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

Kokkos is a representative approach of C++ template metaprogramming 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. OpenACC is a high-level, directive-based programming model. This model allows developers to insert hints into their code that help the compiler to parallelize the code. The compiler is responsible for the transformation of the code to parallelize it, which is completely transparent to the programmer. OpenACC offloads programs to an accelerator in a heterogeneous system, which allows non-expert programmers to easily develop code that benefits from accelerators. OpenACC compilers support several architectures: x86 multicore CPUs, accelerators (FPGAs, Intel KNL, AMD and NVIDIA GPUs), OpenPOWER and ARM processors. Maintaining and optimizing multiple device-specific back ends for each new device type can be complex and error-prone. To alleviate these concerns, this paper presents an alternative OpenACC back end for Kokkos: KokkACC. KokkACC provides a high-productivity programming environment and a multi-architecture back end. Next, we enumerate 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 and miniFE). (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) Demonstration of the strengths of using a descriptive (OpenACC) model versus a prescriptive (CUDA) model in terms of performance and programming productivity for C++ metaprograming solutions (Kokkos).

KokkACC implementation: One of the biggest complications for implementing the OpenACC back end involves handling the complex template specializations deployed in the 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.

Memory Management: 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. 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 the primitives *acc_memcpy_to_device* and *acc_memcpy_from_device*).

Parallel Data Execution: Figures 1 illustrates a Kokkos code example for *parallel_for* and the OpenACC implementation respectively. The *Policy* object corresponds to the second parameter of the *parallel_for* Single Range (SR) construct. 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. For the other Kokkos constructs we used other OpenACC pragma as in the case of *parallel_for* Multi-Dimensional (MD) range, where we use the *collapse* clause or *reduce* clause for *parallel_reduce*.

Evaluation: The performance analysis of the OpenACC back-end implementation is divided into two parts. First, we evaluate the new back end on a set of mini-benchmarks (AXPY and DOT product) by comparing the performance against CUDA and OpenMP target back ends. Second, we analyze the performance on an existing set of important 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 compiler or the NVIDIA NVHPC compiler owing to unsupported C++17 and OpenMP features.

Mini-benchmarks: We analyze the performance of the SR constructs (Figure 1). The performance (Figure 2) 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

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


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. 1. The *parallel_for* SR construct in the Kokkos API (top) and the corresponding OpenACC implementation (bottom).

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. We see an important difference in performance among the back ends when using the MD execution policy. In this case, the CUDA performance is considerably lower than the performance achieved by the OpenACC back end, which reaches a speedup of nearly $9\times$ on the biggest matrix size computed for DOT product. 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.

Mini-applications: We use two mini-applications from different domains: (1) LULESH, a molecular dynamics proxy application, and (2) MiniFE, a finite element mini-application.

MiniFE uses both *parallel_reduce* and *parallel_for* constructs with the SR policy. Figure 3 shows the overall execution time for the MiniFE application. 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 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.

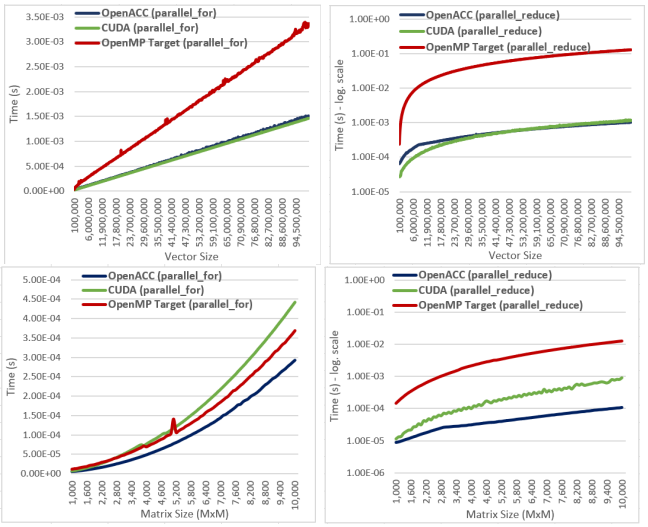

Fig. 2. SR (top) and MD (bottom) range execution policy performance of *parallel_for* (left) and *parallel_reduce* (right).

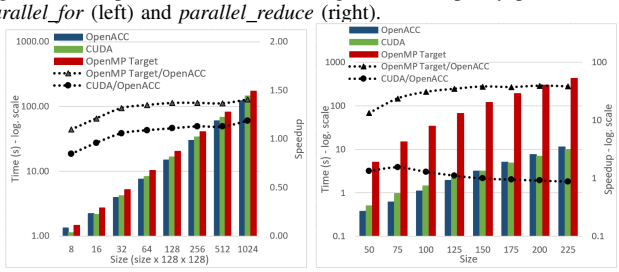

Fig. 3. MiniFE application (left): overall performance. Y-axis: execution time (s) in log. scale. X-axis: dimension input size used for the application. LULESH (right): overall performance. Y-axis: execution time (s); logarithmic scale. X-axis: input problem size; total input size corresponds to *Size* $\times$ *Size* $\times$ *Size*.

Like in MiniFE, LULESH is based on the *parallel_for* and *parallel_reduce* constructs using the SR policy. Figure 3 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. 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. In contrast to the OpenACC case, the OpenMP target back end shows essential slowdown levels. Although OpenMP offers a similar model, the OpenACC back end outperforms the OpenMP backend on the evaluated hardware. We expect these two back ends will continue to offer complementary support across hardware devices.

Conclusions: This work demonstrates the potential benefits of having a high-level, 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, these optimizations can be applied to only a specific type of device, are hard-coded, 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.