MiniKokkos: A Calculus of Portable Parallelism

Feiyang Jin
College of Computing
Georgia Institute of Technology
Atlanta, USA
fjin35@gatech.edu

John Jacobson III
School of Computing
University of Utah
Salt Lake City, USA
john.jacobson@utah.edu

Samuel D. Pollard
Sandia National Laboratories
Livermore, USA
spolla@sandia.gov

Vivek Sarkar
College of Computing
Georgia Institute of Technology
Atlanta, USA
vsarkar@gatech.edu

***Abstract*—Kokkos is a C++ library and ecosystem for writing parallel programs on heterogeneous systems. One of the primary goals of Kokkos is *portability*: programs in Kokkos are expressed through general parallel constructs which can enable the same code to compile and execute on different parallel architectures. However, there is no known formal model of Kokkos's semantics, which must be generic enough to support current and future CPU and accelerator architectures. As a first step of formalizing Kokkos, We introduce MiniKokkos: a small language capturing the main features of Kokkos, and then prove that MiniKokkos ensures portability across all possible parallel executions. We also provide a case study of how MiniKokkos can help reason about Kokkos programs and help find a bug in the Kokkos implementation.**

***Index Terms*—parallel programming, semantics, programming languages**

I. Introduction

Kokkos [1] is a C++ library and ecosystem designed around the abstraction of data management and parallel execution across different devices. Kokkos arose from the need to write maintainable, high-performance parallel software for our current world of highly heterogeneous computer architectures.

To achieve these goals, Kokkos is based on the concept of *data parallel primitives* (DPP) [2], where algorithms are expressed using a small number of parallel operations and parallelism is assumed to be unbounded. Kokkos abstracts over two primary concepts: data and parallel execution. A Kokkos user writes a parallel program by describing the data (such as the layout and location of multidimensional array elements) and execution model (the set of parallel operations) over these data. Then, Kokkos translates this program into one of several back-ends (e.g., CUDA, OpenMP, or HIP). This permits a unified semantics and allows the programmer to express a program at a high-level, without worrying about the details of each back-end. Achieving good performance across many different architectures is called *performance portability*.

The underlying concepts of Kokkos are elegant, but the generality of Kokkos allows users to write programs that deadlock or contain data races. The complexity of various back-end platforms complicates the debugging process and can cause misunderstanding when porting from different programming paradigms. One example is a fence (also called a synchronization point or barrier), which has subtle semantic differences across different CPU and GPU architectures. Furthermore, the complex semantics of C++ adds to the challenge of reasoning about a Kokkos program.

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

Fig. 1. Syntax of MiniKokkos

TABLE I
Important symbols and notations

Symbol	Explanation
$\mho$	a value, could be c or $\mathfrak{vw}$
c	a natural number, has type $\mathbb{N}$
$\mathfrak{vw}$	a view, has type $\mathbb{V}$
x, y	local variables
$t_{mem}[x] = \mho$	x has value $\mho$ in thread-local memory
$\mathtt{HostSM}[\mathfrak{vw} \mapsto vdata(c)]$	$\mathfrak{vw}$ holds c natural numbers in `HostSM`
$len(\mathfrak{vw})$	The number of elements (length) of $\mathfrak{vw}$
$*\mathfrak{vw}\langle c\rangle = \mho$	View $\mathfrak{vw}$ at index c has value $\mho$
$*tcHost = n$	There are n active threads on the `Host`

To help Kokkos users and tool builders understand program behaviors, detect potential bugs, and correctly interpret functionality, we aim to formalize Kokkos. We start by abstracting away the complexity of C++ and back-ends and instead model a simplified core calculus we call MiniKokkos. Our semantics formalizes the informal semantics described by the developers of Kokkos and have been developed and refined through collaboration with the Kokkos team. In summary, the contributions of this paper are:

- a core calculus, MiniKokkos, which captures the main concepts of Kokkos;
- proofs for key properties that MiniKokkos maintains, such as deadlock freedom and portability; and
- a case study where MiniKokkos helped detect a bug in the implementation of Kokkos.

II. Syntax & Types

In this section, we introduce the syntax (Fig. 1) and types (Fig. 2) of MiniKokkos.

A MiniKokkos program consists of a sequence of statements, where a statement is either an instruction or a `ret`. Instructions include assignments, conditionals, synchronization

$$\textsc{R-Nat}\,\frac{}{c \Downarrow c} \qquad \textsc{R-Add}\,\frac{e_1 \Downarrow c_1 \quad e_2 \Downarrow c_2}{e_1 + e_2 \Downarrow c_1 + c_2}$$

$$\textsc{R-Assign}\,\frac{e \Downarrow e'}{x \leftarrow e \Downarrow x \leftarrow e'}$$

$$\textsc{T-Nat}\,\frac{}{\Gamma \vdash c : \mathbb{N}} \qquad \textsc{T-Add}\,\frac{\Gamma \vdash e_1 : \mathbb{N} \quad \Gamma \vdash e_2 : \mathbb{N}}{\Gamma \vdash e_1 + e_2 : \mathbb{N}}$$

$$\textsc{T-Var}\,\frac{}{\Gamma, x : \tau \vdash x : \tau} \qquad \textsc{T-Ret}\,\frac{}{\Gamma \vdash \texttt{ret} : \mathit{void}}$$

$$\textsc{T-View}\,\frac{\Gamma \vdash e : \mathbb{N}}{\Gamma \vdash \texttt{View}(ES, e) : \mathbb{V}}$$

$$\textsc{T-Assign}\,\frac{\Gamma \vdash e : \tau \quad \Gamma, x : \tau \vdash s : \tau'}{\Gamma \vdash x \leftarrow e\,;s : \tau'}$$

$$\textsc{T-Seq}\,\frac{\Gamma \vdash i : \tau \quad \Gamma \vdash s : \tau'}{\Gamma \vdash i\,;s : \tau'}$$

$$\textsc{T-ParFor}\,\frac{\Gamma \vdash e : \mathbb{N}}{\Gamma \vdash \texttt{Parfor}(ES, e, s) : \mathit{void}}$$

$$\textsc{T-Fence}\,\frac{}{\Gamma \vdash \texttt{Fence}(ES) : \mathit{void}}$$

$$\textsc{T-DeepCopy}\,\frac{\Gamma \vdash x : \mathbb{V} \quad \Gamma \vdash y : \mathbb{V}}{\Gamma \vdash \texttt{DeepCopy}(x, y) : \mathit{void}}$$

$$\textsc{T-If}\,\frac{\Gamma \vdash e : \mathbb{N} \quad \Gamma \vdash s_1 : \tau \quad \Gamma \vdash s_2 : \tau}{\Gamma \vdash \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2 : \tau}$$

$$\textsc{T-Read-View}\,\frac{\Gamma \vdash y : \mathbb{V} \quad \Gamma \vdash e : \mathbb{N} \quad \Gamma, x : \mathbb{N} \vdash s : \tau'}{\Gamma \vdash \texttt{rd}\ x\ y\langle e\rangle\,;s : \tau'}$$

$$\textsc{T-Write-View}\,\frac{\Gamma \vdash x : \mathbb{N} \quad \Gamma \vdash y : \mathbb{V} \quad \Gamma \vdash e : \mathbb{N}}{\Gamma \vdash \texttt{wr}\ x\ y\langle e\rangle : \mathit{void}}$$

Fig. 2. Reduction (R-) and typing (T-) rules for MiniKokkos.

points (*fences*), parallel loops, and memory operations. We use x to denote a thread-local variable (which cannot be accessed by memory read/write operations) and c as a constant (in our case, the only constants are natural numbers).

Two simplifying features of MiniKokkos are that variables are single-assigned, and the only operation permitted on values is addition. These are not major limitations from the viewpoint of formal semantics since a multiple-assignment local variable can be renamed to multiple single-assignment variables, and the semantics of additional operators is usually a simple extension to that of a basic operator like addition.

A MiniKokkos program has two *execution spaces*, `Host` and `Dev`. Each execution space has its own *shared memory*, which we call `HostSM` and `DevSM`. Data are only accessible by statements instances executed in their matching execution spaces (to capture the semantics that accelerators have separate memory address spaces in general). A statement instance cannot access data in another execution space unless data are explicitly copied via `DeepCopy`. A thread also has its own *local memory*, which is only visible to itself. We further discuss shared and local memory in Sec. III.

We next explain the instructions. Assignment rules of the form $x \leftarrow e$ will create a new local variable x and assign it the value computed from e. The instruction $x \leftarrow \texttt{View}(ES, e)$ allocates a new view in the shared memory of the corresponding execution space, which is saved in the local variable x. While the variable is immutable, the data stored in the View is mutable. We require e to evaluate to a natural number ($e \Downarrow c$).

The instruction $\texttt{Parfor}(ES, e, s)$ creates a new parallel-for region with c threads and body s. We require e to evaluate to a natural number ($e \Downarrow c$).

$\texttt{Fence}(ES)$ can be called by the master thread (the thread that starts the program and remains active until the program finishes) to wait for all submitted work on ES to finish. $\texttt{DeepCopy}(x, y)$ can be invoked by the master to copy the contents of View saved in y to the View saved in x.

Reading and writing of views are handled by $\texttt{rd}\ x\ y\langle e\rangle$ and $\texttt{wr}\ x\ y\langle e\rangle$, which access index c of view y. And so, `rd` makes a new variable, and `wr` requires an existing local variable. We also require $e \Downarrow c$.

We next describe our type system, shown in Fig. 2. There are three types in MiniKokkos: natural numbers ($\mathbb{N}$), Views ($\mathbb{V}$), and the unit type (*void*). We handle variables (T-Var) similarly to the simply-typed λ-calculus [3]: they are created assuming a valid string in the program text, and the most recent binding is looked up in the typing context Γ. T-Assign states that the newly created variable x will have the type of the expression it is assigned. T-DeepCopy rules that `DeepCopy` only makes sense on views, and T-Write-View ensures that only natural numbers can be written to a view.

One important implication of our type system's simplicity is its lack of expressiveness: execution spaces are not types (but simply syntactic elements), and there are no function types. Rather, a list of statements is what gets executed inside of a parallel loop. In practice, this is closer to the actual behavior of Kokkos embedded in C++.

Type systems also typically have proofs of type safety (progress and preservation); we do not include these for two reasons: space limitation, and the execution model, not the type system is the main focus of MiniKokkos. Extending the type system is a future research direction because many powerful C++ type features (such as *concepts*) provide more type-safety to Kokkos programs.

For reference, we provide Table I to list the notation we use throughout this work. We next discuss program state, which is required to describe the semantics in Sec. IV.

III. Program state

We represent the state σ of a MiniKokkos program as a combination of shared memories and *computation graph*, denoted as $\sigma = (\texttt{HostSM}, \texttt{DevSM}, G)$. `HostSM` and `DevSM` are maps that save each view, plus the count of threads in the execution space. They have the notation

$$\begin{aligned}\texttt{HostSM} = \{&\mathit{tcHost} \mapsto k_h,\\ &\mathfrak{vw}_1 \mapsto \mathit{vdata}(c_1), \cdots, \mathfrak{vw}_m \mapsto \mathit{vdata}(c_m)\}\\ \texttt{DevSM} = \{&\mathit{tcDev} \mapsto k_d,\\ &\mathfrak{vw}_1 \mapsto \mathit{vdata}(c_1), \cdots, \mathfrak{vw}_n \mapsto \mathit{vdata}(c_n)\}.\end{aligned}$$

The computation graph G corresponds to a execution history (trace) of a program. The rules for constructing G are defined in Fig. 3. A thread node t is a quadruple $[f, ES, LM, s]$

$$
\begin{array}{rrll}
\text{Computation graph} & G & ::= & t \quad \text{Thread node} \\
 & & \mid & G + t_1 \xrightarrow{Seq} t_2 \\
 & & \mid & G + t_1 \xrightarrow{Fork} t_2 \\
 & & \mid & G + t_1 \xrightarrow{Seq} t' \xleftarrow{Join} t_2 \\
\text{Thread node} & t & ::= & [f, ES, LM, s]
\end{array}
$$

Fig. 3. Computation graph for MiniKokkos

that includes a thread id f, a flag ES that indicates the execution space of the thread, a map LM that represents the local memory, and a statement s that the node will execute. Map LM has the notation $\{x_1 \mapsto \mho_1, \dots, x_n \mapsto \mho_n\}$. The notation $t_{mem}[x]$ retrieves the value saved in x in t_{mem}. The root node R for a program P is a thread node of the form $[0, \texttt{Host}, \{\}, P]$. We assume the master thread has id 0 and executes on the Host. The master thread is allowed to perform several operations on both `HostSM` and `DevSM`, see Fig. 5. For convenience, we may also refer to each element of the tuple as $[f = t_{tid}, ES = t_{es}, LM = t_{mem}, s = t_{code}]$.

The notation $G + t_1 \xrightarrow{Seq} t_2$ indicates that G is extended with an edge labeled *Seq* and the edge connects two nodes t_1, t_2. Edges denoted $\xrightarrow{Seq}$ capture sequential regions (e.g., statements within a single thread), while *Fork* edges capture new thread creation in `Parfor`; *Join* edges capture thread synchronization in `Fence`. If $t_1 \xrightarrow{Seq} t_2$, we say t_2 is t_1's *continuation node*. We present an example MiniKokkos program and its state transition in Fig. 4.

IV. Semantics

We describe the small-step semantics for MiniKokkos in Figs. 5, 6. Here we briefly explain each rule. At a program's start (K-START), the two shared memory spaces (host and device) have thread count information and the computation graph contains the root node. Note that without an explicit copy, views in `HostSM` are not visible to `DevSM`, and vice versa. For simplicity, We assume unbounded memory in both cases.

Creating a new View allocates a location in the shared memory of the given execution space, initializes every entry to 0, and extends the local memory with a variable that references the new View. Reading and writing to a View accesses the shared memory directly, and reading from a View extends the local memory with the value read.

Parfor commands start a new parallel-for region in the provided execution space ES. The invoking thread will have an execution node that runs asynchronously with the Parfor body. Newly created threads execute the Parfor body and inherit (by copying) the local memory of the invoking thread; a unique loop index is added to each thread's local memory.

$\texttt{Fence}(ES)$ can only be invoked by the master thread, and the master thread will wait for all work submitted to ES to be completed before it can continue. Whether to wait is determined by checking the thread count information in the shared memory. For node v, the notation $out(v)$ denotes the number of outgoing edges from v.

$\texttt{DeepCopy}(x, y)$ can only be invoked by the master thread. The master thread will perform the copy operation if the two views have the same length, and all previous submitted work are finished. Because the master thread itself is executing the copy operation, no new work can be submitted to either `Host` or `Dev` before the copy finishes. We assume that whenever new threads or variables are created, they are given unique and consistent names across different executions.

We finish this section by discussing the essential design choices of Kokkos and their implementation in MiniKokkos.

1) View: Views in Kokkos behave like pointers with metadata describing their dimension and location. This design is reflected in MiniKokkos as well. Each read or write is performed to the shared memory directly because no thread-local copy of a View exists. Assignment of one View to another will perform shallow copy. At creation, Kokkos initializes each slot in the View by the default constructor of the type it holds; MiniKokkos behave similarly by initializing all entries to be 0 when created.
2) Fence: Kokkos requires that `Kokkos::fence` is not called inside an existing parallel region (i.e., inside the `operator()` of a functor or lambda) [4]. In MiniKokkos, we only allow the master thread to invoke a fence. This restriction prevents deadlock in MiniKokkos, as proved in Theorem VI.1.
3) DeepCopy: Kokkos provides two versions of DeepCopy. If no execution space argument is passed in, the call is blocking: it will wait for all work on all execution space to finish, perform the copy, and return after the copy finishes. The call is potentially asynchronous if an execution space argument is passed in. The DeepCopy in MiniKokkos corresponds to the first version; we discuss the second version in our case study (Sec. VII).

V. Preliminaries

We now state some preliminary definitions and prove auxiliary lemmas, which we use in the following section.

For a program P, let notation $P \Downarrow \sigma$ be the execution of program P, defined as $[0, \texttt{Host}, \varnothing, P] \rightarrow^{\star} \sigma$ and P can take no more steps (that is, σ is the state at the end of the execution).

Definition V.1 (Happens-before)**.** Node v *precedes* or *happens-before* node u if and only if there exists at least one directed path from v to u in the computation graph G. We denote it as $v \rightsquigarrow u$. If it is not the case, we denote it as $v \not\rightsquigarrow u$. Node v is *in parallel* with u, denoted $v \parallel u$, if $u \not\rightsquigarrow v$ and $v \not\rightsquigarrow u$.

Definition V.2 (Read and Write)**.** A thread node t on Host *reads* from index d of view $\mathfrak{vw}$ if 1). t_{code} = $\texttt{rd}\ x\ y\langle e\rangle$; s, where $t_{mem}\ [y]$ = $\mathfrak{vw}$, $\mathfrak{vw} \in \texttt{HostSM}$ and $e \Downarrow d$; or 2). t_{code} = $\texttt{DeepCopy}(x, y)$, where $t_{mem}\ [y]$ = $\mathfrak{vw}$ and $\mathfrak{vw} \in \texttt{HostSM}$ or $\mathfrak{vw} \in \texttt{DevSM}$.

A thread node t on Host *writes* to index d of view $\mathfrak{vw}$ if 1). t_{code} = $\texttt{wr}\ x\ y\langle e\rangle$, where $t_{mem}\ [y]$ = $\mathfrak{vw}$, $\mathfrak{vw} \in \texttt{HostSM}$ and $e \Downarrow d$; or 2). t_{code} = $\texttt{DeepCopy}(x, y)$, where $t_{mem}[x]$ = $\mathfrak{vw}$ and $\mathfrak{vw} \in \texttt{HostSM}$ or $\mathfrak{vw} \in \texttt{DevSM}$. We omit similar definitions

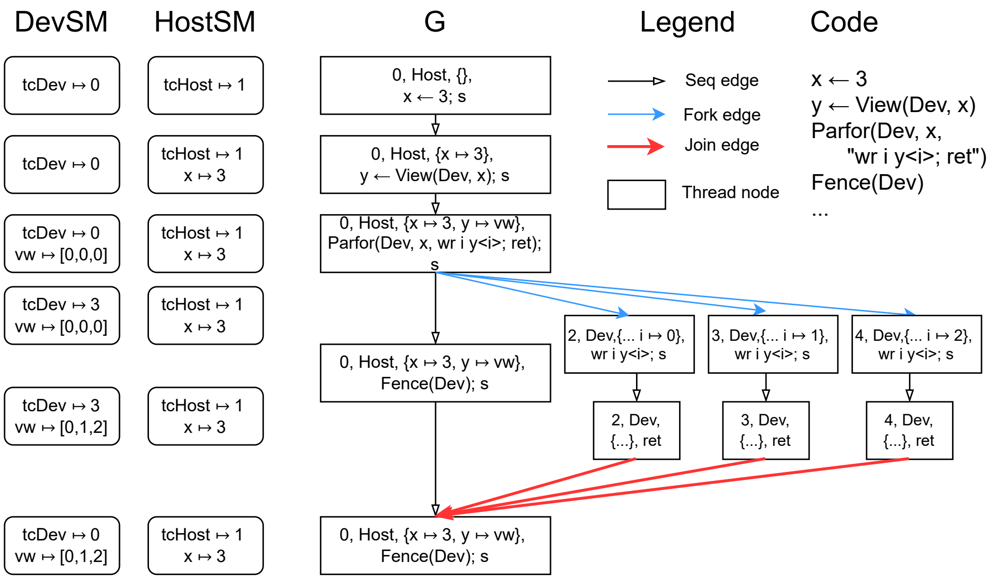


Fig. 4. Example MiniKokkos program and state transition.

$$\textsc{K-Start}\ \frac{}{(\texttt{HostSM}, \texttt{DevSM}, G) = (\{tcHost \mapsto 1\}, \{tcDev \mapsto 0\}, [0, \texttt{Host}, \{\}, P])}$$

$$\textsc{K-View-Host}\ \frac{t_{code} = x \leftarrow \texttt{View}(\texttt{Host}, e)\ ;\ s \quad e \Downarrow n \quad \mathfrak{vw} \notin \texttt{HostSM} \quad t_{es} = \texttt{Host}}{(\texttt{HostSM}, \texttt{DevSM}, G) \rightarrow (\texttt{HostSM}\ \cup\ \{\mathfrak{vw} \mapsto vdata(n), \forall 0 \leq i < n, *\mathfrak{vw}\langle i\rangle \mapsto 0\}, \texttt{DevSM}, G + t \xrightarrow{Seq} [t_{tid}, t_{es}, t_{mem} \cup \{x \mapsto \mathfrak{vw}\}, s]}$$

$$\textsc{K-View-Cross}\ \frac{t_{code} = x \leftarrow \texttt{View}(\texttt{Dev}, e)\ ;\ s \quad e \Downarrow n \quad \mathfrak{vw} \notin \texttt{DevSM} \quad t_{tid} = 0}{(\texttt{HostSM}, \texttt{DevSM}, G) \rightarrow (\texttt{HostSM}, \texttt{DevSM}\ \cup\ \{\mathfrak{vw} \mapsto vdata(n), \forall 0 \leq i < n, *\mathfrak{vw}\langle i\rangle \mapsto 0\}, G + t \xrightarrow{Seq} [t_{tid}, t_{es}, t_{mem} \cup \{x \mapsto \mathfrak{vw}\}, s]}$$

$$\textsc{K-View-Read}\ \frac{\begin{array}{c} t_{code} = \texttt{rd}\ x\ y\langle e\rangle\ ;\ s \qquad t_{mem}[y] = \mathfrak{vw} \qquad e \Downarrow n \\ \mathfrak{vw} \in \texttt{HostSM} \quad len(\mathfrak{vw}) > n \quad *\mathfrak{vw}\langle n\rangle = \mho \quad t_{es} = \texttt{Host} \end{array}}{(\texttt{HostSM}, \texttt{DevSM}, G) \rightarrow (\texttt{HostSM}, \texttt{DevSM}, G + t \xrightarrow{Seq} [t_{tid}, t_{es}, t_{mem}\ \cup\ \{x \mapsto \mho\}, s])}$$

$$\textsc{K-View-Write}\ \frac{t_{code} = \texttt{wr}\ x\ y\langle e\rangle\ ;\ s \quad t_{mem}[x] = \mho \quad t_{mem}[y] = \mathfrak{vw} \quad e \Downarrow n \quad \mathfrak{vw} \in \texttt{HostSM} \quad len(\mathfrak{vw}) > n \quad t_{es} = \texttt{Host}}{(\texttt{HostSM}, \texttt{DevSM}, G) \rightarrow (\texttt{HostSM}[*\mathfrak{vw}\langle n\rangle \mapsto \mho], \texttt{DevSM}, G + t \xrightarrow{Seq} [t_{tid}, t_{es}, t_{mem}, s])}$$

$$\textsc{K-ParFor-Host}\ \frac{t_{code} = \texttt{Parfor}(\texttt{Host}, e, s')\ ;\ s \quad e \Downarrow n \quad f_0 \ldots f_{n-1} \notin G \quad i \notin t_{mem} \quad t_{es} = \texttt{Host} \quad *tcHost = n_0}{\begin{array}{c} (\texttt{HostSM}, \texttt{DevSM}, G) \rightarrow (\texttt{HostSM}[tcHost \mapsto n_0 + n], \texttt{DevSM}, G + t \xrightarrow{Seq} [t_{tid}, t_{es}, t_{mem}, s] \\ +t \xrightarrow{Fork} [f_0, \texttt{Host}, t_{mem} \cup \{i \mapsto 0\}, s'] + \ldots + t \xrightarrow{Fork} [f_{n-1}, \texttt{Host}, t_{mem} \cup \{i \mapsto n-1\}, s']) \end{array}}$$

$$\textsc{K-ParFor-Cross}\ \frac{t_{code} = \texttt{Parfor}(\texttt{Dev}, e, s')\ ;\ s \quad e \Downarrow n \quad f_0 \ldots f_{n-1} \notin G \quad i \notin t_{mem} \quad t_{tid} = 0 \quad *tcDev = n_0}{\begin{array}{c} (\texttt{HostSM}, \texttt{DevSM}, G) \rightarrow (\texttt{HostSM}, \texttt{DevSM}[tcDev \mapsto n_0 + n], G + t \xrightarrow{Seq} [0, \texttt{Host}, t_{mem}, s] \\ +t \xrightarrow{Fork} [f_0, \texttt{Dev}, t_{mem} \cup \{i \mapsto 0\}, s'] + \ldots + t \xrightarrow{Fork} [f_{n-1}, \texttt{Dev}, t_{mem} \cup \{i \mapsto n-1\}, s']) \end{array}}$$

$$\textsc{K-Return}\ \frac{t_{code} = \texttt{ret} \quad t_{es} = \texttt{Host} \quad *tcHost = n}{(\texttt{HostSM}, \texttt{DevSM}, G) \rightarrow (\texttt{HostSM}[tcHost \mapsto n-1], \texttt{DevSM}, G)}$$

$$\textsc{K-Fence1}\ \frac{t_{code} = \texttt{Fence}(\texttt{Host})\ ;\ s \quad t_{tid} = 0 \quad *tcHost = 1 \quad S = \{u \mid u_{code} = \texttt{ret}, u_{es} = \texttt{Host}, out(u) = 0\}}{(\texttt{HostSM}, \texttt{DevSM}, G) \rightarrow (\texttt{HostSM}, \texttt{DevSM}, G + t \xrightarrow{Seq} [t_{tid}, t_{es}, t_{mem}, s] \xleftarrow{Join} u \in S)}$$

$$\textsc{K-Fence2}\ \frac{t_{code} = \texttt{Fence}(\texttt{Dev})\ ;\ s \quad t_{tid} = 0 \quad *tcDev = 0 \quad S = \{u \mid u_{code} = \texttt{ret}, u_{es} = \texttt{Dev}, out(u) = 0\}}{(\texttt{HostSM}, \texttt{DevSM}, G) \rightarrow (\texttt{HostSM}, \texttt{DevSM}, G + t \xrightarrow{Seq} [t_{tid}, t_{es}, t_{mem}, s] \xleftarrow{Join} u \in S)}$$

$$\textsc{K-DeepCopy}\ \frac{\begin{array}{c} t_{code} = \texttt{DeepCopy}(x, y)\ ;\ s \quad t_{mem}[x] = \mathfrak{vw}_1 \quad t_{mem}[y] = \mathfrak{vw}_2 \quad \mathfrak{vw}_1 \in \texttt{HostSM} \quad \mathfrak{vw}_2 \in \texttt{DevSM} \\ len(\mathfrak{vw}_1) = len(\mathfrak{vw}_2) = n \qquad *tcHost = 1 \qquad *tcDev = 0 \qquad t_{tid} = 0 \end{array}}{(\texttt{HostSM}, \texttt{DevSM}, G) \rightarrow (\texttt{HostSM}[\forall 0 \leq i < n, *\mathfrak{vw}_1\langle i\rangle \mapsto *\mathfrak{vw}_2\langle i\rangle], \texttt{DevSM}, G + t \xrightarrow{Seq} [0, \texttt{Host}, t_{mem}, s])}$$

Fig. 5. Small-step semantics for threads on Host. We omit the semantics for DeepCopy for other combinations of Host and Dev since they behave identically. Specifically, there are three additional rules for $\mathfrak{vw}_1, \mathfrak{vw}_2$ set to: 1). $\mathfrak{vw}_1 \in \texttt{DevSM}, \mathfrak{vw}_2 \in \texttt{HostSM}$ 2). $\mathfrak{vw}_1 \in \texttt{HostSM}, \mathfrak{vw}_2 \in \texttt{HostSM}$, and 3). $\mathfrak{vw}_1 \in \texttt{DevSM}, \mathfrak{vw}_2 \in \texttt{DevSM}$.

$$\text{K-VIEW-DEV}\ \frac{t_{code} = x \leftarrow \mathtt{View}(\mathtt{Dev}, e)\ ;\ s \quad e \Downarrow n \quad \mathfrak{vw} \notin \mathtt{DevSM} \quad t_{es} = \mathtt{Dev}}{(\mathtt{HostSM}, \mathtt{DevSM}, G) \rightarrow (\mathtt{HostSM}, \mathtt{DevSM}\ \cup\ \{\mathfrak{vw} \mapsto vdata(n), \forall 0 \leq i < n, *\mathfrak{vw}\langle i\rangle \mapsto 0\}, G + t \xrightarrow{Seq} [t_{tid}, t_{es}, t_{mem} \cup \{x \mapsto \mathfrak{vw}\}, s]}$$

$$\text{K-VIEW-READ-DEV}\ \frac{\begin{array}{c} t_{code} = \mathtt{rd}\ x\ y\langle e\rangle\ ;\ s \qquad t_{mem}[y] = \mathfrak{vw} \qquad e \Downarrow n \\ \mathfrak{vw} \in \mathtt{DevSM} \quad len(\mathfrak{vw}) > n \quad *\mathfrak{vw}\langle n\rangle = \mho \quad t_{es} = \mathtt{Dev} \end{array}}{(\mathtt{HostSM}, \mathtt{DevSM}, G) \rightarrow (\mathtt{HostSM}, \mathtt{DevSM}, G + t \xrightarrow{Seq} [t_{tid}, t_{es}, t_{mem}\ \cup\ \{x \mapsto \mho\}, s])}$$

$$\text{K-VIEW-WRITE-DEV}\ \frac{t_{code} = \mathtt{wr}\ x\ y\langle e\rangle\ ;\ s \quad t_{mem}[x] = \mho \quad t_{mem}[y] = \mathfrak{vw} \quad e \Downarrow n \quad \mathfrak{vw} \in \mathtt{DevSM} \quad len(\mathfrak{vw}) > n \quad t_{es} = \mathtt{Dev}}{(\mathtt{HostSM}, \mathtt{DevSM}, G) \rightarrow (\mathtt{HostSM}, \mathtt{DevSM}[*\mathfrak{vw}\langle n\rangle \mapsto \mho], G + t \xrightarrow{Seq} [t_{tid}, t_{es}, t_{mem}, s])}$$

$$\text{K-PARFOR-DEV}\ \frac{t_{code} = \mathtt{Parfor}(\mathtt{Dev}, e, s')\ ;\ s \quad e \Downarrow n \quad f_0 \ldots f_{n-1} \text{ not in G} \quad i \notin t_{mem} \quad t_{es} = \mathtt{Dev} \quad *tcDev = n_0}{\begin{array}{c} (\mathtt{HostSM}, \mathtt{DevSM}, G) \rightarrow (\mathtt{HostSM}, \mathtt{DevSM}[tcDev \mapsto n_0 + n], G + t \xrightarrow{Seq} [t_{tid}, t_{es}, t_{mem}, s] \\ +t \xrightarrow{Fork} [f_0, \mathtt{Dev}, t_{mem} \cup \{i \mapsto 0\}, s'] \\ +t \xrightarrow{Fork} [f_1, \mathtt{Dev}, t_{mem} \cup \{i \mapsto 1\}, s'] + \ldots + t \xrightarrow{Fork} [f_{n-1}, \mathtt{Dev}, t_{mem} \cup \{i \mapsto n-1\}, s']) \end{array}}$$

$$\text{K-RETURN-DEV}\ \frac{t_{code} = \mathtt{ret} \quad t_{es} = \mathtt{Dev} \quad *tcDev = n}{(\mathtt{HostSM}, \mathtt{DevSM}, G) \rightarrow (\mathtt{HostSM}, \mathtt{DevSM}[tcDev \mapsto n - 1], G)}$$

$$\text{K-IF-T}\ \frac{t_{code} = \mathtt{if}\ e\ \mathtt{then}\ s_1\ \mathtt{else}\ s_2; s \quad e \Downarrow n \quad n \geq 1}{(\mathtt{HostSM}, \mathtt{DevSM}, G) \rightarrow (\mathtt{HostSM}, \mathtt{DevSM}, G + t \xrightarrow{Seq} [t_{tid}, t_{es}, t_{mem}, s_1\ ;\ s])}$$

$$\text{K-IF-F}\ \frac{t_{code} = \mathtt{if}\ e\ \mathtt{then}\ s_1\ \mathtt{else}\ s_2; s \quad e \Downarrow 0}{(\mathtt{HostSM}, \mathtt{DevSM}, G) \rightarrow (\mathtt{HostSM}, \mathtt{DevSM}, G + t \xrightarrow{Seq} [t_{tid}, t_{es}, t_{mem}, s_2\ ;\ s])}$$

Fig. 6. Semantics for statements, and threads on Dev.

for nodes on Dev. Node t *accesses* a view $\mathfrak{vw}$, index d if node t reads from $\mathfrak{vw}\langle d\rangle$ or t writes to $\mathfrak{vw}\langle d\rangle$.

Definition V.3 (Data race and Data-race free)**.** A data race on view $\mathfrak{vw}$ index d occurs iff thread nodes t_1 and t_2 access $\mathfrak{vw}\langle d\rangle$, at least one of them conducts a write, and $t_1 \parallel t_2$.

We say that a computation graph G is *Data-Race Free* (DRF) if and only if for any view $\mathfrak{vw}$ index d, no data races occur on $\mathfrak{vw}\langle d\rangle$. We say that a program P is *Data-Race Free* if for any σ such that $P \rightarrow^{\star} \sigma$, $\sigma.G$ is data-race free.

Definition V.4 (Deadlock and Deadlock free)**.** Let $P \Downarrow \sigma$. A thread node $t \in \sigma.G$ is a *pending node* if $t_{code} = \mathtt{Fence}(ES); s$ and t has no outgoing continuation edge. We say that $P \Downarrow \sigma$ is *deadlocked* if there exists at least one pending node in $\sigma.G$. We say that program P is *Deadlock Free* (DF) if all σ such that $P \Downarrow \sigma$, are not deadlocked.

Definition V.5 (Spawn tree)**.** A spawn tree for a computation graph G, denoted as $st(G)$, is the subgraph containing all thread nodes and Fork/Seq edges.

Definition V.6 (Path)**.** A path in graph G, notation $W \in G$, is an alternating sequence $W = x_1 e_1 x_2 e_2 \ldots x_{k-1} e_{k-1} x_k$ of nodes x_i and edges e_j from G such that 1). the tail of e_i is x_i and the head of e_i is x_{i+1} for every i; and 2). all nodes and edges are distinct. The length of a path is the number of its arcs. We use :: as the "cons" operator for lists; that is, $W :: x$ appends x to the end of W. We say W is a path from x_1 to x_k or a $[x_1, x_k]$-path.

VI. PROPERTIES

In this section, we prove the *portability* of MiniKokkos programs. The property essentially means that without data races, different traces of a MiniKokkos program will always generate the same program state (same computation graph, same shared memories) when executed, regardless of the specific back-end used. In Kokkos itself, back-end refers to the target architectures for Host and Device (such as CUDA or OpenMP), but in MiniKokkos we can think of it as simply different thread counts and potential thread schedules; in MiniKokkos we treat back-ends as opaque and just denoted as Host or Dev.

Recall that in our computation graph, a thread node t is a quadruple $[f, ES, LM, s]$. This quadruple defines a node's identity. When we write "if t exists in G then t exists in H", we specify which node in H we are referring to: namely, the node that has the same $[f, ES, LM, s]$ as node t in G. It also assists us in interpreting path equality because path equality builds on node equality.

Lemma VI.1 (DRF property)**.** Let P be DRF and $P \Downarrow \sigma$. For $\sigma.G$, all writes to a view $\mathfrak{vw}$, index d are well-ordered. This means $W_1 \rightsquigarrow W_2 \rightsquigarrow \ldots \rightsquigarrow W_{n-1} \rightsquigarrow W_n$ if thread node W_i writes to $\mathfrak{vw}\langle d\rangle$. Any read to $\mathfrak{vw}\langle d\rangle$ is well-ordered between two consecutive writes. That is, we have $(R_1, R_2, \ldots, R_k) \rightsquigarrow W_0 \rightsquigarrow (R_{01}, R_{02}, \ldots, R_{0i}) \rightsquigarrow W_1 \rightsquigarrow (R_{11}, R_{12}, \ldots, R_{1j}) \rightsquigarrow (\ldots) \rightsquigarrow W_n \rightsquigarrow (R_{n1}, R_{n2}, \ldots, R_{nk})$.

Proof: Implied by the definition of DRF in Definition V.3. ■

Lemma VI.2 (Fence correctness)**.** The $\mathtt{Fence}(ES)$ rule correctly synchronizes all previous submitted work on ES.

Proof: Each Fence(Dev) statement will invoke the rules K-FENCE2 when the thread count is 0 on Dev; each Fence(Host) will invoke the rule K-FENCE1 when the thread count is 1 on Host. Thread count is decreased when an active thread finishes all its statement. This implies that when Fence is executed, any thread (except the master thread if $ES \mapsto \mathtt{Host}$) on the target ES is finished.

The continuation node t' will join all nodes v on the target ES where v_{code} = `ret` and v has no outing edges. The `ret` statement guarantees that v is the last node of a thread. The fact v has no outgoing edges indicates it has not been joined before. For those nodes u on ES such that u_{code} = `ret` and u has outgoing edge, they were joined by one previous thread node t^* such that ${t^*}_{code}$ = `Fence`$(ES), {t^*}_{tid} = 0$. We have $t^* \rightsquigarrow t'$; as a result, $u \rightsquigarrow t'$, which correctly maintains the happens-before relationship. ■

Theorem VI.1 (Deadlock Freedom/Progress)**.** A well-typed MiniKokkos program is deadlock free.

Proof: It suffices to prove the rules for synchronization(K-FENCE1,K-FENCE2,K-DEEPCOPY) will progress. The key is that Fence or DeepCopy statement cannot be called inside a parallel region. This means that they cannot be passed into a Parfor statement. Only the master thread can wait for other threads; the statements executed on other threads are non-blocking. As a result, circular waits cannot be generated by the syntax of MiniKokkos. ■

Theorem VI.2 (Spawn tree portability)**.** Let a MiniKokkos program P be well-typed and DRF. If $P \Downarrow \sigma_1$ and $P \Downarrow \sigma_2$, then $st(G_1) = st(G_2)$ where G_1 is $\sigma_1.G$ and G_2 is $\sigma_2.G$.

Proof: To conclude $st(G_1) = st(G_2)$, it is enough to show that for all $W \in st(G_1)$ where W is a $[root, v]$-path, we have $W \in st(G_2)$. We prove this claim by inducting on the length of W.

Base case: Length of W is 0, thus $v = root$ and $W = [root, root]$-path. The proof is trivial, since $root$ is in $st(G_2)$ and therefore $W \in st(G_2)$.

Inductive hypothesis: if $W \in st(G_1)$, $W = [root, v]$-path and the length of W is less than or equal n, then W exists in G_2.

Inductive step: We have $W \in st(G_1)$, $W = [root, u]$-path, the length of W is $n + 1$ and $W = W' :: (v, u)$. By our inductive hypothesis we know $W' \in st(G_2)$ and we want to prove $W \in st(G_2)$.

To show $W \in st(G_2)$, it is enough to show node $u \in st(G_2)$ and edge $(v, u) \in st(G_2)$. Because v is in both $st(G_1)$ and $st(G_2)$, they execute the same instruction. To proceed we now do a case analysis on the semantic rule used by $v \in st(G_2)$.

1) K-FENCE1,K-FENCE2: node u and edge (v, u) must exist in $st(G_2)$. When the rule is evoked in $st(G_2)$, v will create a continuation node and connect it with a cont edge. The continuation node will have the same thread id, execution space and local memory as v; furthermore, the statement is what follows the Fence statement in v. We can conclude that the continuation node is the same as node u in $st(G_1)$. We thus show that node u exist in $st(G_2)$, so does edge (v, u).
2) K-DEEPCOPY: node u and edge (v, u) must exist in $st(G_2)$ because this operation is non-blocking. When the rule is evoked in G_2, v will create a continuation node and connect it with a cont edge based on the rule. We can conclude that the continuation node is the same as node u in $st(G_1)$. We thus show that node u exist in $st(G_2)$, so does edge (v, u).
3) K-PARFOR-*: node u and edge (v, u) must exist in $st(G_2)$ because this operation is non-blocking and our naming system is consistent. There are two potential conditions based on the type of edge connecting v and u: $v \xrightarrow{Seq} u$ or $v \xrightarrow{Fork} u$. If the edge is a Seq edge, u is v's continuation node. In $st(G_2)$, we create u as v's continuation node when v performs the Parfor instruction. It will inherit the local memory of v because Parfor does not change the local memory of the calling thread.
   If the edge is a Fork edge, u is the first node in one of the newly created threads. In $st(G_2)$, when v performs the Parfor operation, we will create n new thread nodes. These n thread nodes will have the same thread ids as those thread nodes in $st(G_1)$ because our naming system is consistent. We then extend each thread node's local memory with a loop index, and connect v to each node with a Fork edge. Assume in $st(G_1)$, u has a loop index equals j. In $st(G_2)$, the newly created node with loop index j in its local memory must be u.
4) K-VIEW-READ-*: node u and edge (v, u) must exist in $st(G_2)$ because this operation is non-blocking and our naming system is consistent. In G_2, when v performs `wr` $x\ y\langle e\rangle$ where $v_{mem}[y] \mapsto \mathfrak{vw}$, we will create u as $v's$ continuation node. u will extend the local memory with the value read from $\mathfrak{vw}\langle d\rangle$ where $e \Downarrow d$. The value read from r will be the same as in $st(G_1)$ because of our DRF assumption.
5) K-VIEW-WRITE-*: node u and edge (v, u) must exist in $st(G_2)$ because this operation is non-blocking. The write operation does not change the local memory. When $v \in st(G_2)$ performs the write, we will create u as its continuation node.
6) All other rules: node u and edge (v, u) must exist in $st(G_2)$ because these operations are non-blocking and our naming system is unique and consistent.

■

Theorem VI.3 (Synchronization portability)**.** Let a MiniKokkos program P be well-typed and DRF. If $P \Downarrow \sigma_1, P \Downarrow \sigma_2$, edge $(v, u) \in \sigma_1.G$ and (v, u) is a join edge, then $(v, u) \in \sigma_2.G$.

Proof: By Theorem VI.2, we know $st(\sigma_1.G) = st(\sigma_2.G)$, so that node v, u exist in $\sigma_2.G$. We need to show that v is joined by u in $\sigma_2.G$. Recall that only master thread can call the `Fence`(ES) statement. In $\sigma_1.G$, nodes on master thread that joins other threads are well-ordered in the form $t_1 \rightsquigarrow t_2 \rightsquigarrow \cdots \rightsquigarrow t_{j-1} \rightsquigarrow t_j \rightsquigarrow u \rightsquigarrow \cdots$.

In $\sigma_1.G$, we have the relation $t_j \rightsquigarrow v \rightsquigarrow u$. This relationship indicates the statement that creates v is between t_j and u. In $\sigma_2.G$, v cannot be joined by a node t_k where $k \le j$ because it cannot be created before t_j.

By Theorem VI.1, the join edge (v, u) will be created after node u is created because v has not been joined before. As a result, v cannot be joined by any master thread node happens after u. ■

Theorem VI.4 (Memory portability)**.** Let a MiniKokkos program P be well-typed and DRF. If $P \Downarrow \sigma_1$ and $P \Downarrow \sigma_2$, then $\sigma_1.\texttt{HostSM} = \sigma_2.\texttt{HostSM}$ and $\sigma_1.\texttt{DevSM} = \sigma_2.\texttt{DevSM}$.

Proof: By Theorems VI.2 and VI.3 we know $\sigma_1.G = \sigma_2.G$. By Lemma VI.1, access to each shared memory location is well-ordered. Combined them together we can conclude that for each view $\mathfrak{vw}$, index d, all the access to $\mathfrak{vw}\langle d\rangle$ in $\sigma_1.G$ must also happen in $\sigma_2.G$ in the same order. When the program terminates, the shared memories will be the same for σ_1 and σ_2. ■

Theorem VI.5 (Portability)**.** Let a MiniKokkos program P be well-typed and DRF. If $P \Downarrow \sigma_1$ and $P \Downarrow \sigma_2$, then $\sigma_1 = \sigma_2$.

Proof: By Theorems VI.2 to VI.4. ■

This concludes the proofs about MiniKokkos, we now describe how designing and proving these properties of MiniKokkos can help reason about Kokkos itself.

VII. Case Study

MiniKokkos serves as a basis for formalizing Kokkos program behavior, particularly the concurrency patterns. We design MiniKokkos by consulting its wiki page, communicating with developers and carefully testing numerous programs. The result is a core language that users and developers can use for reference.

This section presents one bug in Kokkos's source code [4] that we identified in the process of developing MiniKokkos, which further reinforces the benefits of formalizing real-world parallel libraries like Kokkos. One of the DeepCopy templates in Kokkos accepts an execution space argument `exec_space`; the intended semantics are that "the call returns before the copy operation is executed. In that case the copy operation will occur only after any already submitted work to `exec_space` is finished, and the copy operation will be finished before any work submitted to `exec_space` after the DeepCopy call returns is executed." [5] The latest released code implementation is shown in Listing 1.

```
exec_space.fence(...);
Impl::view_copy(cpy_exec_space(),dst,src);
cpy_exec_space().fence(...);
```

Listing 1. Kokkos DeepCopy implementation (accessible on Github)

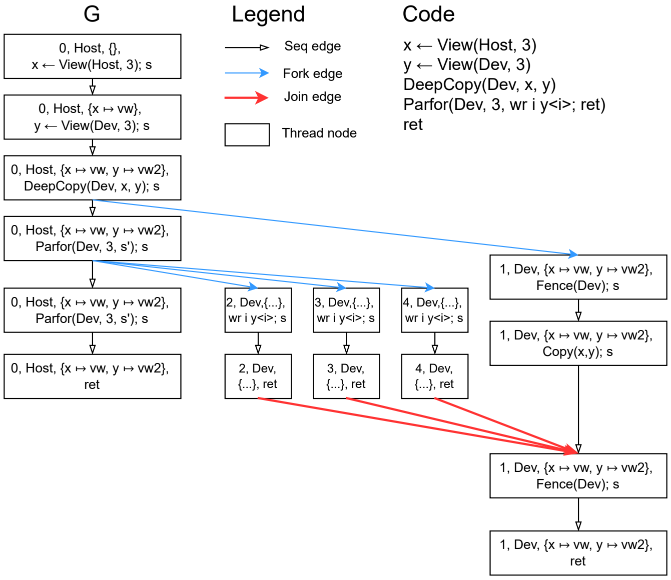
The corresponding MiniKokkos syntax that achieves the same purpose is shown in Fig. 7.

$$\frac{t_{code} = \texttt{DeepCopy}(\texttt{Dev}, x, y)\ ;\ s \qquad t_{tid} = 0}{\begin{array}{l} * \to (\texttt{HostSM}, \texttt{DevSM}, G + t \xrightarrow{Seq} [0, \texttt{Host}, t_{mem}, s] \\ +t \xrightarrow{Fork} [f, \texttt{Dev}, t_{mem}, \texttt{Fence}(\texttt{Dev}); Copy(x, y); \texttt{Fence}(\texttt{Dev}); \texttt{ret}]) \end{array}}$$

Fig. 7. Rule for $\texttt{DeepCopy}(\texttt{Dev}, x, y)$ that follows the implementation. The function Copy(x,y) copies the values in view y into view x if they exist in shared memory and have the same length.

In this rule, the $\texttt{DeepCopy}(\texttt{Dev}, x, y)$ can only be called by the master thread. The master thread will create a new thread on the device to do the copy, while itself will continue the execution on the host. The new thread will first wait for all work on the device to finish, then perform the copy and finally wait for the copy to finish before returning.

Some MiniKokkos rules would need changes to add this DeepCopy rule, such as adding the Copy function, allowing the copy thread to call the fence and access views in different shared memories. While we tried to integrate this rule into MiniKokkos, we realized that the functionality would not obey the wiki description. Although the implementation calls Fence before and after the copy operation, such a pattern does not prevent new work from being submitted to $exec\_space$ between line 1 and line 3. If any new work is submitted, it could run asynchronously with the copy statement on line 2; a race could occur if the new work also accesses the `dst` or `src` View. A possible scenario written in MiniKokkos is given in Fig. 8.



Code

```
x ← View(Host, 3)
y ← View(Dev, 3)
DeepCopy(Dev, x, y)
Parfor(Dev, 3, wr i y<i>; ret)
ret
```

Fig. 8. Example of potential data race in current DeepCopy implementation.

The computation graph records that new work was submitted to the device between the two fences in the copy region. Because there is no path connecting the node executing $Copy(x, y)$ in thread 1 and nodes executing $\texttt{wr}\ i\ y\langle i\rangle$ in threads 2–4, there exist races on each index of the view saved in y.

We reported this issue to Kokkos developers on Slack and opened a new GitHub issue [6]. One developer confirmed that the implementation needs modification to meet the desired behavior and that the fix would not be trivial.

VIII. Related Work

Programming language formalization has been widely used to analyze programs' behavior. Lee and Palsberg present a core calculus of the parallelism (async and finish) of X10 in their paper on Featherweight X10 [7]. They give a proof of deadlock-freedom, build a type system and perform static analysis of programs. We also took inspiration from the Legion programming language [8] and its computation graph. The key difference between MiniKokkos and past works is the modelling of execution spaces in MiniKokkos.

Atzeni and Gopalakrishnan introduce operational semantics for OpenMP programs [9], and it serves as a basis for a data race checker [10]. Designing a race detector based on MiniKokkos is also an exciting direction of future research. The reason is that MiniKokkos already records computation graph, and computation graph is often utilized by graph-based race detectors. A more recent work formalized cost semantics for CUDA [11]. Building a cost semantics is much more difficult for Kokkos because of the various supported backends and data movement between them.

Dynamic profiling and debugging of Kokkos has been integrated into Kokkos as an interface [12], however we are in the process of Merging MiniKokkos into the static analysis of Kokkos.

IX. Conclusion and Future Work

We presented MiniKokkos, a simple formal language used to model Kokkos. We described its syntax and semantics, then proved that desirable properties held for MiniKokkos, such as deadlock freedom and portability. A natural question then arose: how does MiniKokkos help with Kokkos itself? We provided one concrete piece of evidence suggesting its utility in Sec. VII, where we showed how the development of MiniKokkos resulted in the discovery of a bug in Kokkos itself. By designing MiniKokkos, our goal is to show a simple model of Kokkos to facilitate porting Kokkos to new back-ends and porting existing code bases to Kokkos. We plan to continue developing and formalizing the extensive work being undertaken by the Kokkos team to help improve their ecosystem. We are currently using MiniKokkos to guide development of an extension to the KLEE symbolic execution engine. This extension will model the semantics of Kokkos inside KLEE in order to detect bugs (such as data races) in Kokkos programs using static analysis.

A core calculus representing Kokkos opens up many avenues for research. The most straightforward direction is to add more features that Kokkos supports. Examples include multidimensional views and nested parallelism, both of which represent additional sources of bugs in Kokkos programs. As mentioned in Sec. II, we also wish to extend MiniKokkos's type system to model Kokkos and the features of C++ it uses, such as *concepts*, which are semantic constraints on types, similar to typeclasses in functional languages.

Programming in a smaller, simpler design language to reason about behavior, then re-implementing in a general-purpose programming language has shown success in distributed systems using languages such as TLA+ [13]. On a related note, one direction of MiniKokkos could be the development of an interpreter for MiniKokkos to enable rapid prototyping and simpler reasoning about parallel programs.

Acknowledgments

We thank Christian Trott for his insight and availability in helping us understand Kokkos and its design decisions and Shuo Ding from Georgia Tech for reviewing the operational semantics and formalization.

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

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, Sand No. SAND2022-13937 C.

References

[1] C. R. Trott, D. Lebrun-Grandié, D. Arndt, J. Ciesko, V. Dang, N. Ellingwood, R. Gayatri, E. Harvey, D. S. Hollman, D. Ibanez, N. Liber, J. Madsen, J. Miles, D. Poliakoff, A. Powell, S. Rajamanickam, M. Simberg, D. Sunderland, B. Turcksin, and J. Wilke, "Kokkos 3: Programming model extensions for the exascale era," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 4, pp. 805–817, 2022.

[2] G. E. Blelloch, *Vector Models for Data-Parallel Computing*. Cambridge, MA, USA: MIT Press, 1990.

[3] A. Church, "A formulation of the simple theory of types," *The Journal of Symbolic Logic*, vol. 5, no. 2, pp. 56–68, 1940. [Online]. Available: http://www.jstor.org/stable/2266170

[4] Sandia National Labs, "Kokkos," https://github.com/kokkos/kokkos, 2015.

[5] ——, "Kokkos documentation - deep copy semantics," https://kokkos.github.io/kokkos-core-wiki/API/core/view/deep_copy.html#semantics, 2015.

[6] F. Jin, "Potential race in deep_copy implementation," Jul., available at https://github.com/kokkos/kokkos/issues/5213.

[7] J. K. Lee and J. Palsberg, "Featherweight X10: A core calculus for async-finish parallelism," in *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '10. New York, NY, USA: Association for Computing Machinery, Jan. 2010, pp. 25–36. [Online]. Available: https://doi.org/10.1145/1693453.1693459

[8] S. Treichler, M. Bauer, and A. Aiken, "Language support for dynamic, hierarchical data partitioning," in *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, ser. OOPSLA '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 495–514. [Online]. Available: https://doi.org/10.1145/2509136.2509545

[9] S. Atzeni and G. Gopalakrishnan, "An operational semantic basis for building an openmp data race checker," in *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2018, pp. 395–404.

[10] S. Atzeni, G. Gopalakrishnan, Z. Rakamaric, I. Laguna, G. L. Lee, and D. H. Ahn, "Sword: A bounded memory-overhead detector of openmp data races in production runs," in *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2018, pp. 845–854.

[11] S. K. Muller and J. Hoffmann, "Modeling and analyzing evaluation cost of cuda kernels," *Proceedings of the ACM on Programming Languages*, vol. 5, no. POPL, Jan. 2021. [Online]. Available: https://doi.org/10.1145/3434306

[12] S. D. Hammond, C. R. Trott, D. Ibanez, and D. Sunderland, "Profiling and debugging support for the kokkos programming model," in *High Performance Computing*, R. Yokota, M. Weiland, J. Shalf, and S. Alam, Eds. Cham: Springer International Publishing, 2018, pp. 743–754.

[13] L. Lamport, *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.