Implementing a SYCL Backend for Kokkos

Daniel Arndt

Oak Ridge National Laboratory
What is Kokkos?

- A C++ Programming Model for Performance Portability
  - Template library on top CUDA, HIP, OpenMP, SYCL, ...
  - Aligns with developments in the C++ standard, e.g., `mdspan`, `atomic_ref`
- Expanding solution for common needs of modern science and engineering codes
  - Math libraries based on Kokkos
  - Tools for debugging, profiling and tuning
  - Interoperability with Fortran and Python
- Open Source project with a growing community
  - Maintained and developed at https://github.com/kokkos
  - Hundreds of users at many large institutions
Kokkos as Portability Layer

Applications
- NREL/SNL NALU
  Wind Turbine CFD
- SNL LAMMPS
  Molecular Dynamics
- UT Uintah
  Combustine

Frameworks
- ORNL Raptor
  Large Eddy Sim

Frameworks
- ORNL Frontier
  Cray / AMD GPU
- LANL/SNL Trinity
  Intel Haswell / Intel KNL
- ANL Aurora
  Intel Xeon CPUs + Xe Compute
- SNL Astra
  ARM Architecture
- LLNL SIERRA
  IBM Power9 / NVIDIA Volta


Kokkos Tools: C.R. Trott, V. Kale, D. Lebrun-Grandie, D. Ibanez, S. Moore, A. Powell
Kokkos - Applications

More than 50% of C++ Codes in the Exascale Computing Project use Kokkos.

Example of Applications:
- Trilinos (Linear Algebra)
- PETSc (Linear Algebra)
- deal.II (Finite Element Library)
- LAMMPS (Molecular Dynamics)
- XGC (Fusion Reactor Simulation)
- ArborX (Geometry Search)
- Uintah (Chemical Reactions)
- VTK-m (Visualization)
- ...
Kokkos

Kokkos’ basic capabilities:
- Simple 1D data parallel computational patterns
- Deciding where code is run and where data is placed
- Managing data access patterns for performance portability
- Multidimensional data parallelism

Kokkos’ advanced capabilities:
- Thread safety, thread scalability, and atomic operations
- Hierarchical patterns for maximizing parallelism
- Task based programming with Kokkos

Kokkos’ tools and Kernels:
- How to profile, tune and debug Kokkos code
- Interacting with Python and Fortran
- Using KokkosKernels math library
Feature Status Kokkos+SYCL

Near feature-complete since release 3.4.00 (April 2021).

Unsupported

- atomic operations for big types (since Kokkos 4.0.00)
- WorkGraphPolicy
- Tasks
- Graphs
- Virtual functions/function pointer
Kokkos Core Functionalities, Mapping to SYCL

Constructs

- `parallel_for` $\rightarrow$ `sycl::parallel_for`
- `parallel_reduce` $\rightarrow$ `sycl::parallel_for`
- `parallel_scan` $\rightarrow$ `sycl::parallel_for`

Policies

- `RangePolicy` $\rightarrow$ `sycl::range`
- `MDRangePolicy` $\rightarrow$ `sycl::nd_range`
- `TeamPolicy` $\rightarrow$ `sycl::nd_range`

Memory

- `View` $\rightarrow$ `sycl::malloc/sycl::free`
Translate Kokkos to SYCL

```cpp
#include <Kokkos_Core.hpp>

int main() {
    Kokkos::ScopeGuard scope_guard;
    auto do_work = KOKKOS_LAMBDA(int) {};
    Kokkos::parallel_for(100, do_work);
}
```

is loosely translated to

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

int main()
{
    sycl::queue q;
    auto do_work = [] (int) {}; 
    q.parallel_for(sycl::range<1>(100),
                   [=] (sycl::item<1> item){
                      do_work(item.get_id());
                   });
}
```
#include <Kokkos_Core.hpp>

int main() {
    Kokkos::ScopeGuard scope_guard;
    Kokkos::parallel_for(1, KOKKOS_LAMBDA(int){
        printf("Hello World!\n");
    });
}

Problem:

- `printf` doesn’t work in SYCL kernels.
- Can’t pass `sycl::stream` anywhere.
#include <Kokkos_Core.hpp>

int main() {
    Kokkos::ScopeGuard scope_guard;
    Kokkos::parallel_for(1, KOKKOS_LAMBDA(int) {
        KOKKOS_IMPL_DO_NOT_USE_PRINTF("Hello World!
");
        // sycl::ext::oneapi::experimental::printf(
        //     "Hello World!
");
    });
}
#include <Kokkos_Core.hpp>

int main() {
    Kokkos::ScopeGuard scope_guard;
    const int N = 100;
    Kokkos::View<int*> view("view", N);
    Kokkos::parallel_for(N, KOKKOS_LAMBDA(int i){
        view(i) = i;
    });
}

Problem:

- Kokkos::View is not trivially copyable.
- sycl::is_device_copyable?
It is unspecified whether the implementation actually calls the copy constructor, move constructor, copy assignment operator, or move assignment operator of a class declared as `is_device_copyable` when doing an inter-device copy.

Likewise, it is unspecified whether the implementation actually calls the destructor for such a class on the device since the destructor must have no effect on the device.

Issue:

- Implementations actually call special member functions\(^1\)
- We need another workaround!

\(^1\)https://github.com/intel/llvm/issues/5320
union TrivialWrapper {
    TrivialWrapper(){};
    TrivialWrapper(const Functor& f) {
        std::memcpy(&m_f, &f, sizeof(m_f));
    }
    TrivialWrapper(const TrivialWrapper& other) {
        std::memcpy(&m_f, &other.m_f, sizeof(m_f));
    }
    TrivialWrapper(TrivialWrapper&& other) {
        std::memcpy(&m_f, &other.m_f, sizeof(m_f));
    }
    TrivialWrapper& operator=(const TrivialWrapper& other) {
        std::memcpy(&m_f, &other.m_f, sizeof(m_f));
        return *this;
    }
    TrivialWrapper& operator=(TrivialWrapper&& other) {
        std::memcpy(&m_f, &other.m_f, sizeof(m_f));
        return *this;
    }
    ~TrivialWrapper(){};
    Functor m_f;
};
template <typename Functor>
class SYCLFunctionWrapper {
    union TrivialWrapper m_functor;

public:
    SYCLFunctionWrapper(const Functor& functor, Storage&) : m_functor(functor) {}
    const Functor& get_functor() const {
        return m_functor.m_f;
    }
};

template <typename Functor>
struct sycl::is_device_copyable<
    SYCLFunctionWrapper<Functor, Storage, false>> :
    std::true_type {};

Minor annoyances compared with CUDA/HIP

- `sycl::group::barrier(sycl::sub_group)` doesn’t allow masking for active threads, yet.\(^2\)
- Address space annotations (`sycl::multi_ptr`) important, also for `sycl::atomic_ref`.
- Backend-specific information limited on Intel GPUs; difficult to choose good group sizes, e.g., according to occupancy, register usage.
- Status of SYCL standard implementation unclear, CUDA and HIP just tell what they have.
- Important features are only oneAPI extensions.
- Compile time for Kokkos unit tests (Release)
  - SYCL+Cuda: 100 min
  - CUDA/HIP: 45 min

---

\(^2\)https://github.com/intel/llvm/pull/6169
Used Extensions

- `sycl::ext::oneapi::experimental::this_nd_item`
- `sycl::ext::oneapi::experimental::printf`
- `sycl::ext::oneapi::experimental::device_global`
- `sycl::ext::oneapi::experimental::properties`
- `sycl::ext::oneapi::experimental::device_image_scope`
- `sycl::ext::oneapi::experimental::this_sub_group`
- `sycl::ext::oneapi::group_ballot`
- `sycl::ext::oneapi::sub_group_mask`
A lot of complaints but SYCL/DPC++

- integration was otherwise pretty smooth
- works well on Intel GPUs (Aurora)
- works much better than OpenMP-target
- has better support for newer C++ features than nvcc
- is a real portability solution with an interface close to Kokkos
Questions?
Acknowledgments

- This manuscript has been authored by UT-Battelle, LLC, under Contract No. DE-AC0500OR22725 with the U.S. Department of Energy.

- This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.