# KokkACC: Enhancing Kokkos with OpenACC

Pedro Valero-Lara, Seyong Lee, Marc Gonzalez-Tallada, Joel Denny, and Jeffrey S. Vetter
*Oak Ridge National Laboratory (ORNL)*
{valerolarap},{lees2},{gonzaleztalm},{dennyje},{vetter}@ornl.gov

***Abstract*—Template metaprogramming is gaining popularity as a high-level solution for achieving performance portability on heterogeneous computing resources. Kokkos is a representative approach that offers programmers high-level abstractions for generic programming while most of the device-specific code generation and optimizations are delegated to the compiler through template specializations. For this, Kokkos provides a set of device-specific code specializations in multiple back ends, such as CUDA and HIP. Unlike CUDA or HIP, OpenACC is a high-level and directive-based programming model. This descriptive model allows developers to insert hints (pragmas) into their code that help the compiler to parallelize the code. The compiler is responsible for the transformation of the code, which is completely transparent to the programmer. This paper presents an OpenACC back end for Kokkos: KokkACC. As an alternative to Kokkos's existing device-specific back ends, KokkACC is a multi-architecture back end providing a high-productivity programming environment enabled by OpenACC's high-level and descriptive programming model. Moreover, we have observed competitive performance; in some cases, KokkACC is faster (up to 9×) than NVIDIA's CUDA back end and much faster than OpenMP's GPU offloading back end. This work also includes implementation details and a detailed performance study conducted with a set of mini-benchmarks (AXPY and DOT product) and three mini-apps (LULESH, miniFE and SNAP, a LAMMPS proxy mini-app).**

***Index Terms*—OpenACC, C++ Metaprogramming, Kokkos, CUDA, OpenMP Target, Parallel Programming Models**

## I. Introduction

Template metaprogramming (TMP) languages, such as C++, allow for generic programming, in which programmers focus on the general structure of an application while target-specific code specialization can be handled by the compiler (i.e., different alternative specializations of a template class). In this way, different binaries can be built from the same TMP source code. This technique can be used effectively for performance portability. RAJA [1] and Kokkos [2] are two of the most important examples of C++ TMP-based libraries that enable performance portability.

Kokkos is an open-source, performance-portable C++ template metaprogramming library that aims to be architecture agnostic to enable programmers to move past the low-level details of vendor or target-specific programming models and the varying characteristics of the targeted hardware architectures. Like other TMP approaches, Kokkos also offers device-specific code generation and optimizations through template specialization. For this, Kokkos provides multiple device-specific back ends that are implemented as template libraries on top of some popular high-performance computing (HPC) programming models (e.g., CUDA, HIP, OpenMP, HPX) that are aligned with developments in the C++ standard. However, maintaining and optimizing multiple device-specific back ends for each current and future device type will be complex and error-prone endeavors. To alleviate these problems, this paper presents an OpenACC-based alternative back-end implementation for Kokkos: KokkACC.

OpenACC [3] is a high-level, directive-based programming model which supports C, C++, and Fortran. It was developed to allow programmers to interact with heterogeneous high-performance computing architectures without the effort that is required to fully understand all the low-level programming details and underlying hardware features [4]. This programming model allows developers to insert hints into their code that help the compiler interpret how to parallelize the code. In this way, the compiler is responsible for the transformation of the code, which is completely transparent to the programmer. OpenACC defines a mechanism to offload programs to an accelerator in a heterogeneous system [5]. Because OpenACC is a directive-based programming model, the code can be compiled serially, ignoring the directives and still produce correct results, allowing a single code to be portable across different platforms [6]. This simple model allows non-expert programmers to easily develop code that benefits from accelerators [7]. Currently, OpenACC compilers support several platforms such as x86 multicore CPUs, accelerators (GPUs, FPGAs), OpenPOWER processors, KNLs, and ARM processors.

The motivations/objectives behind this effort include the following:

1) Improved programming productivity to efficiently maintain Kokkos back ends.
   - The OpenACC back end is simpler to implement and maintain compared with other device-specific back ends (i.e., CUDA, HIP) [8].
   - Both Kokkos and OpenACC models aim to be architecture agnostic, which makes the models very similar and facilitates the implementation of Kokkos features within the OpenACC programming model.
2) Better support for heterogeneous devices. OpenACC can target different types of devices, so one single back end can be used to target different hardware architectures.
   - The OpenACC back end will complement the existing OpenMP target back end by providing an alternative, directive-based solution to address the varying levels of language support and maturity across existing directive compilers and different

target devices (see Section IV).

3) Existing Kokkos applications are easier to optimize by using OpenACC features [9], [10] and tools that complement Kokkos.
4) Porting OpenACC applications to Kokkos are simplified by allowing incremental code changes.

To the best of our knowledge, the work described here is the first to use OpenACC as a Kokkos back end. Indeed, KokkACC is being integrated into the upstream Kokkos repository right now, and it is expected to be fully integrated in the upcoming Kokkos releases. Here are the main contributions of this work:

1) A novel, portable, and efficient Kokkos back end based on the OpenACC programming model.
2) A detailed performance analysis using state-of-the-art mini-benchmarks (e.g., AXPY and DOT product) and mini-apps (e.g., LULESH, miniFE and SNAP-LAMMPS).
3) Demonstration of performance portability by achieving competitive or better performance of the tested applications with the proposed OpenACC back end when compared with the existing CUDA and OpenMP target back ends.
4) A detailed analysis about what a descriptive model (OpenACC) can offer against vendor-specific prescriptive models (CUDA) for TMP solutions, like Kokkos.

The rest of this paper is organized as follows: Section 2 covers the necessary background in meta-programming for the case of the Kokkos framework. Section 3 describes the implementation of the OpenACC back end for Kokkos. Section 4 evaluates the implementation, and Section 5 includes some important references and related works. Finally, Section 6 concludes the paper.

## II. Background

The Kokkos programming model builds on two major components: data structures and parallel execution constructs.

### A. Memory Management

The Kokkos memory model follows the general accelerator memory model adopted by most existing accelerator programming models (e.g., CUDA, OpenMP, OpenACC) in which the host and accelerator devices have separate memory spaces that may or may not be shared. The *View* construct is the fundamental data structure used to represent user data in Kokkos programming. It provides abstractions on three core concepts of the Kokkos memory model: *Memory Space*, *Memory Layout*, and *Memory Trait*. *View* is a logical structure that represents an array of zero or more dimensions.

As shown in Figure 1, a programmer can create a *View* object by setting the type of entries and the number of dimensions in the construct—for which the memory space, memory layout, and memory trait are optional—which can then be determined implicitly by the compiler or determined explicitly by the programmer. To copy data from one *View* to another, Kokkos provides memory transfer APIs such as *deep_copy*. For advanced programming, such as intermixing the high-level Kokkos codes with low-level, device-specific codes, Kokkos also provides low-level device memory management APIs such as *kokkos_malloc* and *kokkos_free*.

### B. Parallel Data Execution

Kokkos has two primary data-parallel constructs: *parallel_for* and *parallel_reduce*. Additionally, there are three different execution policies that can be used for these constructs: single range (SR), multidimensional range (MD), and hierarchical parallelism (HR).

Figure 1 depicts examples of the *parallel_for* Kokkos constructs using SR. This example computes a simple AXPY operation. Parallel Kokkos constructs are composed of three main components: (1) a string used for identification for debugging and profiling, (2) the number of iterations of the for-loop, which is implicitly converted into *RangePolicy*, and (3) a C++ lambda expression that acts like a function and can be used as an additional data type. The lambda stores information about the computation for use in every iteration of the loop.

```
Kokkos::View<double*> X("X", N);
Kokkos::View<double*> Y("Y", N);
Kokkos::parallel_for( "axpy_init", N,
    KOKKOS_LAMBDA ( int n ) {
        X(n) = InitValue; Y(n) = InitValue;
    } );
Kokkos::parallel_for( "axpy_computation", N,
    KOKKOS_LAMBDA ( int n ) {
        double alpha = ALPHA; Y(n) += alpha * X(n);
    } );
```

Fig. 1. The *parallel_for* SR construct in the Kokkos API.

The MD approach is very similar to the SR one (see Figure 2). The main difference is the use of *MDRangePolicy*. Using MDRangePolicy, we can use more than one single parameter regarding the number of iterations. This can be seen like a set of nested for-loop. This approach is usually used for multidimensional arrays.

```
Kokkos::View<double**> X("X", M , N);
Kokkos::View<double**> Y("Y", M , N);
typedef Kokkos::MDRangePolicy< Kokkos::Rank<2> >
    mdrange_policy;
Kokkos::parallel_for( "axpy_init",
    mdrange_policy( {0, 0}, {M, N} ),
    KOKKOS_LAMBDA ( int m, int n  ) {
        X(m, n) = InitValue; Y(m, n) = InitValue;
    } );
Kokkos::parallel_for( "axpy_computation",
    mdrange_policy( {0, 0}, {M, N} ),
    KOKKOS_LAMBDA ( int m, int n ) {
        double alpha = ALPHA; Y(m, n) += alpha * X(m, n);
    } );
```

Fig. 2. The *parallel_for* MD range construct in the Kokkos API.

Finally, the HR approach (see Figure 3) is completely different from the other execution policies. It requires using *TeamPolicy*. This approach can exploit up to three different levels of parallelism, which are similar to the concepts of gang, worker, and vector level parallelism used by OpenACC. Although much more complicated to use, this approach allows

developers to have a higher control on the management of the parallelism and potentially improve performance.

```
Kokkos::View<double**> X("X", M , N);
Kokkos::View<double**> Y("Y", M , N);
typedef Kokkos::TeamPolicy<>               team_policy;
typedef Kokkos::TeamPolicy<>::member_type  member_type;
Kokkos::parallel_for( "axpy_init",
    team_policy( M, Kokkos::AUTO ),
    KOKKOS_LAMBDA ( const member_type &teamMember ) {
        const int i = teamMember.league_rank();
        Kokkos::parallel_for(
            Kokkos::TeamThreadRange( teamMember, N ),
            [&] ( const int j ) {
                X(i, j) = InitValue; Y(i, j) = InitValue;
            } );
    } );
Kokkos::parallel_for( "axpy_computation",
    team_policy( M, Kokkos::AUTO ),
    KOKKOS_LAMBDA ( const member_type &teamMember ) {
        const int i = teamMember.league_rank();
        Kokkos::parallel_for(
            Kokkos::TeamThreadRange( teamMember, N ),
            [&] ( const int j ) {
            double alpha = ALPHA; Y(i, j) += alpha * X(i, j);
            } );
    } );
```

Fig. 3. The *parallel_for* HR construct in the Kokkos API.

The *parallel_reduce* constructs are identical to the *parallel_for* constructs except they use one extra parameter to store the result of the reduction. Figure 4 shows examples of the different *parallel_reduce* constructs.

```
Kokkos::parallel_reduce( "dotproduct_computation", N,
    KOKKOS_LAMBDA ( int n, double &tmp ) {
        tmp += X(n) * Y(n);
    }, result );
Kokkos::parallel_reduce( "dotproduct_computation",
    mdrange_policy( {0, 0}, {M, N}),
    KOKKOS_LAMBDA ( int m, int n, double &tmp ) {
        tmp += X(m, n) * Y(m, n);
    }, result );
Kokkos::parallel_reduce( "dotproduct_comp",
    team_policy( M, Kokkos::AUTO ),
    KOKKOS_LAMBDA ( const member_type &teamMember,
        float &update ){
         const int m = teamMember.league_rank();
         float tmp = 0.0;
         Kokkos::parallel_reduce(
            Kokkos::TeamThreadRange ( teamMember, N ),
            [&] ( const int n,
                float &innerUpdate ) {
                    innerUpdate += X(m, n) * Y(m, n);
            }, tmp );
            if ( teamMember.team_rank() == 0 )
                update += tmp;
        }, result );
```

Fig. 4. The different *parallel_reduce* parallelism constructs in the Kokkos API.

### C. Atomic Operations

The Kokkos framework supports several atomic operations, which are offered to programmers as run-time primitives. Arithmetic operations (e.g., $+$, $-$, $\cdot$, $\div$) are supported with different data types (e.g., integer, floating point numbers using 32/64 bits). Logical operators (e.g., or, and, xor) are supported along with min/max and bit-wise operations. In general, each Kokkos back end includes an implementation of all these atomic primitives according to the existing support in the target architecture.

## III. Kokkos OpenACC Back End

Although there is a clear connection between the parallel constructs of Kokkos and the OpenACC specification, the implementation presents some complications. Every Kokkos template class must follow a very specific template pattern and must be re-implemented by using OpenACC pragmas. Everything must be compatible with the OpenACC compiler. One of the biggest complications for implementing the OpenACC back end involves handling complex template specializations deployed in a complex hierarchy; the existing Kokkos implementations rely heavily on various template specializations to optimize the performance on specific targets and patterns, some of which are allowed only for specific cases. Therefore, it is a nontrivial task to identify which parts of the hierarchical implementations of the Kokkos programming model are the ideal targets for optimization.

### A. Memory Management

The Kokkos library's core architecture is designed in a hierarchical and modular manner using the C++ object-oriented programming paradigm. Therefore, when implementing the Kokkos memory model in the new OpenACC back end, we could reuse most of the high-level structures in the existing Kokkos memory management implementations, including various interfaces to the *View* data structures and the dynamic reference counting mechanism for automatic lifespan management of *View* objects. Implementing the Kokkos memory model in the OpenACC back end mostly boils down to implementing low-level device memory management operations, such as allocating device memory, transferring data between the host and device memories, and so on. Thanks to the similarities between the Kokkos and OpenACC memory models, most of the basic memory management operations have one-to-one mapping between the Kokkos and OpenACC constructs (e.g., *Kokkos::malloc* can be implemented using *acc_malloc*; *deep_copy* can be implemented using *acc_memcpy_to_device*, *acc_memcpy_from_device*, and *acc_memcpy_device* primitives).

### B. Parallel Data Execution

Figure 5 illustrates an example implementation of the Kokkos template class for the OpenACC back end's *parallel_for* SR construct. The implementations are intended to be as simple as possible. The *Policy* object corresponds to the second parameter of the *parallel_for* SR construct (see Section II). The *a_functor* object is the lambda passed as the third argument of the Kokkos construct, which acts like a function and must be copied to GPU memory explicitly. Then, the parallelization is carried out by using *#pragma acc parallel loop gang vector*. The parameters of the functor (lambda) must be consistent with the Kokkos specification.

The Kokkos template class implementation for the OpenACC back end's *parallel_for* MD construct is similar to the SR counterpart (see Figure 6). The main differences correspond to the use of multiple indexes and nested for-loops. For simplicity, we show the implementation details that

```
template <class FunctorType, class... Traits>
class ParallelFor< FunctorType,
    Kokkos::RangePolicy<Traits...>,
    Kokkos::Experimental::OpenACC > {
 private:
  using Policy    = Kokkos::RangePolicy<Traits...>;
  using WorkTag   = typename Policy::work_tag;
  using WorkRange = typename Policy::WorkRange;
  using Member    = typename Policy::member_type;
  const FunctorType m_functor;
  const Policy m_policy;
 public:
  inline void execute() const
  {      execute_impl<WorkTag>(); }
  template <class TagType>
  inline void execute_impl() const {
    OpenACCExec::verify_is_process(
        "Kokkos::Experimental::OpenACC␣parallel_for");
    OpenACCExec::verify_initialized(
        "Kokkos::Experimental::OpenACC␣parallel_for");
    const auto begin = m_policy.begin();
    const auto end   = m_policy.end();
    if (end <= begin) return;
    const FunctorType a_functor(m_functor);
    #pragma acc parallel loop gang vector
                             copyin(a_functor)
    for (auto i = begin; i < end; i++) { a_functor(i); }
  }
  ...
};
```

Fig. 5. OpenACC implementation of *parallel_for* SR.

correspond to the MD construct implementation for a nesting level of two (*Rank = 2* in Figure 6). To map the behavior defined by the Kokkos specification for MD operations, we use OpenACC's *collapse* clause.

```
template <class TagType, int Rank>
inline typename std::enable_if<Rank == 2>::type
 execute_functor(
  const FunctorType& functor,
  const Policy& policy ) const {
 const FunctorType a_functor(functor);
 int begin1 = policy.m_lower[0];
 int end1 = policy.m_upper[0];
 int begin2 = policy.m_lower[1];
 int end2 = policy.m_upper[1];
 #pragma acc parallel loop gang vector
           collapse(2) copyin(a_functor)
 for (auto i0 = begin1; i0 < end1; i0++) {
  for (auto i1 = begin2; i1 < end2; i1++) {
   a_functor(i0, i1);
}}}
```

Fig. 6. OpenACC implementation of *parallel_for* MD range.

The OpenACC implementation of the Kokkos template class for the *parallel_for* HR construct can be seen in Figure 7. For simplicity, we show the implementation of the two-level nested lambda case. The top-level implementation is similar to the SR implementation, but it uses *Kokkos::TeamPolicy* instead of the number of iterations (i.e., *RangePolicy*), which is then passed as an argument to the HR specification. Also, a corresponding team policy must be created in each iteration of this level and passed as an argument to the functor (second-level lambda). The second level is implemented in a separate function, which must be decorated with *#pragma acc routine worker*. At this level, we compute an OpenACC-decorated for-loop using the same parallelism level indicated in the function decoration (i.e., *worker*).

Regarding the implementation of the *parallel_reduce* Ope-

```
template <class TagType>
inline void execute_impl() const {
 OpenACCExec::verify_is_process(
    "Kokkos::Experimental::OpenACC␣parallel_for");
 OpenACCExec::verify_initialized(
    "Kokkos::Experimental::OpenACC␣parallel_for");
 auto league_size   = m_policy.league_size();
 auto team_size     = m_policy.team_size();
 auto vector_length = m_policy.impl_vector_length();
 const FunctorType a_functor(m_functor);
 #pragma acc parallel loop gang copyin(a_functor)
 for ( int i = 0; i < league_size; i++ ) {
   int league_id = i;
   typename Policy::member_type
     team( league_id, league_size,
        team_size, vector_length );
   a_functor(team);
}}
#pragma acc routine worker
template <typename iType, class Lambda>
KOKKOS_INLINE_FUNCTION
void parallel_for(
    const Impl::TeamThreadRangeBoundariesStruct<
        iType, Impl::OpenACCExecTeamMember>
        & loop_boundaries, const Lambda& lambda) {
 #pragma acc loop worker
 for (iType j = loop_boundaries.start;
           j < loop_boundaries.end;
           j++) { lambda(j); }
}
```

Fig. 7. OpenACC implementation of *parallel_for* HR.

nACC classes, the main difference w.r.t. *parallel_for* implementations consists of adding the OpenACC clause *reduction*. We can see the details in Figure 8.

```
const FunctorType a_functor(m_functor);
value_type tmp; ValueInit::init(a_functor, &tmp);
...
#pragma acc parallel loop gang vector
           reduction(+:tmp) copyin(a_functor)
 for (auto i = begin; i < end; i++)
  a_functor(i, tmp);
 *m_result_ptr = tmp;
```

Fig. 8. OpenACC implementation of *parallel_reduce* SR.

### C. Atomic Operations

Table I lists the available atomic primitives supported within the Kokkos API. The OpenACC back end implements those using the *atomic* directive with the *capture* clause. For instance, the *fetch-add* operation is implemented with the code exposed in Figure 9. Notice the use of the *routine* directive with a *seq* clause to inform the compiler that this subroutine does not contain additional parallelism. The semantics of the Kokkos atomic construct are directly mapped onto the body of the OpenACC atomic directive.

| Primitive | Types | Directive/Runtime |
|---|---|---|
| compare-and-exchange | 32, 64 | CUDA runtime |
| exchange, assign | 32, 64 | OpenACC directive |
| add, sub, mul, div, mod | 32, 64 | OpenACC directive |
| min, max | 32, 64 | OpenACC directive |
| and, or, xor | 32, 64 | OpenACC directive |
| lshift, rshift | 32, 64 | OpenACC directive |

TABLE I
ATOMIC PRIMITIVES WITHIN THE OPENACC BACK END.

## IV. EVALUATION

The performance analysis of the OpenACC back-end implementation is divided into two parts. First, we evaluate the

```
#pragma acc routine seq
inline unsigned int atomic_fetch_add(
                    volatile unsigned int *const dest,
                    const unsigned int &val ) {
  unsigned int retval;
  unsigned int *ptr = const_cast<unsigned int *>(dest);
  #pragma acc atomic capture
  { retval = ptr[0]; ptr[0] += val; }
  return retval;
}
```

Fig. 9. Implementation of *fetch-add* within the OpenACC Kokkos back end.

new back end on a set of mini-benchmarks by comparing the performance against CUDA and OpenMP target back ends. Second, we analyze the performance on an existing set of important mini-applications that leverage the Kokkos framework. All experiments used one NVIDIA Volta V100 GPU from the Oak Ridge Leadership Computing Facility's Summit supercomputer. We used the NVIDIA compilers NVCC (V11.0.3) and NVHPC (V21.3) for the CUDA and OpenACC back ends, respectively, and the LLVM compiler (V15.0.0git) for the OpenMP target back end. We could not build the OpenMP target back end of the Kokkos library using the IBM XL (V16.1.1-10) compiler or the NVIDIA NVHPC compiler owing to unsupported C++17 and OpenMP features.

### A. Mini-benchmarks

This study consists of a set of mini-benchmarks that compute standard and well-known operations such as AXPY and DOT product. These operations are widely used for benchmarking and can be easily implemented using Kokkos (as shown in Figures 1–4). We evaluate the two primary data-parallel Kokkos constructs: *parallel_for* and *parallel_reduce*. We also implement a set of AXPY mini-benchmarks to evaluate the Kokkos *parallel_for* construct and a set of DOT product mini-benchmarks to evaluate the *parallel_reduce* construct. Finally, we evaluate the three different Kokkos execution policies introduced earlier—SR, MD, and HR—along with three different Kokkos back ends—CUDA, OpenMP target, and OpenACC.

First, we analyze the performance of the SR constructs (Figures 10 and 11 [left]). The performance of the CUDA and OpenACC back ends are similar for both AXPY and DOT product mini-benchmarks, with the CUDA back end being slightly faster than OpenACC on smaller vector sizes, and the OpenACC back end being faster than CUDA on bigger vector sizes. By contrast, OpenMP's target back end takes roughly twice as long to run the AXPY (parallel_for) mini-benchmark and roughly two orders of magnitude longer to run the DOT product (parallel_reduce) benchmark when compared with CUDA and OpenACC.

We see an important difference in performance among the back ends when using the MD execution policy (Figures 10 and 11 [center]). In this case, the CUDA performance is considerably lower than the performance achieved by the OpenACC back end, which reaches a speedup of nearly 9× on the biggest matrix size computed for the DOT product test. For AXPY operations, the OpenMP target back end presents better numbers compared with the CUDA back end, but it is still slower than the OpenACC back end. The OpenMP target back end turns out to be the slowest option for DOT product test.

The CUDA and OpenACC back ends achieve similar performance when using the HR execution policy (Figures 10 and 11 [right]) for AXPY operations, with OpenACC achieving slightly higher performance on big matrices. The OpenMP target back end is slower than the other back ends, and the difference increases on bigger matrices. However, we see a different trend on DOT product mini-benchmarks (i.e., *parallel_reduce*), where the execution time of the OpenMP target and OpenACC back ends is about 25% higher than the CUDA back end.

We used the NVIDIA Nsight system to conduct a more precise performance analysis (see Table II). For this analysis, we used the biggest matrix and vector sizes that we tested. Using Nsight, we can evaluate each of the back ends in terms of number of kernels computed, time consumed by each GPU kernel, number of memory operations, hardware occupancy, and active number of warps (blocks of GPU threads) per streaming multiprocessor (SM). The performance numbers shown in Figures 10 and 11 are confirmed by the numbers provided by the Nsight tool. For instance, when comparing the AXPY-SR performance of all back ends, we see that although the initialization (*axpy_init*) is computed faster on OpenACC, the performance of the *axpy_comp* kernel is similar for the CUDA and OpenACC back ends, but the OpenACC implementation achieved higher occupancy and more active warps per SM. We also included the memory operations (i.e., memory transfers between CPU and GPU) and the throughput of the different levels in the GPU memory hierarchy. To enhance Table II, we highlight the best performance numbers for each mini-benchmark (green for CUDA, blue for OpenACC, and red for OpenMP target). We can see that the CUDA back end requires more memory operations than the other two back ends, and the highest memory throughput is provided by the OpenACC implementation.

Nsight analysis confirms important performance differences between the tested back ends on MD mini-benchmarks (Figures 10 and 11 [center]): the CUDA and OpenMP target implementations reach an occupancy of about 10% and 23%, respectively, whereas the OpenACC occupancy reaches about 97%. Also, the number of active warps is much higher for OpenACC than for CUDA or OpenMP target.

For DOT product mini-benchmarks, we see that OpenACC requires two kernels to compute *parallel_reduce*, whereas the CUDA and OpenMP target back ends only require one. However, this extra kernel does not have a significant impact on performance. The number of active warps per SM, hardware occupancy, and memory throughput reached by the OpenACC implementation are all higher than the other back ends.

In general, and according to this performance study, the OpenACC back end is very competitive and achieves similar performance to the CUDA back end and even exceeds CUDA's performance in some cases. The OpenMP target

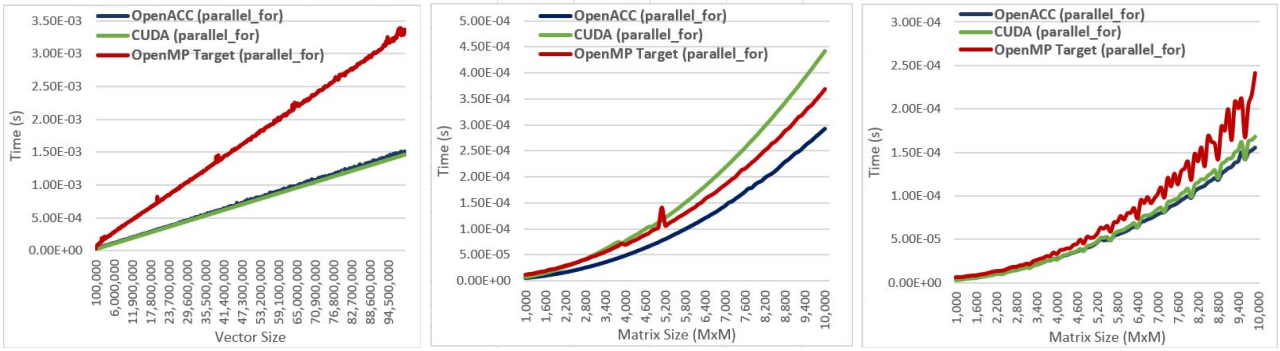


Fig. 10. SR (left), MD (center), HR (right) execution policy performance of *parallel_for*.

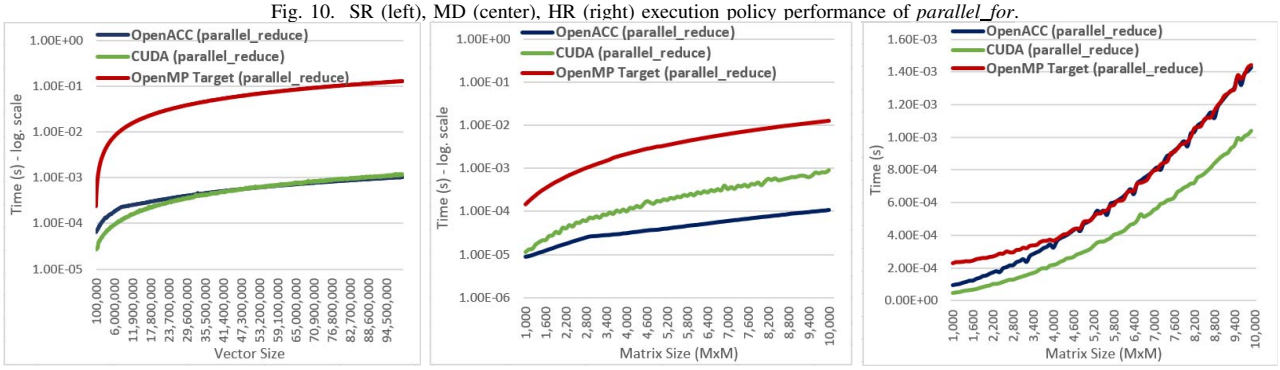


Fig. 11. SR (left), MD (center), HR (right) execution policy performance of *parallel_reduce*

implementation achieved the worst numbers except in a couple of cases. However, we can find an exception to this trend, and if we focus on the DOT-HR benchmark, we see that the OpenACC and OpenMP target implementations reach an occupancy and memory throughput lower than the CUDA implementation, with the CUDA implementation being the fastest one (Figure 10 and 11 [right]).

### B. Mini-applications

We compare performance of the CUDA, OpenMP target, and OpenACC Kokkos back ends on three mini-applications from different domains: (1) LULESH [11], a molecular dynamics proxy application, (2) MiniFE, a finite element mini-application [12], and (3) SNAP-test, a proxy application derived from the molecular dynamics LAMMPS framework [13] [14]. All sources of parallelism have been enabled using the Kokkos parallel constructs.

*1) miniFE mini-application:* This mini-application is divided into two major phases: initialization and computation. Although the initialization is computed once at the very beginning of the execution, the computation phase is computed as many times as the number of iterations (200). For this analysis, we use the largest input size that fits into the device memory (i.e, a mesh of 1,024 $\times$ 128 $\times$ 128 elements). This application uses both *parallel_reduce* and *parallel_for* constructs with the SR policy.

Nsight analysis confirms that the CUDA and OpenACC back ends use different memory management API primitives. For instance, the CUDA back end spends 52% of the time in *cudaMemcpy* and 13% in *cudaMalloc*, whereas the OpenACC back end spends 39% of the time in *cuMemAlloc_v2* and 10% in *cuMemcpyHtoDAsync_v2*. This indicates that the OpenACC back end/compiler (NVHPC) is able to generate a more optimized result than the CUDA back end/compiler (NVCC) in this case. This confirms that different back ends/compilers generate very different output codes with important impacts on performance.

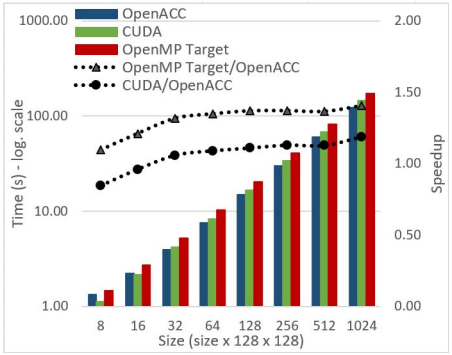


Fig. 12. MiniFE application: overall performance. Y-axis: execution time (s) in logarithmic scale. X-axis: dimension input size used for the application; total input size corresponds to *size* $\times$ 128 $\times$ 128.

Figure 12 shows the overall execution time for the MiniFE application when using the CUDA, OpenMP target, and OpenACC back ends. The general trend is that both CUDA and OpenACC back ends perform similarly, with performance factors ranging from $0.84\times$ (CUDA faster than OpenACC) to $1.29\times$ (OpenACC faster than CUDA). Smaller input sizes clearly present factors close to $1\times$, whereas OpenACC is faster with medium and large input sizes. In particular, for

| —Mini-benchmarks— | | | | | | |
|---|---|---|---|---|---|---|
| **—AXPY—** | | | | | | |
| **–Kokkos execution policy–** | CUDA | | OpenACC | | OpenMP target | |
| **SR** | **Time/Occup.** | **Inst./BS/Warps** | **Time/Occup.** | **Inst./BS/Warps** | **Time/Occup.** | **Inst./BS/Warps** |
| *axpy_init* | 1,286,228/8.18 | 1/128/5.23 | 922,648/94.83 | 1/128/60.69 | 2,717,105/22.59 | 1/128/14.46 |
| *axpy_comp* | 1,445,109/87.72 | 1/128/56.14 | 1,468,756/97.51 | 1/128/62.41 | 3,764,011/21.57 | 1/128/13.81 |
| **Memory operations** | **Time/Inst.** | **RAM/L2/L1/SM** | **Time/Inst.** | **RAM/L2/L1/SM** | **Time/Inst.** | **RAM/L2/L1/SM** |
| *HtD/axpy_init* | 13,471/8 | 69/24/37/13 | 7,359/4 | 97/34/52/23 | 9,538/5 | 69/24/37/13 |
| *DtH/axpy_comp* | 12,225/6 | 92/32/24/12 | 4,322/2 | 92/32/24/13 | 8,927/4 | 92/32/24/12 |
| *Memset* | 4,023/2 | - | 0/- | - | 0/- | - |
| **MD** | **Time/Occup.** | **Inst./BS/Warps** | **Time/Occup.** | **Inst./BS/Warps** | **Time/Occup.** | **Inst./BS/Warps** |
| *axpy_init* | 5,142,724/3.55 | 1/32/2.27 | 922,583/94.77 | 1/128/60.65 | 3,800,173/22.08 | 1/128/14.13 |
| *axpy_comp* | 5,141,252/10.07 | 1/32/6.45 | 1,520,085/97.09 | 1/128/62.14 | 2,817,873/22.82 | 1/128/14.61 |
| **Memory operations** | **Time/Inst.** | **RAM/L2/L1/SM** | **Time/Inst.** | **RAM/L2/L1/SM** | **Time/Inst.** | **RAM/L2/L1/SM** |
| *HtD/axpy_init* | 13,477/8 | 34/12/18/9 | 7,296/4 | 97/34/52/40 | 8,926/5 | 29/10/37/59 |
| *DtH/axpy_comp* | 13,282/6 | 52/18/13/9 | 4,317/2 | 91/32/24/25 | 7,805/4 | 33/11/28/45 |
| *Memset* | 7,819/4 | | 0/- | | 0/- | |
| **HR** | **Time/Occup.** | **Inst./BS/Warps** | **Time/Occup.** | **Inst./BS/Warps** | **Time/Occup.** | **Inst./BS/Warps** |
| *axpy_init* | 916,152/94.95 | 1/128/60.77 | 909,530/72.64 | 1/256/46.49 | 1,046,552/95.10 | 1/256/60.86 |
| *axpy_comp* | 1,690,036/96.88 | 1/128/62.01 | 1,562,549/95.99 | 1/256/61.44 | 1,912,053/97.01 | 1/256/62.09 |
| **Memory operations** | **Time/Inst.** | **RAM/L2/L1/SM** | **Time/Inst.** | **RAM/L2/L1/SM** | **Time/Inst.** | **RAM/L2/L1/SM** |
| *HtD/axpy_init* | 13,345/8 | 96/34/52/6 | 7,456/4 | 86/30/69/41 | 11,937/7 | 83/29/44/17 |
| *DtH/axpy_comp* | 12,255/6 | 79/29/34/5 | 4,545/2 | 85/30/35/22 | 12,606/6 | 69/24/21/10 |
| *Memset* | 4,088/2 | | 0/- | | 0/- | |
| **—DOT product—** | | | | | | |
| **–Kokkos execution policy–** | CUDA | | OpenACC | | OpenMP target | |
| **SR** | **Time/Occup.** | **Inst./BS/Warps** | **Time/Occup.** | **Inst./BS/Warps** | **Time/Occup.** | **Inst./BS/Warps** |
| *dot_init* | 1,286,199/8.57 | 1/128/5.49 | 927,511/94.71 | 1/128/60.62 | 2,717,455/21.57 | 1/128/13.81 |
| *dot_comp* | 1,199,189/40.02 | 1/256/25.61 | 920,312/97.62 | 1/128/62.3 | 133,444,166/24.93 | 1/128/15.96 |
| | | | 91,164/12.43 | 2/256/7.95 | | |
| **Memory operations** | **Time/Inst.** | **RAM/L2/L1/SM** | **Time/Inst.** | **RAM/L2/L1/SM** | **Time/Inst.** | **RAM/L2/L1/SM** |
| *HtD/dot_init* | 13,375/8 | 69/24/37/13 | 6,976/4 | 98/34/52/24 | 11,136/6 | 34/12/40/59 |
| *DtH/dot_comp* | 12,226/6 | 73/25/20/5 | 6,943/3 | 97/34/26/22 | 10,399/5 | 0.6/0.7/1.8/2.7 |
| *Memset* | 3,991/2 | | 1,344/1 | 0.3/0.1/6.9/0 | 0/- | |
| **MD** | **Time/Occup.** | **Inst./BS/Warps** | **Time/Occup.** | **Inst./BS/Warps** | **Time/Occup.** | **Inst./BS/Warps** |
| *dot_init* | 5,119,158/3.49 | 1/32/2.23 | 917,207/94.74 | 1/128/60.63 | 3,170,832/22.08 | 1/128/14.13 |
| *dot_comp* | 9,121,816/40.0 | 1/256/25.60 | 981,304/97.18 | 1/128/62.20 | 10,3469,313/24.90 | 1/128/15.94 |
| *Memset* | | | 89,723/12.43 | 2/256/7.95 | | |
| **Memory operations** | **Time/Inst.** | **RAM/L2/L1/SM** | **Time/Inst.** | **RAM/L2/L1/SM** | **Time/Inst.** | **RAM/L2/L1/SM** |
| *HtD/dot_init* | 13,631/8 | 34/12/18/9 | 7,357/4 | 98/34/52/40 | 10,881/6 | 29/10/37/59 |
| *DtH/dot_comp* | 13,407/6 | 19/5/9/7 | 6,465/3 | 95/33/25/42 | 9,954/5 | 0.8/0.9/2.3/3.6 |
| *Memset* | 8,724/4 | | 1,376/1 | 0.3/0.1/6/0 | 0/- | |
| **HR** | **Time/Occup.** | **Inst./BS/Warps** | **Time/Occup.** | **Inst./BS/Warps** | **Time/Occup.** | **Inst./BS/Warps** |
| *dot_init* | 917,848/95.01 | 1/128/60.81 | 910,710/72.61 | 1/256/46.47 | 1,060,792/94.97 | 1/256/60.78 |
| *dot_comp* | 1,057,367/96.88 | 1/128/62.01 | 1,382,709/74.00 | 1/256/47.36 | 1,463,991/48.03 | 1/256/30.74 |
| | | | 15,484/12.07 | 2/256/7.72 | | |
| **Memory operations** | **Time/Inst.** | **RAM/L2/L1/SM** | **Time/Inst.** | **RAM/L2/L1/SM** | **Time/Inst.** | **RAM/L2/L1/SM** |
| *HtD/dot_init* | 10,780/8 | 96/34/51/6 | 6,144/4 | 85/30/69/41 | 12,642/6 | 85/29/45/18 |
| *DtH/dot_comp* | 10,463/6 | 85/30/25/11 | 5,184/3 | 82/28/20/12 | 11,837/7 | 61/21/20/28 |
| *Memset* | 2,809/2 | | 990/1 | 0.2/0.2/6.2/0 | 0/- | |

TABLE II

PERFORMANCE EVALUATION OF THE AXPY (TOP HALF) AND DOT PRODUCT (BOTTOM HALF) MINI-BENCHMARKS. ALL THREE KOKKOS EXECUTION POLICIES (I.E., SR, MD, AND HR) ARE EVALUATED. WE EVALUATE THE EXECUTION TIME (NS) AND THE HARDWARE OCCUPANCY (%) OF EACH KERNEL AS WELL AS THE THREAD BLOCK SIZE (BS) AND ACTIVE WARPS PER SM (WARPS). ADDITIONALLY, WE EVALUATE TIME (NS) CONSUMED BY THE MEMORY OPERATIONS AND THE NUMBER OF MEMORY TRANSFERS PERFORMED BY EACH OF THE IMPLEMENTATIONS (INST.). FINALLY, WE INCLUDE THE THROUGHPUT (BANDWIDTH % REACHED) OF EACH LEVEL OF THE GPU'S MEMORY HIERARCHY.

an input size of 1,024 $\times$ 128 $\times$ 128 elements OpenACC is $1.29\times$ faster than the CUDA implementation. For OpenMP target, the trends are similar but always show worse executions times compared with OpenACC and CUDA. In particular, the performance factor between OpenMP target and OpenACC is about $1.5\times$ for most of the input sizes.

*2) Lulesh mini-application:* All kernels are based on the *parallel_for* and *parallel_reduce* constructs using the SR policy. Comparing the OpenACC and CUDA back ends (performance factor equal to CUDA time divided by OpenACC time), both perform similarly with performance factors ranging from $0.96\times$ (CUDA faster than OpenACC) to $1.01\times$ (OpenACC faster than CUDA). The OpenMP target back end executes slower than CUDA and OpenACC back ends. While OpenACC roughly matches the CUDA performance (i.e., the speedup factor is close to $1\times$ for most of the kernels), the OpenMP target shows a significant slowdown, especially in the range of kernels that have the largest amount of computation.

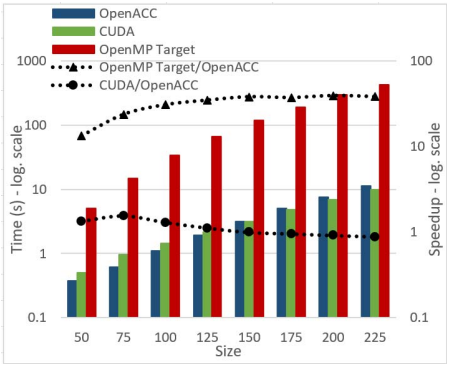


Fig. 13. LULESH: overall performance. Y-axis: execution time (s); logarithmic scale. X-axis: input problem size; total input size corresponds to *Size* × *Size* × *Size*.

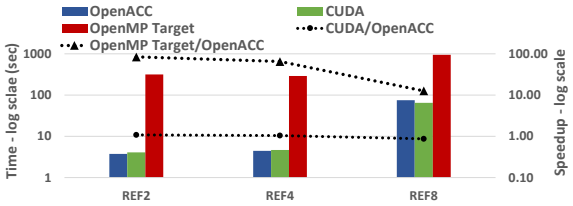


Fig. 14. SNAP: overall performance. Y-axis: execution time (s); logarithmic scale. X-axis: input problem size ( REFx= (atoms, ghost atoms, neighbors, twojmax); REF2 = (2, 33, 8, 2); REF4 = (2, 33, 8, 4) ; REF8 = (2000, 2941, 26, 8) ).

This finding aligns with the results we saw in the mini-benchmark analysis (Figures 10 and 11 [left]). In general, one main observation justifying the different behaviours at application-level is that all back ends instantiate Kokkos constructs with different code. For example, for the CUDA back end, memory management API statistics show that time within runtime execution is distributed as 24% for cudaMalloc, and 5.6% for *cudaMemcpy*. However, for the OpenACC back-end, this distribution changes drastically; 3.6% for *cuMemAlloc_v2* and about 0.3% and 1% for *cuMemcpyDtoHAsync_v2* and *cuMemcpyHtoDAsync_v2* respectively.

Figure 13 shows the overall performance numbers for different problem sizes, which range from 50 to 225 elements in each of the 3 dimensions of the input set. Input sizes greater than 225 elements per dimension exceed the total amount of the device's allocatable memory. For smaller input sizes, OpenACC provides slightly better performance than CUDA; however, for larger input sizes (e.g., 200, 225) OpenACC is 8% slower than CUDA for 200 elements and 13% slower for 225 elements (this is equivalent to the behavior illustrated in Figure 10 [left]). In contrast to the OpenACC case, the OpenMP target back end shows essential slowdown levels.

*3) SNAP-LAMMPS mini-application:* All kernels are based on the *parallel_for* and *parallel_reduce* constructs using the SR policy. The input of the application is parameterized by several factors that determine the actual input size (e.g., number of atoms being simulated, number of neighbors for atoms, etc.). Comparing the OpenACC and CUDA back ends (performance factor equal to CUDA time divided by OpenACC time), both perform similarly (see Figure 14) with performance factors ranging from 0.95x (CUDA faster than OpenACC) to 1.10x (OpenACC faster than CUDA). The OpenMP target back end executes slower than CUDA and OpenACC back ends. Using the OpenMP target back end we get several warnings during compilation, indicating the possibility of not being able to map actual lambda code to the device. This is related to the actual maturity of the OpenMP offload support. In this mini-application, there are four dominating kernels. In some of these kernels, the performance of the OpenACC back end can be up to 34% faster than CUDA back end. In other kernels the performance of the CUDA back end is about 35% faster than the OpenACC back end. In conclusion, OpenACC roughly matches the CUDA performance and the OpenMP-target back end executes slower than CUDA and OpenACC back ends.

### C. Directive-based high-level solutions vs device-specific low-level solutions

It is important to highlight that each back end/compiler (i.e., CUDA/NVCC, OpenACC/NVHPC, OpenMP target/LLVM) generates different parallel code for each Kokkos construct. Therefore, the code compiled for each mini-benchmark and mini-application is completely different. Examples of this can be seen in Table II, where, depending on the back end, a different number of CUDA function calls/kernels and/or CUDA block size is used. This can also be seen in the Lulesh mini-application subsection, where different functions are used for memory management. All these differences have an important impact on performance.

It is not easy to evaluate the set of optimizations applied by each implementation/compiler to explain why a kernel executes faster with one particular back end. Notably, the CUDA version is based on hand-coded run-time calls with no kernel-specific optimizations. In contrast, the OpenACC and OpenMP target versions rely on the compiler optimizations applied on a per-kernel basis. Also, while, the performance of the CUDA back end depends more on the developers' skill and the quality of their implementations, the performance of the other back ends involved in this analysis, OpenACC and OpenMP target, depends more on the quality of the compiler to apply the optimizations. One or the other approach has its own pros and cons. However, for TMP solutions, where the computation to be done is not known in advance and is passed as an argument in the form of a C++ lambda, the directive-based high-level solutions, like OpenACC and OpenMP target, are well positioned when the desired capacity is provided by the compiler.

## V. Related Work

The Kokkos team continues to develop new and important features and optimizations that target performance portability among different architectures, including memory management [15], [16] and vectorization [17]. Kokkos can be successfully integrated or combined with other programming models such as MPI [18], [19] and SYCL [20], among others [21].

Kokkos can also achieve competitive performance compared with other programming models [22]. Owing to these qualities, multiple applications are already using Kokkos [23]–[26].

OpenACC is the de facto standard for directive based programming models on accelerators. One example that summarizes the advantages of using OpenACC is the the work of [8], which evaluates the use of OpenACC, OpenCL and CUDA in terms of performance, productivity, and portability. This work concludes that OpenACC is a robust programming model for accelerators while improving programmer productivity.

## VI. Conclusions

This paper presents KokkACC, which is an OpenACC back end for the Kokkos C++ template metaprogramming library. This work demonstrates the potential benefits of having a high-level and a descriptive programming model such as OpenACC as an alternative to the existing device-specific Kokkos back ends (e.g., CUDA and HIP). Even though device-specific back ends can exploit device-specific features to achieve better performance, the device-specific optimizations can be applied to only a specific type of device, are hard-coded in the back end, and cannot be adjusted for different computing patterns. On the other hand, the descriptive nature of the OpenACC back end allows the compiler to perform advanced optimizations for different computing patterns and device types. The evaluation results also show that the OpenACC back end can complement the existing OpenMP target back end by offering an alternative directive-based approach to solve the practical problems that exist in directive-based, high-level programming, such as varying levels of language support and inconsistent maturity across existing directive compilers and different target devices.

This work focuses on the core constructs required to implement the Kokkos execution and memory models. Implementing the core constructs revealed several limits in the current OpenACC model, such as a lack of custom reductions and constructs for device-specific features. Future work will include adding support for other advanced Kokkos constructs, including Kokkos containers and task parallelism, and exploring possible extensions to support Kokkos features not supported by the current OpenACC standard.

## Acknowledgments

This research used resources of the Experimental Computing Laboratory and the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which are supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725. This research was supported in part by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration. This manuscript has been authored by UT-Battelle, LLC under Contract No. DE-AC05-00OR22725 with the U.S. Department of Energy. The publisher, by accepting the article for publication, acknowledges that the U.S. Government retains a non-exclusive, paid up, irrevocable, world-wide license to publish or reproduce the published form of the manuscript, or allow others to do so, for U.S. Government purposes. The DOE will provide public access to these results in accordance with the DOE Public Access Plan (http://energy.gov/downloads/doe-public-access-plan).

## References

[1] D. Beckingsale, R. D. Hornung, T. Scogland, and A. Vargas, "Performance portable C++ programming with RAJA," in *Proceedings of the 24th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2019, Washington, DC, USA, February 16-20, 2019*, J. K. Hollingsworth and I. Keidar, Eds. ACM, 2019, pp. 455–456. [Online]. Available: https://doi.org/10.1145/3293883.3302577

[2] C. Trott, L. Berger-Vergiat, D. Poliakoff, S. Rajamanickam, D. Lebrun-Grandié, J. Madsen, N. A. Awar, M. Gligoric, G. Shipman, and G. Womeldorff, "The kokkos ecosystem: Comprehensive performance portability for high performance computing," *Comput. Sci. Eng.*, vol. 23, no. 5, pp. 10–18, 2021. [Online]. Available: https://doi.org/10.1109/MCSE.2021.3098509

[3] OpenACC, "OpenACC: Directives for Accelerators," [Online]. Available: http://www.openacc.org, 2011.

[4] S. Chandrasekaran and G. Juckeland, *OpenACC for Programmers: Concepts and Strategies*, 1st ed. Addison-Wesley Professional, 2017.

[5] C. Bonati, E. Calore, S. Coscetti, M. D'elia, M. Mesiti, F. Negro, S. F. Schifano, and R. Tripiccione, "Development of scientific software for HPC architectures using OpenACC: The case of LQCD," in *IEEE/ACM 1st International Workshop on Software Engineering for High Performance Computing in Science*, 2015, pp. 9–15.

[6] R. Dietrich, G. Juckeland, and M. Wolfe, "OpenACC programs examined: A performance analysis approach," in *44th International Conference on Parallel Processing*, 2015, pp. 310–319.

[7] C. Chen, C. Yang, T. Tang, Q. Wu, and P. Zhang, "OpenACC to Intel Offload: Automatic translation and optimization," in *Computer Engineering and Technology*, 2013, pp. 111–120.

[8] J. A. Herdman, W. P. Gaudin, O. Perks, D. A. Beckingsale, A. C. Mallinson, and S. A. Jarvis, "Achieving portability and performance through openacc," in *Proceedings of the First Workshop on Accelerator Programming using Directives, WACCPD '14, New Orleans, Louisiana, USA, November 16-21, 2014*, S. Chandrasekaran, F. S. Foertter, and O. R. Hernandez, Eds. IEEE Computer Society, 2014, pp. 19–26. [Online]. Available: https://doi.org/10.1109/WACCPD.2014.10

[9] L. Toledo, P. Valero-Lara, J. S. Vetter, and A. J. Peña, "Static graphs for coding productivity in openacc," in *28th IEEE International Conference on High Performance Computing, Data, and Analytics, HiPC 2021, Bengaluru, India, December 17-20, 2021*. IEEE, 2021, pp. 364–369. [Online]. Available: https://doi.org/10.1109/HiPC53243.2021.00050

[10] K. Matsumura, S. G. de Gonzalo, and A. J. Peña, "JACC: an openacc runtime framework with kernel-level and multi-gpu parallelization," in *28th IEEE International Conference on High Performance Computing, Data, and Analytics, HiPC 2021, Bengaluru, India, December 17-20, 2021*. IEEE, 2021, pp. 182–191. [Online]. Available: https://doi.org/10.1109/HiPC53243.2021.00032

[11] I. Karlin, J. McGraw, E. Gallardo, J. Keasler, E. A. León, and B. Still, "Abstract: Memory and parallelism exploration using the LULESH proxy application," in *2012 SC Companion: High Performance Computing, Networking Storage and Analysis, Salt Lake City, UT, USA, November 10-16, 2012*. IEEE Computer Society, 2012, pp. 1427–1428. [Online]. Available: https://doi.org/10.1109/SC.Companion.2012.234

[12] M. A. Heroux, D. W. Doerfler, P. S. Crozier, J. M. Willenbring, H. C. Edwards, A. Williams, M. Rajan, E. R. Keiter, H. K. Thornquist, and R. W. Numrich, "Improving Performance via Mini-applications," https://github.com/Mantevo/, 2022, online accessed 20-April-2022.

[13] "ECP Proxy Applications," https://proxyapps.exascaleproject.org/app/snap/, 2022, online accessed 27-July-2022.

[14] A. P. Thompson, H. M. Aktulga, R. Berger, D. S. Bolintineanu, W. M. Brown, P. S. Crozier, P. J. in 't Veld, A. Kohlmeyer, S. G. Moore, T. D. Nguyen, R. Shan, M. J. Stevens, J. Tranchida, C. Trott, and S. J. Plimpton, "LAMMPS - a flexible simulation tool for particle-based materials modeling at the atomic, meso, and continuum scales," *Comp. Phys. Comm.*, vol. 271, p. 108171, 2022.

[15] H. C. Edwards, C. R. Trott, and D. Sunderland, "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," *J. Parallel Distributed Comput.*, vol. 74, no. 12, pp. 3202–3216, 2014.
[16] H. C. Edwards, D. Sunderland, V. Porter, C. Amsler, and S. Mish, "Manycore performance-portability: Kokkos multidimensional array library," *Sci. Program.*, vol. 20, no. 2, pp. 89–114, 2012.
[17] D. Sahasrabudhe, E. T. Phipps, S. Rajamanickam, and M. Berzins, "A portable SIMD primitive using kokkos for heterogeneous architectures," in *Accelerator Programming Using Directives - 6th International Workshop, WACCPD 2019, Denver, CO, USA, November 18, 2019, Revised Selected Papers*, ser. Lecture Notes in Computer Science, S. Wienke and S. Bhalachandra, Eds., vol. 12017. Springer, 2019, pp. 140–163.
[18] G. Hansen, P. G. Xavier, S. P. Mish, T. E. Voth, M. W. Heinstein, and M. W. Glass, "An MPI+X implementation of contact global search using kokkos," *Eng. Comput.*, vol. 32, no. 2, pp. 295–311, 2016.
[19] S. Khuvis, K. Tomko, J. M. Hashmi, and D. K. Panda, "Exploring hybrid mpi+kokkos tasks programming model," in *3rd IEEE/ACM Annual Parallel Applications Workshop: Alternatives To MPI+X, PAW-ATM@SC 2020, Atlanta, GA, USA, November 12, 2020*. IEEE, 2020, pp. 66–73.
[20] B. Joó, T. Kurth, M. A. Clark, J. Kim, C. R. Trott, D. Ibanez, D. Sunderland, and J. Deslippe, "Performance portability of a wilson dslash stencil operator mini-app using kokkos and SYCL," in *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC, P3HPC@SC 2019, Denver, CO, USA, November 22, 2019*. IEEE, 2019, pp. 14–25.
[21] M. M. Wolf, H. C. Edwards, and S. L. Olivier, "Kokkos/qthreads task-parallel approach to linear algebra based graph analytics," in *2016 IEEE High Performance Extreme Computing Conference, HPEC 2016, Waltham, MA, USA, September 13-15, 2016*. IEEE, 2016, pp. 1–7.
[22] J. Eichstädt, M. Vymazal, D. Moxey, and J. Peiró, "A comparison of the shared-memory parallel programming models *OpenMP*, *OpenACC* and *Kokkos* in the context of implicit solvers for high-order FEM," *Comput. Phys. Commun.*, vol. 255, p. 107245, 2020.
[23] K. Teranishi, D. M. Dunlavy, J. M. Myers, and R. F. Barrett, "Sparten: Leveraging kokkos for on-node parallelism in a second-order method for fitting canonical polyadic tensor models to poisson data," in *2020 IEEE High Performance Extreme Computing Conference, HPEC 2020, Waltham, MA, USA, September 22-24, 2020*. IEEE, 2020, pp. 1–7.
[24] R. Halver, J. H. Meinke, and G. Sutmann, "Kokkos implementation of an ewald coulomb solver and analysis of performance portability," *J. Parallel Distributed Comput.*, vol. 138, pp. 48–54, 2020.
[25] S. Rajamanickam, S. Acer, L. Berger-Vergiat, V. Q. Dang, N. D. Ellingwood, E. Harvey, B. Kelley, C. R. Trott, J. Wilke, and I. Yamazaki, "Kokkos kernels: Performance portable sparse/dense linear algebra and graph kernels," *CoRR*, vol. abs/2103.11991, 2021. [Online]. Available: https://arxiv.org/abs/2103.11991
[26] J. A. Ellis and S. Rajamanickam, "Scalable inference for sparse deep neural networks using kokkos kernels," in *2019 IEEE High Performance Extreme Computing Conference, HPEC 2019, Waltham, MA, USA, September 24-26, 2019*. IEEE, 2019, pp. 1–7.

## APPENDIX A
## ARTIFACT DESCRIPTION FOR REPRODUCIBILITY

We have evaluated this work on one node of the ORNL's supercomputer SUMMIT[1], using one NVIDIA Volta V100 GPU. The reader may request access to this system[2]. However, to reproduce the results shown in this work, it is not really necessary to have access to this system. To help to reproduce our results, we provide a detailed description about the software stack used (see Table III).

We are in the middle of a process to include KokkACC within the Kokkos library[3] (expected in the Kokkos 4.0 release) as the new OpenACC back end. In the meantime,

[1]https://www.olcf.ornl.gov/summit/

[2]https://www.olcf.ornl.gov/for-users/documents-forms/olcf-account-application/

[3]https://github.com/kokkos

| **System** | **ORNL's SUMMIT** | | |
|---|---|---|---|
| GPU Architecture | NVIDIA (Volta) V100 | | |
| Kokkos version | 3.6.99 | | |
| Kokkos back end | CUDA | OpenACC | OpenMP Target |
| | Kokkos flags | | |
| | KOKKOS_DEVICES= | | |
| | Cuda | OpenACC | OpenMPTarget |
| | KOKKOS_ARCH="Volta70" | | |
| Compiler | NVCC 11.0.3 | NVHPC 21.3 | LLVM v15.0.0git |
| Compiler flags -std=c++14 -O3 | -Xcudafe -expt-extended-lambda -arch=sm_70 | -acc | -fopenmp -fopenmp-targets= nvptx64 |

TABLE III
SUMMARY OF THE SOFTWARE STACK USED.

the readers can access the code implemented via the Kokkos public ORNL repository[4]. All the codes used for the Evaluation section are accessible via a public GitHub repository[5]. In that repository, we can see two main folders; mini-benchmarks and mini-apps. In the mini-benchmarks folder, we can find multiple folders. The ones that were used for the performance analysis are: openacc-parallel-reduce-single (AXPY and DOT product using SR Kokkos policy), openacc-parallel-reduce-md (AXPY and DOT product using MD range Kokkos policy) and openacc-parallel-reduce-team (AXPY and DOT product using HR range Kokkos policy). We also provide the makefiles for the compilation. Regarding mini-applications, in the mini-apps folder, we can find three folders, one per mini-application. The code used in the analysis can be found in the lulesh, miniFE and TestSNAP folders. For instance, the code and the different makefiles used in each of the mini-applications can be found in the following paths:

- Lulesh: lulesh/lulesh-2.0/kokkos-no-uvm/
- MiniFe: miniFE/kokkos/src
- TestSNAP: TestSNAP/src

There are two ways to reproduce the tests carried out in this study. The binaries to be run can be created by using the Makefiles found in each of the test folders. For instance:

1) In the mini-benchmarks folder in KokkACC-test repository:
   ```
   cd openacc-parallel-reduce-single/
   ```
2) Change Kokkos path in Makefile:
   ```
   KOKKOS_PATH = $(KOKKOS_ROOT)
   ```
3) Compile code via:
   ```
   make -j KOKKOS_DEVICES=OpenACC
   ```

The other way consists of using CMake. In this case, it is necessary to build and install the Kokkos library first, and then compile your test program using CMake. Here is an example of how to build and install the Kokkos library using CMake:

```
1) export KOKKOS_INSTALL_ROOT=$(KOKKOS_ROOT)
2) cd $(KOKKOS_ROOT)
3) mkdir build
4) cd build
5) cmake -DCMAKE_CXX_COMPILER=nvc++
       -DCMAKE_CXX_STANDARD=17
       -DCMAKE_INSTALL_PREFIX=$(KOKKOS_ROOT)
```

[4]https://github.com/ORNL/kokkos-ornl/tree/openacc

[5]https://github.com/pedrovalerolara/KokkACC-tests

```
      -DKokkos_ENABLE_COMPILER_WARNINGS=ON
      -DKokkos_ENABLE_OPENACC=ON ...
```
6) `make -j`
7) `make install`

For instance, these are then the steps to compile and run the Lulesh test case:

1) In mini-apps folder in KokkACC-test repository:
   `cd lulesh/lulesh-2.0/kokkos-no-uvm`
2) `mkdir build`
3) `cd build`
4) `cmake ../ -DKokkos_ROOT=${KOKKOS_ROOT}`
   `-DCMAKE_CXX_COMPILER=nvc++`
5) `make -j`
6) `./lulesh_kokkos`