



# PARALLEL PROGRAMMING...

By Patrick Lemoine 2025.

*Kokkos*

# Κόκκος

SESSION 2

## Kokkos?

### Basic Concepts

- Views,
- Memory spaces,
- Memory access patterns,
- Mirrors

### Advanced Concepts

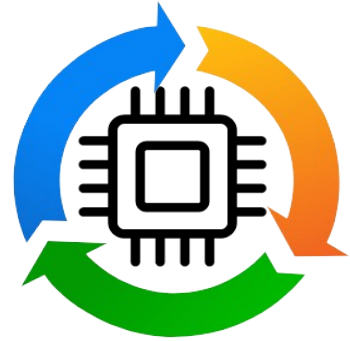
- Asynchronicity, Streams and Task Parallelism
- Multi-level hierarchical parallelism
- Multidimensional Loops and Data Structures
- Advanced Reductions
- MPI (Message Passing Interface) and PGAS (Partitioned Global Address Spaces)
- SIM (Single Instruction Multiple Data) Task Flow

**GOAL**





**A**synchronicity, **S**treams  
**P**arallelism



# Kokkos: Asynchronicity, streams and parallelism of tasks

## Asynchronicity

- Allows non-blocking execution of tasks or kernels.
- Allows overlapping of computations and communications for better resource utilization.

## Stream

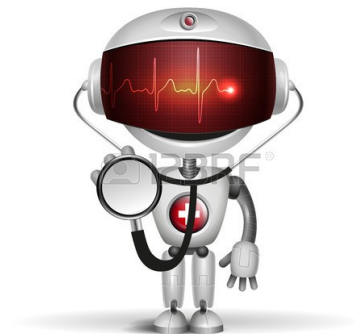
- Independent execution queues for kernel scheduling.
- Useful for running unrelated tasks simultaneously on GPUs or CPUs.

## Task Parallelism

- Manages tasks with dependencies using a task DAG (Directed Acyclic Graph).
- Tasks are scheduled dynamically based on their dependencies.
- Main features:
  - `TaskPolicy`: Defines the allocation and scheduling of tasks.
  - `Future`: represents the result of a task, allowing the management of dependencies.

## Benefits

- Performance: overlap execution to maximize hardware usage.



# Kokkos: Asynchronicity

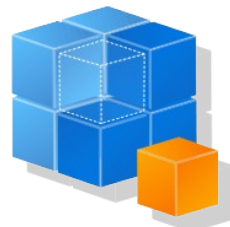
To perform an asynchronous operation:

```
Kokkos::parallel_for("AsyncExample", N, KOKKOS_LAMBDA(const int i) {  
  // Parallel computing  
  data(i) = compute_value(i);  
});
```

```
// Explicit synchronization
```

```
Kokkos::fence(); // Wait for all previous operations to complete
```

The call to `Kokkos::fence` ensures that all previous asynchronous operations are completed before continuing.



# Kokkos: Streams

To organize multiple independent operation streams, one can use specific execution spaces with their own queues. For example, with CUDA the same thing for HIP:

```
Kokkos::Cuda instance1;  
Kokkos::Cuda instance2;
```

```
Kokkos::parallel_for("Stream1", Kokkos::RangePolicy<Kokkos::Cuda>(instance1, 0, N), KOKKOS_LAMBDA(const int i) {  
    // Calculation on flow 1  
    data1(i) = compute_value1(i);  
});
```

```
Kokkos::parallel_for("Stream2", Kokkos::RangePolicy<Kokkos::Cuda>(instance2, 0, N), KOKKOS_LAMBDA(const int i) {  
    // Calculation on flow 2  
    data2(i) = compute_value2(i);  
});
```

```
// Explicit synchronization for each stream  
instance1.fence();  
instance2.fence();
```

*Here, two independent streams (`instance1` and `instance2`) allow the simultaneous execution of two sets of parallel operations.*

# Kokkos: Task parallelism

Task Parallelism in Kokkos is managed through the task system.

## *// Initialize the task pool*

```
auto task_policy=Kokkos::TaskScheduler<Kokkos::DefaultExecutionSpace>(1024 * 1024);
```

## *// Definition of a simple task*

```
auto task=Kokkos::host_spawn(Kokkos::TaskSingle(task_policy), []() {  
    printf("Task executed\n");  
});
```

## *// Scheduling a dependent task*

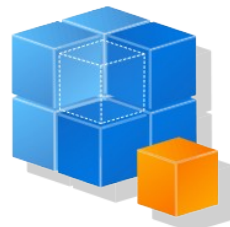
```
auto dependent_task=Kokkos::host_spawn(Kokkos::TaskSingle(task_policy).depends_on(task), []() {  
    printf("Dependent task executed\n");  
});
```

## *// Execution of scheduled tasks*

```
Kokkos::wait(task_policy);
```

In this example:

- A task is defined and scheduled with `host_spawn`.
- A second task is created with an explicit dependency on the first.
- `Kokkos::wait` is used to wait until all tasks are completed



# Kokkos: Combination of the three concepts

```
Kokkos::Cuda instance;  
auto task_policy=Kokkos::TaskScheduler<Kokkos::Cuda>(1024 * 1024);  
  
auto task1=Kokkos::host_spawn(Kokkos::TaskSingle(task_policy), []() {  
    printf("Task 1 on GPU\n");  
});  
  
auto task2=Kokkos::host_spawn(Kokkos::TaskSingle(task_policy).depends_on(task1), []() {  
    printf("Task 2 dependent on GPU\n");  
});  
  
Kokkos::parallel_for("AsyncParallel", Kokkos::RangePolicy<Kokkos::Cuda>(instance, 0, N),  
KOKKOS_LAMBDA(const int i) {  
    // Independent parallel computing  
    data(i) = compute_value(i);  
});  
  
// Global synchronization  
instance.fence();  
Kokkos::wait(task_policy);
```

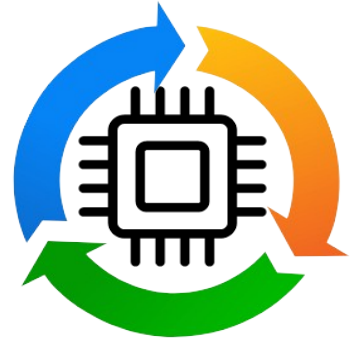
This code shows how to simultaneously use asynchronicity (via `fence`), streams (via `instance`) and Task Parallelism (via `TaskScheduler`) to efficiently coordinate parallel computations.







**Hierarchical Parallelism**



# Kokkos' hierarchical parallelism

Kokkos uses the **hierarchical parallelism** to exploit different levels of parallelism, which naturally adapts to various architectures like multi-core CPUs and GPUs.

## Levels of parallelism:

- League: Higher level, often associated with compute nodes or CPU sockets.
- Team: Corresponds to physical cores or GPU multiprocessors.
- Vector: SIMD parallelism at the instruction level.

**ThreadVectorRange** :Level of *additional parallelism* within a *thread* which exploits the vector capabilities of the hardware.

## Scratch Memory:

- Level 0: Memory private to a thread and persistent between kernel calls.
- Level 1: Memory shared by a team, but private between teams.

**Memory hierarchy**:Hierarchical parallelism efficiently exploits the memory hierarchy, including registers, L1/L2 caches, and GPU shared memory.

# Kokkos' hierarchical parallelism

## Using Thread Teams

```
Kokkos::parallel_for(Kokkos::TeamPolicy<>(N, Kokkos::AUTO),  
KOKKOS_LAMBDA(const Kokkos::TeamPolicy<>::member_type&team) {  
    const int league_rank = team.league_rank();  
    const int team_size = team.team_size();  
    const int team_rank = team.team_rank();
```

```
    // Code using league_rank, team_size and team_rank  
});
```

## Nested parallelism

```
Kokkos::parallel_for(Kokkos::TeamPolicy<>(N, Kokkos::AUTO),  
KOKKOS_LAMBDA(const Kokkos::TeamPolicy<>::member_type&team) {  
    const int league_rank = team.league_rank();
```

### *// Team-level parallelism*

```
Kokkos::parallel_for(Kokkos::TeamThreadRange(team, M), [&](const int& i) {  
    // Code executed in parallel by team threads  
});
```

### *// Parallelism at the vector level*

```
Kokkos::parallel_for(Kokkos::ThreadVectorRange(team, K), [&](const int& j) {  
    // Code executed in parallel by vector lanes  
});  
});
```

# Kokkos' hierarchical parallelism

## Configuring an advanced hierarchical execution policy:

```
int league_size = 1000;  
int team_size = 16;  
int vector_length = 4;
```

```
Kokkos::TeamPolicy<>policy(league_size, team_size, vector_length);  
policy.set_scratch_size(1, Kokkos::PerTeam(1024), Kokkos::PerThread(256));
```

## Combined use of TeamThreadRange and ThreadVectorRange:

```
Kokkos::parallel_for(policy, KOKKOS_LAMBDA(const member_type&team_member) {  
    const int league_rank =team_member.league_rank();  
    const int team_size =team_member.team_size();  
    const int team_rank =team_member.team_rank();
```

```
    Kokkos::parallel_for(Kokkos::TeamThreadRange(team_member, work_per_team), [&](const int& i) {  
        Kokkos::parallel_for(Kokkos::ThreadVectorRange(team_member, vector_work), [&](const int& j) {  
            // Vector work  
        });  
    });  
});
```



# Kokkos' hierarchical parallelism

## Advanced use of scratch memory:

```
Kokkos::parallel_for(policy, KOKKOS_LAMBDA(const member_type&team_member) {  
    Kokkos::View<double*, Kokkos::MemoryUnmanaged>scratch_team(team_member.team_scratch(1), team_size);  
    Kokkos::View<int*, Kokkos::MemoryUnmanaged>scratch_thread(team_member.thread_scratch(0), vector_length);
```

```
// Using scratch memory
```

```
scratch_team(team_member.team_rank()) = compute_value();  
team_member.team_barrier();
```

```
for (int i = 0; i < vector_length; ++i) {  
    scratch_thread(i) = process_data(scratch_team(team_member.team_rank()));  
}  
});
```

- Shared memory space accessible only by threads of the same team
- Lifespan limited to that of the team

### Key Features

- Two levels:
  - Level 0: Small size (tens of KB), quick access
  - Level 1: Larger size (a few GB), average access
- Use with TeamPolicy for private memory per thread or team

### Benefits

- Performance: Faster access than global memory
- Optimization: Acts like a manually managed cache
- Flexibility: Temporary storage by work item

### Use

- Set the required size before launching the kernel
- Create views in scratch memory inside kernels
- Possible synchronization between threads of a team



# Kokkos' hierarchical parallelism

## Hierarchical reduction:

```
double total_sum = 0.0;
Kokkos::parallel_reduce(policy, KOKKOS_LAMBDA(const member_type& team_member, double& team_sum) {
    double thread_sum = 0.0;
    Kokkos::parallel_reduce(Kokkos::TeamThreadRange(team_member, work_per_team),
        [&](const int& i, double& sum) {
            Kokkos::parallel_reduce(Kokkos::ThreadVectorRange(team_member, vector_work),
                [&](const int& j, double& inner_sum) {
                    inner_sum += compute_value(i, j);
                }, sum);
            }, thread_sum);
    team_sum += thread_sum;
}, total_sum);
```



# Kokkos' hierarchical parallelism

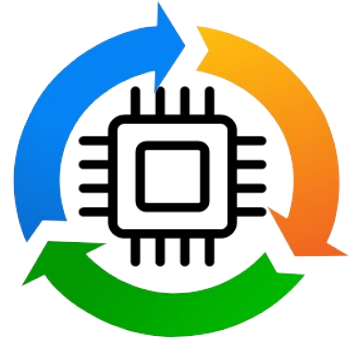
Using `SingleThreadTeam` for sequential operations within a team:

```
Kokkos::parallel_for(policy, KOKKOS_LAMBDA(const member_type& team_member) {  
  Kokkos::single(Kokkos::PerTeam(team_member), [&]() {  
    // Operation executed only once per team  
  });  
  
  // ... Other parallel work ...  
  
  Kokkos::single(Kokkos::PerThread(team_member), [&]() {  
    // Operation executed once per thread  
  });  
});
```





**Multidimensional Loops**  
**Data Structures**





# Kokkos Multidimensional Loops and Data Structures

**MDRangePolicy (Multidimensional Loop Policy):**MDRangePolicy is a feature of Kokkos that allows to parallelize tightly nested loops of 2 to 6 dimensions.

**Views (Multidimensional Views):**Views in Kokkos are multidimensional data containers that adapt to the hardware.

**Layouts (Memory layouts):**Layouts define how multidimensional data is organized in memory. Kokkos mainly offers two types of layouts:

- LayoutLeft: column-major order, optimal for some GPU types
- LayoutRight: row-major order, often preferred for CPUs

**Subviews:**Allows you to create partial views of an existing view without copying data.

# Kokkos Multidimensional Loops and Data Structures

## Advanced usage of MDRangePolicy:

### *// Defining an MDRange policy with tile size*

```
using MDPolicyType = Kokkos::MDRangePolicy<Kokkos::Rank<3>, Kokkos::IndexType<int64_t>>;  
MDPolicyType mdPolicy({0,0,0}, {N0,N1,N2}, {16,16,4});
```

### *// Parallelization with reduction*

```
double sum = 0.0;  
Kokkos::parallel_reduce("MDLoop", mdPolicy,  
    KOKKOS_LAMBDA(const int i, const int j, const int k, double & lsum) {  
        lsum += compute_value(i,j,k);  
    }, sum  
);
```



# Kokkos Multidimensional Loops and Data Structures

Advanced creation and manipulation of multidimensional Views:

*// Creation of a 4D view with specific layout*

```
Kokkos::View<double****, Kokkos::LayoutRight, Kokkos::CudaSpace> data4D("data4D", N0, N1, N2, N3);
```

*// Create a mirror view on the host*

```
cardata4D_host = Kokkos::create_mirror_view(data4D);
```

*// Populating the view on the host*

```
Kokkos::parallel_for("FillHost", Kokkos::MDRangePolicy<Kokkos::Rank<4>>({0,0,0,0}, {N0,N1,N2,N3}),  
  KOKKOS_LAMBDA(int i, int j, int k, int l) {  
    data4D_host(i,j,k,l) = compute_initial_value(i,j,k,l);  
  }  
);
```

*// Copy data to the device*

```
Kokkos::deep_copy(data4D, data4D_host);
```

*// Processing on the device*

```
Kokkos::parallel_for("ProcessDevice", Kokkos::MDRangePolicy<Kokkos::Rank<4>>({0,0,0,0},  
  {N0,N1,N2,N3}),  
  KOKKOS_LAMBDA(int i, int j, int k, int l) {  
    data4D(i,j,k,l) = process_data(data4D(i,j,k,l));  
  }  
);
```



# Kokkos Multidimensional Loops and Data Structures

Advanced use of layouts and subviews:

## *// Creating a view with LayoutStride*

```
Kokkos::View<double***, Kokkos::LayoutStride> strided_view("strided",  
  Kokkos::LayoutStride::padding({N0,N1,N2}, {N1*N2, N2, 2}));
```

## *// Creating a subview*

```
carsub_view= Kokkos::subview(data4D, Kokkos::ALL, Kokkos::ALL, 5, Kokkos::range(10,20));
```

## *// Using the subview*

```
Kokkos::parallel_for("SubviewProcess", Kokkos::MDRangePolicy<Kokkos::Rank<3>>({0,0,0}, {N0,N1,10}),  
  KOKKOS_LAMBDA(int i, int j, int k) {  
    sub_view(i,j,k) *= 2.0;  
  }  
);
```



# Kokkos Multidimensional Loops and Data Structures

## Performance optimization:

*// Using the Unmanaged attribute to avoid memory allocation*

```
double* raw_ptr = /* ... */;  
Kokkos::View<double**, Kokkos::LayoutRight, Kokkos::HostSpace, Kokkos::MemoryTraits<Kokkos::Unmanaged>>  
  unmanaged_view(raw_ptr, N0, N1);
```

*// Using the RandomAccess attribute to optimize random accesses*

```
Kokkos::View<double**, Kokkos::LayoutRight, Kokkos::DefaultExecutionSpace::memory_space,  
  Kokkos::MemoryTraits<Kokkos::RandomAccess>>random_access_view("random", N0, N1);
```

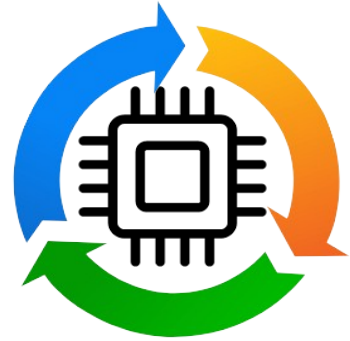
*// Parallelization with optimized access*

```
Kokkos::parallel_for("OptimizedAccess", Kokkos::RangePolicy<>(0, N),  
  KOKKOS_LAMBDA(const int i) {  
    int j = compute_index(i);  
    double value =random_access_view(i, j);  
    // Treatment...  
  }  
);
```





**Advanced Reduction**



# Kokkos Advanced Reductions

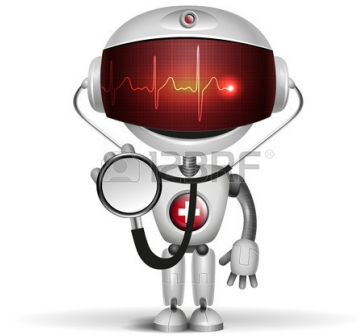
**Parallel reductions:** Combining individual calculation results into a single value.

**Main function:** Using `Kokkos::parallel_reduce` to consolidate data across multiple processing units.

**Key concept of the "reducer":**

- Logic of combining intermediate values.
- Definition of the merge operation.
- Initialization of private variables per thread.
- Management of the localization of the final result.

**Benefits:** Provides a powerful framework for complex reduction operations, optimizing performance in high-performance computing applications.



# Kokkos Advanced Reductions

The "**Reducers**" are classes which:

- Provide information to combine (reduce) two values.
- 
- Know how to initialize private variables for each thread.
- 
- Determine where to store the final result of the reduction.



# Kokkos Advanced Reductions

Kokkos provides **Reducers** for the most common types of reduction, such as:

- BAnd: Binary reduction "and"
- BOr: Binary reduction "or"
- LAnd: Logical reduction "and"
- LOr: Logical reduction "or"
- Max: Find the maximum value
- MaxLoc: Retrieve the maximum value and its associated index
- Min: Find the minimum value
- MinLoc: Retrieve the minimum value and its associated index
- MinMax: Find the minimum and maximum values
- MinMaxLoc: Find the maximum and minimum values and their associated indices
- Prod: Calculate the product of all input values
- Sum: For simple summations

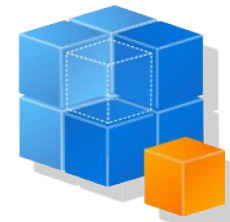
# Kokkos Advanced Reductions

Using **Built-in Reducers** to use a built-in Reducer:

```
double min;
Kokkos::parallel_reduce("MinReduce", N, KOKKOS_LAMBDA (const int& x, double&
lmin) {
    double val = (1.0*x - 7.2) * (1.0*x - 7.2) + 3.5;
    if (val < lmin) lmin = val;
}, Kokkos::Min<double>(min));
printf("Min: %lf\n", min);
```

**Multiple discounts:** Kokkos allows you to perform multiple reductions at the same time.  
For example, to calculate both the maximum value and the sum of the entries in a View:

```
double max_val, sum_val;
Kokkos::parallel_reduce("MultiReduce", N, KOKKOS_LAMBDA (const int& i, double& lmax, double&
lsum) {
    double val = a(i);
    if (val > lmax) lmax = val;
    lsum += val;
}, Kokkos::Max<double>(max_val), Kokkos::Sum<double>(sum_val)) ;
```

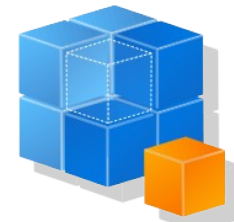


# Kokkos Advanced Reductions

## Using Reducers with localization

For the `MinLoc`, `MaxLoc`, and `MinMaxLoc` Reducers, the reduction type is a complex scalar type that contains `val` and `loc` members to store the reduction value and index, respectively. Here is an example:

```
typedef Kokkos::MinLoc<double,int>::value_type minloc_type;
minloc_type minloc;
Kokkos::parallel_reduce("MinLocReduce", N, KOKKOS_LAMBDA (const int& x, minloc_type& lminloc) {
    double val = (1.0*x - 7.2) * (1.0*x - 7.2) + 3.5;
    if (val < lminloc.val) {
        lminloc.val = val;
        lminloc.loc = x;
    }
}, Kokkos::MinLoc<double,int>(minloc));
printf("Min: %lf at %i\n", minloc.val, minloc.loc);
```



# Kokkos Personalized Reductions

For completely arbitrary reductions, you must provide an implementation of a Reducer class.

```
template<class Scalar>
Structure CustomReducer{
    typedef CustomReducer reducer;
    typedef Scalar value_type;
    Scalar& value;

    KOKKOS_INLINE_FUNCTION
    CustomReducer(Scalar& value_) : value(value_) {}

    KOKKOS_INLINE_FUNCTION
    void join(Scalar& dest, const Scalar& src) const {
        // Define your custom reduction operation here
        dest = /* your operation */;
    }

    KOKKOS_INLINE_FUNCTION
    void init(Scalar& val) const {
        // Initialize the reduction value
        val = /* initial value */;
    }
};
```

```
// Use
Scalar result;
Kokkos::parallel_reduce("CustomReduce", N, KOKKOS_LAMBDA
(const int& i, Scalar& update) {
    // Your discount code here
}, CustomReducer<Scalar>(result));
```

This approach allows you to define completely arbitrary reduction operations tailored to your specific needs.





Thank you for your attention!

