This paper describes objective technical results and analysis. Any subjective views or opinions that might be expressed in the paper do not necessarily represent the views of the U.S. Department of Energy or the United States Government.

SAND2022-12060C

Sandia National Laboratories

Exceptional service in the national interest

Toward Automatic Test Synthesis for Performance Portable Programs

Keita Teranishi, Shyamali Mukherjee, Richard Rutledge, Samuel Pollard, Noah Evans, Alessandro Orso, and Vivek Sarkar

KLEE Workshop 2022

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.

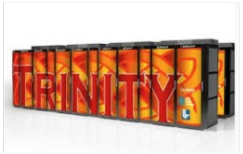
Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia LLC, a wholly owned subsidiary of Honeywell International Inc. for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

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.

Motivation: Tackling the Diversity of HPC Programming Systems

- CPUs (Intel, AMD, ARM, IBM)
- GPUs (NVIDIA, AMD, Intel)
- Heterogeneity
- Diversity of Programming Systems (**OpenMP, OpenACC, CUDA, HIP, DPC++)**

Current Generation: Programming Models OpenMP 3, CUDA and OpenACC depending on machine



LANL/SNL Trinity
Intel Haswell / Intel KNL
OpenMP 3

LLNL SIERRA
IBM Power9 / NVIDIA Volta
CUDA / *OpenMP*[a]

ORNL Summit
IBM Power9 / NVIDIA Volta
CUDA / *OpenACC / OpenMP*[a]

SNL Astra
ARM CPUs
OpenMP 3

Riken Fugaku
ARM CPUs with SVE
OpenMP 3 / *OpenACC*[b]

Upcoming Generation: Programming Models OpenMP 5, CUDA, HIP and DPC++ depending on machine

NERSC Perlmutter
AMD CPU / NVIDIA GPU
CUDA / *OpenMP 5*[c]

ORNL Frontier
AMD CPU / AMD GPU
HIP / *OpenMP 5*[d]

ANL Aurora
Xeon CPUs / Intel GPUs
DPC++ / *OpenMP 5*[e]

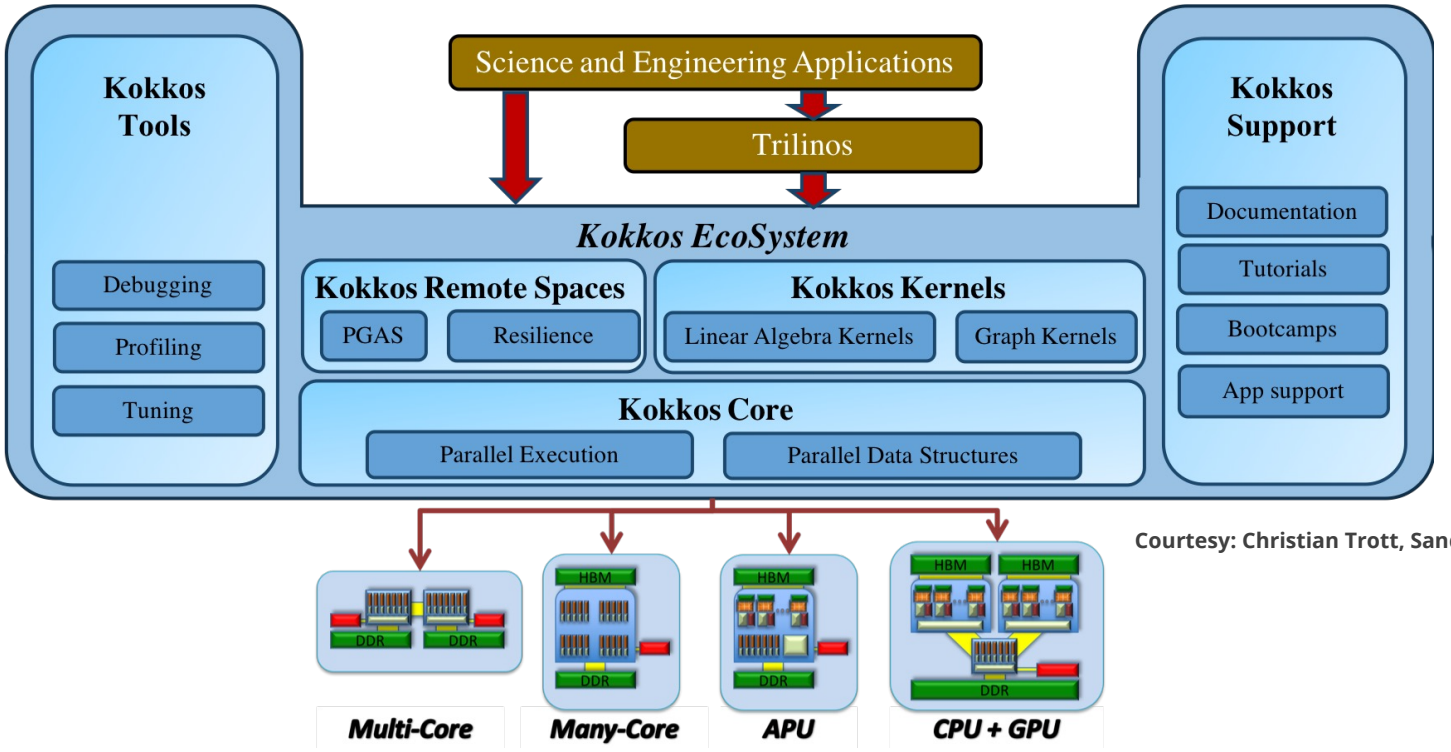
LLNL El Capitan
AMD CPU / AMD GPU
HIP / *OpenMP 5*[d]

Courtesy: Christian Trott, Sandia National Labs, NM

Solution Kokkos Ecosystem Provides Performance Portable Programming Environment -- Same code for any HPC platforms

kokkos

https://github.com/kokkos



Courtesy: Christian Trott, Sandia National Labs, NM

Kokkos Performance Portable Programming

Serial

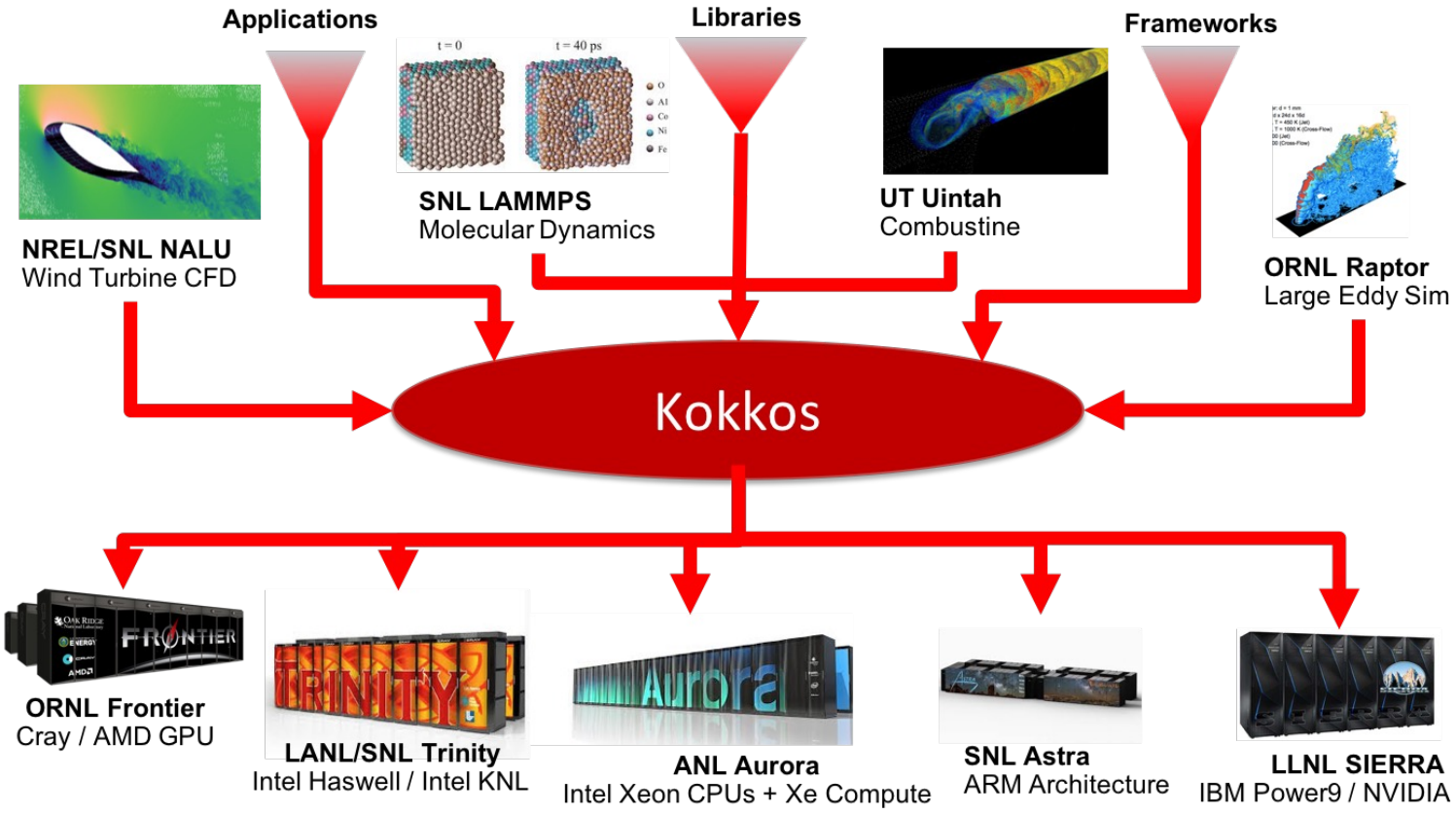
```
double A[100];
for (int i = 0; i < N; ++i)
{
   A[i] = i+N;
}
```

Kokkos

```
Kokkos::View<double *, DefaultSpace::mem> A(100);  // Allocated in the default
device
Kokkos::parallel_for (Kokkos::range_policy<DefaultExecutionSpace>(100),
          KOKKOS_LAMBDA (int &i)
          {
            A(i) = i+N;
          }
);
```

- Modern C++ (C++17) metaprogramming
- Abstraction of data object such as memory allocation/location and data layout (**View**)
- Abstraction of execution patterns and underlying runtime/hardware
  (parallel_for, parallel_reduce, parallel_scan)
- Single Source for Multiple Platforms!

Kokkos enables extreme scale scientific/engineering applications



Courtesy: Christian Trott, Sandia National Labs, NM

Performance Portable Programming is still mistake prone

- Kokkos provides portable abstractions (ironically) allows non-portable implementation.
- Bugs manifest only on specific platforms.
  - Crash
  - Incorrect results
  - Poor performance
  - Major causes are race conditions (GPUs) and lack sync between host and devices
- Still requires good understandings of target 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;
   });
}
```

Kokkos, Heterogeneous

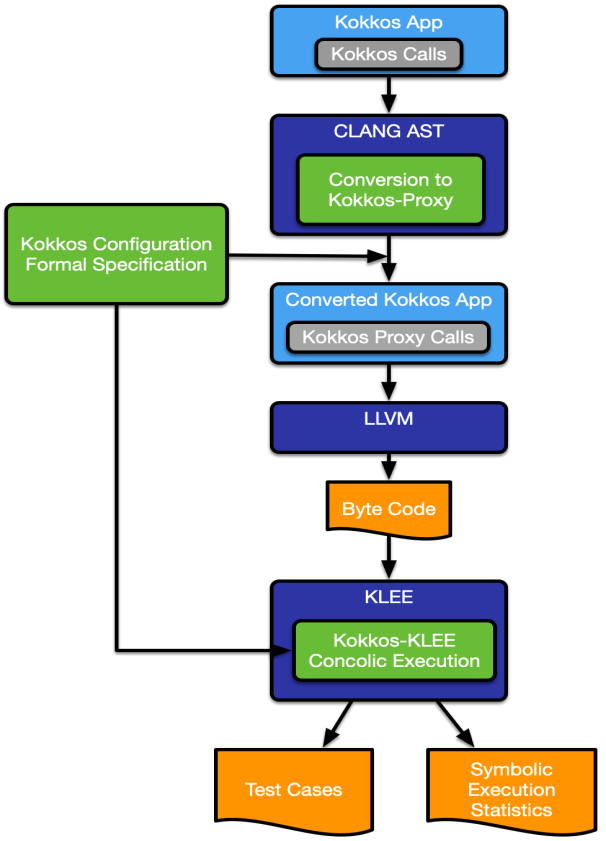
```
Kokkos::View<double **> A(N,N);  // Allocated in the default device
Kokkos::View<double **>::HostMirror HostA = Kokkos::create_mirror(A);
for( int i = 0; i < N; ++i ) {
   Kokkos::parallel_for (N, KOKKOS_LAMBDA(const size_t &j)
   {
      A(i,j) = i*N*j;
   });
}
Kokkos::fence();
Kokkos::deep_copy(HostA,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

Kokkos , Heterogeneous, Efficient

```
Kokkos::View<double **> A(N,N);  // Allocated in the default device
Kokkos::View<double **>::HostMirror HostA = Kokkos::create_mirror(A);
Kokkos::team_policy<> myTeam = Kokkos::team_policy<>(N);
Kokkos::parallel_for (myTeamm, KOKKOS_LAMBDA (Kokkos::team_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
```

KLOKKOS, Auto test-code Generation Framework

- Leverages KLEE
- Establish a portable formal specification of Kokkos APIs for model checking.
- Treats all Kokkos method calls as uninterpreted function calls
  - Symbolic analysis in the level of Kokkos' abstractions
- Track the symbolic state of Kokkos' data representation
- 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.



Kokkos Proxy Allows Symbolic Analysis

Biggest problem in symbolic testing:

- State explosion
- Name mangling of C++
- We really care the states relevant to the correct use of Kokkos APIs.

Solution: Convert Kokkos prorams C-like programs

- Extract API calls, demangle namespace, remove templates, simplify Kokkos
- All Kokkos methods are treated as C-like function

Embody Kokkos formal semantics and models in the proxy representation

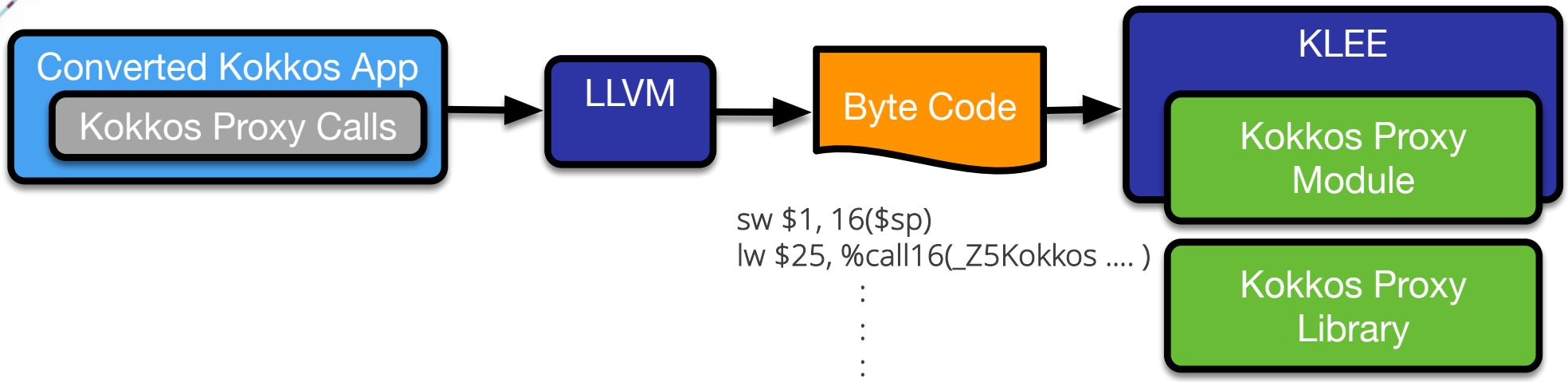
Kokkos

```
Kokkos::View<double *> A("View A", 100);  // Allocated
in the default device
// Kokkos Range Policy, it launches a kernel, i =
[0,100)
Kokkos::parallel_for (100, KOKKOS_LAMBDA (const int
&i)
{
   A(i) = i;
});
```

Kokkos Proxy

```
int input[1]  = {100};
KokkosView A = DeclareView( "View A", 1, input ,
DOUBLE, DefaultMemSpace, LeftLayout);

ParallelForRangePolicyBegin( A, 0, 100,
DefaultExeSpace );
auto MyFunc = [&](const int &thread_i)
{
   int indices[1];
   indices[0] =thread_i;
   KokkosViewAssgin(A,1,indices,thread_i);
};
MyFunc(i);
ParallelForRangePolicyEnd( A, DefaultExeSpace );
```

We modify KLEE to analyze Kokkos Proxy calls



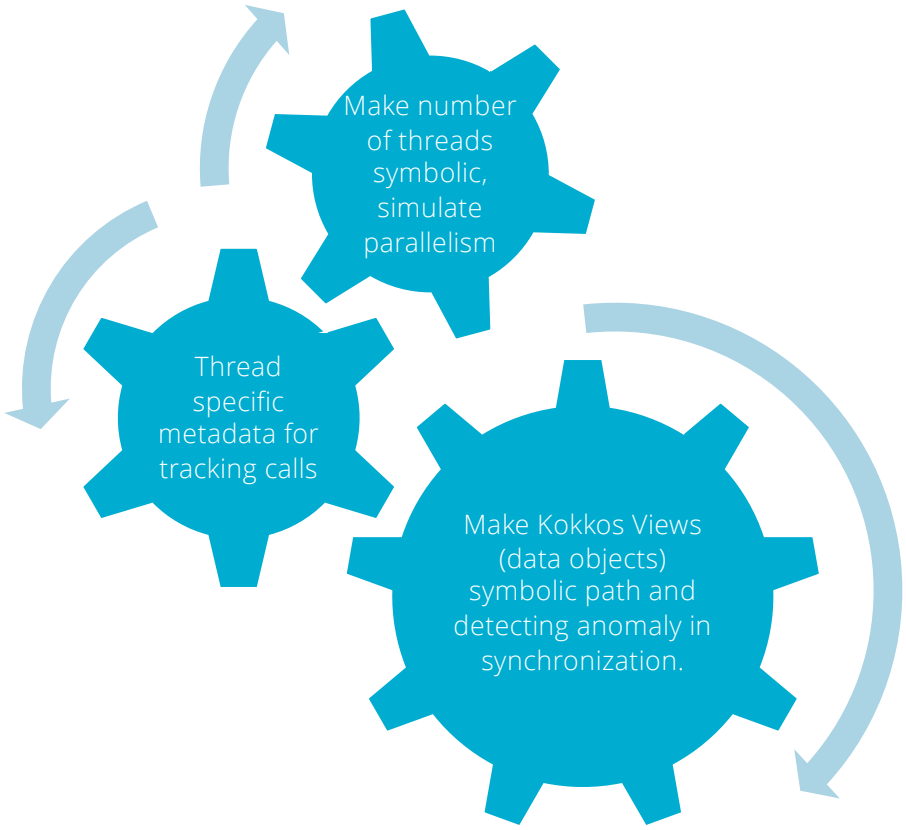
- Two Major Components
  - Kokkos Proxy Module
  - Kokkos Proxy Library
- (Meta) Data Object Centric
  - Maintain meta data of individual Kokkos::View
- Do not perform any floating computation
  - Not scalable
  - We are interested in the common programming mistakes rather than floating point bugs

Kokkos Proxy KLEE Module Allows Tracking Data Object (View) state

Make Kokkos View (data object) Symbolic

Maintain metadata to track mirror and reference of Views and duplicated copies.

Enables to track mirror views on GPUs for detecting heterogeneous data inconsistency



Kokkosn Proxy mistake example

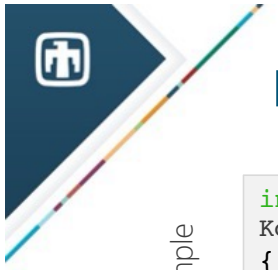
Kokkos VM example

```
int input[1]  = {100};
KokkosView A = DeclareView( "View A", 1, input , DOUBLE, DefaultMemSpace, LeftLayout);
{
   ParallelForRangePolicyBegin( A, 0, 100, DefaultExeSpace );
   auto MyFunc = [&](const int &thread_i)
   {
      int indices[1];
      indices[0] = thread_i%2; //
      KokkosViewAssign(A,1,indices,thread_i);
   };
   MyFunc(i); // We know i = [0,100)
   ParallelForRangePolicyEnd( A, DefaultExeSpace );
}
```

Potential Race Condition. Detected through concolic execution

- The bug shows up with a certain parallel simulation
- KLEE provides a way to concretize the value of **thread_i**
- KLOKKOS internally maintain a table for KokkosView specific to Parallel_For.

- Use a symbolic task or thread id
- Fork the execution for various simultaneous tasks
- Add a constraint on task and range policy to the path condition



Kokkos VM example

Kokkos Proxy mistake example

```
int input[1]  = {100};
KokkosView A = DeclareView( "View A", 1, input , DOUBLE, DefaultMemSpace, LeftLayout);
{
   ParallelForRangePolicyBegin( A, 0, 100, DefaultExeSpace );
   auto MyFunc = [&](const int &thread_i)
   {
      int indices[1];
      indices[0] = my_index[thread_i];
      KokkosViewAssign(A,1,indices,thread_i);
   };
   MyFunc(i); // We know i = [0,100)
   ParallelForRangePolicyEnd( A, DefaultExeSpace );
}
```

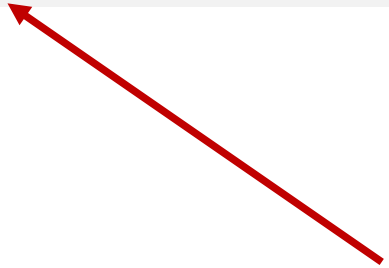
Overwriting in View region indices, as indices is non deterministic

- The bug shows up with a certain parallel simulation
- Result is an incorrect computation. due to lack of data cohesiveness in a non deterministic way

- Use a symbolic task or thread id
- Fork the execution for various simultaneous tasks
- Add a constraint on task and range policy to the path condition

Kokkos Proxy mistake example

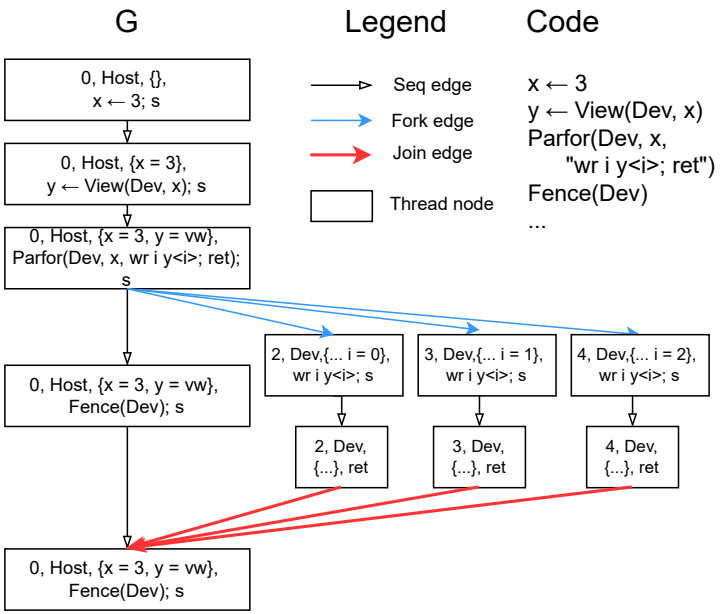
```
int input[1]  = {100};
int inputb[1] = {10};
KokkosView A = DeclareView( "View A", 1, input , DOUBLE, DefaultMemSpace, LeftLayout);

KokkosView B = DeclareView( “View B”, 1, input, DOUBLE, DefaultmemSpace, LeftLayout);

KokkosDeepCopy( A, 1.0 ); // Assign 1s to all entries of A.
KokkosDeepCopy( B, A );   // The size of B and A does not match. Runtime Error
```



- Result will report bug as boundary check fails in Kokkos Proxy plug-in in KLOKKOS.
- KLOKKOS maintains the metadata of individual KokkosViews

MiniKokkos and Memory Models

- MiniKokkos paper submitted to Correctness 22 workshop at SC22
  - Key results:
    - Syntax and semantics for simplified language
    - capture views, fences, parallel loops
    - Proof of *portability* in this language
    - Found a bug in `kokkos::deep_copy` [1]
- Memory Models
  - Literature review
    - Many modeling tools: TLA+, Murphi, CIVL; none were at right abstraction
    - Memory consistency models more specific: herd, alloy
    - Found existing work for NVIDIA's PTX
    - Weaker memory model means more optimizations
    - Kokkos memory model very weak; weakener than PTX



MiniKokkos execution graph

[1] https://github.com/kokkos/kokkos/issues/5213

Ongoing Work on Kokkos Formal Specification

- MiniKokkos
  - More complete model of Kokkos: Add (prioritized)
    1. Nested parallelism
    2. Multidimensional views, more types than just integers
    3. Parallel reduce and scan
- Memory Model
  - Weaken PTX, see which theorems still hold
  - Prove behavior and code transformations make sense on Kokkos' memory model
- Other Potential Directions
  - Model potential Kokkos features (multiple user-level Kokkos threads)
  - Prove theorem: Data-race freedom + (weak) Kokkos memory model implies sequential consistency
  - Interpreter for MiniKokkos – may be useful as modeling language?

$$
\begin{array}{rrcl}
\text{Execution space} & ES & ::= & \texttt{Host} \mid \texttt{Dev} \\
\text{Types} & \tau & ::= & \mathbb{N} \mid \mathbb{V} \mid R \mid \mathit{void} \\
\text{Expressions} & e & ::= & x \mid c \mid e_1 + e_2 \mid \texttt{View}(ES, e) \\
\text{Statement} & s & ::= & i\,; s \mid \texttt{ret} \\
\text{Instruction} & i & ::= & m \mid x \leftarrow c \mid x \leftarrow e_1 + e_2 \\
 & & & \mid x \leftarrow \texttt{View}(ES, e) \\
 & & & \mid \texttt{Parfor}(ES, e, s) \mid \texttt{Fence}(ES) \\
 & & & \mid \texttt{Fence}(\texttt{Host}); \texttt{Fence}(\texttt{Dev}); \texttt{DeepCopy}(x, y) \\
 & & & \mid \texttt{if } e \texttt{ then } s_1 \texttt{ else } s_2 \\
\text{Memory operation} & m & ::= & \texttt{alloc } x \mid \texttt{rd } x\ y \mid \texttt{rd } x\ y\langle e \rangle \\
 & & & \mid \texttt{wr } x\ y \mid \texttt{wr } x\ y\langle e \rangle
\end{array}
$$

Fig. 1. Syntax of MiniKokkos

Kokkos has many more features than MiniKokkos

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 Proxy	One input for multiple platforms Concolic testing indicates which part of code needs to be covered by Dynamic analysis.
Concolic (Concrete-Symbolic) Testing	**Formal Specification**, Compiler, SMT Solver, **Kokkos Proxy**	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 indicates which part of code needs to covered by differential testing.

Future and Ongoing Work

- Evaluation with a suite of Kokkos Mistake Examples
- Application of Partial Symbolic Analysis
  - Analysis applied only to a specific portion of an application
  - Lazy initialization adapted for Kokkos and Scientific applications
- Evaluation with mini-applications
  - Mantevo (**https://mantevo.github.io/**)
  - MiniMD (Molecular Dynamics, Mini-version of LAMMPS)
  - MiniFE (Finite Element Analysis for mechanical applications)
  - MiniAero (CFD, Driven Cavity Flow)
- Integration of Kokkos' Formal Specifications