The Kokkos EcoSystem: Comprehensive Performance Portability for High Performance Computing

Christian Trott, Luc Berger-Vergiat, David Poliakoff, and Sivasankaran Rajamanickam, *Sandia National Laboratories, Albuquerque, NM, 87123, USA*

Damien Lebrun-Grandie, *Oak Ridge National Laboratory, Oak Ridge, TN, 37830, USA*

Jonathan Madsen, *Lawrence Berkeley National Laboratory, Berkeley, CA, 94720, USA*

Nader Al Awar and Milos Gligoric, *The University of Texas at Austin, Austin, TX, 78712, USA*

Galen Shipman and Geoff Womeldorff, *Los Alamos National Laboratory, Los Alamos, NM, 87545, USA*

State-of-the-art engineering and science codes have grown in complexity dramatically over the last two decades. Application teams have adopted more sophisticated development strategies, leveraging third party libraries, deploying comprehensive testing, and using advanced debugging and profiling tools. In today's environment of diverse hardware platforms, these applications also desire performance portability—avoiding the need to duplicate work for various platforms. The Kokkos EcoSystem provides that portable software stack. Based on the Kokkos Core Programming Model, the EcoSystem provides math libraries, interoperability capabilities with Python and Fortran, and Tools for analyzing, debugging, and optimizing applications. In this article, we overview the components, discuss some specific use cases, and highlight how codesigning these components enables a more developer friendly experience.

Today's science and engineering codes bear little resemblance to their predecessors from 25 years ago. Back then, a couple of motivated graduate students were able to write state-of-the-art applications, which were comparable in capabilities to the leading codes in their field. As an example, take the widely used Molecular Dynamics Code LAMMPS,[1] which regularly is one of the top codes used on many HPC platforms. In 1999, it was still written in Fortran and amounted to just 17,000 lines of code. Today, LAMMPS tops 1,300,000 lines—a 75× increase.

A driver and enabler for such complexity increases is the adoption of Open Source software development approaches. It allows for a project to leverage an amount of development resources, which used to be available only to commercial software development. It also enables scientists to simply extend existing code, instead of developing something from scratch. As a consequence, communities now often coalesce around a few comprehensive solutions, which cover many use cases, instead of developing a zoo of unique applications, each supporting only a specific use case.

However, these applications face a new challenge: the proliferation of new High Performance Computing (HPC) architectures. For a decade or two, most applications used MPI for parallelism and otherwise wrote sequential code in languages such as Fortran, C, and C++. On these new platforms however, new paradigms need to be introduced—largely to deal with on-node parallelism. Depending on the system's vendor, that means writing algorithms in CUDA, OpenMP, SYCL, or HIP for example.

To avoid reimplementing a code for each hardware platform, software teams are now adopting solutions that promise *performance portability*—solutions that

Digital Object Identifier 10.1109/MCSE.2021.3098509
Date of publication 2 August 2021; date of current version 23 September 2021.

enable developers to write code once, which then gets mapped to each of the platforms by some portable runtime, compiler, or abstraction layer. However, it is not enough to just have a programming model. Complex applications regularly use a number of fundamental capabilities that are also provided by vendors, such as math libraries, which often differ in API and functionality. Special casing their use for every platform is cumbersome and error prone, and thus should be avoided. Software teams also need tools to debug and profile their applications. If those tools do not understand the portability layers and cannot provide information on each targeted platform, productivity suffers.

To address this predicament, the Kokkos team has developed the Kokkos EcoSystem—a set of capabilities that form the foundational layer for implementing and maintaining complex scientific and engineering applications. It is based on the Kokkos C++ Performance Portability Programming Model,[2] and provides math functions, language interoperability facilities, as well as tools integration.

In this article, we provide a short overview of the primary capabilities in Kokkos, describe why these are important as a foundational layer for scientific software, and give some examples where codesigning these efforts is critical for enabling developer productivity.

KOKKOS ECOSYSTEM OVERVIEW

The Kokkos EcoSystem[a] is fundamentally an abstraction layer that isolates applications from the details of the targeted hardware architecture, as well as platform-specific scientific libraries (Figure 1). The origins of Kokkos date back to 2008, when it started as a C++ project to provide some basic linear algebra capabilities that would work both on CPUs, and on the then new GPU systems. A few years later, a second iteration was started based on the insight that transparently managing data access patterns is critical for performance portability. This led to the development of the general programming model, which is now the Kokkos Core Project. It is implemented as a C++ abstraction layer on top of native programming models such as OpenMP, CUDA, HIP, and SYCL. One key aspect of Kokkos is that a strong data abstraction is an integral part of the model. Not only does data know where it is allocated, the abstractions also allow for transparent memory layout changes to satisfy different data access pattern requirements on different hardware, without rewriting algorithms. Kokkos Kernels was introduced on top of the Kokkos Core programming model, reimplementing dense and sparse linear algebra capabilities, and providing access to vendor libraries under a common interface. However, application developers had a hard time correlating performance data collected by platform-specific tools back to the Kokkos expressions used in their code. To remedy this productivity issue, the Kokkos Tools effort was conceived. It introduced a tooling callback infrastructure in Kokkos, which tools can subscribe to in order to get Kokkos-specific information. This allows for the development of platform-agnostic tools, and enables leveraging platform-specific instrumentation functionality, via thin translation layers.

Another issue of increasing urgency with the growing user base is language interoperability. Kokkos is C++ based, but many HPC applications are written in Fortran or want to use—at least in part—Python for higher productivity. In order to meet application teams where they are, the Kokkos project incorporated efforts to interact with C++ Kokkos code from both Fortran and Python. Recently a new contribution to the Kokkos project is PyKokkos, which lets developers write kernels in Python.

Originally, Kokkos was developed by a team at Sandia National Laboratories. Today, the EcoSystem is largely funded by the US Department of Energy Exascale Computing Project (ECP). This allowed the Kokkos team to expand, leading to a development team now spanning five US National Laboratories as well as multiple non-DOE partners. This dedicated team of two dozen developers enables the Kokkos EcoSystem to succeed and support more than 100 application teams relying on it for running across a variety of computing architectures and achieving performance portability.

KOKKOS CORE: PROGRAMMING MODEL

The central piece of the Kokkos EcoSystem is its Programming Model. The Kokkos Programming Model was designed from the ground up with performance portability in mind. Its primary guiding principle is to be *descriptive* and not *prescriptive*. Developers express their algorithms in terms of general parallel programming concepts, instead of explicitly saying how an algorithm is mapped to specific hardware. This gives Kokkos the freedom to map an algorithm to underlying hardware in whatever the way it deems best.

For example, in Kokkos, a simple parallel loop is expressed through the `parallel_for` construct. It only asserts i) that the user wants the loop to be

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

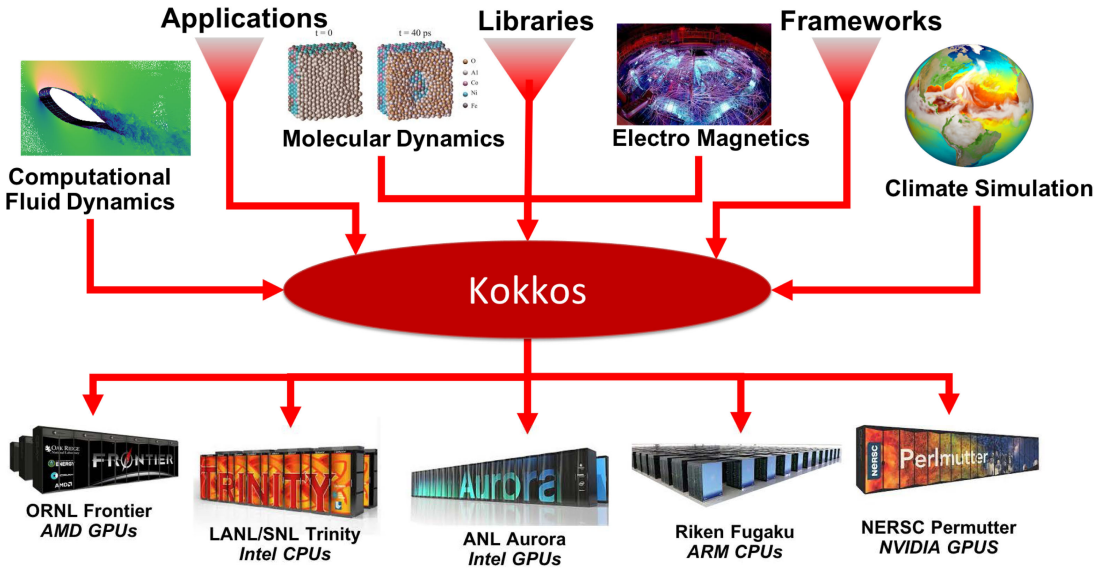

FIGURE 1. Kokkos isolates applications, libraries, and frameworks from the details of underlying hardware. With over 100 projects leveraging it, Kokkos is now used in the vast majority of HPC application domains.

parallelized, and ii) that there are no dependencies between the iterations.

In contrast, *pre*scriptive programming models, such as CUDA, explicitly map loop iterations to threads, limiting the ability to map an algorithm to hardware that is not a close match for this GPU centric model.

The Kokkos semantics on the other hand allow the parallel loop to be threaded, vectorized, mapped to GPUs, or even pipelined through data flow architectures.

Among the key insights while developing the Kokkos Programming Model were the six Kokkos abstractions for performance portability (see Figure 2). They give the descriptive flexibility to express codes detailed enough to enable effective mapping to diverse sets of architectures.

Consider the following code snippet as an example for how these abstractions allow portable code:

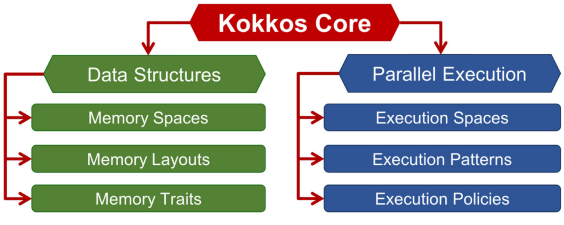

FIGURE 2. Kokkos Core abstractions for performance portability.

```
using exe_t = DefaultExecutionSpace;
using mem_t = exe_t::memory_space;
using in_t = View<double**, mem_t>;
using out_t = in_t::HostMirror;

void foo(in_t contribution,
         out_t host_output) {

 View<double**,mem_t,MemoryTraits<Atomic>>
  output = create_mirror_view(exe_t(),
                              host_output);

 int N = contribution.extent(0);
 int M = contribution.extent(1);

 MDRangePolicy<exe_t, Rank<2>>
  p({0,0},{N,M});

 parallel_for("Label", p,
  KOKKOS_LAMBDA(int i, int j) {
   output(i+1,j) += contribution(i,j);
   output(i,j+1) += contribution(i,j);
 });

 deep_copy(host_output, output);
}
```

First we define what kind of execution mechanism to use. While one could explicitly request for example the `Kokkos::Cuda` *execution space*, in most cases it suffices to request the default execution mechanism. We then define a 2-D array type with the `Kokkos::View` class, using the preferred *memory space* of the chosen *execution space*. After that we define the type for a compatible array on the host. Since the function `foo` takes a device input, but wants the output on the host, we first

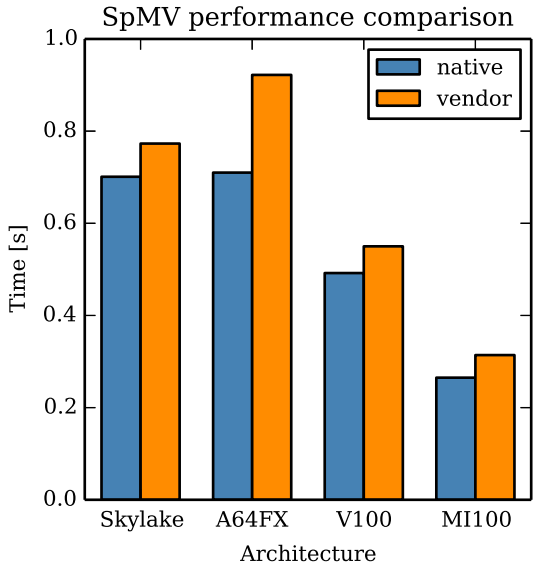

FIGURE 3. Performance of Kokkos Kernels native algorithms (native) and vendor optimized libraries (MKL, ArmPL, cuSPARSE, and rocSPARSE) on two CPU and two GPU architectures for SpMV, showing the performance portability of the kernels across architectures.

create a device version of the output array. When compiling for pure CPU architectures, `output` and `host_output` would alias each other, eliminating unnecessary extra copies. Since the algorithm would have race conditions in the updates, we request the `Atomic` memory trait so that all updates will be done by atomic operations.

Then, an execution policy is created, which uses the execution space to tell the `parallel_for` *execution pattern* how and where to execute. Finally, we use `deep_copy` to move the data back from the device to the host.

Other than the use of potentially separate *memory spaces*, nothing in this code references, utilizes, or expresses any assumptions about hardware properties, and yet it expresses the intent of the developer sufficiently to allow an efficient mapping to a diverse set of architectures.

KOKKOS KERNELS: MATH/GRAPH LIBRARY

Dense and sparse linear algebra functions are critical for solving many physics and engineering problems. As a consequence, vendors provide optimized libraries for these functions: Intel MKL, NVIDIA cuBLAS/cuSPARSE, AMD rocBLAS/rocSPARSE, IBM ESSL/WSMP, and HPE/Cray LibSci. However, several performance portability issues exist in this space. The interfaces, supported operations, and data structures for these libraries are not standardized. Most of the C-based interfaces do not have the type information available in programming models such as Kokkos. These libraries are often only optimized for a single architecture, have restrictions on scalar/ordinal types as well as data layouts, and might not support options like mixed precision computation.

Kokkos Kernels[3] is designed to address these issues by implementing a set of performance portable linear algebra and graph functions using the Kokkos Programming Model. These functions support arbitrary scalar types, mixed precision calculations, and even arbitrary data layouts. The functions can deduce a correct *execution space* from the call arguments, since the Kokkos data structures are *memory space* aware. `View` also contains all its size information, allowing Kokkos Kernels to have a simpler interface than that found in C based math libraries. For example, the ubiquitous Basic Linear Algebra Subroutines' GEMM function requires 13 arguments, including 6 integral parameters describing data layout. The corresponding Kokkos Kernels function only requires 7 arguments. This simplification also reduces inadvertent user errors in the arguments.

Besides providing a generic implementation with maximum flexibility, Kokkos Kernels can also call vendor libraries when available. Figure 3 shows the performance of a sparse matrix-vector product (SpMV) kernel in Kokkos Kernels and vendor libraries on four different architectures. The comparison uses two finite element discretizations focused matrices[b] from the SuiteSparse matrix collection. Note that matrices with different structural properties will lead to varying performance for each of the libraries. In the case of these finite element based matrices, the performance portable Kokkos Kernels SpMV implementation is slightly faster than the vendor variants, for other types of matrices it may be the other way around. Accessing these high performance implementations with a single function call is an effective strategy for applications that need to run on a variety of architectures.

Kokkos Kernels also implements functionality needed by applications that is not generally available in vendor libraries yet. These algorithms are general enough to be useful for more than one application, and are key to their performance. Demonstrating the existence and usefulness of a new function often

[b]https://suitesparse-collection-website.herokuapp.com/MM/Wissgott/parabolic_fem.tar.gz, https://suitesparse-collection-website.herokuapp.com/MM/Botonakis/FEM_3D_thermal2.tar.gz

leads to collaborations with the vendor math library teams, who end up implementing architecture optimized variants in their products. We eventually benefit from such improvements by using them as third party libraries.

Two notable examples are batched and hierarchical dense linear algebra routines. Batched dense linear algebra can be made very efficient using an interleaved layout.[4] After demonstrating the performance benefit of interleaved data layouts, we worked with Intel and ARM who now support interleave-batch functions in MKL and the ARMPL, respectively.

A second example for innovation driven by Kokkos Kernels is the introduction of hierarchical linear algebra kernels. Many applications use hierarchical parallelism, where a subproblem is given to a team of threads. All the primary programming models for GPUs (CUDA, HIP, OpenMP 5.0, and OpenCL) require such expressions when writing algorithms. Kokkos Kernels adds a missing component in this style of programming, allowing the call of linear algebra functions from such a team. Existing vendor libraries mostly support only device scope functions, which cannot be called inside a running kernel. We are working with vendors to support these in the future.

PYTHON AND FORTRAN INTEROPERABILITY

While the Kokkos Programming Model and Kokkos Kernels are C++ libraries, many HPC applications are using languages besides C++. Arguably the most important ones are Python and Fortran. Python is a very powerful and popular language for data analysis and visualization, thus a rapidly growing number of projects are adopting Python as the "driver" layer in computational applications, where performance sensitive sections are implemented in C or C++. The machine learning frameworks PyTorch and TensorFlow are examples of this model.

Fortran is one of the earliest programming languages for numerical computing and is still widely used in HPC. To leverage new architectures however, many Fortran applications offload computational kernels to C++ dialects, such as CUDA. Furthermore, some projects plan to move away from Fortran long term, but have to do so in an incremental fashion.

In order to facilitate these use case scenarios, the Kokkos Ecosystem provides language interoperability capabilities. pyKokkos-base is the fundamental interoperability layer between Kokkos and Python. It provides Python bindings for some critical Kokkos functions—such as `Kokkos::initialize()` and `Kokkos::finalize()`—as well as bindings to Kokkos data structures, `Kokkos::View` and `Kokkos::DynRankView`, so that the Python "driver" can pass data between the performance sensitive sections as well as analyze and visualize the data. Since many existing analysis and visualization packages expect data in the form of a NumPy array, the view data structures can be converted to and from NumPy arrays. pyKokkos-base also provides bindings to Kokkos Tools, which allows users to annotate regions and assign tool callbacks to Python-defined functions.

Similarly, the Fortran Language Compatibility Layer (FLCL) provides Fortran bindings for Kokkos `initialize` and `finalize` as well as facilities to allocate `Views` and wrap existing Fortran arrays. In order to deal with the limitation that C++ bindings in Fortran are actually pure C external functions, FLCL introduces `nd_array` structure, which is a C representation of a `Kokkos::View`.

Together pyKokkos-base and FLCL enable interoperability of Kokkos with Python and Fortran, in a way such that only computational kernels that need to be performance portable are written in C++, while the main application remains Python or Fortran.

An additional new development is the introduction of pyKokkos.[5] This library lets developers write computational kernels directly in Python. The Python interface exposes the majority of the Kokkos Programming Model concepts and capabilities. Providing common semantics in C++ and Python makes it easy to reason about the code and its parallelism constraints. Kernels written that way are automatically translated to C++ Kokkos at runtime. The Python calls to the function are intercepted and replaced by a call to the generated Python binding for the compiled C++ Kokkos variant.

KOKKOS TOOLS: DEBUGGING, PROFILING, AND TUNING

Just as the variety of programming models and architectures motivated the development of Kokkos to provide a unified programming abstraction, the variety of tools for them led to the development of a Kokkos Tools system to unify profiling, debugging, and tuning. In Kokkos, this is achieved with simple instrumentation of constructs in the Core model, and a shared-library based system for loading tools. Critically, this system introduces virtually no overhead when not in use, and does not require recompiling an application to switch or remove tools.

Most importantly, this enables the development of tools that give a unified experience regardless of the

architecture and backend being developed for. Application developers can use Kokkos Tools to see where they spend time and allocate memory, regardless of the supercomputer they are working on, with the exact same interface. For the average developer, this is much more productive than learning a specific tool supported by each hardware vendor.

At the same time, for those advanced developers who do like the powerful platform specific tools, the Kokkos team focuses heavily on codesign efforts with vendors to support Kokkos Tools. For example, many vendor tools initially displayed code performance using the symbol names generated by a compiler. In the case of a code written in a template C++ framework like Kokkos, these names could be hundreds or even thousands of characters long—most reflecting implementation details of Kokkos and not user identifiable names. To address this, the team works with vendors to support the Kokkos instrumentation system to use the labels developers provide as part of the programming model interface.

Another advantage to this approach is that it allows seamless profiling of large software stacks built on top of Kokkos. Consider that the Sandia National Laboratories SPARC application uses the Sierra framework, which uses the Trilinos framework, Kokkos Kernels, and the Kokkos Programming Model. Normally all these projects would need to agree on a common tool instrumentation layer, but thanks to the integrated tools subsystem of Kokkos, that uniform instrumentation is already there.

The most exciting recent work in the tools subsystem is extending this approach from profiling to autotuning. As the architectures and backends that Kokkos has to support proliferate, simple things like maintaining heuristics for runtime parameters of Kokkos dispatches grow less tractable. The autotuning interface is both used to tune internal Kokkos runtime parameters, and can be called directly by user code to tune user level parameters.

KOKKOS SUPPORT

HAVING the best solution for addressing performance portability is not enough however, if potential users cannot easily figure out how to use it. Recognizing this concern, the DOE Exascale Computing Project funded the development of training material, writing documentation and conducting training sessions. Today, this support effort has at its heart three primary thrusts: lectures with associated exercises covering the capabilities of the Kokkos EcoSystem, GitHub based Wikis with API documentation, and continuous online support through a Slack Kokkos community. Being developers of scientific applications ourselves, the Kokkos team approached the support effort primarily through the lens of *what resources did we find most useful when developing code.*

One of the first answers to this question is tutorials. The Kokkos Team developed the Kokkos Lectures for that purpose. Originally taught as a two day tutorial, these lectures now cover all aspects of the Kokkos EcoSystem in over 15 hours of recorded lectures that are accompanied by hands-on exercises. The lectures are available on YouTube with exercises hosted on GitHub—including reference solutions. A content registry is located at https://kokkos.link/the-lectures.

The next thing active users need is an API reference to look up syntax and behavior. For Kokkos, these references are hosted in the respective GitHub wikis for the projects and they are largely modeled after the supremely useful C++ reference found at https://cppreference.com.

And last but not least, all of us considered the ability to just go down the hall to ask a colleague a question very useful. While that is not a particular support model, we found an alternative in the messaging service Slack. The Kokkos Slack channel (https://kokkosteam.slack.com) currently has 600 members, a critical mass that not only makes it worthwhile for Kokkos team members to be constantly present to answer questions, but also is a big enough community that members can help each other. This capability is now considered absolutely critical for sustaining the community.

PERFORMANCE CONSIDERATION

Many Kokkos users have reported performance results in the literature, and providing a comprehensive review of those goes beyond the scope of this article. However, it is well worth summarizing some common lessons learned.

Choosing the right algorithm is the most important factor in achieving good performance. Compared to getting the algorithm right, the choice of programming model is a second order concern. An additional challenge is finding an algorithm that is also performance portable. Sometimes it is trivial given the abstractions Kokkos provides, sometimes finding a performance portable algorithm is difficult, and sometimes there is no performance portable algorithm. It is important to note that even if two specialized algorithms are required, one does not need to leave the Kokkos programming model. One can simply implement two

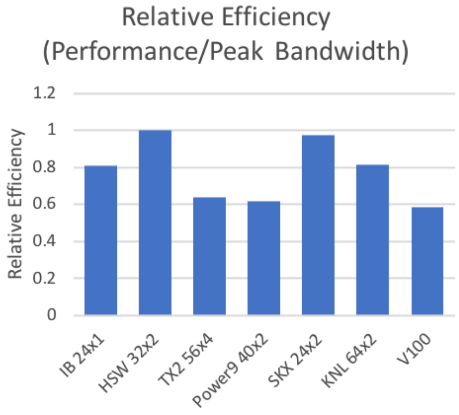

FIGURE 4. Efficiency on various architectures normalized by the efficiency on Intel Haswell CPUs. The data are based on the performance numbers reported by Bertagna *et al.*[6]

different algorithms using Kokkos and choose the right one based on the concurrency of an architecture. This strategy still avoids introducing multiple programming models. Concurrency is generally the main factor for specialization. Some algorithms simply cannot make effective use of 100,000 concurrent GPU threads.

From our experience, somewhere between 50% to 80% of the parallel loops fall into the "trivially performance portable" category. The majority of the rest, which, unfortunately, is often responsible for most of the computation time, falls into the difficult but possible category. Only a very small fraction of problems, accounting for a very small fraction of code lines, cannot be solved with a performance portable solution at all, and one has to implement specialized algorithms.

Another observation is that porting a legacy code to Kokkos often results in a code base that out-performs the original code even on classical CPU systems. Partly that is to be expected when rewriting old code and taking the time to improve it. But Kokkos is also designed to provide guard rails that lead to good coding practices. The Kokkos programming model constrains developers in a way that is beneficial to performance.

One example of such code is HOMMEXX,[6] a climate simulation code that is part of the E3SM Project. The HOMMEXX team compared the performance of the Kokkos based code with the original HOMME, and found it to be on-par or slightly faster on CPU based systems. They also compared single node performance on different architectures. We can use that data to estimate how efficient HOMMEXX is on each architecture. To that end, we take the performance numbers for 1536 elements, and divide it by the peak bandwidth (dual socket DDR bandwidth for CPU nodes, Highbandwidth Memory bandwidth for a single KNL/V100) of each node type. Though this method is a rather crude estimate of achieved efficiency, since it neither takes into account caching nor compute throughput, it is likely a good first approximation for many codes. Normalizing those numbers by the efficiency number of the Haswell node results in a relative efficiency of 0.5 to 1 (see Figure 4), indicating a high level of performance portability is achieved in practice.

RELATED WORK

As mentioned previously, the set of capabilities provided by the Kokkos EcoSystem is also commonly found in vendor solutions. CUDA, HIP, and OneAPI all provide a programming model, math libraries, and tools to analyze performance. *The CUDA* ecosystem only works on NVIDIA hardware. *HIP* does work across a number of hardware platforms—however some of that support is through third party efforts, and not through the primary HIP release. While the programming model is in principle supported on various platforms, that is not necessarily true for the math libraries and the tools, which by-and-large only work for AMD products. *Intel's OneAPI* also comes with a programming model (or actually two, counting both OpenMP and SYCL/DPC++), math libraries and tools. As with HIP, the SYCL/DPC++ programming model has compilers that can target various platforms. The OneAPI math libraries (specifically, oneMKL) has partial support for NVIDIA GPUs in addition to CPUs and Intel GPUs. The tool support in OneAPI is primarily vendor specific.

OpenMP is a vendor independent programming model, and is supported on all the platforms Kokkos supports. It does come with a tool interface—OMPT—similar to the Kokkos Tools interface integration. However, no specific set of math libraries is associated with the OpenMP standard. While many common math libraries do support OpenMP, they do so for older OpenMP standards that only support CPU like architectures. Thus, OpenMP based codes for the exascale architectures need to explicitly leverage each vendor's math library, and support the different interfaces provided by them.

MAINTAINING KOKKOS FOR THE LONG HAUL

A particular concern for application teams considering the adoption of a programming model is the question of its longevity. While such concerns apply to all third party software, programming models are much more invasive into a code base—in particular programming

models, such as Kokkos and SYCL, that also provide fundamental data structures.

The Kokkos team is systematically executing a strategy to alleviate these concerns: i) the Kokkos EcoSystem is developed publicly under a permissive licence; ii) the Kokkos team actively works on expanding the developer team across institutions; iii) the team works on integrating some of the most critical components into the ISO C++ standard; and iv) backwards compatibility is paramount.

Using a permissive license is the basis for building an Open Source community. It makes it easier for external developers to contribute, and also opens up the fail safe option for users to be able to take all the source code and continue developing it on their own, should the original Kokkos team cease to exist.

The original Kokkos team at Sandia recognized that there is strength in numbers. Working with funding agencies, it worked to diversify the team and build core competency at the laboratories running leading US super computers. That effort resulted in half of today's contributions to the Kokkos EcoSystem being developed outside of Sandia. Currently Kokkos is developed by that team through an informal community software development approach, however efforts are underway to draft a more formal project governance processes for the post-ECP era.

In addition to the software development efforts, the Kokkos team works on making widely used concepts and capabilities from the Kokkos EcoSystem part of the ISO C++ standard. This includes helping to shape future heterogeneous computing support in C++, the introduction of atomic capabilities needed by HPC in C++20, the proposal of a multidimensional array class with layouts and accessor extensions for C++23 and the development of a linear algebra interface for the C++ standard. While these efforts will not eliminate the need for the Kokkos EcoSystem, making some hard-to-optimize core capabilities part of the standard will help lessen the maintenance burden of the Kokkos team, and increase the sustainability of the overall effort in the long term.

Last but not least the Kokkos team has strict policies regarding backward compatibility. While we expect the need to occasionally break backward compatibility, such changes are limited to major release versions occurring every few years. Previews of these changes are available long in advance, allowing code teams to adopt changes gradually. In practice, necessary changes in application codes have been minimal or even nonexistent. In fact, tutorial examples from 2016 require only two changes—renaming dimension to extent, and making sure that allocations are freed before `Kokkos::finalize`—in order to work with the latest Kokkos on AMD GPUs using the HIP backend, which did not exist at the time. Notably, neither change breaks compatibility with the Kokkos version from 2016.

Kokkos' stated goal is to provide reliable performance portability for all significant HPC platforms today and in the future. By building an Open Source community with a well-established presence across some of the most important HPC institutions and a strong engagement in the ISO C++ committee, the Kokkos EcoSystem is set up to achieve that goal for the long haul.

CONCLUSION

Performance portability is a critical need for today's HPC applications. But to achieve true performance portability—in particular without loosing much productivity—it is not enough to have a programming model that can target diverse architectures. One needs math libraries, tools, language interoperability, training material and support, well integrated with the programming model: developers need and deserve a Performance Portable EcoSystem. Kokkos provides such an EcoSystem through a community effort led by a team from a diverse set of DOE laboratories. Pushed forward by a decade of application-feedback driven development, Kokkos is now a mature solution, which can meet the needs of the most complex application areas, and provide practical performance portability today.

ACKNOWLEDGMENTS

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA-0003525. Los Alamos National Laboratory is operated by Triad National Security LLC for the U.S. Department of Energy under contract 89233218CNA000001. Approved for unlimited release LA-UR-21-25,501. This manuscript has been authored by UT-Battelle, LLC, under contract DE-AC05-00OR22725 with the US Department of Energy (DOE).

This research was supported 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.

REFERENCES

1. S. Plimpton, "Fast parallel algorithms for short-range molecular dynamics," *J. Comput. Phys.*, vol. 117, no. 1, pp. 1–19, 1995, doi: 10.1006/jcph.1995.1039.
2. H. Edwards, C. Trott, and D. Sunderland, "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," *J. Parallel Distrib. Comput.*, vol. 74, no. 12, pp. 3202–3216, 2014, doi: 10.1016/j.jpdc.2014.07.003.
3. S. Rajamanickam *et al.*, "Kokkos kernels: Performance portable sparse/dense linear algebra and graph kernels," 2021, *arXiv:2103.11991*.
4. K. Kim *et al.*, "Designing vector-friendly compact blas and lapack kernels," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2017, pp. 1–12, doi: 10.1145/3126908.3126941.
5. N. A. Awar, S. Zhu, G. Biros, and M. Gligoric, "A performance portability framework for python," in *Proc. ACM Int. Conf. Supercomput.*, 2021, pp. 467–478, doi: 10.1145/3447818.3460376.
6. L. Bertagna *et al.*, "Hommexx 1.0: A performance-portable atmospheric dynamical core for the energy exascale earth system model," *Geoscientific Model Develop.*, vol. 12, no. 4, pp. 1423–1441, 2019, doi: 10.5194/gmd-12-1423-2019.

CHRISTIAN TROTT is a Principal Member of technical staff and Sandia National Laboratories, where he has worked since acquiring a Ph.D. degree in theoretical physics at TU Ilmenau, Germany. He leads the Kokkos Core Project and represents Sandia on the ISO C++ Committee. Contact him at crtrott@sandia.gov.

LUC BERGER-VERGIAT is a limited term employee at Sandia National Laboratories. He co-leads the Kokkos Kernels projects. Contact him at lberge@sandia.gov.

DAVID POLIAKOFF has spent seven years working at various DOE National Laboratories working on tools in multiphysics applications. He currently leads the Kokkos Tools effort. Contact him at dzpolia@sandia.gov.

SIVASANKARAN RAJAMANICKAM is a Principal Member of technical staff at Sandia National Laboratories. He leads Kokkos Kernels and Trilinos solver projects. Contact him at srajama@sandia.gov.

DAMIEN LEBRUN-GRANDIÉ is a Computational Scientist with Oak Ridge National Laboratory. He co-leads the Kokkos Core Project and represents ORNL on the ISO C++ Standards Committee. Contact him at lebrungrandt@ornl.gov.

JONATHAN MADSEN is an Application Performance Specialist for the National Energy Research Scientific Computing Center, Lawrence Berkeley National Laboratory. He is a Developer for the Kokkos Core project, leads the development of a modular toolkit for software monitoring at LBNL (timemory), and represents LBNL on the ISO C++ standards committee. Contact him at jrmadsen@lbl.gov.

NADER AL AWAR is currently working toward the Ph.D. degree with the Department of Electrical and Computer Engineering, University of Texas at Austin, Austin, TX, USA. He works on making HPC more accessible from tools and productivity languages. Contact him at nader.alawar@utexas.edu.

MILOS GLIGORIC is an Assistant Professor with the Department of Electrical and Computer Engineering, University of Texas at Austin, Austin, TX, USA. His research interests include software engineering and formal methods with focus on improving software quality and developers' productivity. Contact him at gligoric@utexas.edu.

GALEN SHIPMAN is a Scientist with Los Alamos National Laboratory, where he leads a next-generation programming models integration project as part of the Exascale Computing Project. Contact him at gshipman@lanl.gov.

GEOFF WOMELDORFF is a Scientist with Los Alamos National Laboratory, where he develops HPC applications for parallel architectures. Contact him at womeld@lanl.gov.