LA-UR-20-26335
Approved for public release; distribution is unlimited.

Title: Fortran Language Compatibility Library for Kokkos

Author(s): Womeldorff, Geoffrey Alan
Gaspar, Andrew James
Halverson, Scot Alan

Intended for: Performance, Portability, and Productivity in HPC Forum (P3HPC),
2020-09-01/2020-09-02 (Online, New Mexico, United States)

Issued: 2020-08-17

Los Alamos
NATIONAL LABORATORY
EST. 1943

Los Alamos NATIONAL LABORATORY

EST. 1943

Delivering science and technology
to protect our nation
and promote world stability

Fortran Language Compatibility Library for Kokkos

Performance, Portability, and Productivity in HPC (P3HPC) Forum

Geoff Womeldorff, Andrew Gaspar, Scot Halverson

9/1/2020

Talk Structure

- BLUF
- Motivation
- Incremental Porting on Hosts
  - Memory Allocated by Fortran
  - Usage Example
- Incremental Porting on Hosts and Devices
  - DualViews
  - Usage Examples
- Conclusions
  - Open Source
  - Future Ideas

Bottom Line Up Front

- Wrappers to allow Fortran memory to be used as Kokkos Views
  - (1D, 2D, …,7D) x (real, integer, complex) x (32,64), (also logical!)
- Routines to allocate memory with Kokkos from Fortran
  - (1D, 2D, 3D)x(real, integer)x(32,64), (also logical!)
- Lots of compatibility testing
  - x86 x gnu x 7.4 x (serial, openmp, cuda) x (release, debug) x (3.0, 3.1)
  - x86 x intel x (19,20) x (serial, openmp) x (release, debug) x (3.0, 3.1)
  - ppc x gnu x 7.4 x (serial, openmp, cuda) x (release, debug) x (3.0, 3.1)
  - ppc x XL x 16 x serial x (release, debug) x (3.0, 3.1)
- Open source: https://github.com/kokkos/kokkos-fortran-interop

Motivation

- HPC world owns many Fortran LOC!
  - Which we use every day! And is not going anywhere!
- But we generally cannot port it all at once.
- Thus we need an incremental porting strategy
- Keep our e.g. Fortran mains, drivers, physics packages
  - But port relevant infrastructure, or hotspot kernels to C++
- But C++ doesn’t have multi-dimensional arrays as first class citizens!
- So we chose Kokkos for its Views which gives us `everything’ from Fortran arrays, plus first-class knowledge of memory spaces

Fortran Language Compatibility Layer (FLCL)

- Our open-source contribution to the Kokkos ecosystem. Features:
  - Fortran compatible types for Kokkos View and DualViews, and routines for their allocation and deallocation, to share memory allocated from C++ to Fortran and C++ kernels.
  - Structs and helper routines to use Fortran memory from C++ kernels.
  - Unit tests testing and examples using above.
  - Utility routines for interfacing with Kokkos from Fortran.
  - CI-like scripts for testing against various configurations of Kokkos library and compiler families.

- See: https://github.com/kokkos/kokkos-fortran-interop

Incremental Porting on Hosts

- Memory Allocated by Fortran
- Initialization/Finalization of Kokkos
- Usage Example

Memory Allocated by Fortran

- Motivated by the particular scenario of incrementally porting kernels, or infrastructure (e.g MPI), but only running on host-based systems
- We’re not re-writing the mains/drivers for our codes.
- So – how best to share a chunk of memory across an ABI?
  - Our answer – recreate most of Fortran’s dope vector.
  - Then share that recreation across the ABI.
  - And on the far side, wrap things up in a Kokkos View.
- Then we have access to all of the very useful features of Kokkos with respect to parallel execution, and a convenient multidimensional access to memory.

nd_array_t

```fortran
integer, parameter :: ND_ARRAY_MAX_RANK = 8

type, bind(C) :: nd_array_t
  integer(c_size_t) :: rank
  integer(c_size_t) :: dims(ND_ARRAY_MAX_RANK)
  integer(c_size_t) :: strides(ND_ARRAY_MAX_RANK)
  type(c_ptr) :: data
end type nd_array_t
```

- nd_array_t keeps track of:
  - an array's rank
  - dimensions per rank
  - strides per rank
  - pointer to the beginning of the memory allocation

- How do we populate an nd_array_t?
  - A routine called: to_nd_array_(l|i32|i64|r32|r64|c32|c64)_(1|2|3|4|5|6|7)d
  - That's a little verbose, so thankfully we can wrap it up in an interface and just call: result = to_nd_array(foo)
  - Where foo is a Fortran array and result is an nd_array_t

nd_array_t

- Rank is inferred from the dummy argument's rank.
- Each rank's dimension can be read via size()
- Stride is a little trickier, but we settled on taking the difference between successive elements in each rank.
    - This avoids the need for the CONTIGUOUS attribute. (And thus a copy, if it's not already true!)
- Pointer to data is straightforward, as well.

Initializing and finalizing Kokkos

- Think of this as a once per program operation. Similar to MPI_Init/MPI_Finalize.
  - Generally, we call Kokkos::Initialize() directly after MPI_Init, and Kokkos::Finalize() directly before MPI_Finalize.
  - In FLCL, we provide:
    - kokkos_initialize() (this version parses command line Kokkos arguments)
    - kokkos_initialize_without_args()
    - kokkos_finalize()
  - See: https://github.com/kokkos/kokkos/wiki/Initialization

Usage Example

- How about an AXPY? Everyone loves the AXPY.

```fortran
program example_axpy
  use, intrinsic :: iso_c_binding
  use :: flcl_mod
  use :: axpy_f_mod

  implicit none

  real(c_double), dimension(:), allocatable :: c_y
  real(c_double), dimension(:), allocatable :: x
  real(c_double) :: alpha
  integer :: mm = 5000

  ... setup here ...

  call kokkos_initialize()
  call axpy(c_y, x, alpha)
  call kokkos_finalize()

end program example_axpy
```

Usage Examples

- The axpy() we call from our main.

```
module axpy_f_mod
    use, intrinsic :: iso_c_binding
    use :: flcl_mod
    public
      interface
        subroutine f_axpy ...
      end interface

      contains

        subroutine axpy( y, x, alpha )
          use, intrinsic :: iso_c_binding
          use :: flcl_mod
          implicit none
          real(c_double), dimension(:), intent(inout) :: y
          real(c_double), dimension(:), intent(in) :: x
          real(c_double), intent(in) :: alpha
          call f_axpy(to_nd_array(y), to_nd_array(x), alpha)
        end subroutine axpy
end module axpy_f_mod
```

Usage Examples

- The binding we invoke from our axpy()

```
module axpy_f_mod
    use, intrinsic :: iso_c_binding
    use :: flcl_mod
    public
      interface
        subroutine f_axpy( nd_array_y, nd_array_x, alpha ) &
          & bind(c, name='c_axpy')
          use, intrinsic :: iso_c_binding
          use :: flcl_mod
          type(nd_array_t) :: nd_array_y
          type(nd_array_t) :: nd_array_x
          real(c_double) :: alpha
        end subroutine f_axpy
      end interface

      contains
        subroutine axpy ...
end module axpy_f_mod
```

Usage Examples

- The C++ implementation of AXPY we ultimately invoke.

```
#include "flcl-cxx.hpp"
extern "C" {
  void c_axpy( flcl_ndarray_t *nd_array_y,
               flcl_ndarray_t *nd_array_x,
               double *alpha )
  {
    using flcl::view_from_ndarray;

    auto y = view_from_ndarray<double*>(*nd_array_y);
    auto x = view_from_ndarray<double*>(*nd_array_x);

    Kokkos::parallel_for( "axpy", y.extent(0), KOKKOS_LAMBDA( const size_t idx)
    {
      y(idx) += *alpha * x(idx);
    });

    return;
  }
}
```

Usage Example

- More details:
  - https://github.com/kokkos/kokkos-fortran-interop/tree/master/examples/01-axpy
- More examples:
  - https://github.com/kokkos/kokkos-fortran-interop/tree/master/examples

Incremental Porting on Hosts and Devices

- DualViews
  - Usage Examples

DualViews

- A Kokkos DualView is a view that has a backing memory allocation on both a Host and Device.
- Motivation for using one is that we want to give Fortran access to more exotic memory spaces in an incremental way.
- And that if possible, we would like the same user-facing implementation for multiple platforms.
- If we use DualView, we can write our kernels such that they use the device memory, and the right thing will happen on host-only platforms.

kokkos_allocate_dualview

- kokkos_allocate_dualview_(l|i32|i64|r32|r64)_(1|2|3)d
- Accepts as input
  - a Fortran pointer (to hold the View's host data)
  - an opaque pointer to the View (for scope)
  - a string to populate the View's label
  - extents, one per dimension

kokkos_allocate_dualview code flow

- A little more complicated than just wrapping up Fortran memory
- kokkos_allocate_dualview() (matched to type/rank)
  - Invokes a matching f_kokkos_allocate_dualview()
    - Which is bound to a matching c_kokkos_allocate_dualview()
      - Which stringifies a Fortran char array
      - Creates a matching type/rank DualView with the stringified label
      - And sets a temporary passed-in pointer to DualView's h_view.data()
  - Then we wrap up the Fortran pointer using c_f_pointer()
- Now we have a Fortran accessible DualView.

Usage Examples

- Let’s walk through allocating a DualView from Fortran
- First we start in some application specific wrapper:

```
! allocate 'physics arrays'
real(c_double), dimension(:), pointer :: array_x
real(c_double), dimension(:), pointer :: array_y
type(c_ptr) :: v_x
type(c_ptr) :: v_y
... setup here ...

call kokkos_allocate_dualview(array_x, v_x, ”array_x", length)
call kokkos_allocate_dualview(array_y, v_y, ”array_y", length)
```

Usage Examples

- Which goes through an interface and selects kokkos_allocate_dualview_r64_1d()

```
subroutine kokkos_allocate_dualview_r64_1d(A, v_A, n_A, e0)
  use, intrinsic :: iso_c_binding
  implicit none
  real(REAL64), pointer, dimension(:), intent(inout) :: A
  type(c_ptr), intent(out) :: v_A
  character(len=*), intent(in) :: n_A
  integer(c_int), intent(in) :: e0
  type(c_ptr) :: c_A

  character(len=:, kind=c_char), allocatable, target :: f_label
  call char_add_null( n_A, f_label )
  call f_kokkos_allocate_dualview_r64_1d(c_A, v_A, c_loc(f_label), e0)
  call c_f_pointer(c_A, A, shape=[e0])
end subroutine kokkos_allocate_dualview_r64_1d
```

Usage Examples

- f_kokkos_allocate_dualview_r64_1d() just exists to bind to its C counterpart.

```
interface
  subroutine f_kokkos_allocate_dualview_r64_1d(c_A, v_A, n_A, e0) &
    bind (c, name='c_kokkos_allocate_dualview_r64_1d’)
    use, intrinsic :: iso_c_binding
    implicit none
    type (c_ptr), intent(out) :: c_A
    type (c_ptr), intent(out) :: v_A
    type (c_ptr), intent(in) :: n_A
    integer (c_int), intent(in) :: e0
  end subroutine f_kokkos_allocate_dualview_r64_1d
end interface
```

Usage Examples

- The actual allocation then happens.

```
void c_kokkos_allocate_dualview_r64_1d( double** A,
                                        dualview_r64_1d_t** v_A,
                                        const char** f_label,
                                        const int* e0)
{
  const int e0t = std::max(*e0, 1);
  std::string c_label( *f_label );
  *v_A = (new dualview_r64_1d_t(c_label, e0t));
  *A = (*v_A)->h_view.data();
}
```

Usage Examples

- Finally back here, where we call c_f_pointer so that A wraps around h_view.data()

```
subroutine kokkos_allocate_dualview_r64_1d(A, v_A, n_A, e0)
  use, intrinsic :: iso_c_binding
  implicit none
  real(REAL64), pointer, dimension(:), intent(inout) :: A
  type(c_ptr), intent(out) :: v_A
  character(len=*), intent(in) :: n_A
  integer(c_int), intent(in) :: e0
  type(c_ptr) :: c_A

  character(len=:, kind=c_char), allocatable, target :: f_label
  call char_add_null( n_A, f_label )
  call f_kokkos_allocate_dualview_r64_1d(c_A, v_A, c_loc(f_label), e0)
  call c_f_pointer(c_A, A, shape=[e0])
end subroutine kokkos_allocate_dualview_r64_1d
```

Usage Examples

- For a usage example which is a little more complex, see a mesh operations proxy which usages FLCL / DualViews:
  - https://github.com/lanl/xkt
- In addition, while we show DualView in this section, we also have a version which uses just Views. This would be fine for CPU-based systems, but for GPU-based/accelerator-based systems requires some sort of UVM / coherent memory interface. Both are useful, but be aware of design constraints.

Conclusions

- Open Source
- Future Work

Open Source

- We released these ideas as open source.
- We want our lessons learned to be shared with the broader HPC community (and others).
- We `dogfood' this method in production, so it is battle-tested.
  - But, we're not omniscient.
  - So, we're happy to have help and new ideas!
  - Please feel free to file issues and/or merge requests:
  - https://github.com/kokkos/kokkos-fortran-interop
- Development is somewhat interrupt driven, so requirements for new features truly do matter.

Future Work

- We see the need for some fashion of memory manager on device-compute based systems.
  - For as long as host and device memories are not equal on a node.
  - UMPIRE is one choice (moreso, as Kokkos and RAJA become more compatible) (see Jeff Miles work to integrate UMPIRE into Kokkos)

Thanks

- We would like to thank ISO_C_BINDING for letting us do all of this in a standard way. Thank you, ISO_C_BINDING!

Thank you! Questions?

Thank you for listening!

If you have questions, please ask – womeld@lanl.gov, file an issue on github, or ask on the Kokkos slack.

Backup Slides

Usage Examples : Safety Features

FortranIndex<T>

- EAP has LOTS of indirection arrays
- This means dealing with LOTS of index lists
- And if you want to share these index lists without a copy between Fortran and C++…

- It means dealing with a lot of 1-based indices

FortranIndex<T>
Motivation

- As we've shown, arrays can be shared between C++ and Fortran
- Those arrays are, for better or worse, 0-based in C++ and 1-based Fortran
- However, indices don't get mapped in this way!
- Easy to mess-up: when porting a kernel, you have to manually "fix" 1-based indices
- Separate 0-based copy of index arrays is not feasible:
  - Easy for these arrays to end up out of sync, leading to hard to debug problems
  - Additional O(num_cell) allocations quickly add up

What is FortranIndex<T>?

LOGICALLY *0-based indexes*

```
Kokkos::View<FortranIndex<int32_t> *> x = {2, 5, 6, 3};
```

REPRESENTATIONALLY *1-based indexes*

```
integer(INT32) :: x(:) = [3, 6, 7, 4]
```

FortranIndex<T>

- C++ custom type
- Same size and layout as type T, where T is an integer type
- Internally stores integer value of type T by an offset of 1
- Works in non-host Kokkos execution and memory spaces
- ”Looks” like any integer
  - Assignment operators (including =, +=, -=, etc.) overloaded
  - Converts to other integer types, like a normal integer
  - Impossible to observe “internal” value without reinterpret_cast
- Benchmark/Code-gen
  - Literally just an additional inc/dec
  - No overhead vs. explicit ”- 1” (in my testing!)

Getting the Views to the right place, when Fortran knows nothing about them

- But, when we want to run a device-based kernel, how do ask Fortran to convey which View(s) we want it to use?
  - Assuming of course, we have some physics-motivated code which passes around multi-dimensional arrays (raise your hand if you're like us!)
- We spit-balled a few different ideas, and implemented one of them
  - The first, is some sort struct or dictionary which would hold a list of View pointers
    - Used to passing around some blob of state, so this would be one more thing to add to that
  - The second, a hashing between the Fortran pointers and the DualView pointers
    - A little more automatic!

Usage Examples

- What if we add a bookkeeping hashmap to our allocation routine?

```
void c_kokkos_allocate_dualview_r64_1d( double** A,
                                        dualview_r64_1d_t** v_A,
                                        const char** f_label,
                                        const int* e0)
{
  const int e0t = std::max(*e0, 1);
  std::string c_label( *f_label );
  *v_A = (new dualview_r64_1d_t(c_label, e0t));
  *A = (*v_A)->h_view.data();
  insert_dualview_reference(A, v_A);
}
```

Usage Examples

- Which looks something like this:

```
void insert_dualview_reference( void* host_ptr,
                                void* dualview_ptr )
{
  void** hp_temp1 = (void**)host_ptr;
  void* hp_temp2 = *(hp_temp1);
  void** dv_temp1 = (void**)dualview_ptr;
  void* dv_temp2 = *(dv_temp1);
    if (host_to_dualview_map == NULL) {
    host_to_dualview_map = new std::map<void*,void*>();
  }
  if ( (*host_to_dualview_map).find(hp_temp2) !=
       (*host_to_dualview_map).end() ) {
    std::cout << "Key already exists!\n";
  }
  (*host_to_dualview_map)[hp_temp2] = dv_temp2;
}
```

Usage Examples

- Then when we get to the C++ part of the kernel wrapper (but before we get to the kernel launch on the device, we could simply:

```
void* retrieve_dualview_reference(void** host_ptr){
  return (*host_to_dualview_map)[host_ptr];
}
```

- And then we're off to the races and we have our DualView back where it matters (at the parallel_X launch sites).