SC22
Dallas, TX | hpc accelerates.

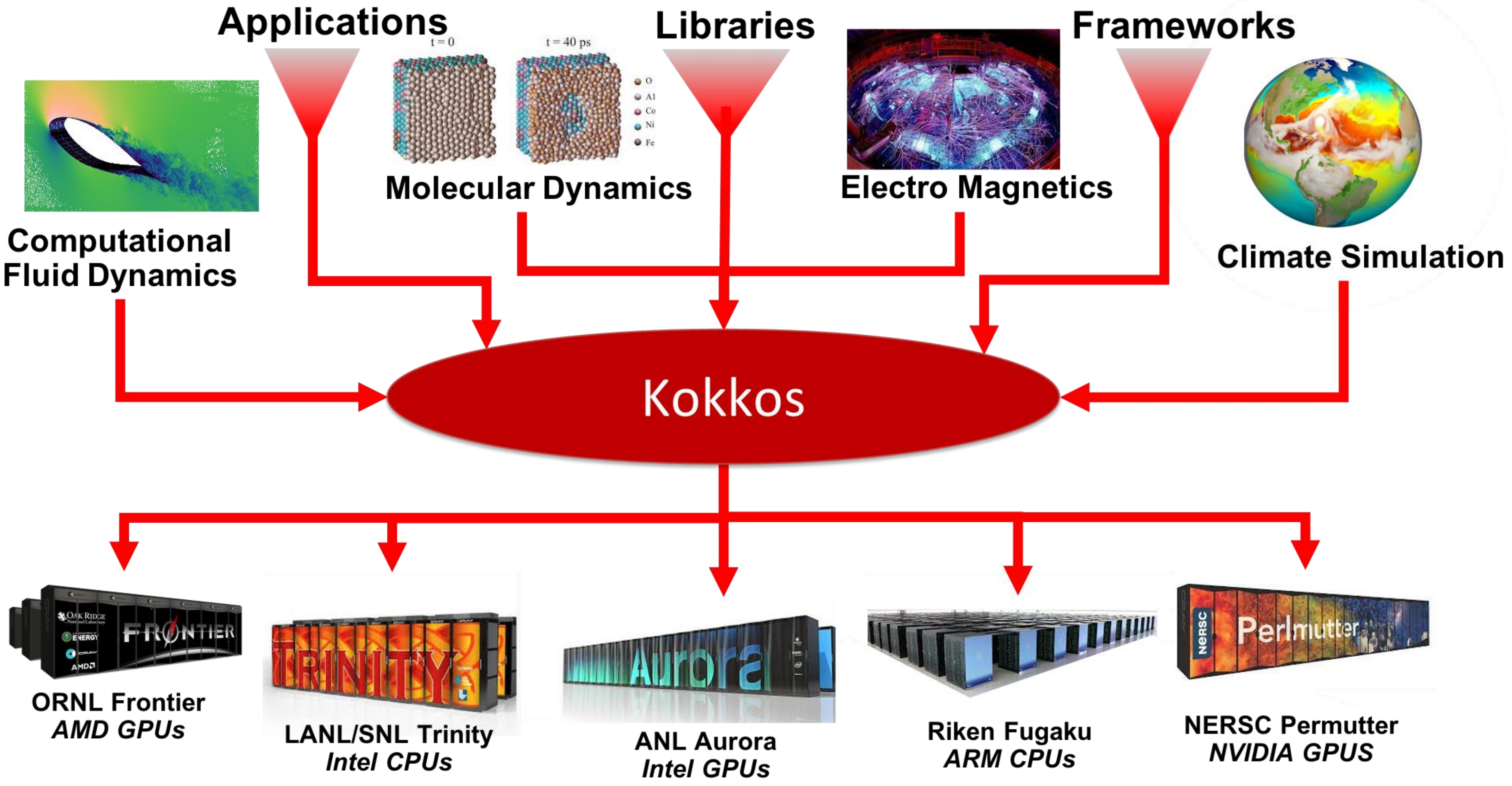
Paving the way to exascale for scientific applications

Damien Lebrun-Grandié

Oak Ridge National Laboratory

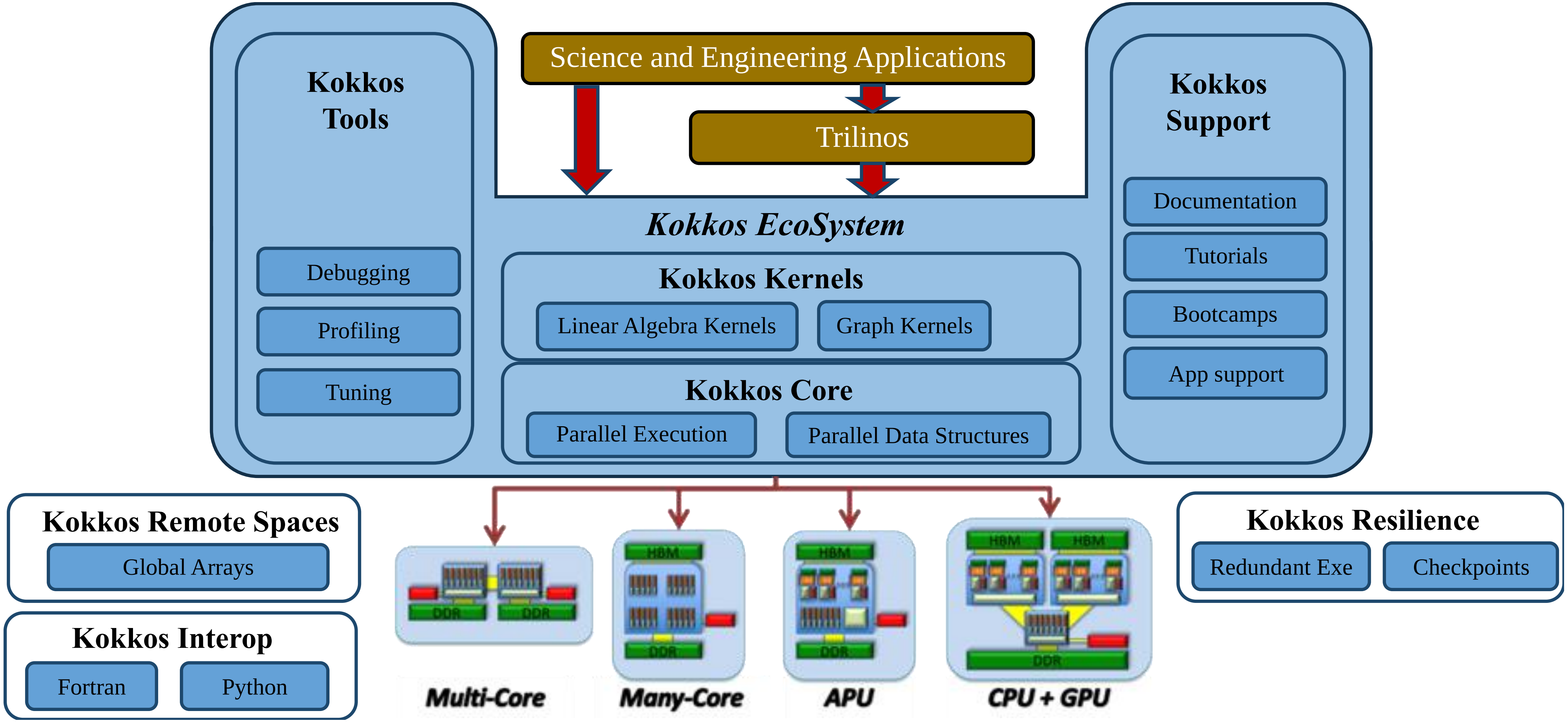
Content

- Brief intro on the challenges from heterogeneous computing
- STL-like parallel algorithms
- Synchronization primitives (atomics)


Applications
Libraries
Frameworks
Computational Fluid Dynamics
t = 0
t = 40 ps
O
Al
Co
Ni
Fe
Molecular Dynamics
Electro Magnetics
Climate Simulation
Kokkos
ORNL Frontier
AMD GPUs
LANL/SNL Trinity
Intel CPUs
ANL Aurora
Intel GPUs
Riken Fugaku
ARM CPUs
NERSC Permutter
NVIDIA GPUS

2

The Kokkos ecosystem



The Kokkos team

Sandia National Laboratories

Oak Ridge National Laboratory

Argonne National Laboratory

Los Alamos National Laboratory EST. 1943

Berkeley Lab

NGA

CSCS

Parallel algorithms

Standard C++ parallel algorithms

- Overloads that accept execution policies
  - Implementations may define additional exec policies as an extension
  - It is programmer's responsibility to avoid data races and deadlocks

std::execution::seq (C++17)
std::execution::par (C++17)
std::execution::par_unseq (C++17)
std::execution::unseq (C++20)

```
std::for_each_n(
    std::execution::par,
    begin(v), n,
    [](auto& val){ val *= 2; });
```

```
auto squared_sum =
std::transform_reduce(
    std::execution::par_unseq,
    cbegin(v), cend(v), 0L,
    std::plus{},
    [](auto val) { return val * val; });
```

```
std::sort(policy, begin(v), end(v));
```

Parallel algorithms available today

- CUDA/ROCm Thrust library
  - Allow programmers to nest their algorithm calls within functors
  - Some support for async algos (copy, for_each, reduce, scan, sort, transform)
- NVIDIA HPC SDK compiler with `-stdpar`
  - CUDA Unified Memory
  - Other restrictions and limitations (e.g. `__device__` annotations, random-access iterators, interoperability with `std::atomic`)
- Intel oneAPI DPC++ library

```
std::fill(oneapi::dpl::execution::make_device_policy(queue), begin(v), end(v), 42);
```

  - Supporting a number of _async algorithms

Kokkos example: exclusive prefix sum

Native construct

```
parallel_scan(
    RangePolicy(exec, 0, n),
    KOKKOS_LAMBDA(int i,
                                  long long int& partial_sum,
                                  bool is_final) {
      auto const v_i = v(i);
      if (is_final) v(i) = partial_sum;
      partial_sum += v_i;
    });
```

Using STL-like algorithms

```
exclusive_scan(
    exec,
    cbegin(v), cend(v), begin(v), 0ll);
```

Example: dot product

Native construct

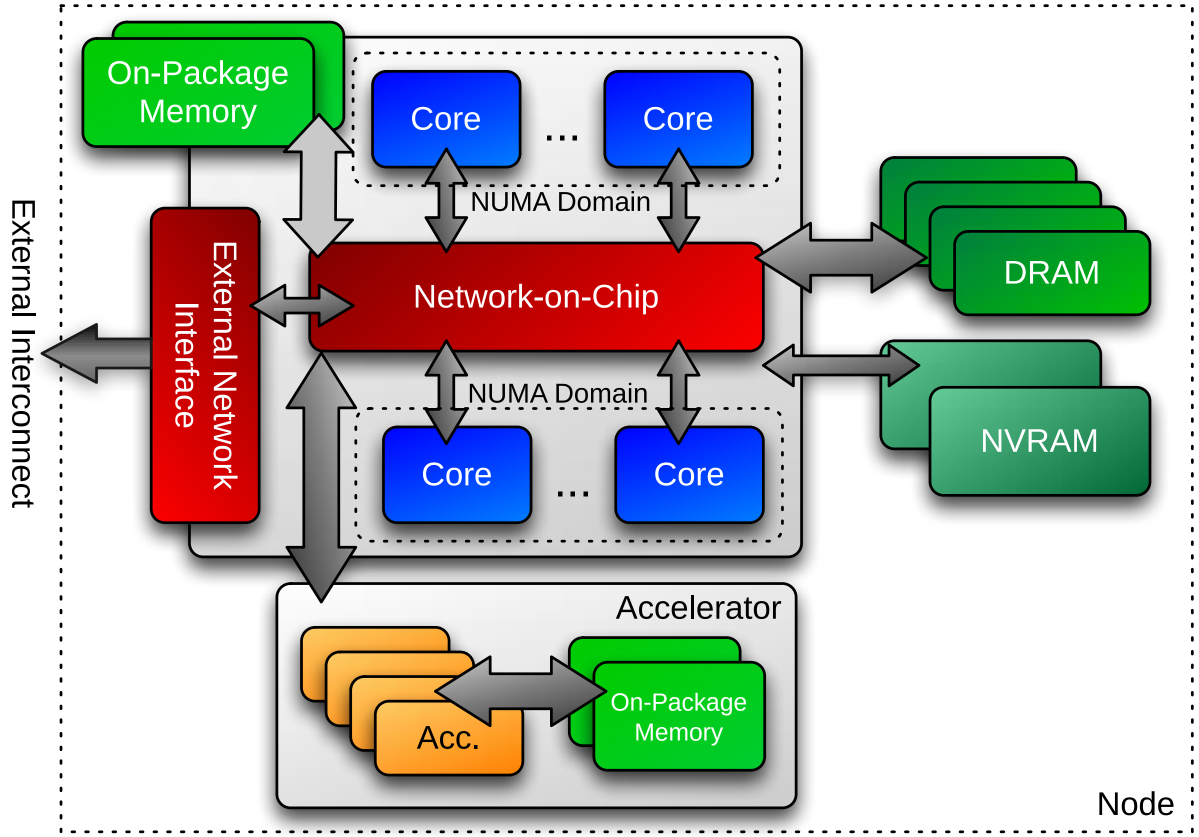
```
float sum;
parallel_reduce(
  RangePolicy(exec, 0, n),
  KOKKOS_LAMBDA(int i, float &partial_sum) {
    partial_sum += v(i) * w(i);

  }, sum);
```

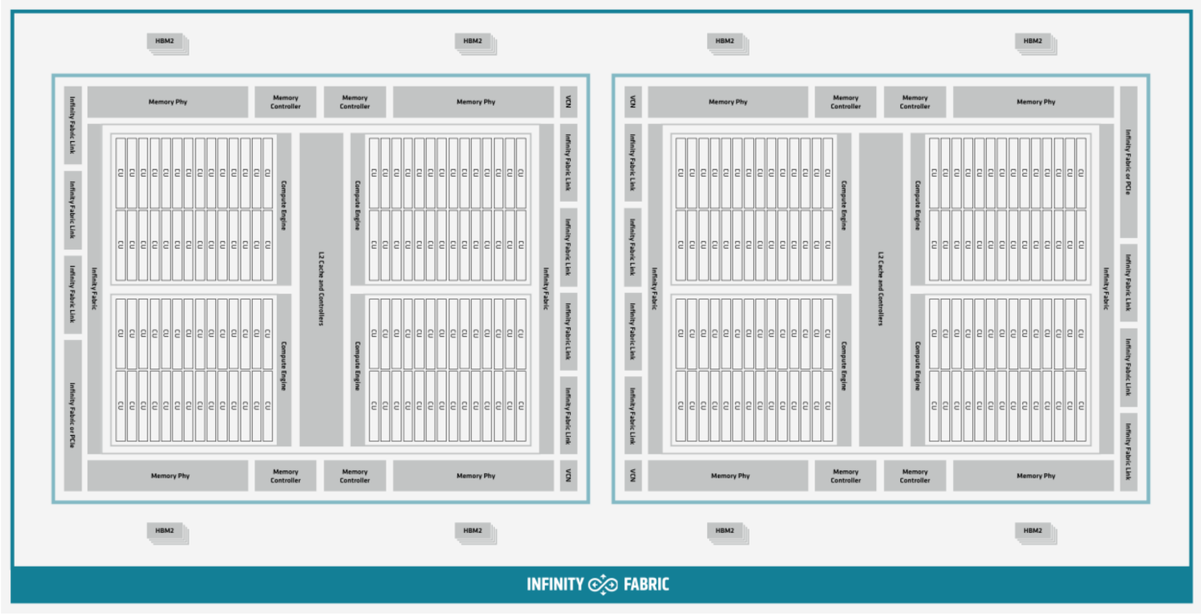
Using STL-like algorithms

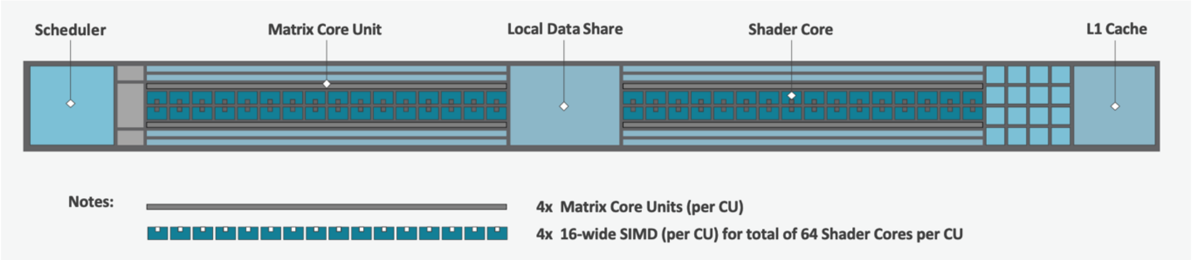
```
auto sum = transform_reduce(
  exec,
  cbegin(v), cend(v), cbegin(w), 0.f,
  Plus{},
  KOKKOS_LAMBDA(float v_i, float w_i) {
    return v_i * w_i;
  });
```

Target machine



Frontier compute node





[1x] 64-core AMD "Optimized 3rd Gen EPYC" CPU
[4x] MI250x each with 2 GCDs
Each GCD contains 110 CUs
64 GB of HBM accessible at 1.6 TB/s

4 CEs which dispatch wavefronts to CUs
All wavefronts from a single workgroup are assigned to the same CU

Work items in a wavefront are scheduled in units of 64 called wavefronts
Up to 64 KB of LDS can be allocated

Each CU has 4 MCUs and 4 16-wide SIMD units
Each wavefront is assigned to a single 16-wide SIMD unit

Each CU maintains an instruction buffer for 10 wavefronts

Expose more parallelism

- 8 logical GPUs
  each with 110 CUs x 2560 threads = 281600 threads

```
for (int i = 0; i < num_rows, ++i) {
  auto A_i = subview(A, i, ALL);
  double y_i = 0.;
  for (int j = 0; j < num_columns, ++j) {
    y_i += A_i(j) * x(j);
  }
  y(i) = y_i;
}
```

- Not enough parallelism exposed from a single level (outer loop)
  - parallelize inner loops!

Hierarchical parallelism in Kokkos

- Exploit multiple level of shared-memory parallelism
  These levels include thread teams, threads within a team, and vector lanes.
- Able to nest these levels of parallelism, and execute parallel_for(), parallel_reduce(), or parallel_scan() at each level
- Syntax differs only by the execution policy which is the 1st argument to the parallel_*
- Also exposing a “scratch pad” memory which provides thread private and team private allocations

Matrix-vector multiplication (1/2)

Native construct

```
parallel_for(
   TeamPolicy(exec, num_rows, AUTO),
   KOKKOS_LAMBDA(
            TeamHandle const &team_handle) {
     int const i = team_handle.league_rank();
     double y_i;
     parallel_reduce(
       TeamThreadRange(team_handle,
                                num_cols),
       [&](int j, double &lsum) {
              lsum += A(i, j) * x(j);
       }, y_i);
     y(i) = y_i;
   });
```

Mixing native construct and STL-algo

```
parallel_for(
   TeamPolicy(exec, numRows, AUTO),
   KOKKOS_LAMBDA(
            TeamHandle const &team_handle) {
     int const i = team_handle.league_rank();
     auto A_i = subiew(A, i, ALL);
     y(i) = transform_reduce(
        team_handle,
        cbegin(A_i), cend(A_i), cbegin(x), 0.,
        Plus{},
        [](double A_ij, float x_j) {
         return A_ij * x_ij;
     });
   });
```

Matrix-vector multiplication (2/2)

Mixing native construct and STL-algo

```
parallel_for(
  TeamPolicy(exec, numRows, AUTO),
  KOKKOS_LAMBDA(
        TeamHandle const &team_handle) {
   int const i = team_handle.league_rank();
   auto A_i = subview(A, i, ALL);
   y(i) = transform_reduce(
     team_handle,
     cbegin(A_i), cend(A_i), cbegin(x), 0.,
     Plus<double>(),
     [](double A_ij, float x_j) {
      return A_ij * x_ij;
   });
  });
```

Using STL-like algorithms only

```
for_each(
  exec,
  CountingIterator(0), CountingIterator(n),
  KOKKOS_LAMBDA(int i) {
   auto A_i = subiew(A, i, ALL);
    y(i) = transform_reduce(
      ???, // falling back to serial
      cbegin(A_i), cend(A_i), cbegin(x), 0.,
      Plus{},
      [](double A_ij, float x_j) {
       return A_ij * x_ij;
    });
  });
```

Looking forward

- Have a solution in Kokkos but trying to figure out how this can be done with std algorithms
- Open questions
  - Not sure what to do with algorithms that allocate memory (shift_left, shift_right, rotate)
- Proposed for C++26
  - Adding basic linear algebra support to C++: std::linalg
  - Taking std::mdspan (C++23) as argument

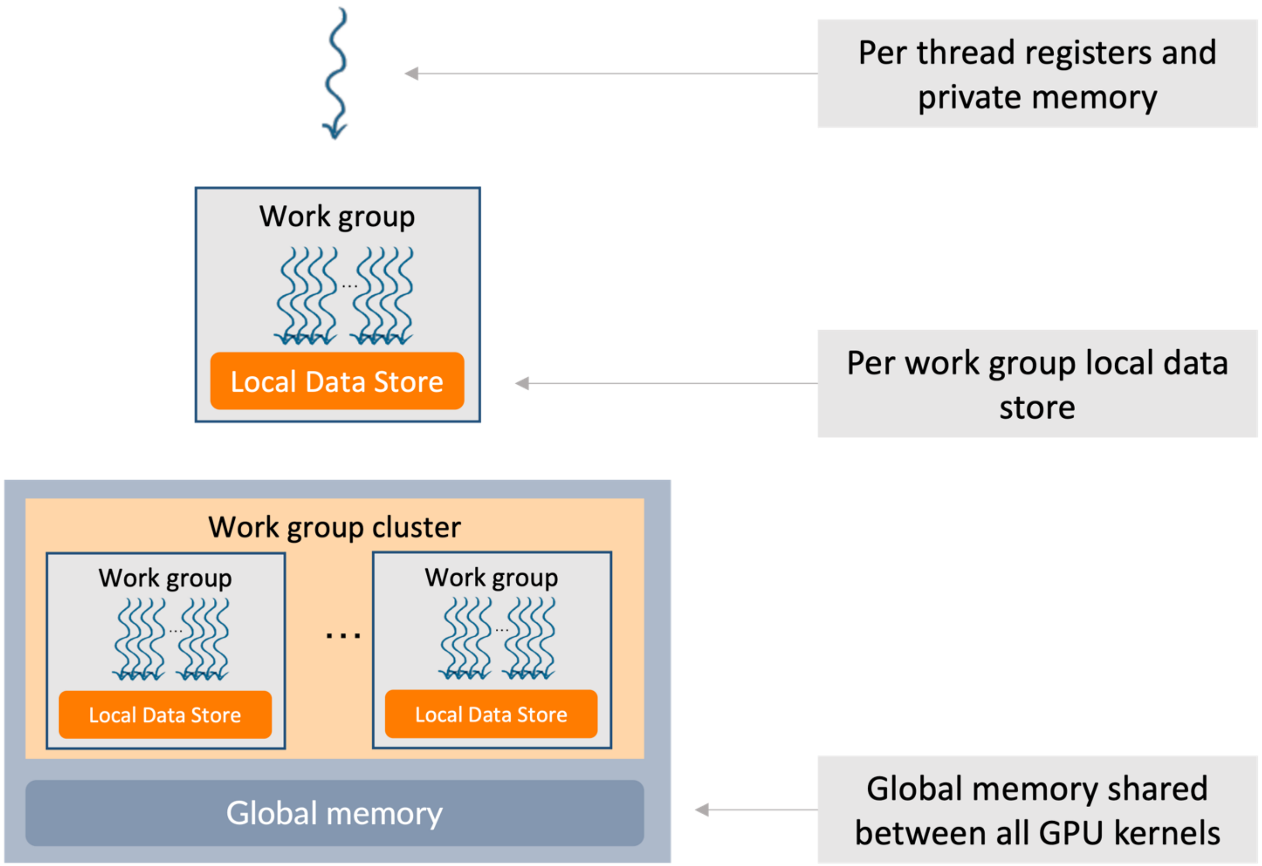
Atomics

Memory hierarchy

- C++ memory model specifies memory ordering

  std::memory_order_seq_cst
  std::memory_order_acq_rel
  std::memory_order_acquire
  std::memory_order_release
  std::memory_order_relaxed

- But…
- Flat storage space
- Single contiguous address space
- No notion of hierarchy (caches, etc.)
- No concept of host or device memory, nor accessibility



OAK RIDGE National Laboratory

Atomic functions in CUDA/HIP

- Sequential consistency and acquire/release are not expressible
- Device only!
- Only supported for a few arithmetic types
  `atomicAdd()` supports int, unsigned int, unsigned long long int, float, double (CC 6.0+)
  `atomicSub()` supports only int, unsigned int
  `atomicInc()` supports unsigned int and that is it
- Widen or narrow the scope with the `_system` or `_block` suffixes (CC 6.0+)
  HIP only supports system scope atomic operations
- Other operations must be implemented in terms of `atomicCAS()` (Compare And Swap)

Atomic operations in desul

- Department of Energy Standard Utility Library (desul)
  - Shared facility between Kokkos and RAJA
  - Hosted on GitHub under https://github.com/desul/desul
- Defined in header <desul/atomics.hpp>
- Generic fallback implementation
  - lock-free (compare-and-swap loop) when possible
  - using lock array otherwise
- Specialize when corresponding hardware instructions are available

Atomic operations in desul

```
template <class T, class MemoryOrder, class MemoryScope>
class atomic_ref;
```

```
struct MemoryOrderSeqCst {};
struct MemoryOrderAcqRel {};
struct MemoryOrderAcquire {};
struct MemoryOrderRelease {};
struct MemoryOrderRelaxed {};
```

```
struct MemoryScopeSystem {};
struct MemoryScopeNode {};
struct MemoryScopeDevice {};
struct MemoryScopeCore {};
struct MemoryScopeCaller {};
```

MemoryOrder specifies how memory accesses are to be ordered around an atomic operation

MemoryScope specifies where the ordering constraint applies

Atomics scope in other programming models

libcu++ (NVIDIA C++ Standard Library)

```
template <
  typename T,
  cuda::thread_scope Scope
    = cuda::thread_scope_system
>
class cuda::atomic;

enum cuda::thread_scope {
  thread_scope_system,
  thread_scope_device,
  thread_scope_block,
  thread_scope_thread
};
```

SYCL

```
template <
  typename T,
  memory_order DefaultOrder,
  memory_scope DefaultScope,
  access::address_space Space
    = access::address_space::generic_space
>
class sycl::atomic_ref;

enum class sycl::memory_scope {
  work_item,
  sub_group,
  work_group,
  device,
  system
};
```

Broad set of generic atomic operations in desul

- Provides generic atomic operations on **non-atomic** objects
  - In contrast, std atomics operate on atomic types

```
template <class T, class MemoryOrder, class MemoryScope>
atomic_is_lock_free
atomic_store
atomic_load
atomic_exchange
atomic_compare_exchange
atomic_fetch_add
atomic_fetch_sub
atomic_fetch_max //
atomic_fetch_min //
atomic_fetch_mul //
atomic_fetch_div //
atomic_fetch_mod //
atomic_fetch_inc //
atomic_fetch_dec //
atomic_fetch_inc_mod //
atomic_fetch_dec_mod //
atomic_fetch_and
atomic_fetch_or
atomic_fetch_xor
atomic_fetch_nand //
atomic_fetch_lshift //
atomic_fetch_rshift //
```

// denotes operations not provided by the C++ standard library

Atomics in the C++ standard library

```
template< class T >
struct std::atomic; // (since C++11)

template< class T >
struct std::atomic_ref; // (since C++20)
```

- Shortcomings
  - With std::atomic objects are atomic
  - Arithmetic operations with std::atomic_ref are always sequentially consistent (too strong)
- Proposed for C++26
  - std::atomic_ref_{relaxed,acq_rel,seq_cst}

Putting it all together

```
template <class ExecutionHandle, class InputIterator>
KOKKOS_FUNCTION void compute_histogram(ExecutionHandle const &exec,
                                       InputIterator first, InputIterator last,
                                       float bin_size,
                                       mdspan<int, dextents<int, 1>> hist) {
  for_each(exec, first, last, [=](float val) {
    int bin = abs(val) / bin_size;
    if (bin > hist.extent(0)) bin = hist.extent(0) - 1;
    atomic_increment(&hist[bin],
                     DeduceMemoryScopeT<ExecutionHandle>{});
  });
}
```

Wrap up

- We want nested algorithms
- Need to pass execution handles through to the next layer
- Execution handles correlate with memory scopes

Thank you!

- C++ standard support for HPC is limited but we are making progress
- You can get involved too!

Damien L-G <lebrungrandt@ornl.gov>