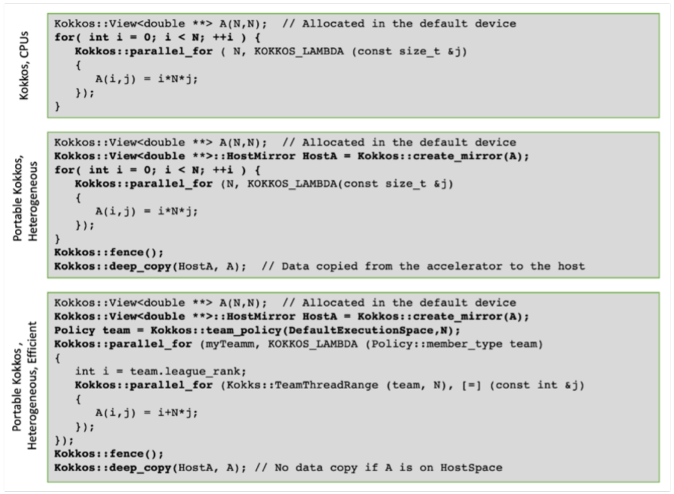
SAND2022-10988C

Toward Automatic Test Synthesis for Performance Portable Programs

Keita Teranishi*, Shyamali Mukherjee*, Richard Rutledge†, Samuel D. Pollard*, Nicolas Morales*, Noah Evans*, Alessandro Orso†, and Vivek Sarkar†
*Sandia National Laboratories, CA, USA †Georgia Institute of Technology, GA, USA
Email: {knteran,smukher,spolla,nmmoral,nevans}@sandia.gov, {rrutledge,orso,vsarkar}@gatech.edu

I. Motivation

The unprecedented increase in parallelism and the corresponding complexity of HPC systems imposes a formidable burden on application developers and users to verify that their code meets the performance, correctness, and reliability requirements for conducting scientifically-relevant simulations and analyses. Large-scale DOE applications, for instance, execute on different vendor platforms containing thousands of nodes with heterogeneous accelerators and must exploit multiple levels of parallelism for high performance. The combination of platform diversity, extreme heterogeneity, and massive scales of parallelism vastly increases the set of possible dynamic behaviors of HPC applications, thereby making testing and verification extremely challenging.


Kokkos, CPUs
Portable Kokkos, Heterogeneous
Portable Kokkos , Heterogeneous, Efficient

```
Kokkos::View<double **> A(N,N);  // Allocated in the default device
for( int i = 0; i < N; ++i ) {
   Kokkos::parallel_for ( N, KOKKOS_LAMBDA (const size_t &j)
   {
      A(i,j) = i*N*j;
   });
}
```

```
Kokkos::View<double **> A(N,N);  // Allocated in the default device
Kokkos::View<double **>::HostMirror HostA = Kokkos::create_mirror(A);
for( int i = 0; i < N; ++i ) {
   Kokkos::parallel_for (N, KOKKOS_LAMBDA(const size_t &j)
   {
      A(i,j) = i*N*j;
   });
}
Kokkos::fence();
Kokkos::deep_copy(HostA, A);  // Data copied from the accelerator to the host
```

```
Kokkos::View<double **> A(N,N);  // Allocated in the default device
Kokkos::View<double **>::HostMirror HostA = Kokkos::create_mirror(A);
Policy team = Kokkos::team_policy(DefaultExecutionSpace,N);
Kokkos::parallel_for (myTeamm, KOKKOS_LAMBDA (Policy::member_type team)
{
   int i = team.league_rank;
   Kokkos::parallel_for (Kokks::TeamThreadRange (team, N), [=] (const int &j)
   {
      A(i,j) = i+N*j;
   });
});
Kokkos::fence();
Kokkos::deep_copy(HostA, A); // No data copy if A is on HostSpace
```

Fig. 1. Incremental Addition to Kokkos Code Source toward portable and heterogeneous computing.

To address this increasing complexity of HPC application development, performance portable programming models, such as Kokkos [1] and Raja [2], provide common platform-independent abstractions for data representation and parallel loops. By doing so, these models shield the users from addressing the details of the underlying node architecture, while still achieving performance comparable to those of applications implemented using platform-specific languages or frameworks. Ideally, the users gain a significant productivity improvement in their application development from only maintaining one program source to support multiple architectures. However, designing test cases and applications for these programming frameworks is still a tedious manual process because their application programming interface (API) specifications and their behaviors require a detailed understanding of the architecture on which the application is to be deployed.

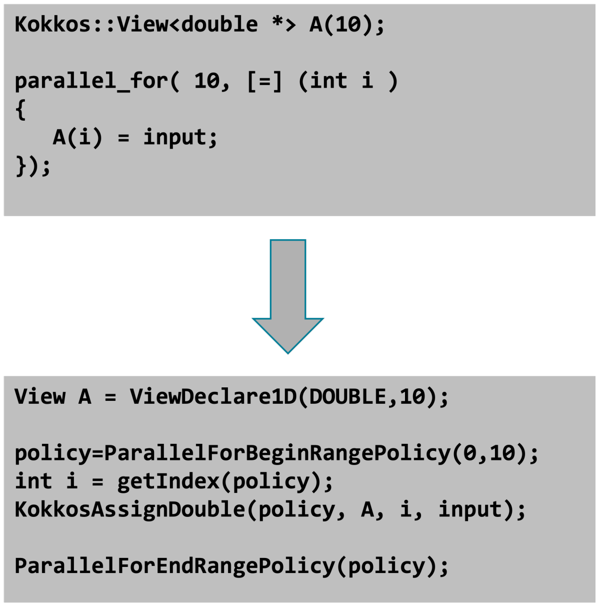
For example, only a few basic Kokkos APIs are needed to parallelize computations on CPU platforms. For convenience, many applications are typically developed on traditional CPU-based architectures at their initial stage. Adapting the initial implementation to heterogeneous systems for improved performance portability involves incremental additions of advanced API calls, as illustrated in Figure 1. A side effect of this approach is that writing robust portable test cases for applications that use these more advanced APIs becomes even more daunting than before, leading to either poor test coverage or Herculean developer efforts. Furthermore, many application developers and domain scientists may have limited experience with respect to modern software engineering practices in C++. As a result, their codes may exhibit subtle bugs related to the use of these advanced APIs, thereby further motivating the need for automatic test generation.

Typically, only a small fraction of the source code of Kokkos applications contains calls to Kokkos APIs, but the templated nature of C++ hides the true complexity of Kokkos behavior in scientific applications. As a performance portability layer, Kokkos adapts and extends program behavior to current and future runtimes by specializing the behavior of a scientific program to use a desired set of concrete parallel runtime APIs for a given platform. However, widespread use of template instantiation makes it hard to separate the behavior of Kokkos runtime instantiations from the underlying scientific application. Thus, any attempt to apply automated static analysis and traditional test generation techniques must explore the state space and the behavior not just of the scientific application but also of each instantiation of individual Kokkos runtime behavior models. Consequently, any automated test framework used to test these kinds of applications will typically require runtime-specific test cases to test all behaviors of an application on different platforms.

II. Our Approach

Our framework, called KLOKKOS, will use KLEE [3], a well-known automatic symbolic execution framework built on

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.



```
Kokkos::View<double *> A(10);

parallel_for( 10, [=] (int i )
{
   A(i) = input;
});
```

```
View A = ViewDeclare1D(DOUBLE,10);

policy=ParallelForBeginRangePolicy(0,10);
int i = getIndex(policy);
KokkosAssignDouble(policy, A, i, input);

ParallelForEndRangePolicy(policy);
```

Fig. 2. An Example of Code Transformation through Clang AST. All modern C++ features are converted to C-like functions to simplify the symbolic analysis.

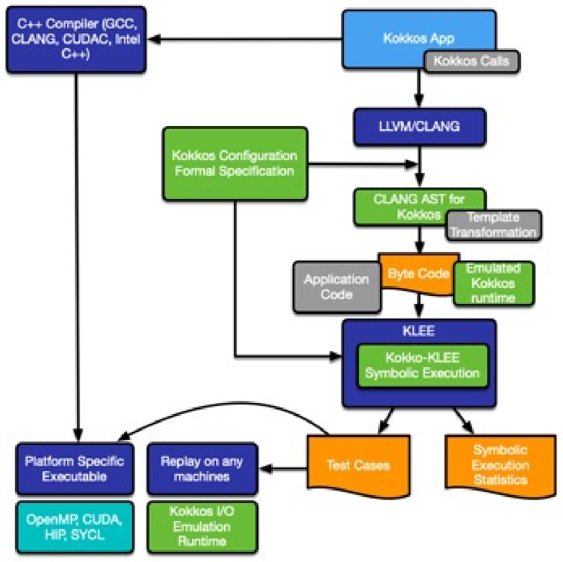

C++ Compiler (GCC, CLANG, CUDAC, Intel C++)
Kokkos App
Kokkos Calls
LLVM/CLANG
Kokkos Configuration Formal Specification
CLANG AST for Kokkos
Template Transformation
Application Code
Byte Code
Emulated Kokkos runtime
KLEE
Kokko-KLEE Symbolic Execution
Platform Specific Executable
Replay on any machines
Test Cases
Symbolic Execution Statistics
OpenMP, CUDA, HIP, SYCL
Kokkos I/O Emulation Runtime

Fig. 3. The Klokkos testing framework. Our proposed work is indicated by green boxes. The grey boxes are existing elements our framework leverages.

top of the LLVM compiler infrastructure, to address the aforementioned challenges. To support the testing of applications written for heterogeneous computing systems, we propose to adapt KLEE to analyze HPC applications written in Kokkos. A critical challenge in test generation is limiting the number, size, and complexity of the generated tests, but Kokkos' APIs and abstraction exploit modern templated C++, which can generate a large number of executable variants. To address this problem, we leverage our own (1) Kokkos formal specifications and (2) compiler (Clang-AST) source code transformation to simplify the way in which Kokkos operations are processed by our toolchain. Our Kokkos specification provides a rigorous formal semantics that makes it possible for domain scientists and application developers to separate the testing of scientific applications from the testing of the behavior of specific runtimes.

Specifically, our source code transformation converts all Kokkos operations into C-like function calls (see Figure 2), allowing KLEE to capture and interpret such operations at a higher level, that is, before they are lowered into low-level LLVM IR operations. This separation of concerns can greatly reduce the execution time and resources required for test generation, and make computationally expensive formal methods practical by limiting state space exploration to the subset relevant to a given scientific application. Invariants specifying the behavior of the performance portability layer and property-based testing will be used not only to generate tests but also to identify bugs and potential sources of non-deterministic behavior. As illustrated in Figure 3, our KLOKKOS framework, a new test generation framework for HPC based on dynamic symbolic execution, combines advances in compiler technology and formal methods, aiming to generate test cases using a combination of concrete and symbolic execution.

In the presentation, we will discuss the design of the KLOKKOS framework and its current implementation status. We will also illustrate our approach by showing how it can address some common bugs in Kokkos programs, such as data consistency with heterogeneous node architectures and race conditions. Finally, we will discuss how our Kokkos formal specification and specialized testing strategies (e.g., differential testing) will allow us to create a comprehensive testing framework for performance portable programs.

Acknowledgement

This material is based upon work supported by the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research, under Award Number DE-FOA-0002460. 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-NA0003525.

References

[1] C. R. Trott, D. Lebrun-Grandie, D. Arndt, J. Ciesko, V. Dang, N. Ellingwood, R. Gayatri, E. Harvey, D. S. Hollman, D. Ibanez *et al.*, "Kokkos 3: Programming model extensions for the exascale era," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 4, pp. 805–817, 2021.
[2] R. Hornung and J. Keasler, "The RAJA portability layer: overview and status," Lawrence Livermore National Laboratory, Tech. Rep. LLNL-TR-661403, 2014.
[3] C. Cadar, D. Dunbar, and D. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'08. USA: USENIX Association, 2008, p. 209–224.