SAND2022-0409C

Sandia National Laboratories

Exceptional service in the national interest

Automated Test Generation for Performance Portable Programs Using Clang/LLVM and Formal Methods

Keita Teranishi, Noah Evans, Sam Pollard, Shyamali Mukherjee, Alessandro Orso, Vivek Sarkar

Jan. 14, 2022. DOE ASCR XSTACK Kick-Off Meeting

Team Members

Keita Teranishi (Sandia National Labs, PI)

Noah Evans (Sandia National Labs)

Shyamali Mukherjee (Sandia National Labs)

Samuel Pollard (Sandia National Labs)

Alessandro Orso (Georgia Tech)

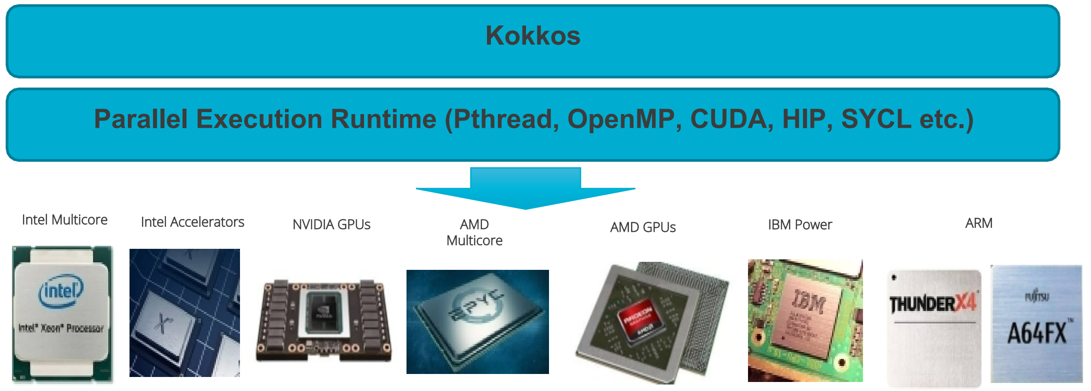
Vivek Sarkar (Georgia Tech, Institutional-coPI)

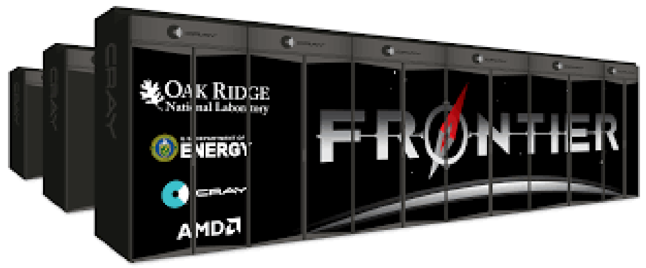
Outline

- Motivation
  - Performance Portable Programming Model
- Our Approach
  - Klokkos Automatic test case generation framework
- Project Highlights
  - Formal Specification
  - Compiler Assisted Checking/Testing
  - Differential Testing
- Risks
- Q&A

Motivation

- Performance Portable Programming Models
  - Standard for DOE HPC Systems (Kokkos, Raja)
  - Heterogeneity
  - Performance Tuning
- Writing performance portable code with correct behavior is challenging
  - Templated C++ Programming
  - Need to test all heterogeneous HPC systems
  - Availability of Future computing platforms
  - Some “vague” specifications
    - The code runs on CPUs, but crashes on GPUs
    - Get correct results on CPUs, but wrong on GPUs
- Our Goal
  - Create automatic testing framework for Performance Portable Programming Model





Kokkos Performance Portable Programming

Serial

```
double A[100][100];
for (size_t i = 0; i < N; ++i)
{
  for (size_t j = 0; j < N; ++j ) {
     A[i][j] = i+N*j;
  }
}
```

Kokkos

```
Kokkos::View<double **> A(100,100);  // Allocated in the default device
Kokkos::View<double **>::HostMirror HostA = Kokkos::create_mirror(A);
Policy team = Kokkos::team_policy(CUDA,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);  // Data copied from the accelerator to the host
```

- Single Source for Multiple Platforms!

Writing Portable and Correct Program Across Multiple Platform

- Kokkos allows non-portable implementation.
  - A program can run on CPUs, but crashes on GPUs.
  - A program can run correctly on CPUs but incorrect on GPUs.
- Writing correct and portable code requires good knowledge of multiple platforms.
  - It is not what Kokkos is intended for.

Kokkos, CPUs

```
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;
   });
}
```

Portable Kokkos, Heterogeneous

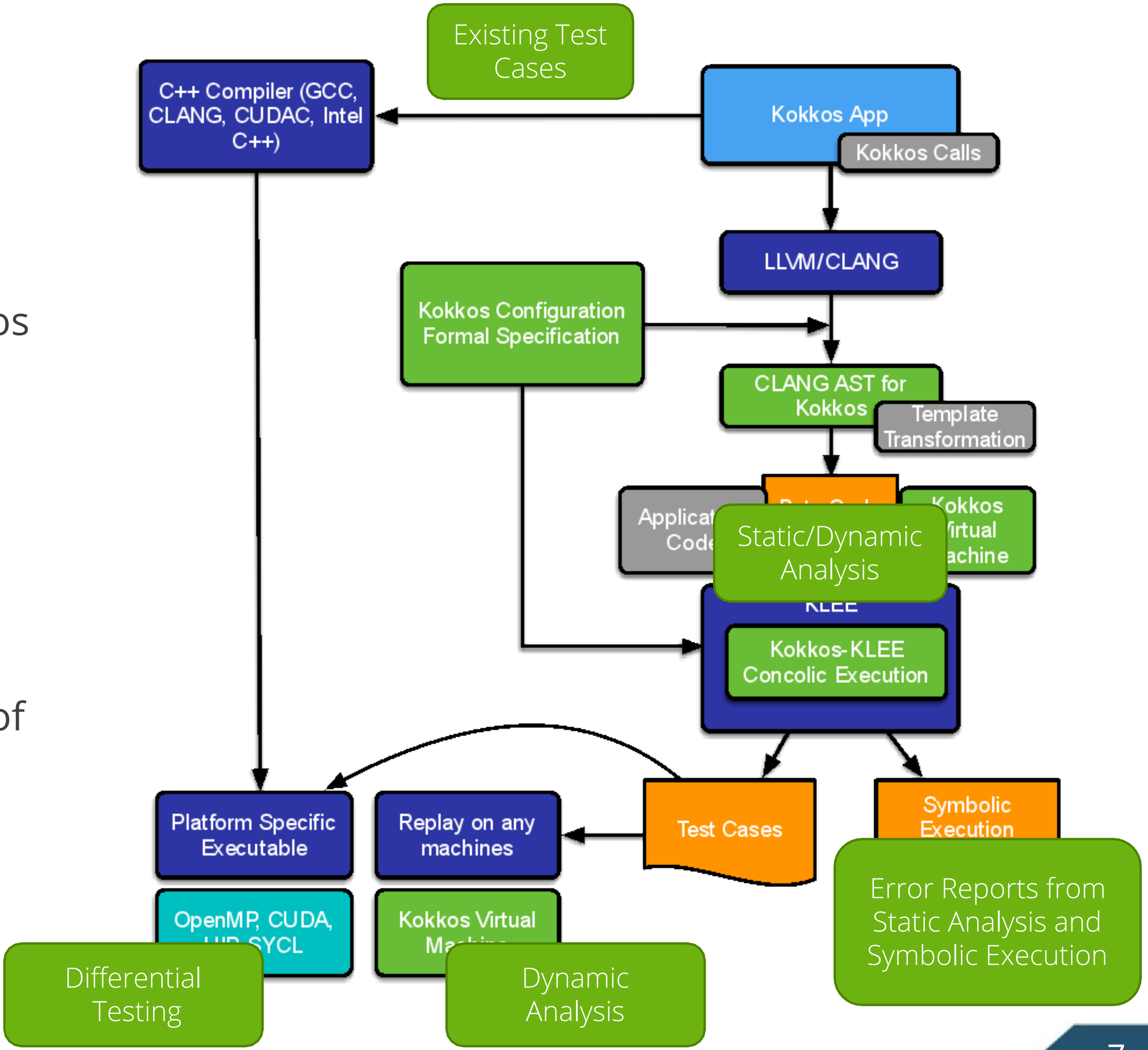
```
Kokkos::View<double **> A(N,N);  // Allocated in the default device
Kokkos::View<double **>::HostMirror HostA = Kokkos::create_mirror(A);
for( int i = 0; i < N; ++i ) {
   Kokkos::parallel_for (N, KOKKOS_LAMBDA(const size_t &j)
   {
      A(i,j) = i*N*j;
   });
}
Kokkos::fence();
Kokkos::deep_copy(HostA,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

Portable Kokkos, Heterogeneous, Efficient

```
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
```

Our Approach: Auto test-code Generation Framework

- **KLOKKOS**
  - Inspired by KLEE Project https://klee.github.io
  - Use the source code analysis capability of Clang-LLVM
  - Establish a portable formal specification of Kokkos APIs for model checking.
  - Apply multiple code testing techniques together to **reduce search space**
    - Static analysis
    - Symbolic (concolic) analysis
    - Dynamic analysis
    - Differential testing
  - Automatic Test Generation for "suspicious" part of program source
  - Ultimately, users **do not access the target platforms** to check the correctness of their Kokkos programs.



Our Approach

- Formal Specification of Kokkos
- Kokkos Virtual Machine
- Automatic Testing through Symbolic Execution
  - Compiler Analysis (Clang AST/LLVM-IR)
    - Static Analysis
  - Concolic Testing (AKA Dynamic Symbolic Execution)
  - Differential Testing
  - Dynamic Analysis
- Test Programs

Formal Specification (Pollard)

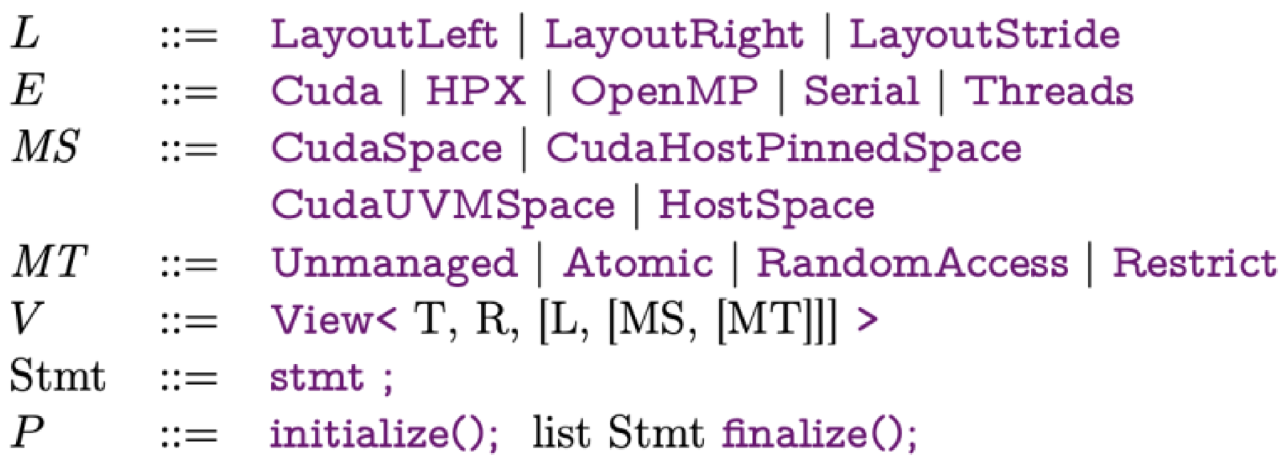
- Work by John Jacobson (U. of Utah) and Sam Pollard (SNL-CA)
- Goal: Write an operational semantics for Kokkos
  - Do this by translating Kokkos' behavior into inference rules
- Will codify the canonical behavior of Kokkos to inform tools and developers
- Decided on small-step operational semantics to granularly describe concurrency
- Process consists of carefully reading wiki, code examples, and Kokkos source and talking with developers to get an overview, then translating these into inference rules

$$\frac{N : \mathbb{N} \quad P : \mathrm{ExecutionPolicy} \quad Fn : \mathrm{Functor} \quad S : \mathrm{Stmt}}{\mathrm{ParallelFor}(N, P, Fn); S \to \mathrm{dispatch}(N, P, Fn) \,||\, S}$$

An example inference rule stating that `Kokkos::parallel_for` is asynchronous

Formal Specification Progress

- Insight: Machine-readable specs are not at the right abstraction levels (e.g. CIVL, murphi)
- Progress
  - A set of inference rules capturing Kokkos datatypes and operations
- Challenges
  - What level of granularity?
    - Modeling the entire C++ semantics is infeasible
  - How to model concurrency?
    - Start with Communicating Sequential Processes (CSP), but Kokkos concurrency is more restricted than that
    - Assume very little to work across all architectures
    - Multiple execution space instances in the latest Kokkos-3.5.



$$\begin{array}{lcl}
L & ::= & \texttt{LayoutLeft} \mid \texttt{LayoutRight} \mid \texttt{LayoutStride} \\
E & ::= & \texttt{Cuda} \mid \texttt{HPX} \mid \texttt{OpenMP} \mid \texttt{Serial} \mid \texttt{Threads} \\
MS & ::= & \texttt{CudaSpace} \mid \texttt{CudaHostPinnedSpace} \\
 & & \texttt{CudaUVMSpace} \mid \texttt{HostSpace} \\
MT & ::= & \texttt{Unmanaged} \mid \texttt{Atomic} \mid \texttt{RandomAccess} \mid \texttt{Restrict} \\
V & ::= & \texttt{View<}\ \mathrm{T, R, [L, [MS, [MT]]]}\ \texttt{>} \\
\mathrm{Stmt} & ::= & \texttt{stmt ;} \\
P & ::= & \texttt{initialize();}\ \ \mathrm{list\ Stmt}\ \texttt{finalize();}
\end{array}$$

An early draft of the grammar

Kokkos Virtual Machine (Evans)

Biggest problem in symbolic testing: state explosion

Many testing frameworks don't test all of runtime and system libraries

- unimportant, application state space matters
- solution: write a simplified virtual machine which can model the runtime while minimizing the implementation details that need to be explored by the framework.

Goal: Embody Kokkos Formal Semantics in a virtual machine such that the developer can symbolically test their application in tractable time and have a reasonable degree of assurance that their application will work correctly.

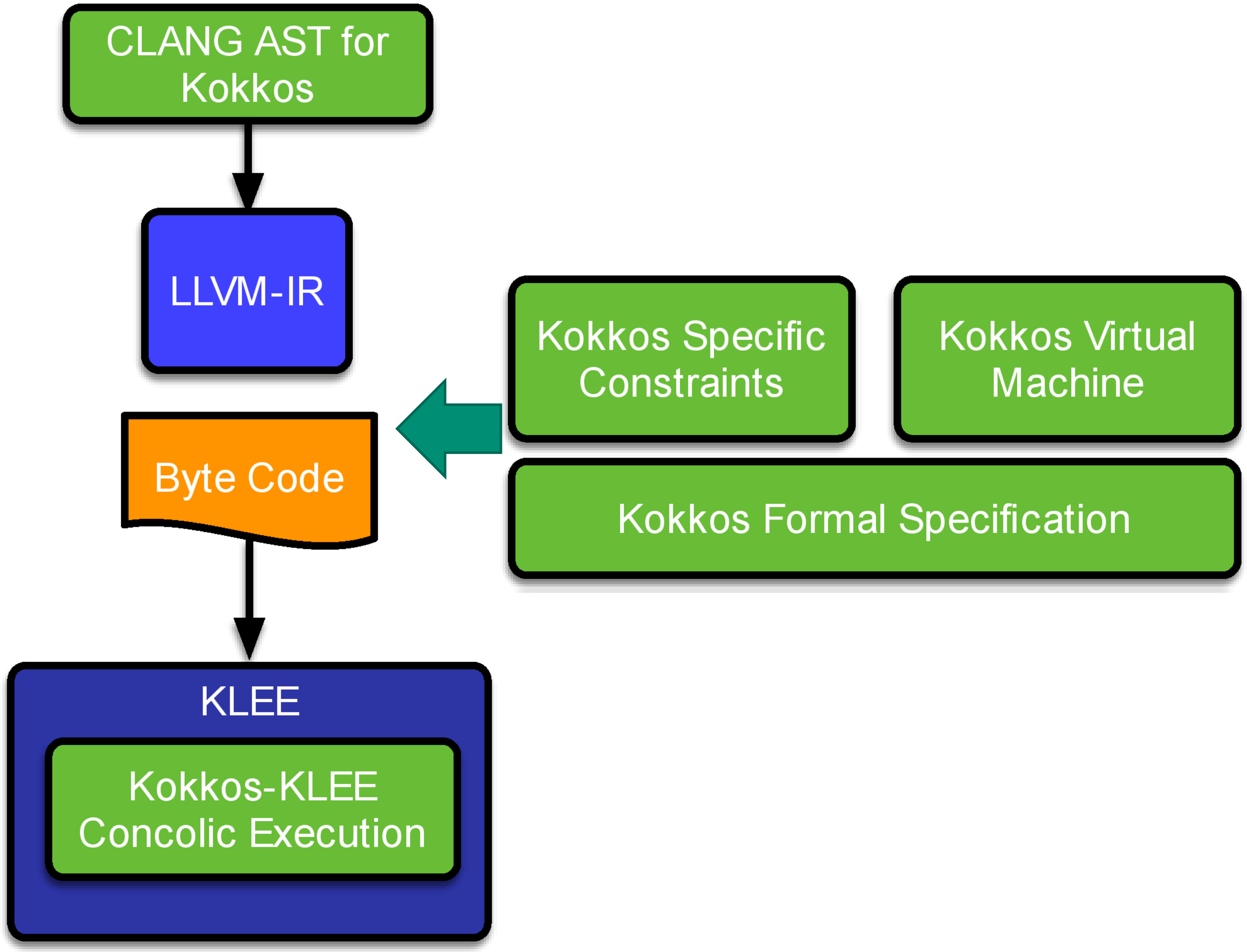
Technical Details: Catch template instantiations at the AST level and replace them with library calls to the virtual machine. The virtual machine is an interpreter for the formal Kokkos semantics (sequential initially). This abstracts away the Kokkos implementation while ensuring correct use.

Testing Coverage Overview

TEST TYPE	Important Techniques	Behavior and Platform Coverage
Static Analysis	Formal Specification, Compiler AST	All possible executions paths and platforms
Dynamic Analysis	Compiler, Kokkos Virtual Machine	One input for multiple platforms
Concolic Testing	Formal Specification, Compiler, SMT Solver, Kokkos Virtual Machine	All possible execution paths and platforms for a subset of concrete inputs
Differential Testing	Knowledge Base, Kokkos Virtual Machine	Heterogeneous, Application-Driven (Sequential VS Parallel)

Concolic Testing (Mukherjee)

- Our Approaches are:
  - Address the application state through **Kokkos** API and Class (View) level
  - Focus on Kokkos View's **metadata** (template properties, array sizes) rather than their numerical contents
  - Use inputs to force code to symbolically execute towards path of Exception or Error return block
  - **CLANG AST** has the capability to source match, which will very valuable for testcase validation and correctness.
  - Feasible to control state explosion using bias in terms of function name, instead of symbol or branch name in assembly code generated by **LLVM-IR serving Kokkos VM**


CLANG AST for Kokkos
LLVM-IR
Byte Code
Kokkos Specific Constraints
Kokkos Virtual Machine
Kokkos Formal Specification
KLEE
Kokkos-KLEE Concolic Execution

Partial Symbolic Execution (Orso and Sarkar): Overview

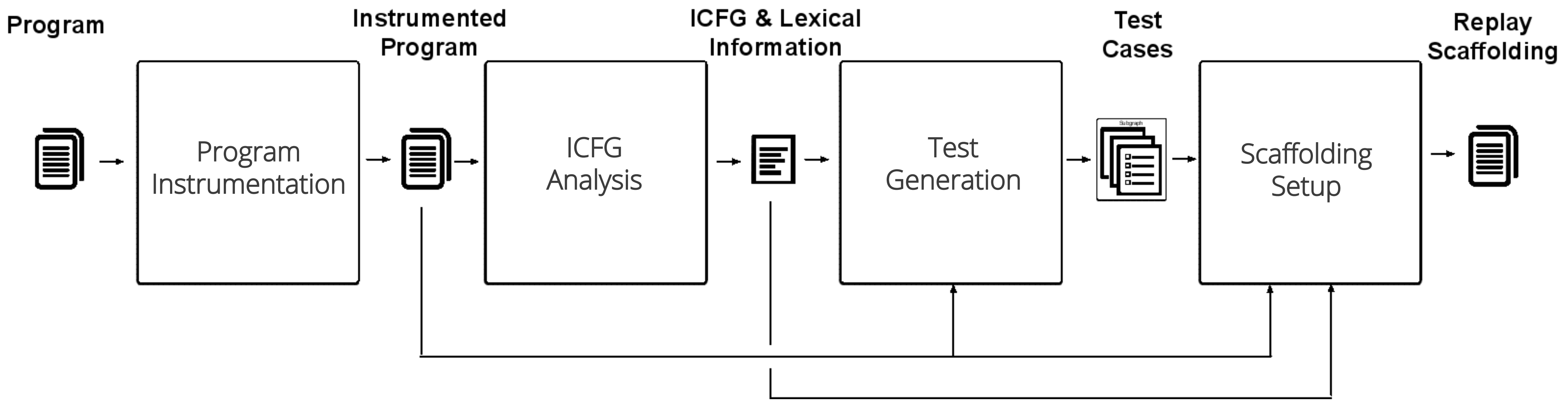
- Symbolic execution is a powerful and sophisticated test-input generation strategy but has many limitations
  - Highly structured inputs
  - Constraint solver limitations (theories, sheer complexity)
  - External libraries, whether symbolically executed or modeled (large, inaccessible)
  - Path explosion

➡ Limited coverage, affecting tasks that depend on it (testing, dynamic analysis)

- To mitigate this problem, we propose *Partial Symbolic Execution* [1,2]
  - Consider increasingly smaller parts of the program
  - Utilize scaffolding to execute program fragments
  - Different strategies:
    - Symbolic input state: Traditional SE
    - Symbolic external state: Execution of each function with symbolic input parameters and symbolic global state
    - Symbolic stubs: Execution of each function with symbolic input parameters, symbolic global state, and all callees replaced by symbolic stubs

[1] R. Rutledge and A. Orso, “PG-KLEE: Trading Soundness for Coverage”, 42nd IEEE and ACM SIGSOFT International Conference on Software Engineering (ICSE, demo track), 2020.
[2] F. Ye, J. Zhao, and V. Sarkar. “Detecting MPI usage anomalies via partial program symbolic execution” International Conference for High Performance Computing, Networking, Storage, and Analysis (SC), 2018

Partial Symbolic Execution: Technique



- Program Instrumentation: identify *paths of interest* and performs some refactoring
- ICFG Analysis: collect information to support Input Generation and Scaffolding Setup
- Test Generation: perform symbolic execution
- Scaffolding Setup: leverage the Kokkos Virtual Machine to put in place drivers and stubs needed to run the generated tests on code fragments

Differential Testing

- Leverage existing inputs to identify potential issues that results in different behaviors [1]
  - Between serial and parallel executions of Kokkos code (fragments)
  - Between versions compiled for different platforms (e.g., CPUs (OpenMP), NVIDIA CUDA, AMD ROCm, SYCL)
- Our approach
  - Serial executions: run Kokkos code using the "sequential execution space".
  - Different platforms: run the code compiled for different targets as described in our overview.
  - For each pair of versions v1 and v2 and for each input i:
    - Run v1(i) and v2(i)
    - Compare their output
    - Report an issue in case of discrepancies
- Additional details
  - The oracle can be defined at different levels of detail (I/O, return values, internal state)
  - The Kokkos Virtual Machine can be used to run the tests while tracking I/O and internal state

[1] R. Rutledge and A. Orso, "Automating Differential Testing with Overapproximate Symbolic Execution", 15th IEEE International Conference on Software Testing, Verification and Validation (ICST), 2022

Dynamic Analysis (Sarkar)

- Concurrency bugs and erroneous memory accesses may occur in Kokkos programs, e.g., Data races, Uses of uninitialized memory (UUM), Uses of stale data (USD), Buffer overflows
- Our approach to dynamic analysis
  - Use test cases generated via partial symbolic execution as inputs
  - Execute instrumented Kokkos program on Kokkos Virtual Machine
  - Build on our past work on dynamic analysis of data mapping issues in OpenMP [1] to identify illegal data accesses and data mapping issues using runtime information collected on the host, e.g.,
    - Happens-before relations between the host program and kernels
    - View accesses with offsets and lengths
  - Perform dynamic analysis on host using summary information collected on devices
    - Data race ➡ happens-before analysis between memory accesses
    - UUM & USD ➡ states checking on the accessed view
    - Buffer overflow ➡ bounds checking on the accessed view

[1]: Yu, Lechen, Joachim Protze, Oscar Hernandez, and Vivek Sarkar. "ARBALEST: Dynamic Detection of Data Mapping Issues in Heterogeneous OpenMP Applications." 2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS). IEEE, 2021.

Dynamic Analysis Example

- A buggy example

```
Kokkos::initialize(argc,argv);
Kokkos::View<double *>  A("A",100);
Kokkos::deep_copy( A, 1.0 );
```

UUM & Buffer Overflow Detected

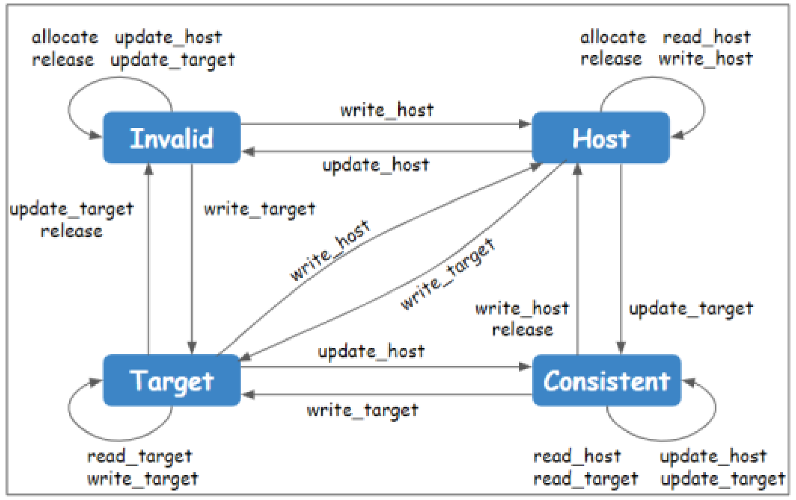
```
// A buffer overflow on A
Kokkos::parallel_for(200, KOKKOS_LAMBDA(const int i)
{
   A(i) = static_cast<double>(i);
});
Kokkos::finalize();
```

- Dynamic analysis for UUM and buffer overflow

Create a shadow memory for view A (length: 100)

Mark the state of all A's elements as "Host"

- "Host" denotes the view has valid value on the host



Launch states and bounds check on shadow memory

For i in 0 – 99
- States check: pass
- Bounds check: pass

For i in 100 – 199
- States check: **fail**
- Bounds check: **fail**

Test Cases

- Mini Applications
  - ExaMiniMD (ECP MiniApp Collection)
  - MiniFE, MiniAero (Mantevo Collection)
  - NimbleSM (Mini Multi-Physics App, Funded by ASC)

- Collection of Kokkos Mistakes
  - Team is writing a suite of common Kokkos mistake examples
  - Team is contacting Kokkos application developers to collect more examples and bug report related to Kokkos programming mistakes.

RISKS

- Availability Kokkos Application Program
  - Scarcity of Example Programs
  - Mini Applications (Mantevo, NimbleSM, Exascale Mini-App Projects) may not be sufficient to reflect real applications
  - Large real application is hard to test due to the requirements other than Kokkos (other TPLs, difficulty of software build)
- (Mitigation)
  - Partial symbolic analysis address the size and complexity of program source
  - Curation of Kokkos-mistake program source in collaboration with Kokkos, Trilinos and other Kokkos-users (in ECP and Sandia)

- Integration of all ideas
  - Tools to represent formal specification
  - SMT Solvers
  - Kokkos Virtual Machine
- (Mitigation)
  - Maximize software reuse from well-established frameworks (e.g. KLEE)

Project Plan Year 1

- Kokkos Virtual Machine Development

- Initial Formal Specification of Kokkos Operational Semantics

- Demonstration of Concolic Testing for Kokkos programs
  - Parallel_for (no nest)
  - Heterogeneity (cudaSpace/ and hostSpace)

- Design of Clang Abstract Syntax Tree (AST) for Kokkos

- Local (Partial) Symbolic Execution of Kokkos Programs

Project Plan Year 2 and 3

- Formal Specification of Kokkos Operational Semantics
  - Parallel_for with nesting
  - Atomics
  - Kokkos 3.5 capability (multiple execution space form the same device)

- Demonstration of Concolic Testing for more sophisticated Kokkos programs

- Static Analysis of Kokkos Programs using Clang AST
  - Prune out unnecessary states/paths in symbolic analysis using Kokkos/Application specific knowledge
  - Adapting SMT solver to high-level Kokkos programming context

- Differential Testing of Kokkos Programs across multiple platforms

- Application to Kokkos program source from real applications

Q1 Outcome

- Initial Formal Specification Work on Kokkos Programming Model
- Test Suite of Common Programming Mistakes in Kokkos
  - Set of small program source to represent Kokkos' programming mistakes that triggers runtime errors and inconsistent behavior
  - Heterogeneity (nested parallel_for, Memory Space)
  - Concurrency Issues (race, atomics, nested parallel_for)
  - Soft Copy vs Deep Copy

  ➔ This test suite has been a useful guide for initial design of static/symbolic/concolic/dynamic analysis algorithms for Kokkos

- Evaluation of KLEE
  - Took much longer than expected to analyze a simple Kokkos example program
  - This motivated why we need a special version dedicated for Kokkos
- Kokko Virtual Machine

  Special Sequential Version

  Working on de-templated version

Collaboration Opportunities

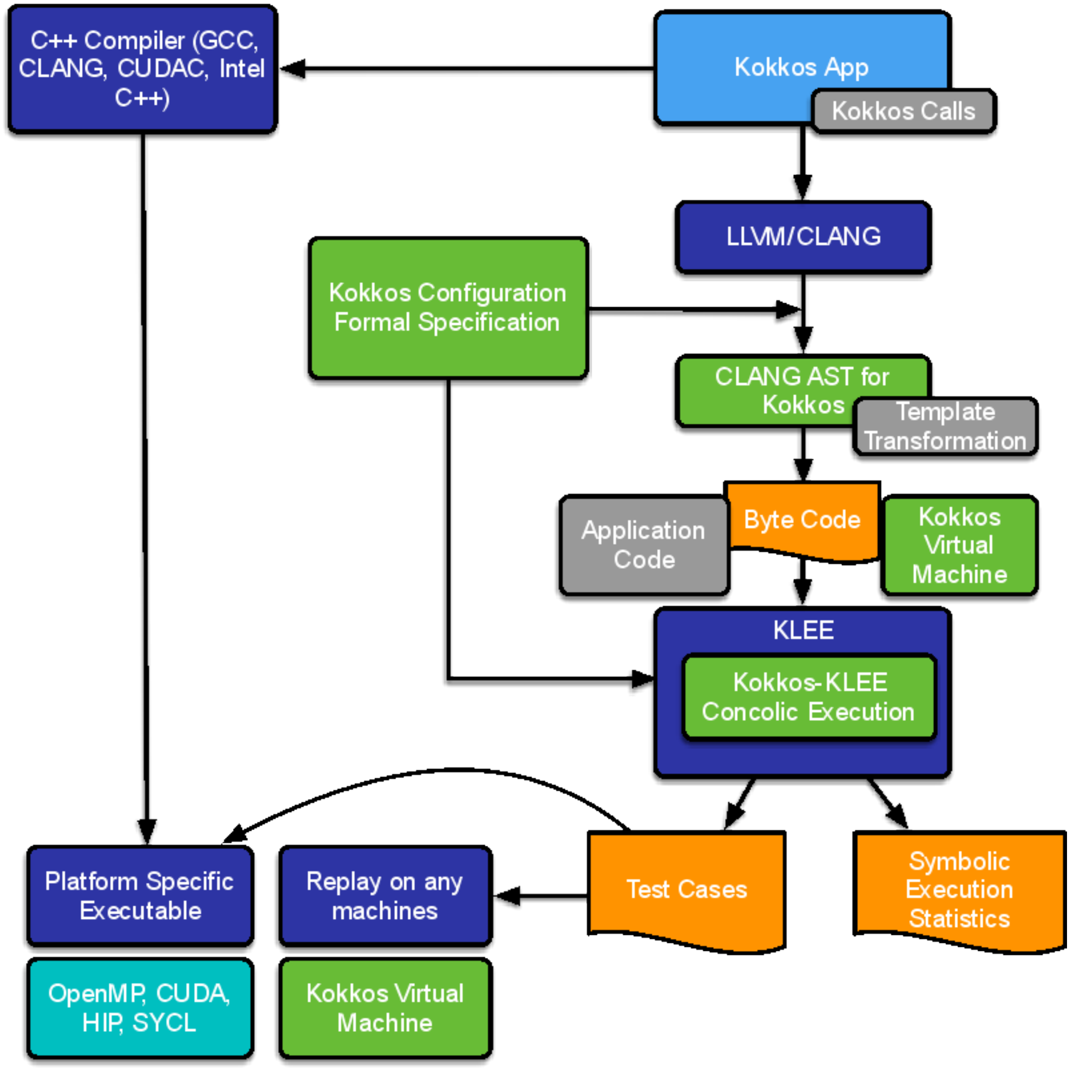
- Kokkos Team (SNL, ANL, ORNL)
- Ganesh Gopalakrishnan (U of Utah)
- Application and Library Developers
  - Scientific Libraries (ANL,ORNL and SNL)
    - Kokkos, PETSc and Trillnos
  - Applications using Kokkos
    - ExaWind
- Tool Developers
  - Debuggers
  - Performance Tool Developers

The XSTACK KLOKKOS Project

Performance Portable Programming Model has become a standard for HPC application development at ASCR, but writing a correct code for any platforms is still challenging.

We develop Automatic Test Generation Tool for Kokkos through integration of

- Kokkos Formal Specification
- Kokkos Virtual Machine
- State-of-the-art concolic analysis methodology
- Ensemble of multiple test/analysis to reduce state explosion of symbolic analysis
- Domain specific knowledge and Kokkos-level abstraction


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
Kokkos Virtual Machine
KLEE
Kokkos-KLEE Concolic Execution
Platform Specific Executable
Replay on any machines
Test Cases
Symbolic Execution Statistics
OpenMP, CUDA, HIP, SYCL
Kokkos Virtual Machine