Kokkos 3.7 Release Briefing

New Capabilities

October 4, 2022

Sandia National Laboratories is a multi-mission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.
UPDATEME SAND2020-7755 PE

3.7 Release Higlights

- Documentation
- SIMD
- HIPManagedSpace
- More View Allocation Properties Support
- Initialization
- Deprecations
- Kokkos 4.0 preview

Online Resources:

- https://github.com/kokkos:
  - Primary Kokkos GitHub Organization
- https://github.com/kokkos/kokkos-tutorials/wiki/Kokkos-Lecture-Series:
  - Slides, recording and Q&A for the Full Lectures
- https://kokkos.github.io/kokkos-core-wiki:
  - Wiki including API reference
- https://kokkosteam.slack.com:
  - Slack channel for Kokkos.
  - Please join: fastest way to get your questions answered.
  - Can whitelist domains, or invite individual people.

Would like to strengthen community bonds and discoverability

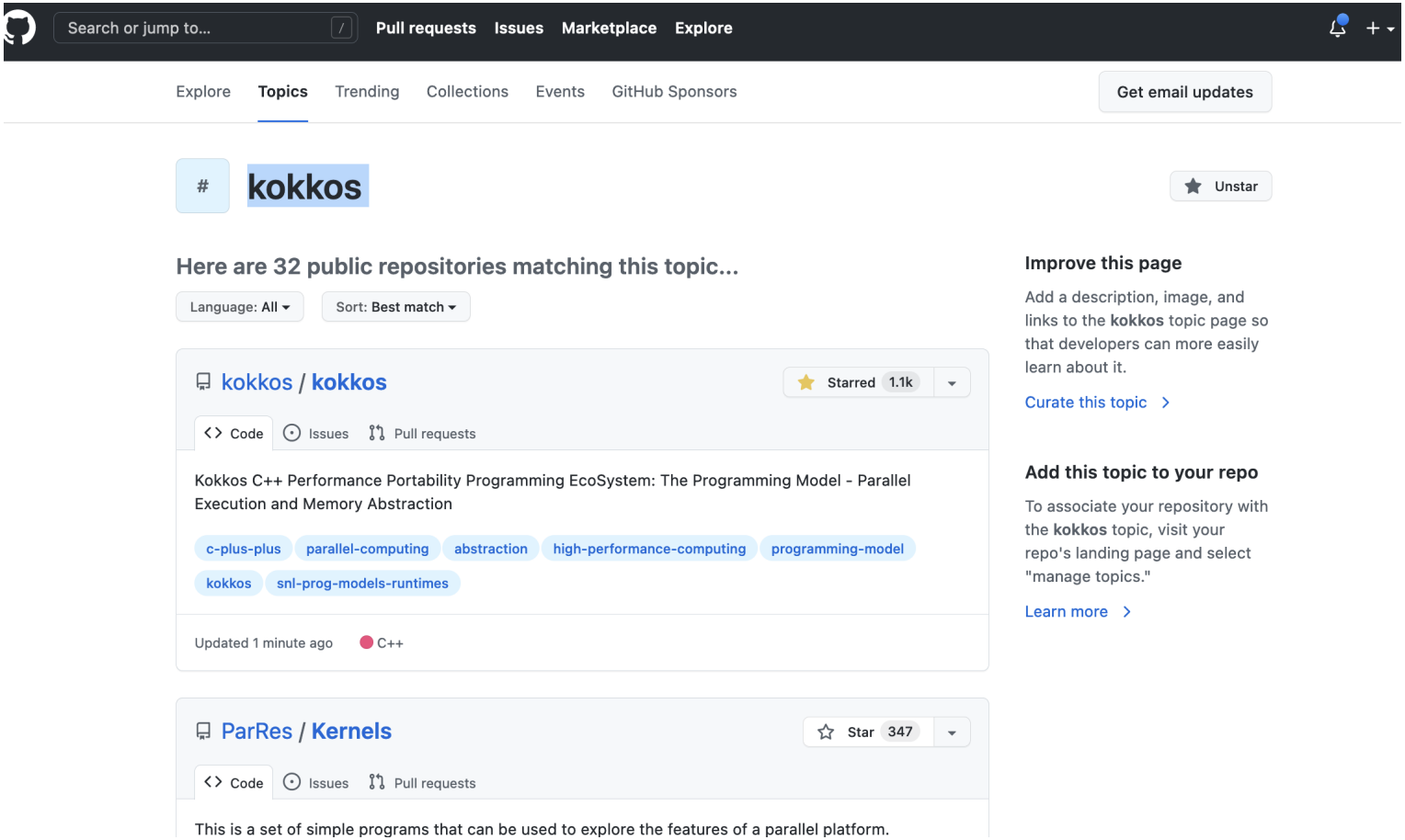
List of Applications and Libraries

- Add your app to https://github.com/kokkos/kokkos/issues/1950
- We are planning to add that to a Kokkos website.
- Helps people discover each other when working on similar things.

GitHub Topics

- Use *kokkos* tag on your repos.
- If you click on the topic you get a list of all projects on github with that topic.

Kokkos Topic



New Documentation Website

Kokkos Documentation Now on https://kokkos.github.io

- Transition to Sphinx syntax
- More flexibility in site layout and style
- Better update processes
  - Source for core documentaiton at https://github.com/kokkos/kokkos-core-wiki
  - Using pull requests with auto deploy
  - Pull requests to improvde documentation are welcome!

Kokkos: The Programming Model

C++ Performance Portability Programming Model

Kokkos Core implements a programming model in C++ for writing performance portable applications targeting all major HPC platforms. For that purpose it provides abstractions for both parallel execution of code and data management. Kokkos is designed to target complex node architectures with N-level memory hierarchies and multiple types of execution resources. It currently can use CUDA, HIP, SYCL, HPX, OpenMP and C++ threads as backend programming models with several other backends development.

The Kokkos EcoSystem includes:

Name	Info	
`kokkos`	(this library) Programming Model - Parallel Execution and Memory Abstraction	Github link
`kokkos-kernels`	Sparse, dense, batched math kernels	Github link
`kokkos-tools`	Profiling and debugging tools	Github link
`pykokkos`	Provides Python bindings to the Kokkos performance portable parallel programming.	Github link
`kokkos-remote-spaces`	Shared memory semantics across multiple processes	Github link
`kokkos-resilience`	Resilience and Checkpointing Extensions for Kokkos	Github link

Math functions are now in the primary namespace

- Now call `Kokkos::sin` etc. instead of `Kokkos::Experimental::sin`

Enabled nested reduce with teams that are not power-of-two size!

SIMD Moving into Core Repository

Previously maintained as its own repository.

- Incrementally moving capabilities over.
  - Started with classic CPU SIMD: AVX512 and Scalar implementation.
- Planned: all capabilities will be moved by Kokkos 4.2.
- Will update lecture on SIMD after that.

We encourage testing of this capability, and providing us feedback!

Old

```
using simd_t = simd::simd<
   double, simd::simd_abi::native>
>;
```

New

```
using simd_t = Kokkos::Experimental::simd<
   double,
   Kokkos::Experimental::simd_abi::native<double>
>;
```

```
using simd_t = Kokkos::Experimental::simd<
   double,
   Kokkos::Experimental::simd_abi::native<double>
>;

int N = N_in/simd_t::size();

Kokkos::View<simd_t**> data("D",N,M);
Kokkos::View<simd_t*> results("R",N);

double a = 3.0;
Kokkos::parallel_for("Combine",data.extent(0), KOKKOS_LAMBDA(con
  simd_t tmp = 0.0;
  double b = a;
  for(int j=0; j<data.extent(1); j++) {
    tmp += b * data(i,j);
    b+=a+1.0*(j+1);
  }
  results(i) = tmp;
});
```

`HIP` MemorySpaces

`HIP` backend got a new page-migrating memory space

- Page migrating memory space similar to `CudaUVMSpace` for `HIP`
- Coarse-grained memory, thus changes are visible at kernel exit (also changed for `HIPHostPinnedSpace`)
- OS migrates memory to local memory on access (**not** zero copy access like `HIPHostPinnedSpace`)

No configure time option to make it the default!

Requirements for page-migration on `HIP`

- Have a GPU that supports the feature (e.g. $\geq$ MI200)
- Have a OS that supports the feature (e.g. Kernel HMM module)
- Compile with `xnack:+` or `xnack:any` (default)
- Set runtime variable `HSA_XNACK=1` or `amdgpu.noretry=0` at boot time

Without working xnack (compiletime and runtime) the memory will not migrate but be zero copy access via the interconnecting bus

More View Allocation Properties Support

Content: Support `view_alloc` argument for

- `resize`
- `realloc`
- `create_mirror`
- `create_mirror_view`
- `create_mirror_view_and_copy`

for `View`-like types

Motivation: Avoid initialization, avoid fencing, structured syntax, ...

Relevant properties:

- `WithoutInitializing`
- memory spaces
- execution spaces(**new**)
- labels

Example:

```
Kokkos::view_alloc(Kokkos::WithoutInitializing,
                   Kokkos::HostSpace{},
                   Kokkos::OpenMP{});
```

```
template <class I, class T, class... ViewCtorArgs>
void
resize(const Impl::ViewCtorProp<ViewCtorArgs...>& arg_prop,
       View<T, P...>& v,
       size_t n0, size_t n1, size_t n2, size_t n3,
       size_t n4, size_t n5, size_t n6, size_t n7);

template <class T, class... P, class... ViewCtorArgs>
void
resize(const Impl::ViewCtorProp<ViewCtorArgs...>& arg_prop,
       View<T, P...>& v,
       const typename View<T, P...>::array_layout& layout);
```

Resizes `v` to have the new dimensions while preserving the contents for the common subview of the old and new `View`. The new `View` is constructed using the View constructor properties `arg_prop`, e.g., `WithoutInitializing`.

```
template <class T, class... P, class... ViewCtorArgs>
void
realloc(const Impl::ViewCtorProp<ViewCtorArgs...>& arg_prop,
        View<T, P...>& v,
        size_t n0, size_t n1, size_t n2, size_t n3,
        size_t n4, size_t n5, size_t n6, size_t n7);

template <class I, class T, class... ViewCtorArgs>
void
realloc(const Impl::ViewCtorProp<ViewCtorArgs...>& arg_prop,
        View<T, P...>& v,
        const typename View<T, P...>::array_layout& layout);
```

Resizes `v` to have the new dimensions whithout preserving the contents of the old `View`. The new `View` is constructed using the View constructor properties `arg_prop`, e.g., `WithoutInitializing`.

```
template <class ViewType, class... ViewCtorArgs>
auto
create_mirror(
  const Impl::ViewCtorProp<ViewCtorArgs...>& arg_prop,
  ViewType const& src);
```

Creates a new `View` with the same layout and padding as `src`. The new `View` is constructed using the View constructor properties `arg_prop`, e.g., `WithoutInitializing`. The memory space for the new View is host-accessible if no memory space was specfifed.

```
template <class ViewType, class... ViewCtorArgs>
auto
create_mirror_view(
  const Impl::ViewCtorProp<ViewCtorArgs...>& arg_prop,
  ViewType const& src);
```

If `src` can't access the specified memory space, it creates a new `View` in that memory space with the same layout and padding as `src` using the View constructor properties `arg_prop`, e.g., `WithoutInitializing`.

```
template <class ViewType, class... ViewCtorArgs>
typename ViewType::HostMirror
create_mirror_view_and_copy(
  const Impl::ViewCtorProp<ViewCtorArgs...>& arg_prop,
  ViewType const& src);
```

If `src` can't access the specified memory space, it creates a new `View` in that memory space with the same layout and padding as `src` using the View constructor properties `arg_prop`, e.g., `WithoutInitializing`. The contents of `src` are copied.

Build System Updates

Objectives:

- New Architectures
- Compiler Versions

New explicit CPU architectures

- Kokkos_ARCH_SKL: Intel Skylake Client CPUs
- Kokkos_ARCH_ICL: Intel Ice Lake Client CPUs
- Kokkos_ARCH_ICX: Intel Ice Lake Xeon Server CPUs
- Kokkos_ARCH_SPR: Intel Sapphire Rapids Xeon Server CPUs

New explict GPU architecture:

- Kokkos_ARCH_INTEL_PVC: Intel GPU Ponte Vecchio (AOT)
- Kokkos_ARCH_INTEL_GEN: Intel GPUs (JIT)

NATIVE: Optimize for local CPU architecture

Updated minimum compiler requirements:

	3.6.01	3.7.00
Clang (OMPT):	13.0.0	14.0.0
IntelLLVM (SYCL):	2022.0.0	2022.1.0

Minimum version for other compilers stayed the same.

Kokkos execution environment

A few reminders on the Kokkos execution environment

- `Kokkos::initialize` must be called before any other API function or constructor
- Kokkos can be initialized at most once
- Kokkos must be finalized before exiting the program if it has been initialized
- All Kokkos objects must be destroyed before `Kokkos::finalize` gets called
- Once `Kokkos::finalize` has been called, no Kokkos API function nor constructor may be called

Added `Kokkos::is_finalized`

- `Kokkos::is_initialized` returning `false` does not mean it is safe to call `Kokkos::initialize()`
- `Kokkos::is_finalized()` fills that gap
- Just like `Kokkos::is_initialized`, `Kokkos::is_finalized` can be called at any time (exception to the rule on the previous slide)

	Before `initialize()`	**After** `initialize()` but **before** `finalize()`	**After** `finalize()`
`is_initialized` returns	`false`	`true`	`false`
`is_finalized` returns	`false`	`false`	`true`

ScopeGuard behavior change

- Old behavior: constructor calls `Kokkos::initialize()` if `Kokkos::is_initialized()` is `false`
  - *Problem*: silently ignores potentially inconsistent settings, like device-id or number of threads
- New behavior: constructor simply forwards arguments to `Kokkos::initialize()`
  - *Consequence*: the constructor may indirectly abort if Kokkos was already initialized.
- Unchanged: destructor calls `Kokkos::finalize()`

```
// if you really think you need the old behavior
auto guard = std::unique_ptr<Kokkos::ScopeGuard>(
    Kokkos::is_initialized() ? new Kokkos::ScopeGuard()
                             : nullptr);
```

ScopeGuard typical use cases

```
int main(int argc, char* argv[]) {
  // Does not require introducing a scope as with
  // Kokkos::initialize();
  // {
  //   Kokkos::View<int> foo("foo");
  //   /*...*/
  // }
  // Kokkos::finalize();
  Kokkos::ScopeGuard guard(argc, argv);
  Kokkos::View<int> foo("foo");
  /*...*/
}

// With more complex control flow ensures that
// Kokkos::finalize() is called upon scope exit
Kokkos::ScopeGuard guard;
/*...*/
if (COND)
  return EXIT_FAILURE;  // finalize here
/*...*/
return EXIT_SUCCESS;    // and here
```

Prefer `Kokkos::initialize(int& argc, char* argv[])` **unless you have a good reason!**

```
#if KOKKOS_VERSION < 30700
// Deprecated
Kokkos::InitArguments args;
args.device_id = 1;
args.set_num_threads = 2;
args.disable_warnings = true;
Kokkos::initialize(args);
#else
// Since Kokkos 3.7
Kokkos::initialize(Kokkos::InitializationSettings()
                        .set_device_id(1)
                        .set_num_threads(2)
                        .set_disable_warnings(false));
#endif
```

Why changing?

- Difficult to get rid of deprecated data members (e.g. `numa`) or rename them
- Consistency with command-line arguments and environment variables
- Cannot distinguish user-specified from defaulted options

Settings available

command line arguments

```
--kokkos-num-threads=1
--kokkos-device-id=2
--kokkos-disable-warning=false
--kokkos-map-device-id-by=random
--kokkos-print-configuration
--kokkos-tune-internals=true
--kokkos-tools-libs="f.so;b.so"
--kokkos-tools-args="\"-x 2\""
--kokkos-tools-help
```

environment variables

```
KOKKOS_NUM_THREADS=1
KOKKOS_DEVICE_ID=2
KOKKOS_DISABLE_WARNINGS=false
KOKKOS_MAP_DEVICE_ID_BY=random
KOKKOS_PRINT_CONFIGURATION=1
KOKKOS_TUNE_INTERNAL=true
KOKKOS_TOOLS_LIBS="f.so;b.so"
KOKKOS_TOOLS_ARGS="\"-x 2\""
KOKKOS_TOOLS_HELP=YES
```

```
Kokkos::InitializationSettings
set_num_threads(1)
set_device_id(2)
set_disable_warnings(false)
set_map_device_id_by("random")
set_print_configuration(true)
set_tune_internals(true)
set_tools_libs("f.so;b.so")
set_tools_largs("\"-x 2\"")
set_tools_help(true)
```

- All non-prefixed options (other than `--help`) are deprecated
- Boolean as string (case insensitive)
  - `true|yes|1`
  - `false|no|0`
- Pay attention to the warnings that are raised when using deprecated settings
- Use `--kokkos-help` to find out the list of available options

- Added `Kokkos::is_finalized`
  May initialize if `!is_initialized()&&!is_finalized()`
- `Kokkos::ScopeGuard` constructor unconditionally forwarding arguments to `Kokkos::initialize`
- `Kokkos::InitArguments` deprecated in favor of `Kokkos::InitializationSettings`
- Command line arguments and environment variables updated to increase consistency

Further Deprecations in Release 3.7

Removing `finalize_all`

- Deprecated `Kokkos::finalize_all()`
- Use `Kokkos::finalize()` instead

Non-volatile member function called

```
struct DeprecatedReducer {  // just happened to work
  KOKKOS_FUNCTION
  void join(value_type volatile& dst,
            value_type const volatile& src) const;
};

struct Release37AndBeyondReducer {
  KOKKOS_FUNCTION
  void join(value_type& dst,
            value_type const& src) const;
};

struct BackwardCompatibleReducer {
  KOKKOS_FUNCTION
  void join(value_type& dst,
            value_type const& src) const;
  KOKKOS_FUNCTION
  void join(value_type volatile& dst,
            value_type const volatile& src) const;
};
```

Compilation error when including private Kokkos headers

```
<kokkos>/core/src/Kokkos_View.hpp:47:1: error: static_assert failed
  "Including non-public Kokkos header files is not allowed."
static_assert(false,
^             ~~~~~
1 error generated.
```

- See GitHub Issue #4856
- Why can't I just include <Kokkos_View.hpp> or <Kokkos_Parallel.hpp>

Symbol	header #include
Kokkos::View	<Kokkos_Core.hpp>
Kokkos::parallel_for	<Kokkos_Core.hpp>
Kokkos::fence	<Kokkos_Core.hpp>
KOKKOS_FUNCTION	<Kokkos_Core.hpp> or <Kokkos_Macros.hpp>
Kokkos::Cuda	<Kokkos_Core.hpp>
Kokkos::atomic_fetch_add	<Kokkos_Core.hpp> or <Kokkos_Atomic.hpp>
Kokkos::pi	<Kokkos_Core.hpp> or <Kokkos_MathematicalConstants.hpp>
Kokkos::cos	<Kokkos_Core.hpp> or <Kokkos_MathematicalFunctions.hpp>
Kokkos::sort	<Kokkos_Sort.hpp>

Kokkos headers you may `#include`

Core

- `<Kokkos_Core.hpp>`
- `<Kokkos_Macros.hpp>`
- `<Kokkos_Atomic.hpp>`
- `<Kokkos_DetectionIdiom.hpp>`
- `<Kokkos_MathematicalConstants.hpp>`
- `<Kokkos_MathematicalFunctions.hpp>`
- `<Kokkos_NumericTraits.hpp>`
- `<Kokkos_Array.hpp>`
- `<Kokkos_Complex.hpp>`
- `<Kokkos_Pair.hpp>`
- `<Kokkos_Half.hpp>`
- `<Kokkos_Timer.hpp>`

Algorithms

- `<Kokkos_StdAlgorithms.hpp>`
- `<Kokkos_Random.hpp>`
- `<Kokkos_Sort.hpp>`

Containers

- `<Kokkos_Bit.hpp>`
- `<Kokkos_DualView.hpp>`
- `<Kokkos_DynRankView.hpp>`
- `<Kokkos_DynamicView.hpp>`
- `<Kokkos_ErrorReporter.hpp>`
- `<Kokkos_Functional.hpp>`
- `<Kokkos_OffsetView.hpp>`
- `<Kokkos_ScatterView.hpp>`
- `<Kokkos_StaticCrsGraph.hpp>`
- `<Kokkos_UnorderedMap.hpp>`
- `<Kokkos_Vector.hpp>`

Example usage of `WorkTag` argument

```
struct DoTheThing {
  template <class ExecSpace>
  DoTheThing(ExecSpace const& exec, int n) {
    if (n < 100) {
      Kokkos::parallel_for("FooSmall",
          Kokkos::RangePolicy<Small, ExecSpace>(exec, 0, n),
          *this);
    } else {
      Kokkos::parallel_for("FooBig",
          Kokkos::RangePolicy<Big, ExecSpace>(exec, 0, n),
          *this);
    }
  }
  KOKKOS_FUNCTION void operator()(Small, int i) const { /*...*/ }
  KOKKOS_FUNCTION void operator()(Big, int i) const { /*...*/ }
};
```

WorkTag must be an empty type

```
struct Small {
  float pi = 3.14f;  // <- not an empty class!
};
struct Big {};
```

yields

```
<kokkos>/core/src/impl/Kokkos_AnalyzePolicy.hpp:172:7: error: implicit instantiation of
  undefined template 'Kokkos::Impl::show_name_of_invalid_execution_policy_trait<Small>'
      show_name_of_invalid_execution_policy_trait<Trait>{};
      ^
<kokkos>/core/src/impl/Kokkos_AnalyzePolicy.hpp:159:7: note: in instantiation of template class
  'Kokkos::Impl::AnalyzeExecPolicyUseMatcher<void, Kokkos::Impl::type_list<>, Small, Kokkos::Serial>'
  requested here
    : AnalyzeExecPolicyUseMatcher<void, type_list<TraitSpecs...>, Trait,
      ^
<snip>
```

Motivating example

```
Kokkos::parallel_for(
    Kokkos::RangePolicy<Kokkos::HostSpace>(0, 1), // <- Uuups
    KOKKOS_LAMBDA(int i){ /*...*/ });
```

- GitHub PR #5230 warning if `std::is_empty<WorkTag>::value` is `false`
- Warning becomes an error in Kokkos 4.0
- Static data member and member types are fine

Pass label as the first argument instead!

```
// DEPRECATED
Kokkos::parallel_for(N, func, "MyLabel");
Kokkos::parallel_for(policy, func, "MyLabel");
Kokkos::parallel_scan(N, func, "MyLabel");
Kokkos::parallel_scan(policy, func, "MyLabel");
Kokkos::parallel_scan(N, func, val, "MyLabel");
Kokkos::parallel_scan(policy, func, val, "MyLabel");
```

Removed (no replacement)

- Trailing boolean whether to force using `Kokkos::BinSort`
- Was defaulted to `false`
- Looking forward: support for non-arithmetic types and custom comparison

```
// DEPRECATED
Kokkos::sort(v, true);
Kokkos::sort(exec, v, true);
```

- Disable deprecated code (configure with `-DKokkos_ENABLE_DREPRECATED_CODE_3=OFF`)
- Reducer `join` member function taking `volatile`-qualified argumens are deprecated
- Do not include private Kokkos headers
- `WorkTag` must be an empty type
- Name your kernels by passing a string as **argument** argument
- `Kokkos::sort` does not accept trailing boolean argument any more
- `InitArguments` replaced by `InitializationSettings`
- `ScopeGuard` behavior change with respect to prior initialization

Important upcoming changes in 4.0

Kokkos 4.0 will require C++17!

- Will support C++17, C++20 and C++23
- Allows us to keep the testing amount manageable
- Will enable new interfaces and streamlined implementation
  - Use of CTAD reduces the need to spell template arguments out
  - Fold expressions help with internal implementation, and improve compile times
  - `constexpr if` reduces use of clunky SFINAE patterns

New Compiler Minimums

Compiler	Version
GCC	8.2
Clang	8.0
Clang as CUDA compiler	10.0
Intel	19.0.5
CUDA-NVCC	11.0
CUDA with Clang as CUDA compiler	10.0.1
ROCM	5.2.0
IntelLLVM (CPU)	2021.1.1
IntelLLVM (SYCL)	2022.2.0
NVC++	22.3
MSVC	19.29
IBM XL	Not Supported
Classic PGI	Not Supported

HIP Backend will be promoted from Experimental in 4.0

- Use `Kokkos::HIP` etc. instead of `Kokkos::Experimental::HIP`.
- We will now support ROCM versions longer, and not update with every minor release.
- For a transition time `HIP` will be available in both namespaces.

Removal of deprecated code

- We will start to remove code which was deprecated during the 3.x release cycle
- Turn `Kokkos_ENABLE_DEPRECATED_CODE_3 OFF` to remove that code in 3.7 and check whether you are ready
  - There are just a handful exceptions we will leave in for one or two more minor cycles to give more transition time
- `Kokkos_ENABLE_DEPRECATED_CODE_4` will be used for features deprecated in the 4.x release cycle, slated for removal in 5.0 - maybe 2024/25