, 4>()), uniform(float())); // selects (1)

invoke_simd(sg, bar{}, float(), float());                         // selects (4)
invoke_simd(sg, bar{}, float(), uniform(float()));                // ambiguous: (3) or (4)
```

Determining Sub-group Size

- If all parameters of a function are scalars, we cannot deduce the sub-group size (or prove that a function assumes a sub-group size).

- Proposed solution is a tag type:

```
// Function assumes a sub-group size of 8, uses simd<float, 8> internally
float scale(float* x, float n, simd_tag<8> = {});

// Function compatible with all sub-group sizes
// (e.g. may dispatch internally, or algorithm may be agnostic to SIMD size)
float scale(float* x, float n, simd_tag<std::dynamic_extent> = {});
```

Interoperability with Intrinsics

```
// Function interface uses portable SIMD representation
simd<int, 16> popcnt(simd<int, 16> x)
{
  // Convert from portable SIMD representation to intrinsic type
  __m512i mx = sycl::bit_cast<__m512i>(x);

  // Operate on intrinsic type using device-specific built-ins
  __m512i count = _mm512_popcnt_epi32(mx);

  // Convert back from intrinsic type to portable SIMD representation
  return sycl::bit_cast<simd<int, 16>>(count);
}
```

- Clear separation of mapping steps: 1) SPMD-to-SIMD; 2) SIMD-to-intrinsics
- Facilitates rapid prototyping and user-defined sub-group algorithms!

Summary

- `uniform<T>` and `invoke_simd` enable fine-grained specialization and optimization for SIMD architectures.

- Direction is aligned with specialization approaches in OpenMP, and spelling of explicit SIMD in ISO C++ (i.e. `std::experimental::simd`)

- Specification is designed to support header-only implementation(s) of argument mapping.

Notices and Disclaimers

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.