sKokkos: Enabling Kokkos with Transparent Device Selection on Heterogeneous Systems using OpenACC

PEDRO VALERO-LARA, SEYONG LEE, JOEL DENNY, KEITA TERANISHI, AND JEFFREY S. VETTER, Oak Ridge National Laboratory, USA
MARC GONZALEZ-TALLADA, Universidad Politecnica de Cataluña, Spain

This paper presents a new feature to enable Kokkos with transparent device selection. For application developers, it is not easy to identify which device is the most appropriate to use in a heterogeneous system, since this depends on the characteristics of both the application and the hardware. In Kokkos, a backend is associated with one specific programming model/hardware. Programmers decide which backend to use at compilation time. This new feature implemented on the OpenACC backend eliminates the burden of deciding which device to use, providing a highly productive programming solution for Kokkos applications. This work includes implementation details and a performance study conducted with a set of mini-benchmarks (i.e., AXPY and dot product), kernels (Lattice-Bolzmann method), and two mini-apps (LULESH and miniFE) on two heterogeneous systems with different hardware capabilities. This new Kokkos feature provides high accelerations of up to 35× thanks to automatic and transparent device selection.

CCS Concepts: • **Computing methodologies → Parallel algorithms**; **Parallel programming languages**.

Additional Key Words and Phrases: Kokkos, OpenACC, C++ Metaprogramming, Heterogeneous Systems, CPU, GPU, Parallel Programming Models, Auto-tuning

ACM Reference Format:
Pedro Valero-Lara, Seyong Lee, Joel Denny, Keita Teranishi, and Jeffrey S. Vetter and Marc Gonzalez-Tallada. 2018. sKokkos: Enabling Kokkos with Transparent Device Selection on Heterogeneous Systems using OpenACC. 1, 1 (August 2018), 18 pages. https://doi.org/XXXXXXX.XXXXXXX

1 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 [33] are two of the most significant examples of C++ TMP-based libraries that enable performance portability.

Kokkos is an open-source, performance-portable C++ TMP library that aims to be architecture-agnostic and 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

Authors' addresses: Pedro Valero-Lara, Seyong Lee, Joel Denny, Keita Teranishi, and Jeffrey S. Vetter, Oak Ridge National Laboratory, Oak Ridge, USA, {valerolarap},{lees2},{dennyje},{teranishik},{vetter}@ornl.gov; Marc Gonzalez-Tallada, Universidad Politecnica de Cataluña, Barcelona, Spain, marc.gonzalez@upc.edu.

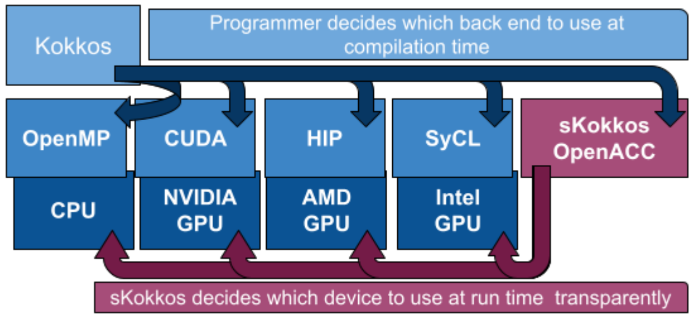
Kokkos
Programmer decides which back end to use at compilation time
OpenMP
CUDA
HIP
SyCL
sKokkos OpenACC
CPU
NVIDIA GPU
AMD GPU
Intel GPU
sKokkos decides which device to use at run time transparently

Fig. 1. sKokkos diagram.

code generation and optimizations through template specialization. For this purpose, Kokkos provides multiple device-specific backends that are implemented as template libraries on top of 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 backends for each current and future device type will be a complex and error-prone endeavor. To alleviate these problems, an OpenACC-based back-end for Kokkos was recently implemented: KokkACC [40].

This work is based on the OpenACC [26] back-end of Kokkos [40]. OpenACC is a high-level, directive-based programming model that supports C, C++, and Fortran. It was developed to allow programmers to interact with heterogeneous HPC architectures without the effort required to fully understand all the low-level programming details and underlying hardware features [5]. 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 [2]. Because OpenACC is a directive-based programming model, the code can be compiled serially, ignoring the directives while still producing correct results, thereby allowing a single code to be portable across various platforms [7]. This simple model allows non-expert programmers to easily develop code that benefits from accelerators [6]. Currently, OpenACC compilers support several platforms such as x86 multicore CPUs, accelerators (e.g., GPUs, FPGAs), OpenPOWER processors, Intel KNLs, and ARM processors.

The motivation/objective of this work is to implement support for transparent device selection as an extension of Kokkos [40], smart Kokkos (sKokkos). This new feature was implemented in the OpenACC back end. OpenACC can target different types of devices, so a single backend can be compiled to target different hardware architectures. sKokkos equips Kokkos with automatic and transparent device selection depending on the computational cost of the applications and the characteristics of the hardware (see Figure 1).

At the time of this writing, the OpenACC back end of Kokkos is being integrated into the upstream Kokkos repository, and it is expected to be fully integrated into the upcoming Kokkos releases. To the best of our knowledge, the work described herein is the first to implement autotuning for transparent device selection in Kokkos. The main contributions of this work are as follows:

(1) A novel Kokkos feature implemented on the OpenACC back end to enable Kokkos with transparent and automatic device selection.
(2) A detailed performance analysis using state-of-the-art mini-benchmarks (i.e., AXPY and dot product), kernels (Lattice-Boltzmann Method), and mini-apps (LULESH and miniFE).

(3) Demonstration of the effectiveness of transparent device selection and performance portability on two heterogeneous systems with different capabilities.

The remainder 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 sKokkos. Section 4 evaluates the implementation, and Section 5 includes some important references and related works. Finally, Section 6 concludes the paper.

2 KOKKOS

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

2.1 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 for three core concepts of the Kokkos memory model: the *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 2, 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 and can 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*.

2.2 Parallel Execution Constructs

Kokkos has two primary data-parallel constructs: *parallel_for* and *parallel_reduce*. Additionally, two execution policies can be used for these constructs: single range (SR) and multidimensional range (MD).

Figure 2 depicts examples of the *parallel_for* Kokkos constructs using SR. This example computes a simple AXPY operation. Kokkos parallel 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.

The MD approach is very similar to that of the SR (see Figure 3). The main difference is the use of *MDRangePolicy policy*. Using MDRangePolicy, users can select more than one parameter regarding the number of iterations. This can be seen like a set of nested for-loops. This approach is usually used for multidimensional arrays.

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

```
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( "computation", N, KOKKOS_LAMBDA(int n) {
        double alpha = ALPHA;
        Y(n) += alpha * X(n);
    }
);
```

Fig. 2. The *parallel_for* SR construct in the Kokkos API.

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

Fig. 3. The *parallel_for* MD range construct in the Kokkos API.

```
Kokkos::parallel_reduce( "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
);
```

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

3 SMART KOKKOS (SKOKKOS): ENABLING KOKKOS WITH TRANSPARENT DEVICE SELECTION

Today, the Kokkos users can decide which device to use in two different ways: (i) Setting the KOKKOS_DEVICES macro before compilation (for instance: KOKKOS_DEVICES=Cuda) or (ii) Using the different execution spaces in Kokkos: Kokkos::DefaultHostExecutionSpace (Threads, OpenMP, etc.) and Kokkos::DefaultExecutionSpace (CUDA, HIP, SyCL, etc.), using the same compiler for both Kokkos execution spaces. In both cases, the users are responsible for making the selection of which device to use. sKokkos enables Kokkos with transparent device selection on heterogeneous systems using the OpenACC back-end (KokkACC). We extended such a back-end to enable Kokkos with transparent device selection, which is carried out at runtime (see Figure 1). As we mentioned at the beginning of this work, a single OpenACC code (or a Kokkos code using the OpenACC back-end) can be deployed on different devices, or in other words, OpenACC compiler is able to generate both codes (CPU and GPU) up front, which makes not only the implementation of sKokkos much simpler but also provides competitive or even faster performance than other back ends [40]. In the following, we explain the main details about the implementation of sKokkos on top of the OpenACC back end of Kokkos:

3.1 KokkACC: The OpenACC back end of Kokkos

Although a clear connection exists 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 "hints," or *pragmas*. Everything must be compatible with the OpenACC compiler. One of the biggest complications for implementing the OpenACC backend 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.

3.2 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 backend, most of the high-level structures in the existing Kokkos memory management implementations could be reused, 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 backend principally involves 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). In the particular case of sKokkos, the device to use is set at the beginning of the execution and cannot be changed along the execution, so no costly memory transfers between devices are carried out.

3.3 Parallel Data Execution

Figure 5 illustrates an example implementation of the Kokkos template class for the OpenACC backend'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 2). 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. In this case, no further modifications were necessary to make on the original implementation of KokkACC. In OpenACC, a *#pragma* can be computed in different ways depending on the target device, adjusting the granularity to the characteristics of the target device, using fine and coarse granularity on GPU and CPU respectively.

```
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 {
    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. sKokkos implementation of *parallel_for* SR using the OpenACC back end.

The Kokkos template class implementation for the OpenACC backend's *parallel_for* MD construct is similar to its SR counterpart (see Figure 6). The main differences correspond to the use of multiple indices and nested for-loops. For simplicity, we show the implementation details that correspond to the MD construct implementation for a nesting level of two (*Rank = 2* in Figure 6). Unlike the implementation of the *parallel_for* SR construct, where the compiler can use a different level of granularity and implement the corresponding memory access pattern depending on the target device without changing the code, in the MD construct, a separate code is required depending on the target device. However, the implementation is straightforward. In fact, the only two differences correspond to the order of the loops and the use of the *collapse* clause [1]. Whereas we need to use a fine-grain granularity (using the *collapse* clause) and a column-major memory access pattern on GPUs, we need to exploit coarse-grain granularity and a row-major memory pattern on CPUs.

For the implementation of the *parallel_reduce* OpenACC classes, the main difference with respect to *parallel_for* implementations consists of adding the OpenACC clause *reduction.* We can see the details in Figure 7.

[1]The *collapse* clause allows to transform a multi-dimensional loop nest into a single-dimensional loop.

```
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 begin0 = policy.m_lower[0];
  int end0 = policy.m_upper[0];
  int begin1 = policy.m_lower[1];
  int end1 = policy.m_upper[1];
  acc_device_t target_dev;
  target_dev = acc_get_device_type();
  if ( target_dev == acc_device_nvidia ) {
    #pragma acc parallel loop gang vector collapse(2) copyin(a_functor)
    for (auto i1 = begin1; i1 < end1; i1++) {
      for (auto i0 = begin0; i0 < end0; i0++) {
        a_functor(i0, i1);
      }
    }
  }
  else if (target_dev == acc_device_host) {
    #pragma acc parallel loop gang vector copyin(a_functor)
    for (auto i0 = begin0; i0 < end0; i0++) {
      for (auto i1 = begin1; i1 < end1; i1++) {
        a_functor(i0, i1);
      }
    }
  }
}
```

Fig. 6. sKokkos implementation of *parallel_for* MD range using the OpenACC back end.

3.4 Auto-tuning in sKokkos

Auto-tuning is used in sKokkos to enable Kokkos with transparent device selection for the different parallel constructs or applications. This process is completely transparent to the programmer, who only has to provide the tuning factor. The tuning factor must be known to efficiently select the proper device to use at the beginning of the execution. Once the device is selected, this is used along the entire execution.

The tuning factor must be related to the computational cost of the operation/s or application to be computed, for instance: this can be the number of operations to be computed by the parallel constructs or a particular characteristic of the application, such as size of the matrix or grid, number of non-zero elements, and so on. For single parallel construct applications, it is recommendable to use the number of operations (computational cost) to be computed as tuning factor. However, for other applications that are composed of multiple parallel constructs, it is better to use the most influential characteristic of the application in terms of performance, for instance: the number of non-zero elements of a sparse matrix for sparse linear algebra operations, the size of the simulation domain for a Computational Fluid Dynamic (CFD) simulation, or the number of operations for dense matrix computations.

It is also important to know the capabilities of the different devices available and how these are connected, such as memory and network bandwidth, number of cores, gigaFLOPs, etc. All this information is needed to estimate the performance of the different devices. Thus, making such an estimation for the different parallel constructs is complex because of the particularities of such operations, synchronizations, data movements, overheads, or latencies [40]. The model used to estimate the time (see Equation 1) is based on the model described by [36], which was proven to be very effective for automatic device (CPU–GPU) selection. This process should be carried out at the beginning of the code,

```
//SR implementation
const FunctorType a_functor(m_functor);
value_type 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;
//MD implementation
value_type tmp;
int begin0 = begin[0];
int end0   = end[0];
int begin1 = begin[1];
int end1   = end[1];
acc_device_t target_dev;
target_dev = acc_get_device_type();
if ( target_dev == acc_device_nvidia ) {
  #pragma parallel loop gang vector collapse(2) reduction(+:tmp) copyin(functor)
  for (auto i1 = begin1; i1 < end1; ++i1) {
    for (auto i0 = begin0; i0 < end0; ++i0) {
      functor(i0, i1, val);
    }
  }
  *m_result_ptr = tmp;
}
else if ( target_dev == acc_device_host ) {
  #pragma parallel loop gang vector reduction(+:tmp) copyin(functor)
  for (auto i0 = begin0; i0 < end0; ++i0) {
    for (auto i1 = begin1; i1 < end1; ++i1) {
      functor(i0, i1, tmp);
    }
  }
  *m_result_ptr = tmp;
}
```

Fig. 7. sKokkos implementation of *parallel_reduce* SR (top) and MD (bottom) using the OpenACC back end.

by using the new Kokkos function: *kokkos::set_device(double tuning_factor)*. This is the only addition to the Kokkos syntax.

$$\begin{aligned} GPU_{performance} &= \frac{Tuning\ Factor}{GPU\ flops} + GPU\ latency \\ CPU_{performance} &= \frac{Tuning\ Factor}{CPU\ flops} \end{aligned} \tag{1}$$

Equation 1 illustrates the formulation used for *parallel_for* constructs. The terms *CPU flops* and *GPU flops* correspond to the hardware peak flops provided by the CPU and GPU. The *GPU latency* term corresponds to the latency associated with the launch of the kernel from the CPU to the GPU. To compute this term, we need to know the bandwidth of the CPU–GPU connection.

$$\begin{aligned} GPU_{performance} &= \frac{Tuning\ Factor}{GPU\ flops} + GPU\ latency \\ &\quad + GPU\ reduction \\ CPU_{performance} &= \frac{Tuning\ Factor}{CPU\ flops} + CPU\ reduction \end{aligned} \tag{2}$$

Estimating the performance for *parallel_reduce* constructs is more complex because of the particulars of such operations, such as multiple kernel launches, synchronization, data movement, among others [40], which requires the use of more terms in the equations (i.e., *CPU reduction* and *GPU reduction* in Equation 2). Essentially, these new terms estimate the necessary time to compute the reduction regarding the capacity of the different architectures (e.g., number of cores, memory bandwidth, connectivity, ...). Unlike the *parallel_for* construct, the time estimation of the *parallel_reduce* construct depends more on how this operation is implemented by the compiler. To know more about implementation details, such as number of kernels used, tools such as Nsight [40] can be used. For instance, if two kernels are used for GPU *parallel_reduce*, we have to multiply by two the GPU latency term in Equation 2.

It is also important to know the number of CPU cores or the number of GPU multiprocessors to make accurate estimations. In particular, this factor is important to compute the last phase of this operator, when the elements of a shared array are reduced into a single result. This operation can be seen as a sequential set of parallel operations [23]; thus, to make a good estimation, we divided the tuning factor into the corresponding number of cores (for CPUs) or multiprocessors (for GPUs). This result is again divided by the ratio between gigaFLOPS and the number of cores/multiprocessors. This (see Equation 3) is a simple and effective way to estimate this operator according to the computational cost of the applications and hardware features [23].

$$\begin{aligned} CPU\ reduction &= \frac{Tuning\ Factor}{\#CPU\ cores} \\ CPU\ reduction\ / &= \frac{CPU\ flops}{\#CPU\ cores} \end{aligned} \tag{3}$$

4 EVALUATION

The performance analysis of the sKokkos implementation is divided into three parts. First, we evaluate our sKokkos on a set of mini-benchmarks by comparing the performance against KokkACC, the OpenACC backend of Kokkos, using GPUs and CPUs. Second, we use a kernel based on the Computational Fluid Dynamics algorithm Lattice-Boltzmann method. Finally, we analyze the performance on an existing set of important mini-applications that leverage the Kokkos framework. We used two different platforms from the Experimental Computing Lab (ExCL) at Oak Ridge National Laboratory (ORNL): Equinox, with one NVIDIA Volta V100 GPU and one Intel Xeon E5-2698 v4 20-Core CPU both connected by PCIe 3.0; and Zenith, with one NVIDIA GeForce RTX 3090 GPU and one AMD Ryzen Threadripper 3970X 32-Core CPU both connected by PCI 4.0. We used the NVIDIA compiler NVHPC installed on both platforms, which is supported on Intel (*haswell*) and AMD (*zen2*) x86-64 CPU architectures.

4.1 Mini-benchmarks

This study consists of a set of mini-benchmarks that compute standard and well-known operations such as AXPY and the dot product. These operations are widely used for benchmarking and can be easily implemented using Kokkos (as

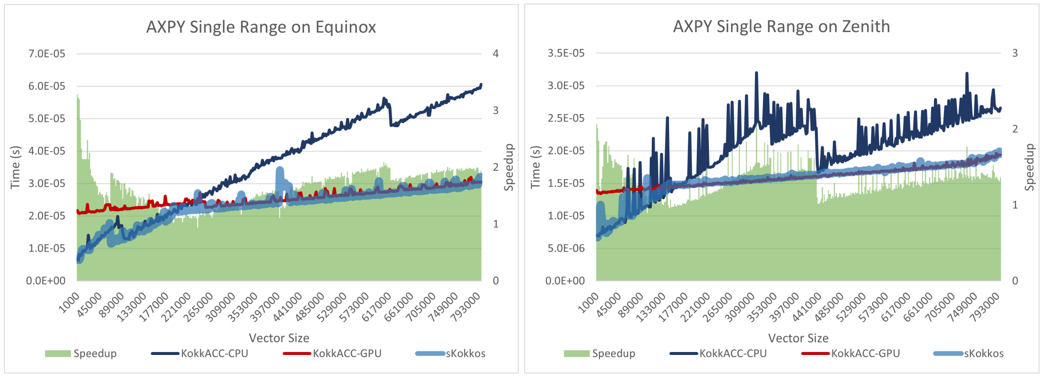

Fig. 8. SR execution policy performance in terms of time (left side of the graph) and speedup (right side of the graph) of *parallel_for* for sKokkos on Equinox (left) and Zenith (right). Speedup is computed as the ratio between the slowest KokkACC time (either using a CPU or a GPU) and sKokkos time.

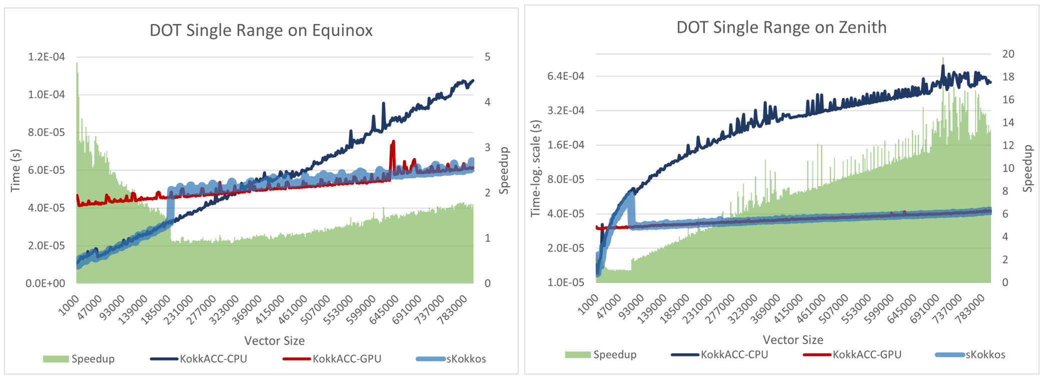

Fig. 9. SR execution policy performance in terms of time (left side of the graph) and speedup (right side of the graph) of *parallel_reduce* for sKokkos on Equinox (left) and Zenith (right). Speedup is computed as the ratio between the slowest KokkACC time (either using a CPU or a GPU) and sKokkos time.

shown in Figures 2–4). We evaluate the two primary data-parallel Kokkos constructs: *parallel_for* and *parallel_reduce*. We implemented 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 two different Kokkos execution policies introduced earlier—SR and MD—along with two different KokkACC configurations (using CPUs and GPUs) and our auto-tuned and heterogeneous sKokkos. For this analysis, the number of operations is used for the tuning factor.

First, we analyze the performance of the SR constructs (Figures 8 and 9). The performance of the OpenACC backend using CPUs is higher than that using GPUs for relatively small- or medium-size vectors, whereas the use of GPUs is more beneficial on larger vectors. This trend is similar for both systems used (Equinox and Zenith) and both mini-benchmarks (AXPY and dot product). We can see that sKokkos can select the proper device to use depending on the hardware characteristics and computational cost of the applications. This can be seen in Figure 8, where the use of sKokkos provides a speedup higher than 3× with respect to KokkACC using CPUs for small vector sizes and a speedup of about 2× with respect to KokkACC using GPUs on Equinox. The trend, although similar, is different in both systems. Whereas

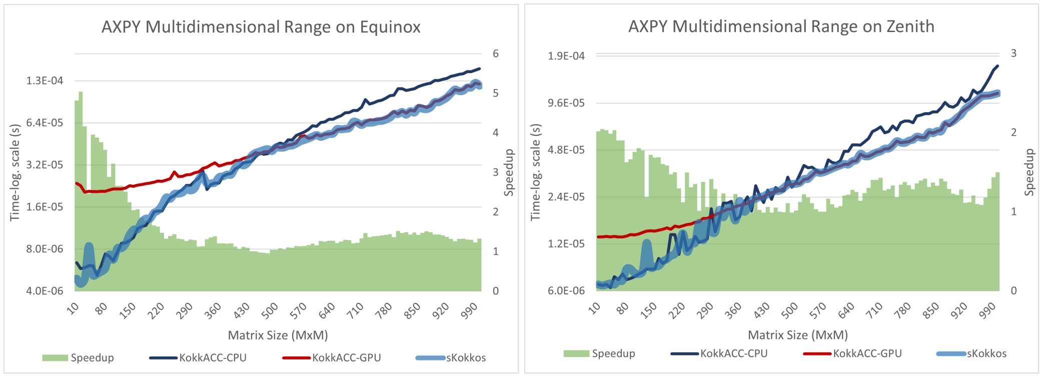

Fig. 10. MD execution policy performance in terms of time (left side of the graph) and speedup (right side of the graph) of *parallel_for* for sKokkos on Equinox (left) and Zenith (right). Speedup is computed as the ratio between the slowest KokkACC time (using either a CPU or a GPU) and sKokkos time.

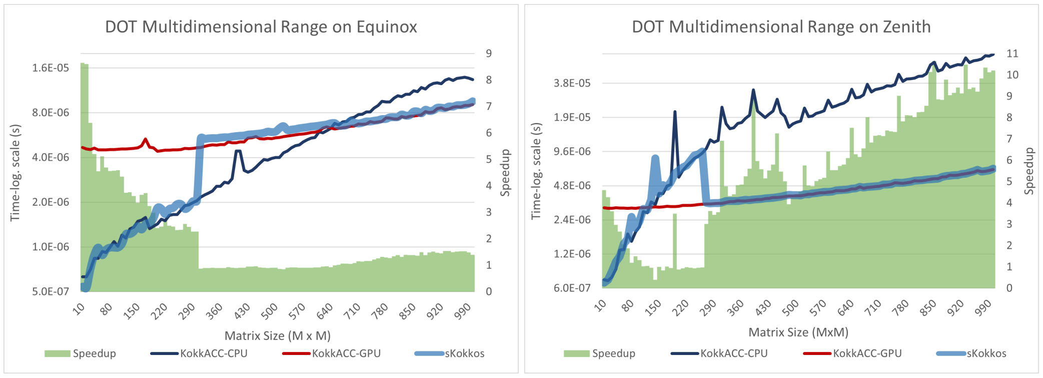

Fig. 11. MD execution policy performance in terms of time (left side of the graph) and speedup (right side of the graph) of *parallel_reduce* for sKokkos on Equinox (left) and Zenith (right). Speedup is computed as the ratio between the slowest KokkACC time (using either a CPU or a GPU) and sKokkos time.

the transition point between CPU and GPU usage occurs for a vector size of 250,000 on Equinox, on Zenith, this vector size is 150,000. Also, the speedup reached is different on both systems. These results are caused by the differences between both systems in terms of hardware and connectivity.

The *parallel_reduce* constructs are more difficult for automatic and transparent device selection because of their complexity (multiple kernel launches, synchronization, data movement, and so on [40]). This is seen in Figure 9, where sKokkos estimates the CPU–GPU transition for a vector size of 185,000 on Equinox. However, this estimation is different on Zenith with a vector size of 90,000. Once again, we can see some differences in the performance reached on both systems. The speedup reached by sKokkos on Equinox and Zenith is about 5× and 2× for small vectors and 2× and 18× for large vectors respectively. These differences in performance are related to the varying hardware and connectivity features of both systems.

We see a similar trend in performance when using the MD execution policy (Figures 10 and 11). In this case, instead of using 1D arrays, we use 2D arrays (i.e., matrices), increasing the complexity and computational intensity of these operations. The time is illustrated by using a logarithmic scale. As shown in Figure 10, the benefit of using sKokkos is higher when used for more complex and computationally expensive operations, showing a greater difference between the CPU and GPU performance. This increments the advantage of using sKokkos, reaching a speedup of up to 5× and 2× on Equinox and Zenith, respectively.

We still experience difficulties in device selection for *parallel_reduce* using the MD execution policy (see Figure 11) but achieve a speedup close to 9× and 11× on Equinox and Zenith. Similar to the previous *parallel_reduce* SR analysis, in this case, we see the most significant difference between both systems in terms of performance. Whereas the difference in performance between CPU and GPU is more significant for small matrices on Equinox, we see the opposite scenario on Zenith.

4.2 Kernel - Lattice-Boltzmann Method

The Lattice-Boltzmann Method (LBM) [37, 38] is an explicit Navier-Stokes solver for weakly compressible flows with lattice-symmetry characteristics which respect to the conservation of the macroscopic moments [15, 27]. LBM does this by modeling the fluid as a distribution function of microscopic particles.

For this study, we implemented the 2-lattice D2Q9 pull algorithm [35, 39]. This algorithm is considered the state-of-the-art fastest algorithm for LBM on both CPUs and GPUs [12, 34]. Like in the previous analysis, we use the number of operations as the tuning factor.

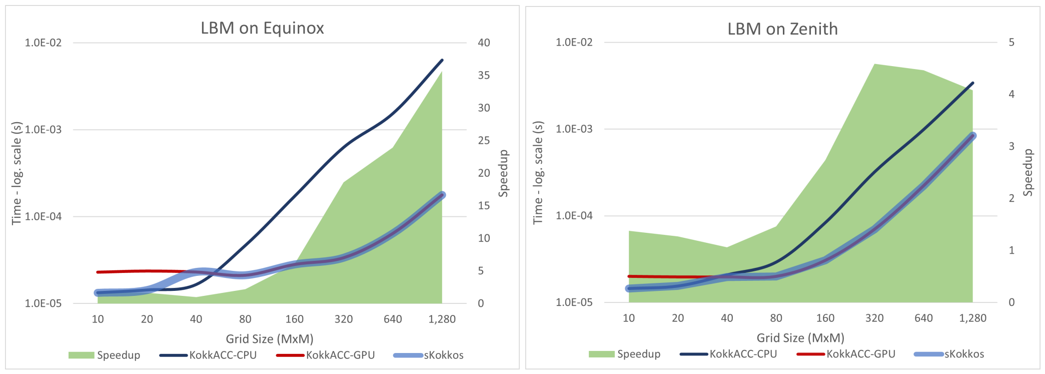

Fig. 12. Lattice-Boltzmann Method Kernel: overall performance for Equinox (left) and Zenith (right). Y-axis: (left) execution time (s) in logarithmic scale, and (right) speedup. X-axis: dimension input size (grid size) used for the kernel. Speedup is computed as the ratio between the slowest KokkACC time (using either a CPU or a GPU) and sKokkos time.

For this analysis, we make use of up to 2GB of data. Unlike the previous study, where we can see a relatively low number of operations because of memory-bound computations, the number of operations is much larger in LBM because of its more computationally expensive operations. These characteristics make the use of GPUs more effective for relatively small mesh sizes, as shown in Figure 12. This is an example of how sKokkos is able to adapt the decision to be taken depending on the characteristics of the applications.

4.3 Mini-applications

In mini-benchmarks, the objective is to get the maximum performance for a single parallel construct. However, in Kokkos applications, which are usually composed of a set of parallel constructs, the objective is to get the best overall performance, even if the chosen device is not the best one for all the parallel constructs of the Kokkos application. So, instead of using the number of operations as the tuning factor for each of the parallel constructs, we have to use the most characteristic tuning factor for the entire application.

We evaluate the performance of sKokkos on two mini applications from different domains: (1) MiniFE [17], a finite-element mini-application and (2) LULESH [19], a molecular dynamics proxy application. All sources of parallelism have been enabled using the Kokkos parallel constructs. All the operations are computed in single precision.

4.3.1 MiniFE mini-application. For this analysis, we increment the size of the inputs until the entire device memory is occupied (see Figure 13). This application uses both *parallel_reduce* and *parallel_for* constructs with the SR policy. The most computationally expensive operation in this application corresponds to the computation of an iterative conjugate gradient algorithm on general sparse matrices [3]. This algorithm is divided into different operations, such as dot products and vector–vector multiplications. However, the most important and computationally expensive operation is the computation of a sparse matrix-vector multiplication [4]. Given the characteristics of this application, we decided to use the number of non-zero elements of the input sparse matrix as the tuning factor, using the Equation 1.

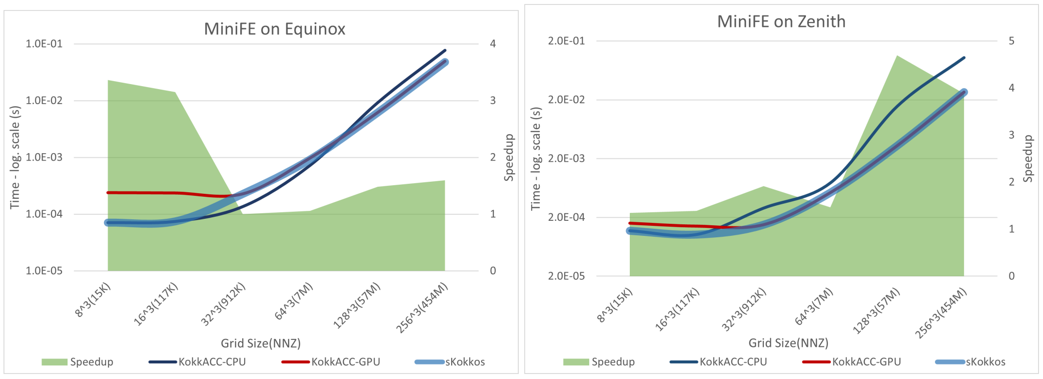

Fig. 13. MiniFE application: overall performance for Equinox (left) and Zenith (right). Y-axis: (left) execution time (s) in logarithmic scale, and (right) speedup. X-axis: dimension input size (grid size and number of non-zero elements) used for the application. Speedup is computed as the ratio between the slowest KokkACC time (using either a CPU or a GPU) and sKokkos time.

Figure 13 shows the overall execution time for the MiniFE application when using KokkACC (on CPU and GPU) and the portable and auto-tuned variant, sKokkos, on both Equinox and Zenith. As expected, CPU usage is more efficient for small input sizes, whereas GPU usage is faster on larger inputs. The general trend confirms the performance presented in the previous analysis (i.e., mini-benchmarks). We see that sKokkos can make a good decision about device selection, achieving a speedup close to 4× and 5× on Equinox and Zenith, respectively.

4.3.2 Lulesh mini-application. As in the analysis carried out on the MiniFE mini-application, we increment the size of the inputs until the entire device memory is used (see Figure 14). Also, all kernels are based on the *parallel_for* and *parallel_reduce* constructs using the SR policy. In this case, the computation is dominated by a set of *parallel_for* constructs that perform relatively simple operations on all the elements of the 3D grid, such as the computation of the force. So, in this case, we use the size of the grid as the tuning factor, using, once a again, the Equation 1.

Figure 14 shows the overall performance numbers for different problem sizes, which range from 16 to 256 elements in each of the three dimensions of the input set. Input sizes greater than 256 elements per dimension exceed the total

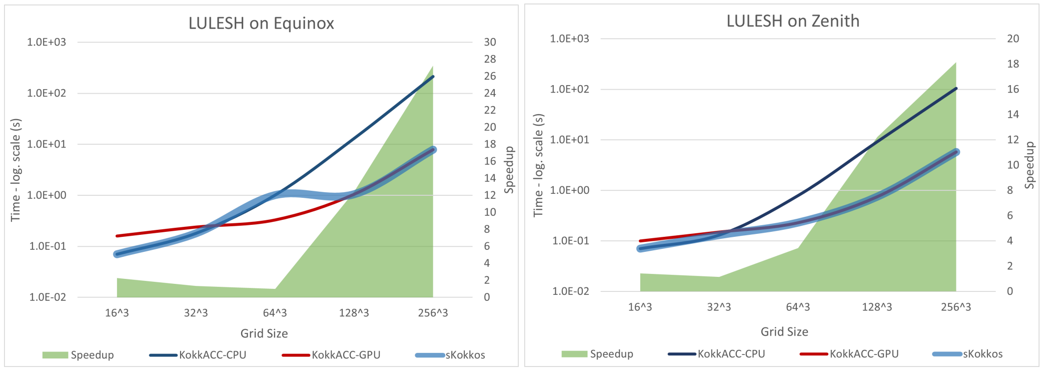

Fig. 14. LULESH: overall performance for Equinox (left) and Zenith (right). Y-axis: (left) execution time (s) in logarithmic scale and (right) speedup. X-axis: input problem size (3D grid). Speedup is computed as the ratio between the slowest KokkACC time (using either a CPU or a GPU) and sKokkos time.

amount of the device's allocatable memory. The computational cost of this application is considerably more expensive than MiniFE. This is clearly illustrated by the time consumed. Once again, sKokkos demonstrates good device selection, providing speedups of about 28× and 19× with respect to KokkACC using CPUs for large input sizes on Equinox and Zenith, respectively. This greater acceleration is related to the computational cost of the application. In other words, the higher the computational cost, the higher the potential benefit.

4.4 Remarks

The analyses above evaluate the benefit of using sKokkos on four mini-benchmarks, one kernel, and two mini-applications. The selection of the proper device to use is not a simple task, as it is related to the particular requirements of the applications as well as the singular characteristics of the hardware and how the different computational components or devices of a heterogeneous node are connected. sKokkos enables Kokkos with transparent device selection, which provides a real performance-portable capability on CPU–GPU heterogeneous systems. Although the device selection is more complicated for *parallel_reduce* constructs, because the computational cost of this operation is more difficult to estimate, the device selection is nonetheless very precise for *parallel_for* constructs.

In the mini-benchmark analysis, we see a different trend in terms of acceleration depending on the system used. For instance, whereas the highest acceleration is achieved for relatively small problems when using the CPU instead of the GPU on Equinox, we see the opposite scenario on Zenith in most of the tests carried out, where the greater speedups are found when running relatively large problems by using the GPU instead of the CPU. This is just one example of the difficulty of selecting the best device to use and how that decision depends on many different factors. Another example of this is the analysis carried out on LBM. Unlike mini-benchmarks, LBM has more computationally expensive operations, and that affects the decision to be taken by sKokkos, which is able to identify such differences and make a good decision. In mini-applications, where more costly operations and multiple parallel constructs are computed, we see a different trend. In these cases, instead of using the number of operations as the tuning factor, we use the most relevant characteristics of the applications, such as the number of non-zero elements (MiniFE) or the size of the domain (Lulesh). It is here that we see the most important benefits, with accelerations of up to 28× and 19× thanks to the transparent and automatic device selection.

sKokkos is able to adapt the decisions depending on the hardware and connectivity features, as well as on the application demands by using a relatively simple model. Although different tuning factors are used, the influence of those on both CPU and GPU hardware, along with some details about the implementation of some operators, such as *parallel_reduce*, provides a good estimation to select the best device.

5 RELATED WORK

The Kokkos library continues integrating new features and optimizations that target performance portability among different architectures, including memory management [8, 9] and vectorization [29]. Kokkos can be successfully integrated or combined with other programming models such as MPI [14, 20] and SYCL [18], among others [41]. Kokkos can also achieve competitive performance compared with other programming models [10]. Owing to these qualities, multiple applications are already using Kokkos [11, 13, 28, 31].

Although OpenACC can be considered one of the most important references for directive-based programming models on GPUs, it is also compatible with other accelerators, such as CPUs and FPGAs. One example that summarizes the advantages of using OpenACC is the works of [16, 32], which evaluate the use of OpenACC in terms of performance, productivity, and portability. These works conclude that OpenACC is a robust programming model for multiple architectures while improving programmer productivity.

This work is an extension of the previously published work of Pedro Valero-Lara et al [40], which describes the efforts for the OpenACC back-end of Kokkos (KokkACC). That work demonstrated the potential benefits of having a high-level and 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.

There are some performance tuning works guided by performance modeling for program executions on heterogeneous node architectures. Kim et al leveraged a Java virtual machine to analyze Java bytecode to select the location of kernel execution calls [21]. COMPASS [22] is an automated performance model generation and performance prediction framework, which generates a structured, parameterized application performance model written in the Aspen domain-specific modeling language [30] by using automated static analysis of the target application and uses the generated application model, combined with a machine model, to predict a right target device between CPU and GPU. MAPredict [25] is another memory access modeling framework built on top of COMPASS, which predicts memory accesses by combining static and dynamic features from applications and hardware. MEPHESTO [24] uses an empirical model to estimate the energy and performance of multiple collocated kernels and devises a scheduling strategy to reach a desired energy-performance balance on heterogeneous SoC systems.

6 CONCLUSIONS & FUTURE WORK

This paper presents sKokkos, which is an extension of the OpenACC backend (KokkACC) for the Kokkos C++ TMP library. This work demonstrates the potential benefits of a high-level and descriptive programming model such as OpenACC to enable Kokkos with automatic and transparent device selection. The descriptive nature of sKokkos's implementation allows the compiler to perform advanced optimizations for varying computing patterns and device types. The results also show that sKokkos not only eliminates the burden for device selection to Kokkos applications providing a real performance-portable and high–programming productivity solution, but it also achieves important accelerations by selecting the proper device, depending on the hardware characteristics and application particularities.

In this work, we used relatively simple performance models, which worked well for the tested applications by carefully choosing the right tuning factors dominating the overall application performance. As future work, we plan to use more advanced performance modeling approaches, which can automatically generate a structured, parameterized application performance model and predict execution times for applications on different devices to find the right target device.

ACKNOWLEDGMENTS

This research used resources of the Oak Ridge Leadership Computing Facility and the Experimental Computing Laboratory at the Oak Ridge National Laboratory, which is supported by DOE's Office of Science 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 DOE's Office of Science and the National Nuclear Security Administration. This material is based upon work by the RAPIDS Institute, supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, and Scientific Discovery through Advanced Computing (SciDAC) program. This manuscript has been authored by UT-Battelle LLC under Contract No. DE-AC05-00OR22725 with the DOE. The publisher, by accepting the article for publication, acknowledges that the US Government retains a non-exclusive, paid-up, irrevocable, worldwide license to publish or reproduce the published form of the manuscript or allow others to do so, for US 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] David Beckingsale, Richard D. Hornung, Tom Scogland, and Arturo Vargas. 2019. 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*, Jeffrey K. Hollingsworth and Idit Keidar (Eds.). ACM, 455–456. https://doi.org/10.1145/3293883.3302577

[2] C. Bonati, E. Calore, S. Coscetti, M. D'elia, M. Mesiti, F. Negro, S. F. Schifano, and R. Tripiccione. 2015. 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*. 9–15.

[3] Sandra Catalán, Xavier Martorell, Jesús Labarta, Tetsuzo Usui, Leonel Antonio Toledo Díaz, and Pedro Valero-Lara. 2019. Accelerating Conjugate Gradient using OmpSs. In *20th International Conference on Parallel and Distributed Computing, Applications and Technologies, PDCAT 2019, Gold Coast, Australia, December 5-7, 2019*. IEEE, 121–126. https://doi.org/10.1109/PDCAT46702.2019.00033

[4] Sandra Catalán, Tetsuzo Usui, Leonel Toledo, Xavier Martorell, Jesús Labarta, and Pedro Valero-Lara. 2020. Towards an Auto-Tuned and Task-Based SpMV (LASs Library). In *OpenMP: Portable Multi-Level Parallelism on Modern Systems - 16th International Workshop on OpenMP, IWOMP 2020, Austin, TX, USA, September 22-24, 2020, Proceedings (Lecture Notes in Computer Science, Vol. 12295)*, Kent F. Milfeld, Bronis R. de Supinski, Lars Koesterke, and Jannis Klinkenberg (Eds.). Springer, 115–129. https://doi.org/10.1007/978-3-030-58144-2_8

[5] Sunita Chandrasekaran and Guido Juckeland. 2017. *OpenACC for Programmers: Concepts and Strategies* (1st ed.). Addison-Wesley Professional.

[6] Cheng Chen, Canqun Yang, Tao Tang, Qiang Wu, and Pengfei Zhang. 2013. OpenACC to Intel Offload: Automatic Translation and Optimization. In *Computer Engineering and Technology*. 111–120.

[7] R. Dietrich, G. Juckeland, and M. Wolfe. 2015. OpenACC Programs Examined: A Performance Analysis Approach. In *44th International Conference on Parallel Processing*. 310–319.

[8] H. Carter Edwards, Daniel Sunderland, Vicki Porter, Chris Amsler, and Sam Mish. 2012. Manycore performance-portability: Kokkos multidimensional array library. *Sci. Program.* 20, 2 (2012), 89–114. https://doi.org/10.3233/SPR-2012-0343

[9] H. Carter Edwards, Christian R. Trott, and Daniel Sunderland. 2014. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *J. Parallel Distributed Comput.* 74, 12 (2014), 3202–3216. https://doi.org/10.1016/j.jpdc.2014.07.003

[10] Jan Eichstädt, Martin Vymazal, David Moxey, and Joaquim Peiró. 2020. 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.* 255 (2020), 107245. https://doi.org/10.1016/j.cpc.2020.107245

[11] J. Austin Ellis and Sivasankaran Rajamanickam. 2019. 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, 1–7. https://doi.org/10.1109/HPEC.2019.8916378

[12] John Gounley, Madhurima Vardhan, Erik W. Draeger, Pedro Valero-Lara, Shirley V. Moore, and Amanda Randles. 2022. Propagation Pattern for Moment Representation of the Lattice Boltzmann Method. *IEEE Trans. Parallel Distributed Syst.* 33, 3 (2022), 642–653. https://doi.org/10.1109/TPDS.2021.3098456

[13] Rene Halver, Jan H. Meinke, and Godehard Sutmann. 2020. Kokkos implementation of an Ewald Coulomb solver and analysis of performance portability. *J. Parallel Distributed Comput.* 138 (2020), 48–54. https://doi.org/10.1016/j.jpdc.2019.12.003

[14] Glen Hansen, Patrick G. Xavier, Sam P. Mish, Thomas E. Voth, Martin W. Heinstein, and Micheal W. Glass. 2016. An MPI+X implementation of contact global search using Kokkos. *Eng. Comput.* 32, 2 (2016), 295–311. https://doi.org/10.1007/s00366-015-0418-x

[15] Xiaoyi He and Li-Shi Luo. 1997. A priori derivation of the lattice Boltzmann equation. *Physical Review E* 55, 6 (1997), R6333.

[16] J. A. Herdman, W. P. Gaudin, Oliver Perks, D. A. Beckingsale, A. C. Mallinson, and Stephen A. Jarvis. 2014. 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*, Sunita Chandrasekaran, Fernanda S. Foertter, and Oscar R. Hernandez (Eds.). IEEE Computer Society, 19–26. https://doi.org/10.1109/WACCPD.2014.10

[17] Michael A. Heroux, Douglas W. Doerfler, Paul S. Crozier, James M. Willenbring, H. Carter Edwards, Alan Williams, Mahesh Rajan, Eric R. Keiter, Heidi K. Thornquist, and Robert W. Numrich. 2022. Improving Performance via Mini-applications. https://github.com/Mantevo/. Online accessed 20-April-2022.

[18] Bálint Joó, Thorsten Kurth, Michael A. Clark, Jeongnim Kim, Christian Robert Trott, Dan Ibanez, Daniel Sunderland, and Jack Deslippe. 2019. 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, 14–25. https://doi.org/10.1109/P3HPC49587.2019.00007

[19] Ian Karlin, Jim McGraw, Esthela Gallardo, Jeff Keasler, Edgar A. León, and Bert Still. 2012. 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, 1427–1428. https://doi.org/10.1109/SC.Companion.2012.234

[20] Samuel Khuvis, Karen Tomko, Jahanzeb Maqbool Hashmi, and Dhabaleswar K. Panda. 2020. 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, 66–73. https://doi.org/10.1109/PAWATM51920.2020.00011

[21] Gloria Y. K. Kim, Akihiro Hayashi, and Vivek Sarkar. 2017. Exploration of Supervised Machine Learning Techniques for Runtime Selection of CPU vs. GPU Execution in Java Programs. In *Accelerator Programming Using Directives - 4th International Workshop, WACCPD 2017, Held in Conjunction with the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2017, Denver, CO, USA, November 13, 2017, Proceedings (Lecture Notes in Computer Science, Vol. 10732)*, Sunita Chandrasekaran and Guido Juckeland (Eds.). Springer, 125–144. https://doi.org/10.1007/978-3-319-74896-2_7

[22] Seyong Lee, Jeremy S. Meredith, and Jeffrey S. Vetter. 2015. COMPASS: A Framework for Automated Performance Modeling and Prediction. In *ACM International Conference on Supercomputing (ICS15)*. https://doi.org/10.1145/2751205.2751220

[23] Wen mei W. Hwu, David B. Kirk, and Izzat El Hajj. 2023. Programming Massively Parallel Processors (Fourth Edition). In *Programming Massively Parallel Processors (Fourth Edition)* (fourth edition ed.), Wen mei W. Hwu, David B. Kirk, and Izzat El Hajj (Eds.). Morgan Kaufmann, xv.

[24] Mohammad Alaul Haque Monil, Mehmet E. Belviranli, Seyong Lee, Jeffrey S. Vetter, and Allen D. Malony. 2020. MEPHESTO: Modeling Energy-Performance in Heterogeneous SoCs and Their Trade-Offs. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques* (Virtual Event, GA, USA) *(PACT '20)*. Association for Computing Machinery, New York, NY, USA, 413–425. https://doi.org/10.1145/3410463.3414671

[25] Mohammad Alaul Haque Monil, Seyong Lee, Jeffrey S. Vetter, and Allen D. Malony. 2022. MAPredict: Static Analysis Driven Memory Access Prediction Framework for Modern CPUs. In *High Performance Computing*, Ana-Lucia Varbanescu, Abhinav Bhatele, Piotr Luszczek, and Baboulin Marc (Eds.). Springer International Publishing, Cham, 233–255.

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

[27] Yue-Hong Qian, Dominique d'Humières, and Pierre Lallemand. 1992. Lattice BGK models for Navier-Stokes equation. *EPL (Europhysics Letters)* 17, 6 (1992), 479.

[28] Sivasankaran Rajamanickam, Seher Acer, Luc Berger-Vergiat, Vinh Q. Dang, Nathan D. Ellingwood, Evan Harvey, Brian Kelley, Christian R. Trott, Jeremiah Wilke, and Ichitaro Yamazaki. 2021. Kokkos Kernels: Performance Portable Sparse/Dense Linear Algebra and Graph Kernels. *CoRR* abs/2103.11991 (2021). arXiv:2103.11991 https://arxiv.org/abs/2103.11991

[29] Damodar Sahasrabudhe, Eric T. Phipps, Sivasankaran Rajamanickam, and Martin Berzins. 2019. 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 (Lecture Notes in Computer Science, Vol. 12017)*, Sandra Wienke and Sridutt Bhalachandra (Eds.). Springer, 140–163. https://doi.org/10.1007/978-3-030-49943-3_7

[30] Kyle Spafford and Jeffrey S. Vetter. 2012. Aspen: A Domain Specific Language for Performance Modeling. In *SC12: ACM/IEEE International Conference for High Performance Computing, Networking, Storage, and Analysis*.

[31] Keita Teranishi, Daniel M. Dunlavy, Jeremy M. Myers, and Richard F. Barrett. 2020. 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, 1–7. https://doi.org/10.1109/HPEC43674.2020.9286251

[32] Leonel Toledo, Pedro Valero-Lara, Jeffrey S. Vetter, and Antonio J. Peña. 2021. 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, 364–369. https://doi.org/10.1109/HiPC53243.2021.00050

[33] Christian Trott, Luc Berger-Vergiat, David Poliakoff, Sivasankaran Rajamanickam, Damien Lebrun-Grandié, Jonathan Madsen, Nader Al Awar, Milos Gligoric, Galen Shipman, and Geoff Womeldorff. 2021. The Kokkos EcoSystem: Comprehensive Performance Portability for High Performance Computing. *Comput. Sci. Eng.* 23, 5 (2021), 10–18. https://doi.org/10.1109/MCSE.2021.3098509

[34] Pedro Valero-Lara. 2014. Accelerating solid-fluid interaction based on the immersed boundary method on multicore and GPU architectures. *J. Supercomput.* 70, 2 (2014), 799–815. https://doi.org/10.1007/s11227-014-1262-2

[35] Pedro Valero-Lara. 2017. Reducing memory requirements for large size LBM simulations on GPUs. *Concurr. Comput. Pract. Exp.* 29, 24 (2017). https://doi.org/10.1002/cpe.4221

[36] Pedro Valero-Lara, Diego Andrade, Raül Sirvent, Jesús Labarta, Basilio B. Fraguela, and Ramon Doallo. 2019. A Fast Solver for Large Tridiagonal Systems on Multi-Core Processors (Lass Library). *IEEE Access* 7 (2019), 23365–23378. https://doi.org/10.1109/ACCESS.2019.2900122

[37] Pedro Valero-Lara, Francisco D. Igual, Manuel Prieto-Matías, Alfredo Pinelli, and Julien Favier. 2015. Accelerating fluid-solid simulations (Lattice-Boltzmann & Immersed-Boundary) on heterogeneous architectures. *J. Comput. Sci.* 10 (2015), 249–261. https://doi.org/10.1016/j.jocs.2015.07.002

[38] Pedro Valero-Lara and Johan Jansson. 2015. LBM-HPC - An Open-Source Tool for Fluid Simulations. Case Study: Unified Parallel C (UPC-PGAS). In *2015 IEEE International Conference on Cluster Computing, CLUSTER 2015, Chicago, IL, USA, September 8-11, 2015*. IEEE Computer Society, 318–321. https://doi.org/10.1109/CLUSTER.2015.52

[39] Pedro Valero-Lara and Johan Jansson. 2017. Heterogeneous CPU+GPU approaches for mesh refinement over Lattice-Boltzmann simulations. *Concurr. Comput. Pract. Exp.* 29, 7 (2017). https://doi.org/10.1002/cpe.3919

[40] Pedro Valero-Lara, Seyong Lee, Marc González Tallada, Joel E. Denny, and Jeffrey S. Vetter. 2022. KokkACC: Enhancing Kokkos with OpenACC. In *9th Workshop on Accelerator Programming Using Directives, WACCPD@SC 2022, Dallas, TX, USA, November 13-18, 2022*. IEEE, 32–42. https://doi.org/10.1109/WACCPD56842.2022.00009

[41] Michael M. Wolf, H. Carter Edwards, and Stephen L. Olivier. 2016. 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, 1–7. https://doi.org/10.1109/HPEC.2016.7761649