Utilisation cheat sheet for Kokkos :: page 1

v4.6.1.20250516

Header

```
#include <Kokkos_Core.hpp>
```

Initialization

Initialize and finalize

```
int main(int argc, char* argv[]) {
    Kokkos::initialize(argc, argv);
    { /* ... */ }
    Kokkos::finalize();
}
```

Scope guard

```
int main(int argc, char* argv[]) {
    Kokkos::ScopeGuard kokkos(argc, argv);
    /* ... */
}
```

Kokkos concepts

Execution spaces

Execution space	Device backend	Host backend
`Kokkos::DefaultExecutionSpace`	On device	On host
`Kokkos::DefaultHostExecutionSpace`	On host	On host

Memory spaces

Generic memory spaces

Memory space	Device backend	Host backend
`Kokkos::DefaultExecutionSpace::memory_space`	On dev.	On host
`Kokkos::DefaultHostExecutionSpace::memory_space`	On host	On host

Specific memory spaces

Memory space	Description
`Kokkos::HostSpace`	Accessible from the host but maybe not from the device
`Kokkos::SharedSpace`	Accessible from the host and the device; copy managed by the driver
`Kokkos::SharedHostPinnedSpace`	Accessible from the host and the device; zero copy access in small chunks

Memory management

View

Create

```
Kokkos::View<DataType, LayoutType, MemorySpace, MemoryTraits>
↪  view("label", numberOfElementsAtRuntimeI,
↪  numberOfElementsAtRuntimeJ);
```

Template arg.	Description
`DataType`	`ScalarType` for the data type, followed by a `*` for each runtime dimension, then by a `[numberOfElements]` for each compile time dimension, mandatory
`LayoutType`	See memory layouts, optional
`MemorySpace`	See memory spaces, optional
`MemoryTraits`	See memory traits, optional

The order of template arguments is important.

Manage

Method	Description
`(i, j...)`	Returns and sets the value at index `i`, `j`, etc.
`size()`	Returns the total number of elements in the view
`rank()`	Returns the number of dimensions
`layout()`	Returns the layout of the view
`extent(dim)`	Returns the number of elements in the requested dimension
`data()`	Returns a pointer to the underlying data

Resize and preserve content

```
Kokkos::resize(view, newNumberOfElementsI, newNumberOfElementsJ...);
```

Reallocate and do not preserve content

```
Kokkos::realloc(view, newNumberOfElementsI, newNumberOfElementsJ...);
```

Memory Layouts

Layout	Description	Default
`Kokkos::LayoutRight`	Strides increase from the right most to the left most dimension, also known as row-major or C-like	CPU
`Kokkos::LayoutLeft`	Strides increase from the left most to the right most dimension, also known as column-major or Fortran-like	GPU
`Kokkos::LayoutStride`	Strides can be arbitrary for each dimension	

By default, a layout suited for loops on the high frequency index is used.

Memory trait

Memory traits are indicated with `Kokkos::MemoryTraits<>` and are combined with the `|` (pipe) operator.

Memory trait	Description
`Kokkos::Unmanaged`	The allocation has to be managed manually
`Kokkos::Atomic`	All accesses to the view are atomic
`Kokkos::RandomAccess`	Hint that the view is used in a random access manner; if the view is also `const` this may trigger more efficient load operations on GPUs
`Kokkos::Restrict`	There is no aliasing of the view by other data structures in the current scope

Deep copy

```
Kokkos::deep_copy(dest, src);
```

The views must have the same dimensions, data type, and reside in the same memory space (mirror views can be deep copied on different memory spaces).

Mirror view

Create and always allocate on host

```
auto mirrorView = Kokkos::create_mirror(view);
```

Create and allocate on host if source view is not in host space

```
auto mirrorView = Kokkos::create_mirror_view(view);
```

Create, allocate and synchronize if source view is not in same space as destination view

```
auto mirrorView = Kokkos::create_mirror_view_and_copy(ExecutionSpace(),
↪   view);
```

Subview

A subview has the same reference count as its parent view, so the parent view won't be deallocated before all subviews go away.

```
auto subview = Kokkos::subview(view, selector1, selector2, ...);
```

Subset selector	Description
`Kokkos::ALL`	All elements in this dimension
`Kokkos::pair(first, last)`	Range of elements in this dimension
`value`	Specific element in this dimension

Scatter view (experimental)

Specific header

```
#include <Kokkos_ScatterView.hpp>
```

Create

```
auto scatterView = Kokkos::Experimental::create_scatter_view<Operation,
↪   Duplication, Contribution>(targetView);
```

Template arg.	Description
`Operation`	See scatter operation; defaults to `Kokkos::Experimental::ScatterSum`
`Duplication`	Whether to duplicate the grid or not; choices are `Kokkos::Experimental::ScatterDuplicated`, and `Kokkos::Experimental::ScatterNonDuplicated`; defaults to the option that is the most optimised for `targetView`'s execution space
`Contribution`	Whether to contribute using atomics or not; choices are `Kokkos::Experimental::ScatterAtomic`, or `Kokkos::Experimental::ScatterNonAtomic`; defaults to the option that is the most optimised for `targetView`'s execution space

Scatter operation

Operation	Description
`Kokkos::Experimental::ScatterSum`	Sum
`Kokkos::Experimental::ScatterProd`	Product
`Kokkos::Experimental::ScatterMin`	Minimum value
`Kokkos::Experimental::ScatterMax`	Maximum value

Scatter, compute, and gather

```
Kokkos::parallel_for(
    "label",
    /* ... */,
    KOKKOS_LAMBDA (/* ... */) {
        // scatter
        auto scatterAccess = scatterView.access();

        // compute
        scatterAccess(/* index */) /* operation */ /* contribution */;
    }
);

// gather
Kokkos::Experimental::contribute(targetView, scatterView);
```

Parallel constructs

For loop

```
Kokkos::parallel_for(
    "label",
    ExecutionPolicy</* ... */>(/* ... */),
    KOKKOS_LAMBDA (/* ... */) { /* ... */ }
);
```

Reduction

```
ScalarType result;
Kokkos::parallel_reduce(
    "label",
    ExecutionPolicy</* ... */>(/* ... */),
    KOKKOS_LAMBDA (/* ... */, ScalarType& resultLocal) { /* ... */ },
    Kokkos::ReducerConcept<ScalarType>(result)
);
```

With `Kokkos::ReducerConcept` being one of the following:

Reducer	Operation	Description
`Kokkos::BAnd`	`&`	Binary and
`Kokkos::BOr`	`|`	Binary or
`Kokkos::LAnd`	`&&`	Logical and
`Kokkos::LOr`	`||`	Logical or
`Kokkos::Max`	`std::max`	Maximum
`Kokkos::MaxLoc`	`std::max_element`	Maximum and associated index
`Kokkos::Min`	`std::min`	Minimum
`Kokkos::MinLoc`	`std::min_element`	Minimum and associated index
`Kokkos::MinMax`	`std::minmax`	Minimum and maximum
`Kokkos::MinMaxLoc`	`std::minmax_element`	Minimum and maximum and associated indices
`Kokkos::Prod`	`*`	Product
`Kokkos::Sum`	`+`	Sum

A scalar value may be passed, for which the reduction is limited to a sum. When using the `TeamVectorMDRange`, the `TeamThreadMDRange`, or the `ThreadVectorMDRange` execution policy, only a scalar value may be passed, for which the reduction is also limited to a sum.

Fences

Global fence

```
Kokkos::fence("label");
```

Execution space fence

```
ExecutionSpace().fence("label");
```

Team barrier

```
Kokkos::TeamPolicy<>::member_type().team_barrier();
```

Execution policy

Create

```
ExecutionPolicy<ExecutionSpace, Schedule, IndexType, LaunchBounds,
↪  WorkTag> policy(/* ... */);
```

Template arg.	Description
`ExecutionSpace`	See execution spaces; defaults to `Kokkos::DefaultExecutionSpace`
`Schedule`	How to schedule work items; defaults to machine and backend specifics
`IndexType`	Integer type to be used for the index; defaults to `int64_t`
`LaunchBounds`	Hints for CUDA and HIP launch bounds
`WorkTag`	Empty tag class to call the functor

Ranges

One-dimensional range

```
Kokkos::RangePolicy<ExecutionSpace, Schedule, IndexType LaunchBounds,
↪  WorkTag> policy(first, last);
```

If the range starts at 0 and uses default parameters, can be replaced by just the number of elements.

Multi-dimensional (dimension 2)

```
Kokkos::MDRangePolicy<ExecutionSpace, Schedule, IndexType, LaunchBounds,
↪  WorkTag, Kokkos::Rank<2>> policy({firstI, firstJ}, {lastI, lastJ});
```

Hierarchical parallelism

Team policy

```
Kokkos::TeamPolicy<ExecutionSpace, Schedule, IndexType, LaunchBounds,
↪  WorkTag> policy(leagueSize, teamSize);
```

Usually, `teamSize` is replaced by `Kokkos::AUTO` to let Kokkos determine it. A kernel running in a team policy has a `Kokkos::TeamPolicy<>::member_type` argument:

Method	Description
`league_size()`	Number of teams in the league
`league_rank()`	Index of the team within the league
`team_size()`	Number of threads in the team
`team_rank()`	Index of the thread within the team

Note that nested parallel constructs do not use `KOKKOS_LAMBDA` to create lambdas. One must use the C++ syntax, for example `[=]` or `[&]`.

Team vector level (2-level hierarchy)

```
Kokkos::parallel_for(
    "label",
    Kokkos::TeamPolicy(numberOfElementsI, Kokkos::AUTO),
    KOKKOS_LAMBDA (const Kokkos::TeamPolicy<>::member_type& teamMember)
↪   {
        const int i = teamMember.team_rank();

        Kokkos::parallel_for(
            Kokkos::TeamVectorRange(teamMember, firstJ, lastJ),
            [=] (const int j) { /* ... */ }
        );
    }
);
```

One-dimensional range

```
Kokkos::TeamVectorRange range(teamMember, firstJ, lastJ);
```

Multi-dimensional range (dimension 2)

```
Kokkos::TeamVectorMDRange<Kokkos::Rank<2>,
↪   Kokkos::TeamPolicy<>::member_type> range(teamMember,
↪   numberOfElementsJ, numberOfElementsK);
```

Team thread vector level (3-level hierarchy)

```
Kokkos::parallel_for(
    "label",
    Kokkos::TeamPolicy(numberOfElementsI, Kokkos::AUTO),
    KOKKOS_LAMBDA (const Kokkos::TeamPolicy<>::member_type& teamMember)
↪   {
        const int i = teamMember.team_rank();
        Kokkos::parallel_for(
            Kokkos::TeamThreadRange(teamMember, firstJ, lastJ),
            [=] (const int j) {
                Kokkos::parallel_for(
                    Kokkos::ThreadVectorRange(teamMember, firstK,
↪   lastK),
                    [=] (const int k) { /* ... */ }
                );
            }
        );
    }
);
```

One-dimensional range

```
Kokkos::TeamThreadRange range(teamMember, firstJ, lastJ);
Kokkos::ThreadVectorRange range(teamMember, firstK, lastK);
```

Utilisation cheat sheet for Kokkos v4.6.1.20250516

Multi-dimensional range (dimension 2)

```
Kokkos::TeamThreadMDRange<Kokkos::Rank<2>,
↪   Kokkos::TeamPolicy<>::member_type> range(teamMember,
↪   numberOfElementsJ, numberOfElementsK);
Kokkos::ThreadVectorMDRange<Kokkos::Rank<2>,
↪   Kokkos::TeamPolicy<>::member_type> range(teamMember,
↪   numberOfElementsL, numberOfElementsM);
```

Scratch memory

Each team has access to a scratch memory pad, which has the team's lifetime, and is only accessible by the team's threads.

Scratch memory space

Space level	Memory size	Access speed
0	Limited (tens of kilobytes)	Fast
1	Larger (few gigabytes)	Medium

Used when passing the team policy to the parallel construct and when creating the scratch memory pad.

Create and populate

```
// Define a scratch memory pad type
using ScratchPad = Kokkos::View<DataType,
↪   Kokkos::DefaultExecutionSpace::scratch_memory_space,
↪   Kokkos::MemoryTraits<Kokkos::Unmanaged>>;

// Compute how much scratch memory is needed (in bytes)
size_t bytes = ScratchPad::shmem_size(vectorSize);

// Create the team policy and specify the total scratch memory needed
Kokkos::parallel_for(
    "label",
    Kokkos::TeamPolicy<>(leagueSize,
↪   teamSize).set_scratch_size(spaceLevel, Kokkos::PerTeam(bytes)),
    KOKKOS_LAMBDA (const Kokkos::TeamPolicy<>::member_type& teamMember)
↪   {
        const int i = teamMember.league_rank();

        // Create the scratch pad
        ScratchPad scratch(teamMember.team_scratch(spaceLevel),
↪   vectorSize);

        // Initialize it
        Kokkos::parallel_for(
            Kokkos::TeamVectorRange(teamMember, vectorSize),
            [=] (const int j) { scratch(j) = getScratchData(i, j); }
        );

        // Synchronize
        teamMember.team_barrier();
    }
);
```

Atomics

Atomic operations

Operation	Replaces
`Kokkos::atomic_add(&x, y)`	`x += y`
`Kokkos::atomic_and(&x, y)`	`x &= y`
`Kokkos::atomic_dec(&x)`	`x--`
`Kokkos::atomic_inc(&x)`	`x++`
`Kokkos::atomic_lshift(&x, y)`	`x = x << y`
`Kokkos::atomic_max(&x, y)`	`x = std::max(x, y)`
`Kokkos::atomic_min(&x, y)`	`x = std::min(x, y)`
`Kokkos::atomic_mod(&x, y)`	`x %= y`
`Kokkos::atomic_nand(&x, y)`	`x = !(x && y)`
`Kokkos::atomic_or(&x, y)`	`x |= y`
`Kokkos::atomic_rshift(&x, y)`	`x = x >> y`
`Kokkos::atomic_sub(&x, y)`	`x -= y`
`Kokkos::atomic_store(&x, y)`	`x = y`
`Kokkos::atomic_xor(&x, y)`	`x ^= y`

Atomic exchanges

Operation	Description
`Kokkos::atomic_exchange(&x, desired)`	Assign desired value to object and return old value
`Kokkos::atomic_compare_exchange(&x, expected, desired)`	Assign desired value to object if the object has the expected value and return the old value

Mathematics

Math functions

Function type	List of functions (prefixed by `Kokkos::`)
Basic ops.	`abs`, `fabs`, `fmod`, `remainder`, `fma`, `fmax`, `fmin`, `fdim`, `nan`
Exponential	`exp`, `exp2`, `expm1`, `log`, `log2`, `log10`, `log1p`
Power	`pow`, `sqrt`, `cbrt`, `hypot`
Trigonometric	`sin`, `cos`, `tan`, `asin`, `acos`, `atan`, `atan2`
Hyperbolic	`sinh`, `cosh`, `tanh`, `asinh`, `acosh`, `atanh`
Error, gamma	`erf`, `erfc`, `tgamma`, `lgamma`
Nearest	`ceil`, `floor`, `trunc`, `round`, `nearbyint`
Floating point	`logb`, `nextafter`, `copysign`
Comparisons	`isfinite`, `isinf`, `isnan`, `signbit`

Note that not all C++ standard math functions are available.

Complex numbers

Create

```
Kokkos::complex<double> complex(realPart, imagPart);
```

Manage

Method	Description
`real()`	Returns or sets the real part
`imag()`	Returns or sets the imaginary part

Utilities

Code interruption

```
Kokkos::abort("message");
```

Utilisation cheat sheet for Kokkos :: page 5

Print inside a kernel

```
Kokkos::printf("format string", arg1, arg2);
```

Similar to `std::printf`.

Timer

Create

```
Kokkos::Timer timer;
```

Manage

Method	Description
`seconds()`	Returns the time in seconds since construction or last reset
`reset()`	Resets the timer to zero

Manage parallel environment

Function	Description
`Kokkos::device_id()`	Returns the device ID of the current device
`Kokkos::num_devices()`	Returns the number of devices available to the current execution space

Macros

Essential macros

Macro	Description
`KOKKOS_LAMBDA`	Replaces capture argument for lambdas
`KOKKOS_CLASS_LAMBDA`	Replaces capture argument for lambdas, captures `this`
`KOKKOS_FUNCTION`	Functor attribute
`KOKKOS_INLINE_FUNCTION`	Inlined functor attribute

Extra macros

Macro	Description
`KOKKOS_VERSION`	Kokkos full version
`KOKKOS_VERSION_MAJOR`	Kokkos major version
`KOKKOS_VERSION_MINOR`	Kokkos minor version
`KOKKOS_VERSION_PATCH`	Kokkos patch level
`KOKKOS_ENABLE_*`	Any equivalent CMake option passed when building Kokkos, see installation cheat sheet
`KOKKOS_ARCH_*`	Any equivalent CMake option passed when building Kokkos, see installation cheat sheet