The Kokkos Lectures

Advanced Reductions and Scans

April 26, 2024

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.
SAND2020-7475 PE

Advanced Reductions

Learning objectives:

- How to use Reducers to perform different reductions.
- How to do multiple reductions in one kernel.
- Using `Kokkos::View`'s as result for asynchronicity.
- Custom reductions

Example: Scalar integration

OpenMP

```
double totalIntegral = 0;
#pragma omp parallel for reduction(+:totalIntegral)
for (int64_t i = 0; i < numberOfIntervals; ++i) {
  totalIntegral += function(...);
}
```

Kokkos

```
double totalIntegral = 0;
parallel_reduce(numberOfIntervals,
  [=] (const int64_t i, double & valueToUpdate) {
    valueToUpdate += function(...);
  },
  totalIntegral);
```

- The operator takes **two arguments**: a work index and a value to update.
- The second argument is a **thread-private value** that is managed by Kokkos; it is not the final reduced value.

So far only "sum" reduction. What about other things? Using a Reducer:

```
double max_value = 0;
parallel_reduce("Label", numberOfIntervals,
  KOKKOS_LAMBDA(const int64_t i, double & valueToUpdate) {
    double my_value = function(...);
    if(my_value > valueToUpdate) valueToUpdate = my_value;
}, Kokkos::Max<double>(max_value));
```

So far only "sum" reduction. What about other things? Using a Reducer:

```
double max_value = 0;
parallel_reduce("Label", numberOfIntervals,
  KOKKOS_LAMBDA(const int64_t i, double & valueToUpdate) {
    double my_value = function(...);
    if(my_value > valueToUpdate) valueToUpdate = my_value;
}, Kokkos::Max<double>(max_value));
```

- Note how the operation in the body matches the reducer op!

So far only "sum" reduction. What about other things? Using a Reducer:

```
double max_value = 0;
parallel_reduce("Label", numberOfIntervals,
  KOKKOS_LAMBDA(const int64_t i, double & valueToUpdate) {
    double my_value = function(...);
    if(my_value > valueToUpdate) valueToUpdate = my_value;
}, Kokkos::Max<double>(max_value));
```

- Note how the operation in the body matches the reducer op!
- The scalar type is used as a template argument.

So far only "sum" reduction. What about other things? Using a Reducer:

```
double max_value = 0;
parallel_reduce("Label", numberOfIntervals,
  KOKKOS_LAMBDA(const int64_t i, double & valueToUpdate) {
    double my_value = function(...);
    if(my_value > valueToUpdate) valueToUpdate = my_value;
}, Kokkos::Max<double>(max_value));
```

- Note how the operation in the body matches the reducer op!
- The scalar type is used as a template argument.
- Many reducers available: `Sum`, `Prod`, `Min`, `Max`, `MinLoc`, see: https://kokkos.github.io/kokkos-core-wiki/API/core/builtin_reducers.html

So far only "sum" reduction. What about other things? Using a Reducer:

```
double max_value = 0;
parallel_reduce("Label", numberOfIntervals,
  KOKKOS_LAMBDA(const int64_t i, double & valueToUpdate) {
    double my_value = function(...);
    if(my_value > valueToUpdate) valueToUpdate = my_value;
}, Kokkos::Max<double>(max_value));
```

- Note how the operation in the body matches the reducer op!
- The scalar type is used as a template argument.
- Many reducers available: `Sum`, `Prod`, `Min`, `Max`, `MinLoc`, see: https://kokkos.github.io/kokkos-core-wiki/API/core/builtin_reducers.html
- Some reducers (like `MinLoc`) use special scalar types!

So far only "sum" reduction. What about other things? Using a Reducer:

```
double max_value = 0;
parallel_reduce("Label", numberOfIntervals,
  KOKKOS_LAMBDA(const int64_t i, double & valueToUpdate) {
    double my_value = function(...);
    if(my_value > valueToUpdate) valueToUpdate = my_value;
}, Kokkos::Max<double>(max_value));
```

- Note how the operation in the body matches the reducer op!
- The scalar type is used as a template argument.
- Many reducers available: `Sum`, `Prod`, `Min`, `Max`, `MinLoc`, see: https://kokkos.github.io/kokkos-core-wiki/API/core/builtin_reducers.html
- Some reducers (like `MinLoc`) use special scalar types!
- Custom value types supported via specialization of `reduction_identity`.

Sometimes multiple reductions are needed

- Provide multiple reducers/result arguments
- Functor/Lambda operator takes matching thread-local variables
- Mixing scalar types is fine.

```
float max_value = 0;
double sum = 0;
parallel_reduce("Label", numberOfIntervals,
    KOKKOS_LAMBDA(const int64_t i,float& tl_max,double& tl_sum){
  float a_i = a[i];
  if(a_i > tl_max) tl_max = a_i;
  tl_sum += a_i;
}, Kokkos::Max<float>(max_value),sum);
```

Reducing into a Scalar is blocking!

- Providing a reference to scalar means no lifetime expectation.
  - Call to `parallel_reduce` returns after writing the result.
- `Kokkos::View` can be used as a result, allowing for potentially non-blocking execution.
- Can provide `View` to host memory, or to memory accessible by the `ExecutionSpace` for the reduction.
- Works with Reducers too!

```
View<double,HostSpace> h_sum("sum_h");
View<double,CudaSpace> d_sum("sum_d");
using policy_t = RangePolicy<Cuda>;

parallel_reduce("Label", policy_t(0,N), SomeFunctor,
  h_sum);

parallel_reduce("Label", policy_t(0,N), SomeFunctor,
  Kokkos::Sum<double,CudaSpace>(d_sum));
```

Pseudocode for Kokkos implementation

```
per_thread:
  value& tmp=init(local_tmp);
  for (i in local range)
    functor(i, tmp)
call join for merging values between threads
  in the same thread group
let one (the last) thread group merge all results
  from all thread groups
call final(result) on one thread
```

Three ingredients

- init (optional), default: default constructor
- join (required)
- final (optional), default: no-op

Rules for choosing reduction behavior

1. If a reducer is specified (return type is a functor with `reducer` alias to itself), use that.
2. If functor implements `join`, use functor as reducer.
3. Otherwise, assume `join` behaves like `operator+`.

Note that the functor's `init`, `join`, `final` members must be tagged if the call operator is tagged. The reducers member functions must never be tagged.

```
class Reducer {
  public:
    using reducer     = Reducer;
    using value_type = ... ;
    using result_view_type = Kokkos::View<value_type, ... >;

    KOKKOS_FUNCTION
    void join(value_type& dest, const value_type& src) const;

    //optional
    KOKKOS_INLINE_FUNCTION
    void init(value_type& val) const;

    //optional
    KOKKOS_INLINE_FUNCTION
    void final(value_type& val) const;

    KOKKOS_INLINE_FUNCTION
    value_type& reference() const;

    KOKKOS_INLINE_FUNCTION
    result_view_type view() const;

    KOKKOS_INLINE_FUNCTION
    Reducer(value_type& value_);

    KOKKOS_INLINE_FUNCTION
    Reducer(const result_view_type& value_);
};
```

Exercise: Geometric Mean

Details:

- Location: `Exercises/advanced_reductions/Begin/`
- Look for comments labeled with “EXERCISE”

Scans/Prefix Sums

Learning objectives:

- How to use parallel_scan

The last parallel construct is `parallel_scan`

What is a (simple) scan:

- Consider you have a a list of numbers: 1 3 4 6 7 8
- an (inclusive) scan gives you the running sum:
  1 4 8 14 21 29

Example inclusive scan:

```
double total = 0;
Kokkos::View<double*> view_inclusive("view_inclusive", n);
parallel_scan("Label", n,
  KOKKOS_LAMBDA(const int64_t i,
                double & valueToUpdate, bool is_final) {
    update += i;
    if (is_final)
      view_inclusive(i) = update;
  }, total);
```

- list: 1 3 4 6 7 8
- result: 1 4 8 14 21 29
- total: 29

Example exclusive scan:

```
double total = 0;
Kokkos::View<double*> view_exclusive("inclusive", n);
parallel_scan("Label", n,
  KOKKOS_LAMBDA(const int64_t i,
                double & valueToUpdate, bool is_final) {
    if (is_final)
      view_exclusive(i) = update;
    update += i;
  }, total);
```

- list: 1 3 4 6 7 8
- result: 0 1 4 8 14 21
- total: 29

Example exclusive and inclusive can:

```
Kokkos::View<double*> view_inclusive("view_inclusive", n);
Kokkos::View<double*> view_exclusive("view_exclusive", n);
parallel_scan("Label", n,
  KOKKOS_LAMBDA(const int64_t i,
                double & valueToUpdate, bool is_final) {
    if (is_final)
      view_exclusive(i) = update;
    update += i;
    if (is_final)
      view_inclusive(i) = update;
  });
```

The last parallel construct is `parallel_scan`
Example exclusive and inclusive can:

```
Kokkos::View<double*> view_inclusive("view_inclusive", n);
Kokkos::View<double*> view_exclusive("view_exclusive", n);
parallel_scan("Label", n,
  KOKKOS_LAMBDA(const int64_t i,
                double & valueToUpdate, bool is_final) {
    if (is_final)
      view_exclusive(i) = update;
    update += i;
    if (is_final)
      view_inclusive(i) = update;
  });
```

Pseudocode for Kokkos implementation

Kernel 1:

```
per_thread:
  value& tmp=init(local_tmp);
  for (i in local range)
    functor(i, tmp, /*is_final*/ false)
call join for implementing a prefix sum
  in the same workgroup
let the last workgroup compute the prefix sum for the
  totals of all workgroups and store the result
store intermediate results on each thread
```

Kernel 2:

```
combine workgroup totals with thread intermediate results
call the functor again for final result (with final=true)
```

Three ingredients similar to `parallel_reduce` but no reducers supported

- init (optional), default: default constructor
- join (required)

Behavior:

- functor is called with `is_final=true`
- functor might not be called with `is_final=false`
- functor might be called with `is_final` more than once

Exercise: Factorial

Details:

- Location: `Exercises/parallel_scan/Begin/`
- Look for comments labeled with "EXERCISE"

Advanced Reductions

- `parallel_reduce` defaults to summation
- Kokkos reducers can be used to reduce over arbitrary operations
- Reductions over multiple values are supported
- Only reductions into scalar arguments are guaranteed to be synchronous
- Support for custom reductions

```
parallel_reduce("Join", n,
  KOKKOS_LAMBDA(int i, double& a, int& b) {
    int idx = foo();
    if(idx > b) b = idx;
    a += bar();
  }, result, Kokkos::Max<int>{my_max});
```

Scans

- `parallel_scan` defaults to summation
- Powerful interface to support many algorithms
- Only scans with scalar result guaranteed to be synchronous
- Support for custom scans

```
parallel_scan("Scan", n,
  KOKKOS_LAMBDA(int i, double& update, bool is_final) {
    if(is_final)
      out_view(i) = update;
    ++update;
  }, result, total);
```