Interactive Debugger for Performance Portable Python HPC Kernels

Ivan Grigorik
The University of Texas at Austin
Austin, Texas, USA
grigorik@utexas.edu

Gabriel Kosmacher
The University of Texas at Austin
Austin, Texas, USA
gkosmacher@utexas.edu

George Biros
The University of Texas at Austin
Austin, Texas, USA
gbiros@acm.org

Milos Gligoric
The University of Texas at Austin
Austin, Texas, USA
gligoric@utexas.edu

***Abstract*—We propose PKDB, the first interactive debugger for GPU and multithreaded low-level kernels written in Python. Python is widely used in high performance computing (HPC), with frameworks such as PyKokkos translating Python-embedded domain-specific languages to native code that runs across OpenMP-threaded CPUs and various GPUs. Yet interactive debugging support for such code is absent: developers resort to print statements, framework-specific assertions, or CPU-only execution, the last of which requires altering the program or its data and can mask device-specific bugs. PKDB enables standard interactive debugging like breakpoints, stepping, and variable inspection while preserving actual on-device execution without source modification. Beyond these fundamentals, PKDB introduces two advanced capabilities that exploit the dynamic nature of Python and PyKokkos: (i) Live code evaluation, which lets developers execute arbitrary Python expressions or entire kernels in the middle of a paused kernel without restarting the process; (ii) Kernel call site substitution, which allows an actively running kernel to be updated and reloaded on the fly, so only the kernel is recompiled and re-executed without restarting the application. Our performance evaluation on Intel, AMD, and NVIDIA CPUs and NVIDIA and AMD GPUs shows that PKDB introduces limited overhead and is practical for everyday use while introducing critical debugging features to the Python HPC ecosystem.**

***Index Terms*—High Performance Computing, Python, Debugging, Performance Portability.**

I. Introduction

The rapid diversification of high performance computing (HPC) hardware platforms, from OpenMP-threaded CPUs to GPUs and specialized accelerators, makes it increasingly challenging to develop HPC applications that run efficiently across different devices. Performance-portable frameworks such as Kokkos [1] address this by enabling developers to write code once and deploy it across different architectures without manual porting. While such frameworks were traditionally implemented in statically typed languages like C++, the growing need for rapid prototyping and seamless integration with machine learning workflows has driven the development of equivalent Python-based frameworks. These frameworks compile Python kernels just-in-time to C++ or other lower-level formats; PyKokkos [2], for instance, translates Python-embedded domain-specific language (eDSL) kernels directly to Kokkos C++, enabling developers to harness performance-portable HPC without leaving the Python ecosystem.

Despite substantial progress on frameworks covering new hardware backends, language features, and compilation pipelines, there has been minimal effort to provide tools that *assist the development process* itself. We designed and developed PKDB, the first interactive debugger for GPU and multithreaded low-level kernels written in Python, targeting PyKokkos kernels in particular.

The current state-of-practice for debugging Python eDSLs is embedded `print` statements [3, 4] or assertions [5, 6, 7]; a small subset of the debugging capabilities available to device-native kernel developers provided by CUDA-GDB [8] or ROCgdb [9]. As an alternative, developers may choose to execute the program serially on the CPU without translation [2, 10] or with partial translation; for complex workloads, this approach forces the developer to alter the program or its data to keep execution time manageable and may fail to expose corrupted behavior specific to parallel execution on the target device. Building a proper interactive debugger for this setting is itself non-trivial: it must bridge the Python host with JIT-compiled device kernels, map Python-level source concepts to dynamically generated device code, and unify distinct low-level debugging protocols for GPU (CUDA-GDB/ROCgdb) and OpenMP-threaded CPU targets.

PKDB is designed to be fully familiar to Python developers by mirroring the interface of pdb [11], integrating naturally into existing development workflows. At the same time, its architecture supports debugging application kernels executing on a device (e.g., a GPU), preserving actual on-device execution and performance, without source modification.

Beyond standard interactive debugging features such as breakpoints, stepping, continuation, and variable inspection, PKDB introduces two novel capabilities that exploit the dynamic nature of Python and PyKokkos: (1) *Live code evaluation*, which lets developers execute arbitrary Python expres-

SC26, November 15-20, 2026, Chicago, Illinois, USA
979-8-3195-4789-7/26/$31.00 ©2026 IEEE

sions or entire kernels in the middle of a paused kernel without restarting the process, enabling inspection or modification of program state at any point during debugging. To reduce the latency of live evaluation, PKDB can execute multiple kernel versions concurrently rather than sequentially. (2) *Kernel call site substitution*, which allows an actively running kernel to be updated and reloaded on the fly, so only the kernel is recompiled and re-executed without restarting the entire application. Due to its modular design, PKDB currently supports Intel, AMD and NVIDIA CPUs and NVIDIA and AMD GPUs; adding a new platform requires only implementing a platform-specific protocol without changes to the rest of the framework. The key contributions of this paper include:

- ⋆ **Conceptualization.** We present PKDB, the first interactive debugger for GPU and multithreaded low-level Python kernels. PKDB enables on-device debugging without source modification and integrates seamlessly into the familiar `pdb` interface, supporting context switching between Python and generated device code, and across heterogeneous accelerator types (e.g., CUDA and OpenMP), within a single session.
- ⋆ **Live code evaluation.** PKDB enables execution of arbitrary code—on the Python host or directly on the device—in the middle of a paused kernel session, giving developers the ability to inspect and modify program state at any point without restarting the application.
- ⋆ **Kernel call site substitution.** PKDB supports in-flight kernel changes, allowing developers to fix and continue kernel execution within the same debugger session without a full application restart.
- ⋆ **Performance evaluation.** We evaluate PKDB on representative PyKokkos workloads: ExaMiniMD, a Boltzmann-kinetics solver, and a periodic Ewald sum for the Stokes potential. We quantify end-to-end debugging over PKDB supported on Intel, NVIDIA and AMD CPUs and NVIDIA and AMD GPUs, compare against baseline debugging workflows, and measure the time saved by call site substitution versus a full application restart.

In doing so, PKDB brings long-overdue, first-class debugging support to developers of performance-portable Python HPC kernels. PKDB is publicly available at https://github.com/EngineeringSoftware/pkdb.

II. Background and Example

In this section, we provide a brief description of Kokkos (§II-A) and PyKokkos (§II-B), as well as showcase an example of a debugging session using PKDB (§II-C).

A. The Kokkos framework

Kokkos [12] is a C++ framework for developing performance portable HPC applications. It is implemented as a heavily templated, header-only library that maps a single source to diverse hardware backends, e.g., Intel, AMD and NVIDIA CPUs (`OpenMP`, `Threads`, `Serial`) and NVIDIA and AMD GPUs (`CUDA`, `HIP`).

Kernels in Kokkos are expressed as C++ functors or lambdas and launched via one of three *parallel dispatch* operations: `parallel_for` (map), `parallel_reduce` (reduction), and `parallel_scan` (prefix scan). Each dispatch is parameterized by two key abstractions: executions spaces and execution policies. An *execution space* (e.g., `CUDA`, `HIP`, `OpenMP`) selects the target device and its associated API. An *execution policy* governs how work is mapped onto that device: `RangePolicy` specifies a one-dimensional iteration range whose iterations are distributed across threads, while `TeamPolicy` introduces hierarchical parallelism by grouping threads into *teams* (analogous to CUDA thread blocks) that can synchronize and share local scratch memory. Data is managed through the `View` data structure, a multi-dimensional array abstraction whose memory spaces (which are logically distinct physical memory resources where data is stored) and layout are parameterized to match the target backend.

Despite its portability, Kokkos presents significant usability challenges. The heavy reliance on C++ template metaprogramming makes compiler diagnostics verbose and difficult to interpret, and explicit management of memory spaces and device placement adds overhead to the development workflow. Furthermore, C++ is a poor fit for developers who work primarily in Python—whether for rapid prototyping, data analysis, or integration with machine learning frameworks and libraries—creating a barrier between HPC kernel development and the broader scientific computing ecosystem.

B. PyKokkos

PyKokkos [2] was developed to remove the limitations of Kokkos for Python developers while preserving its performance portability. Developers write HPC kernels in a subset of Python (eDSL) and PyKokkos automatically translates, compiles, and executes them on the target device with minimal overhead. The eDSL mirrors the Kokkos dispatch API: kernels are functions decorated with `@pk.workunit` (Figure 1, line 4) and later launched via `parallel_for`, `parallel_reduce`, or `parallel_scan` and paired with execution spaces and policies as in Kokkos. Crucially, PyKokkos does not introduce any extra data abstractions: rather than requiring developers to manage Kokkos `Views` explicitly, it accepts standard Python array API [13] objects directly, e.g., NumPy and CuPy arrays, as well as PyTorch tensors. PyKokkos then converts these objects to `Views` internally with minimal overhead; keeping the Python side of the code idiomatic and compatible with the broader scientific and machine learning ecosystem. Because PyKokkos kernels remain ordinary Python functions at runtime, they retain the dynamic properties of Python—introspection, runtime code modification, and late binding—on which PKDB is built.

Figure 1 illustrates the PyKokkos workflow through a fused vector-matrix-vector operation $\boldsymbol{y}\boldsymbol{A}\boldsymbol{x}$, where $\boldsymbol{y} \in \mathbb{R}^N$, $\boldsymbol{x} \in \mathbb{R}^M$ and $\boldsymbol{A} \in \mathbb{R}^{N \times M}$. The input arrays are CuPy arrays instantiated on the GPU (lines 14–16). The kernel `yAx` (line 5) is decorated with `@pk.workunit` and launched by a `parallel_reduce` over a `RangePolicy` of N threads on the `CUDA` execution space (lines 18, 19). PyKokkos translates `yAx` into the Kokkos C++ functor shown in Figure 1b, compiles it just-in-time, and

```python
import cupy as cp
import pykokkos as pk

@pk.workunit    PyKokkos HPC kernel executed on a GPU
def yAx(j, acc, cols, y_view, x_view, A_view):
    temp2: float = 0
    for i in range(cols):
      temp2 += A_view[j][i] * x_view[i]
    acc += y_view[j] * temp2

def run() -> None:
    N = 256   # Rows
    M = 1024  # Cols
    y = cp.random.rand(N)
    x = cp.random.rand(M)
    A = cp.random.rand(N, M)
            Execution space set to be a GPU device
    space = pk.RangePolicy( pk.ExecutionSpace.Cuda , 0,N)
    pk.parallel_reduce(space, yAx, cols=M,
                       y_view=y, x_view=x, A_view=A)

if __name__ == "__main__":
    run()
```

(a) Python code with `PyKokkos` kernel (`yax.py`).

```
KOKKOS_FUNCTION void operator()(const yAx_tag &,
                    int32_t j, double &acc) const {
  double temp2 = 0;
  for (int32_t i = 0; i < cols; i += 1)
    temp2 += A_view(j, i) * x_view(i);
  acc += y_view(j) * temp2; };
```

(b) Generated C++ functor from `yAx` workunit.

Fig. 1: End-to-end PyKokkos example. 1a shows an HPC kernel (function annotated with `@pk.workunit`, lines 5–9) and the code that launches said kernel. All code without the kernel is executed by the standard Python interpreter, while the kernel is automatically translated (at the invocation time) to Kokkos (1b) and executed using the targeted execution space (CUDA in this example).

executes it on the GPU; the results are written back through pointers to the original Python objects and remain accessible to the rest of the Python runtime.

C. PKDB *example*

Figure 2 illustrates a PKDB debugging session on the `yAx` program from Figure 1. The developer sets breakpoints (line 1) by Python source line, either inside the `@pk.workunit` kernel or in the outer Python code, using a single, uniform interface. Internally, PKDB dispatches each command to the appropriate debugger: CUDA-GDB handles breakpoints inside the device kernel, and pdb handles breakpoints in the Python host code. This is entirely transparent to the developer, who never interacts with CUDA-GDB or pdb directly.

The session begins by setting breakpoints and issuing `run` (line 3). Execution first halts at a Python-side breakpoint (line 4), where the developer inspects `y` (line 6) and then continues. Execution halts inside the kernel at the `temp` variable definition (Figure 1, line 6); `continue` (line 8) moves deeper into device execution.

Once inside the kernel, the session illustrates two advanced capabilities. First, array slicing (lines 13–17): rather than transferring arrays into CUDA-GDB or pdb directly, PKDB accepts standard Python range syntax and fetches only the

```
1  (pkdb) break 6 9 18
2  Breakpoint set at yax.py:6 9 18
3  (pkdb) run
4  > /root/yax.py(18)run()
5  -> space = pk.RangePolicy(pk.ExecutionSpace.Cuda, 0,N)
6  (pkdb) print y[0:3]
7  array([0.81069483, 0.0676661 , 0.51309116])
8  (pkdb) continue
9  CUDA kernel breakpoint hit at yax.py:6
10 (pkdb-cuda) where
11 #0 /root/yax.py:6 (yAx)
12 #1 /root/yax.py:18 (main)
13 (pkdb-cuda) print y_view[0:2]
14 list([0.810694828106066, 0.06766610250020871])
15 (pkdb-cuda) print j
16 int(0)
17 (pkdb-cuda) print y_view[j:j+2]
18 list([0.810694828106066, 0.06766610250020871])
19 (pkdb-cuda) continue
20 CUDA kernel breakpoint hit at yax.py:9
21 (pkdb-cuda) eval device_sum(y_view)
22 float(134.55991968052976)
23 (pkdb-cuda) info cuda threads
24 * block_idx=(0,0,0)  thread_idx=(0,0,0)..(0,255,0)
       to_block_idx=(0,0,0)  count=256
25 (pkdb-cuda) cuda thread 0 10
26 (pkdb-cuda) print j
27 int(10)
28 (pkdb-cuda) finish
29 % > /root/yax.py()run()
30 (pkdb) sum(y)
31 array(134.55991968)
```

Fig. 2: Example PKDB debugging session on `yax.py` (from Figure 1).

requested slice, saving memory bandwidth. The values match those printed earlier on the Python side on line 6, confirming the kernel is receiving the correct input.[1] Second, live code evaluation (line 21): PKDB evaluates the expression `device_sum` directly on the GPU, computing the sum of `y_view` without moving data to the host; the result is cross-checked by evaluating the equivalent expression on the Python side (line 30).

The session also demonstrates kernel-specific behavior. By default, PKDB debugs the first thread, as shown on line 15, where printing `j`, the thread ID, returns 0. The developer then switches the CUDA context to block 0, thread 10 (line 25) and prints `j` again (line 26); its value now reads 10, confirming that `j` reflects the active thread's context.

III. Technique and Implementation

We describe the design and implementation of PKDB. We begin with an overview of the architecture in §III-A and specific features are detailed in §III-B–§III-D.

A. Architecture

PKDB is built around three cooperating processes: pdb*, a controller, and a target debugger. pdb*—our extension of the standard Python debugger pdb [11]—communicates with the controller over a Pseudo-Terminal (PTY); the controller in turn communicates with the target debugger over a second PTY. Figure 3 shows the resulting structure. We describe each step of a session below.

[1]Digits differ slightly as the default CUDA-GDB precision is greater than pdb precision.

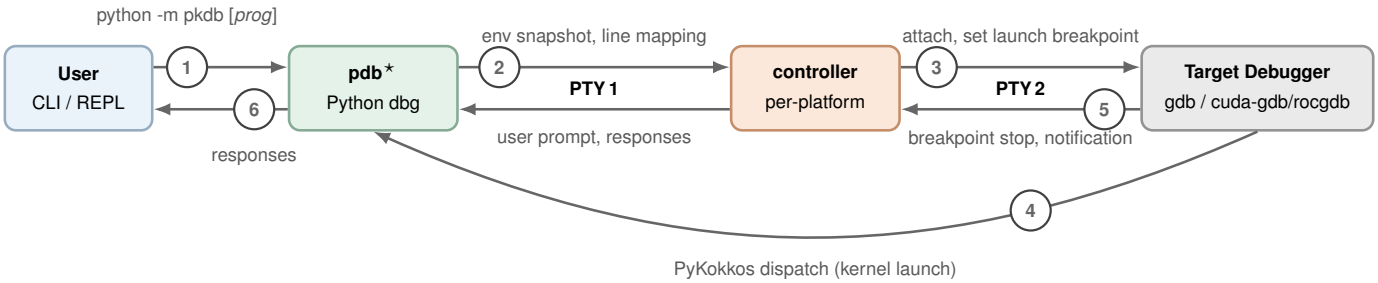

Fig. 3: PKDB architecture. Three cooperating processes communicate over two pseudo-terminals (PTY). The user launches pdb*, our extension of the standard Python debugger (Step 1). On reaching a PyKokkos parallel dispatch, pdb* spawns the controller and establishes PTY 1 (Step 2). The controller spawns and attaches the target debugger over PTY 2 (Step 3). pdb* then resumes and PyKokkos dispatches the kernel; the attached target debugger intercepts the launch (Step 4) and notifies the controller at breakpoints (Step 5). Subsequent user commands are relayed from pdb* through the controller to the target debugger; responses follow the same path in reverse (Step 6).

Step 1: pdb*. Execution begins when the user launches PKDB from the CLI via `python -m pkdb [program]`, which starts pdb*; additional commands are handled through pdb's well-established `cmd.Cmd` command interpreter protocol [14] (without modification)—the interface is immediately familiar to Python developers.

Step 2: Entering a parallel dispatch. Four preparatory actions are performed when execution reaches one of the PyKokkos parallel operations (`parallel_for`, `parallel_reduce`, `parallel_scan`):

(1) **Environment collection.** PKDB snapshots the Python environment—global variable names, available functions, imported modules—at the call site and passes the snapshot to the controller so that Python expressions can be evaluated directly from the device.

(2) **Execution-space discovery.** Arguments supplied to the parallel operation determine the effective execution space (e.g., `CUDA`, `HIP`, `OpenMP`) and PKDB selects the appropriate controller; because this discovery is performed independently at each parallel dispatch, a single PKDB session can debug a program that mixes execution spaces (e.g., a workload that dispatches some kernels to OpenMP and others to CUDA) without restarting the debugger.

(3) **Debug-mode compilation.** The kernel is compiled by PyKokkos with debug flags and optimizations disabled so that all debug information is preserved in the resulting binary; compiling without optimization is the standard approach during debugging.

(4) **Line-mapping construction.** PKDB builds a mapping from each PyKokkos kernel line to the corresponding line in the produced intermediate C++, which was linked to compiled binary in the previous step; the mapping is used to translate user-specified breakpoints for the target debugger.

After preparation, pdb* spawns a controller process, passes it the collected environment and line mapping, and establishes the first PTY channel for communication. The choice of controller implementation depends on the execution space identified above: for example, an `OpenMP` dispatch uses a `GDBController`, while a `CUDA` or `HIP` dispatch uses an `AcceleratorGDBController`. The controller is the only component that must be implemented per platform—all other parts of the architecture are platform-agnostic—and each controller must implement the common controller API (Table I). Table II lists implemented controllers and corresponding debuggers, as well as the platforms they support.

TABLE I: Controller API shared across targets in `pkdb`.

API	Description
`start_attached()`	Start an attached debugger session
`stop()`	Terminate debugger and stop the controller
`_setup_debugger()`	Initialize debugger process/PTY wiring
`send_command()`	Send one command to the target debugger
`_readline_debugger()`	Read one line of debugger output
`_transparent_loop()`	Relay output/events while in debug mode
`_command_mode()`	Run interactive user-input mode
`_read_command_line()`	Read/parse a user command line
`parse_and_execute()`	Parse and dispatch command
`command()`	Append function with command dispatch

TABLE II: Implemented PKDB controllers with the associated target debugger and the supported PyKokkos execution spaces (platforms).

Controller	Target debugger	Platform
`GDBController`	`gdb`	OpenMP
`AcceleratorGDBController`	`cuda-gdb`	CUDA
	`rocgdb`	HIP / ROCm

Step 3: Spawning the target debugger. The controller spawns a platform-provided *target debugger* (e.g., CUDA-GDB or ROCgdb) and attaches it to the pdb* process. A second PTY is established between the controller and the target debugger for all subsequent communication. The controller also configures the target debugger to break execution at kernel launch, so that control returns to pdb* immediately when the kernel begins.

Step 4: Kernel launch. pdb* triggers the kernel launch through the unmodified PyKokkos dispatch path. As soon as the kernel starts, the target debugger fires the launch breakpoint and notifies the controller, which processes the notification, checks if there is a debugging point set at the

TABLE III: PKDB commands exposed through pdb* and handled by the controllers in Table II. Commands marked with • apply to both OpenMP and CUDA/HIP sessions unless otherwise noted. Rows colored blue denote commands that exist in both pdb and PKDB; rows colored green denote commands unique to PKDB.

Command	Role	Platform
`break/delete`	Put breakpoint on workunit lines	•
`continue/step`	Continue execution	•
`quit`	Return to pdb*	•
`eval/print`	Evaluate expression / print variable	•
`locals`	Show local variables	•
`bt/whereami`	Show backtrace	•
`parallel_print`	Print across all threads	•
`hotswap`	Change kernel invocation	•
`parallel_launc`	Parallel launch of multiple kernels	•
`args`	Show frame arguments	•
`threads/thread`	Print / select GDB thread(s)	OMP
`cuda/roc info`	Show framework-tied command(s)	CUDA ROCm
`help/info`	Show command help	•
`send`	Send command directly to debugger	•
`set`	Set debug properties	•
`len`	Print `View` size	•

beginning of the kernel, and instructs the target debugger to continue to the first breakpoint inside the kernel.

Step 5: Breakpoint stops. The target debugger stops at a breakpoint and notifies the controller, which consults the line mapping and the set of user-defined breakpoints to determine whether the stop corresponds to a user breakpoint or execution error (e.g., an illegal memory access); in either case, the controller suspends execution and waits for user input.

Step 6: User interaction. The user issues commands through the pdb*, which is redirected to the controller; supported target commands are listed in Table III. The controller interprets each command, translates it into the appropriate instruction for the target debugger, and relays the response back to the user; default pdb commands are sent directly to pdb. Subsequent sections describe several of our custom commands.

B. Live code evaluation

PKDB allows for the evaluation of any arbitrary Python expression at any point in a session; the expression is executed as Python code in the environment collected at the parallel-dispatch boundary (Step 2 of §III-A), so it has access to the live global variables, functions, and modules at the breakpoint: a non-array assignment made by the evaluated expression (e.g., rebinding a global name) is visible to the rest of the program once execution resumes, just as array mutations are (the array case, described below, requires additional handling because array data may live outside pdb*'s own memory). The syntax for `eval` can be found in Figure 2 line 21: the user writes `eval` followed by the evaluated command, where `f` is any callable in the captured environment and `arr` is a CuPy array. `eval` may call an ordinary Python function, or it may dispatch a new PyKokkos kernel, hence PKDB effectively supports live kernel invocation at a stop without any modification of the debugged program.

When an expression argument is an array, PKDB passes it to the evaluation context without copying the underlying data—for a device array, copying large device buffers to the host is slow, and, for datasets that already fill device memory, can trigger out-of-memory errors; avoiding the copy eliminates both problems.

Algorithm 1 Invocation of device side during live code evaluation

1: **Input:** Device function name f_{name} (e.g., "device_sum"), arguments $\mathcal{D}_{\text{args}}$.
2: **Output:** Result of device-side execution.
3: $\mathcal{D}_{\text{dev}} \leftarrow [\,]$, $\mathcal{P}_{\text{dev}} \leftarrow \emptyset$
4: **for** each argument $d \in \mathcal{D}_{\text{args}}$ **do**
5: **if** `¬hasattr(d, data)` or `¬hasattr(d.data, ptr)` **then**
6: $\mathcal{D}_{\text{dev}}$.append(d)
7: continue
8: **end if**
9: ptr $\leftarrow d$.data.ptr
10: $h_{\text{IPC}} \leftarrow$ cupy.cuda.runtime.ipcGetMemHandle(ptr)
11: $\text{ptr}_{\text{dev}} \leftarrow$ ipcOpenMemHandle($h_{\text{IPC}}.hex$)
12: $d_{\text{dev}} \leftarrow$ cupy.ndarray(d.shape, d.dtype, ptr_{dev})
13: $\mathcal{D}_{\text{dev}}$.append(d_{dev})
14: $\mathcal{P}_{\text{dev}} \leftarrow \mathcal{P}_{\text{dev}} \cup \{\text{ptr}_{\text{dev}}\}$
15: **end for**
16: result $\leftarrow$ pykokkos.parallel_<op>($f_{\text{name}}, \mathcal{D}_{\text{dev}}$)
17: **for** each $\text{ptr}_{\text{dev}} \in \mathcal{P}_{\text{dev}}$ **do**
18: ipcCloseMemHandle(ptr_{dev})
19: **end for**
20: **return** result

PKDB avoids device-to-host copies via GPU IPC memory handles: for each array argument, pdb* obtains an IPC HandleInfo by calling `ipcGetMemHandle` on the array's device pointer and transmits the HandleInfo to the controller; the controller opens the HandleInfo with `ipcOpenMemHandle`, constructs a CuPy array view over the same device memory, and uses that view when invoking the expression. Because both processes alias the same device allocation, writes made by the evaluated expression are visible to the rest of the program; PKDB does not restrict mutability, that choice is left to the user. For host arrays on the OpenMP execution space, the shared memory region is accessed directly (i.e., without an IPC HandleInfo).

Algorithm 1 formalizes the IPC-based array-passing mechanism. PKDB iterates over the call's arguments (lines 4–15), skipping any argument that is not a device array; for each device array, it retrieves the device pointer, requests an IPC memory handle for it, and immediately opens that handle (line 11), reconstructing a CuPy array view over the same device allocation (line 12). The resulting view is added to the argument list $\mathcal{D}_{\text{dev}}$ and the opened device pointer to $\mathcal{P}_{\text{dev}}$ set. Once every argument has been processed, PKDB dispatches the function once with the collected set of CuPy views (line 16) and, once the call returns, closes all opened IPC handles before returning the result.

Algorithm 2 Kernel call site substitution

1: Global map $\mathcal{M}_{\text{glob}}$ and per-line map $\mathcal{M}_{\text{loc}}$ store substitutions $s \to t$.
2: **On** `hotswap` s t [ℓ]: store $s \to t$ in $\mathcal{M}_{\text{glob}}$ or in $\mathcal{M}_{\text{loc}}$ for ℓ.
3: **On** dispatch of a `parallel_<op>` call at line ℓ: let s be the invoked function; if $s \in \mathcal{M}_{\text{glob}}$ or $(s, \ell) \in \mathcal{M}_{\text{loc}}$, call **Apply**$(s, \ell)$.
4: **function** APPLY(s, ℓ)
5: *line* $\leftarrow$ source text at ℓ;
6: $\mathcal{H} \leftarrow \mathcal{M}_{\text{glob}}$ updated by $\mathcal{M}_{\text{loc}}[s, \ell]$ if defined.
7: $t \leftarrow \mathcal{H}[s]$.
8: Build `parallel_<op>`$'$ with t instead of s in *line*;
9: run `parallel_<op>`$'$ and get result r;
10: skip current ℓ by advancing to $\ell + 1$;
11: **return** r
12: **end function**

C. Kernel call site substitution

Kernel call site substitution lets the user replace kernels during a debug session without restarting the process. This is especially valuable in long-running HPC jobs, where a restart means reloading inputs, reallocating device memory, warming caches, and losing the failure point under inspection.

Some CPU debuggers, such as GDB, support reverse execution and time-travel debugging, which can be combined with on-the-fly patching [15]. However, accelerator-oriented debuggers such as CUDA-GDB expose a narrower feature set and do not support record/replay or similar mechanisms. Instead of using record/replay, PKDB relies on the dynamic nature of PyKokkos, where kernels are compiled and loaded on demand at runtime—a capability that, to our knowledge, no existing technique has exploited—and uses call-site interposition [16, 17]: at each `parallel_<op>` dispatch, PKDB checks whether the callee should be replaced and, if so, redirects the call without touching the Python source file on disk by changing the Python's reference to called function.

Commands. PKDB provides two forms of the `hotswap` command for kernel call site substitution: (1) `hotswap kernel1 kernel2` registers a global substitution—every future dispatch that would invoke `kernel1` is redirected to `kernel2` instead; (2) `hotswap lineno kernel1 kernel2` restricts the substitution to the single call site at `lineno`, leaving all other dispatches of `kernel1` unchanged.

The replacement kernel in both commands must match the signature of the original; if the signatures differ, PKDB reports an error and falls back to the original kernel.

Mechanism. Algorithm 2 formalizes how substitutions are recorded and applied. Global substitution map $\mathcal{M}_{\text{glob}}$ stores function that should be substituted $\mathbf{s}$ and function that we should substitute to $\mathbf{t}$; per-line rules in $\mathcal{M}_{\text{loc}}$ (line 1), and in addition to s and t stores location, where s should be substituted to t. HOTSWAP records the substitution. Because HOTSWAP never modifies the source file, a call site keeps invoking the same original function s on every subsequent dispatch, so PKDB must consult the maps again at each dispatch to learn whether s is currently substituted. Dispatch of a `parallel_<op>` call checks whether the invoked function s has a registered substitution in either map and, if so, invokes APPLY (line 3). **Apply** (line 4) resolves what s should be replaced with right now: it builds global state $\mathcal{H}$, which represents all currently-substituted functions. It's done by merging functions from $\mathcal{M}_{\text{glob}}$ overridden by any line-specific entry from $\mathcal{M}_{\text{loc}}$ for (s, ℓ) (line 6), so a per-line substitution always takes precedence over a global one at that call site; it then looks up the replacement $t \leftarrow \mathcal{H}[s]$ (line 7) and executes t in place of s (line 8), advancing the line counter so the original call is skipped entirely.

D. Concurrent kernel launch

At a breakpoint, a user may wish to run several kernels simultaneously rather than one at a time. Running kernels concurrently saves time and, importantly, allows their results to be compared directly—for example, to check whether two implementations of the same computation agree on the live data at a breakpoint. Additionally, kernels in different clauses may target different execution spaces, so that, e.g., variants running on GPU and CPU can be compared in a single command.

The command syntax consists of multiple clauses: `parallel_launch` c_1`;` c_2`; ...;` c_n, where each c_i clause specifies a parallel operation, a kernel name, an optional execution space (with the `pk.` prefix, e.g., `pk.Cuda`), `RangePolicy`, and arguments. All names and expressions are resolved via Python `eval` against the live environment collected at the parallel-dispatch boundary, so symbols such as `N` or array objects refer to current program state. For example, the `yAx` kernel from Figure 1a can be launched on GPU and on OpenMP with `parallel_launch yAx pk.Cuda pk.RangePolicy(0,N) y x A; yAx pk.OpenMP pk.RangePolicy(0,N) y x A`; a failure in one clause is reported but does not prevent the remaining clauses from executing.

By default, array arguments are treated as read-only: each clause receives an isolated copy of the data so that concurrent kernels do not interfere. For CuPy arrays (GPU execution spaces), the copy is taken as a host snapshot; for NumPy arrays (OpenMP execution spaces), a separate buffer is allocated on the CPU .

A user can opt out of this isolation by marking an argument as *mutable* using the `mut:` prefix (e.g., `mut:arr` or `mut:arr=<expr>`). Mutable CuPy arrays are shared across processes via IPC without copying, exactly as in live code evaluation (§III-B); mutable NumPy arrays are passed by reference so the kernel and the parent alias the same storage. If the same buffer is marked mutable in multiple concurrent clauses, races and silent corruption are possible; PKDB applies the annotations directly and leaves correctness to the user.

Algorithm 3 Thread-aware program execution control

Input: Breakpoint set $\mathcal{B}$; active threads $\mathcal{T}$.
Output: Focused thread t stopped at some $b \in \mathcal{B}$.
`gdb-set scheduler-locking off`
send `continue` to all threads in $\mathcal{T}$
Wait until some $t \in \mathcal{T}$ reaches a location in $\mathcal{B}$
`gdb-set scheduler-locking on`
return t

TABLE IV: Machines used in evaluation. For CPU evaluation, we set `OMP_NUM_THREADS` to the number of available CPU cores.

Machine	CPU	GPU
Local	Intel Xeon w5, 32 cores	NVIDIA RTX 5000, 32 GB
V7	NVIDIA Grace, 72 cores	NVIDIA H200, 96 GB
AMC	AMD EPYC, 96 cores	AMD MI300X, 192 GB

E. Thread-aware continue

When debugging OpenMP kernels, a `continue` command can cause the debugger to switch away from the current thread if a different thread hits a breakpoint first. To give the user predictable control, PKDB provides an extended continue command. `continue threads [begin:end]` which resumes only the specified range of threads. Issuing `continue` without a range resumes all threads.

IV. Evaluation

We evaluate the user-independent factors of PKDB by performing experiments to answer the following questions:

(1) **Debug overhead** (§IV-C): How does wall time differ between PKDB's debug mode versus pdb debug execution, on `CUDA`, `HIP` and `OpenMP` backends?
(2) **Comparison with the PyKokkos debug mode** (§IV-D): The existing PyKokkos Debug execution mode lowers all parallel work to serial, Python-only execution for debugging; how do these times differ compared to PKDB, which dispatches to the unmodified GPU/OpenMP backend during debugging?
(3) **Kernel call site substitution cost** (§IV-E): What is the wall-clock cost of substituting a kernel call by editing the source and restarting the session (edit-and-debug) versus issuing `hotswap` inside a running session (call site substitution)?

Additionally, we provide two case studies (§IV-F) where PKDB is used to debug two PyKokkos research applications: a Boltzmann particle-in-cell kinetics code and an Ewald summation code.

We describe the evaluation setup (§IV-A) and subjects (§IV-B) before addressing each experiment (§IV-C-§IV-E) and the case studies (§IV-F).

A. Evaluation setup

Experiments are run on the machines listed in Table IV. Each experiment is run four times, with a dry run discarded and results reported as averages over the remaining times. Moreover, for Section IV-C we disabled PKDB's optimization that skips kernel debugging if there is no breakpoint inside of the kernel. This lets us measure the overhead of the full PKDB framework, not just pdb* in isolation.

B. Subjects

We briefly describe subjects used in our evaluation.

ExaMiniMD. The ExaMiniMD mini-application is a modular codebase designed to investigate performance of common kernels in particle code frameworks [18]. ExaMiniMD was ported from Kokkos C++ to PyKokkos [2], where the total execution time was shown to be competitive. Given an N atom system, the execution flow comprises the following steps: (1) `Initialize` position, velocity, and force arrays of size $3N$; (2) `Compute` temperature, potential energy, and kinetic energy of the system using PyKokkos parallel kernels; (3) `Update` position, velocity, and force arrays; (4) `Repeat` steps 2 and 3 for 100 time steps.

Boltzmann. The Boltzmann kinetic equations solver is a particle-in-cell code that uses NumPy/CuPy for linear algebra and PyKokkos for parallel kernels [19]. The scheme has three main steps: (1) `Collision`: compute collision probabilities, requires on-device random number generation, reshuffling in memory and atomics; (2) `Recombination`: compute a three-body problem, requires two distinct kernels with low arithmetic intensity and atomics; (3) `Particle-to-Cell`: compute per-cell right-hand-side vectors, requires a high arithmetic intensity kernel with atomics.

Ewald. Ewald summation—used to accelerate molecular dynamics simulations and potential theory calculations—comprises the following steps: (1) `P2P`: interactions between particles within a cutoff; (2) `P2G`: spread particle forces onto the grid by convolving with a compactly supported window function; (3) `FFT`: compute Fourier space grid forces; (4) `CNV`: apply convolution in Fourier space for the grid velocities; (5) `IFFT`: compute real space grid velocities; (6) `G2P`: interpolate the grid velocities to particles. The algorithmic implementation, given in [20], uses PyKokkos for parallel kernels and the NumPy/CuPy libraries for data structures and FFTs. The `P2P`, `P2G` and `CNV` provide representative HPC benchmarks: `P2P` has a high arithmetic intensity and a spatially local cache structure, `P2G` uses shared memory and atomic operations, and `CNV` has a low arithmetic intensity.

C. Debug overhead

ExaMiniMD. Figures 4a, 4d and 4g show total ExaMiniMD execution time for pdb and PKDB debugging sessions across multiple backends as the atom count increases. Table V reports min–max wall times and the mean per-point ratio $t^{\text{PKDB}}/t^{\text{pdb}}$for each configuration; for ExaMiniMD, the mean does not exceed $1.75\times$ for OpenMP, $1.95\times$ for CUDA, and $1.54\times$ for HIP.

Boltzmann. Figures 4b, 4e, and 4h show that pdb and PKDB runs track each other closely on each backend across systems. Table V reports min–max wall times and the mean per-point ratio $t^{\text{PKDB}}/t^{\text{pdb}}$for each configuration; for Boltzmann, that mean does not exceed $2.18\times$ for OpenMP, $2.33\times$ for CUDA, and $2.14\times$ for HIP.

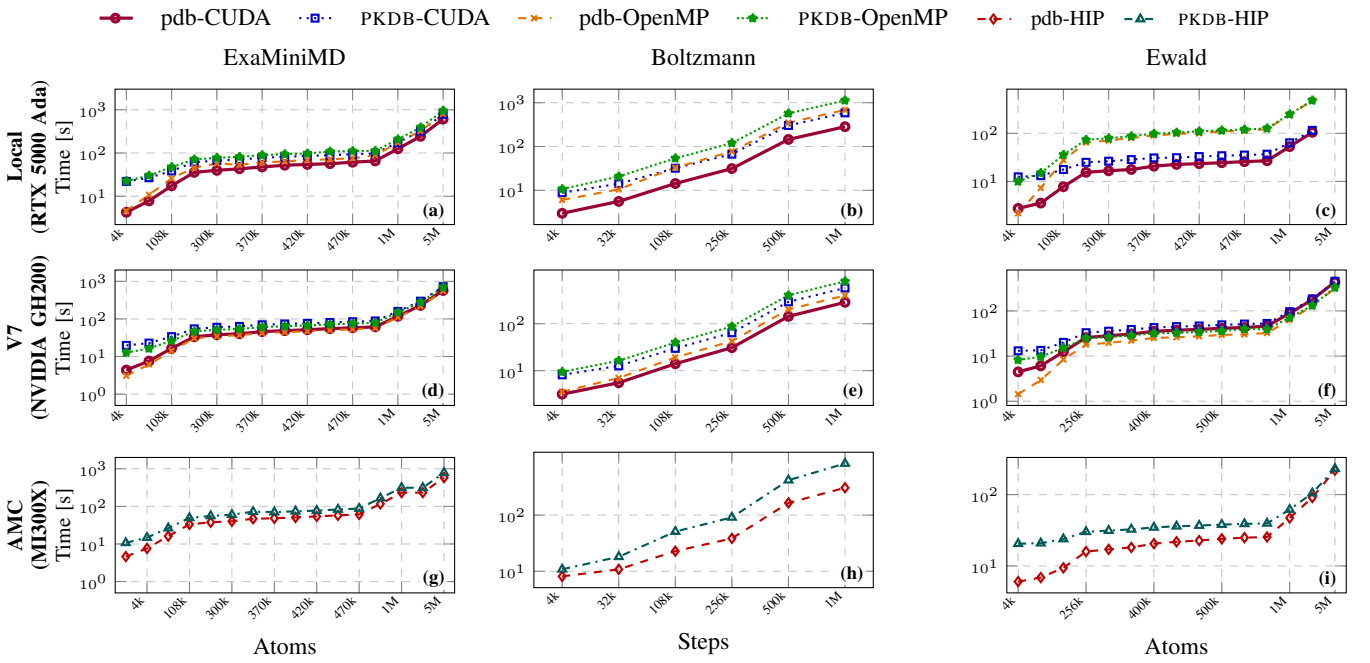

Fig. 4: Wall time for pdb vs. PKDB (instrumented) runs. *Columns* (left to right): ExaMiniMD, Boltzmann solver, Ewald summation. *Rows:* **(a)–(c)** Local/NVIDIA RTX 5000 Ada; **(d)–(f)** V7/NVIDIA GH200; **(g)–(i)** AMC/AMD MI300X.

TABLE V: Wall-time ranges (seconds) for Boltzmann, ExaMiniMD, and Ewald on the Local, V7, and AMC node (Sec. IV-C). Each range is the min–max time over all sizes in the corresponding execution-time figures; the last column is the mean of $t_i^{\text{PKDB}}/t_i^{\text{pdb}}$ over those same points.

Benchmark	Processor	Execution space	pdb (s)	PKDB (s)	Overhead (×)
ExaMiniMD	Xeon/RTX	OpenMP	4.6–800.7	22.4–952.0	1.75
		CUDA	4.3–607.7	22.0–782.9	1.95
	GH200	OpenMP	3.1–551.2	12.7–693.2	1.70
		CUDA	4.4–570.5	19.7–722.1	1.81
	MI300X	HIP	4.6–581.8	10.7–787.3	1.54
Boltzmann	Xeon/RTX	OpenMP	6.0–684.2	10.7–1,128.0	1.70
		CUDA	3.0–283.3	8.8–587.7	2.33
	GH200	OpenMP	3.6–399.7	9.4–800.1	2.18
		CUDA	3.2–285.1	8.1–576.1	2.19
	MI300X	HIP	8.1–306.3	10.8–826.9	2.14
Ewald	Xeon/RTX	OpenMP	2.2–478.7	10.0–482.0	1.41
		CUDA	2.7–104.9	12.5–115.1	1.86
	GH200	OpenMP	1.4–316.0	8.2–329.6	1.70
		CUDA	4.5–438.5	13.1–452.1	1.39
	MI300X	HIP	6.0–219.8	20.5–230.7	1.84

Ewald. Figures 4c, 4f and 4i report total Ewald time for each configuration. Note that for Figure 4c we don't have a runtime for 5M, due to lack of memory on Local server, which is unrelated to PKDB. Table V reports min–max wall times and the mean per-point ratio $t^{\text{PKDB}}/t^{\text{pdb}}$for each configuration; for Ewald, that mean does not exceed 1.70× for OpenMP, 1.86× for CUDA, and 1.84× for HIP.

D. Comparison with PyKokkos Debug mode

Figures 5, 6, and 7 compare ExaMiniMD debug wall time for two strategies. PyKokkos-Debug denotes the existing debug execution mode: parallel kernels are lowered to sequential, Python-only execution and debugged with pdb—a common

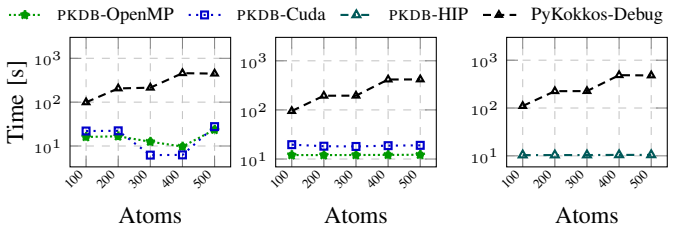

Fig. 5: ExaMiniMD debug wall time (Local, RTX 5000).

Fig. 6: ExaMiniMD debug wall time (V7, GH200).

Fig. 7: ExaMiniMD debug wall time (AMC, HIP).

Python eDSL debugging approach [10, 21]—and compared with PKDB debugging with CUDA, OpenMP, and HIP backends, where we observe significant speedups over PyKokkos-Debug.

We plot timings across backends for atom counts from 100 up to 500; beyond that range the PyKokkos-Debug quickly becomes impractical because each timestep runs sequentially on the host, inside the Python interpreter. Across the full size range, PyKokkos-Debug spans 99.17–457.62s on Local and 94.83–417.80s on V7, while PKDB-OpenMP stays between 9.72 and 23.57s on Local (12.02–12.20s on V7), and PKDB-CUDA between 6.20 and 27.66s on Local (18.03–19.61s on V7); on AMC, PyKokkos-Debug spans 110.13–486.01s and PKDB-HIP stays between 10.36 and 10.46s.

E. Kernel call site substitution cost

To evaluate kernel call site substitution, we modified the ExaMiniMD benchmark by adding four alternate workunits

TABLE VI: ExaMiniMD kernel call site substitution benchmark with atom size of 420k, executed on Local/NVIDIA RTX 5000 Ada. All times in seconds except t_{hs} (milliseconds): t_A is two-run edit-and-debug with pdb total; t_{A1}, t_{A2} are the first and second run of pdb sessions; t_B uses `hotswap` during PKDB interactive session; t_{hs} is substitution command latency.

Preset	t_A **(s)**	t_{A1} **(s)**	t_{A2} **(s)**	t_B **(s)**	**Spd. (×)**	t_{hs} **(ms)**
Temperature	87.11	43.76	43.34	66.55	1.31	8.63
KinE	88.25	44.41	43.84	67.18	1.31	0.65
NVE initial	89.64	44.90	44.75	67.78	1.32	6.21
NVE final	89.08	44.22	44.86	67.77	1.31	6.27
Mean	88.52	44.32	44.20	67.32	1.31	5.44

that a programmer could swap in at the `parallel_<op>` call site[2]. The alternate kernels are:

(1) **Temperature reduction**: `compute_workunit` swapped for `compute_workunit_half`, which is a half-scaled kernel.
(2) **Energy reduction**: `work` swapped for `work_half`, which is a half-scaled kernel.
(3) **First integrator**: `initial_integrate` swapped for `initial_integrate_clean` (zeros `v` and `x` for probing).
(4) **Last integrator**: `final_integrate` swapped for `final_integrate_clean` (zeros `v` for probing).

Regarding size, we picked median value from our sample, which is 420k. All kernels were tested using CUDA execution space and compared against pdb edit-and-debug sessions.

We compare the efficiency of our call site substitution implementation to relaunching separate debugging sessions after each on-disk edit, using a fully automated experiment: (1) launch PKDB in a subprocess; (2) set breakpoints before and after the kernel call; (3) run **edit-and-debug**—save changes on disk and launch two separate PKDB sessions; (4) run **call site substitution**—launch a single session that issues `hotswap` at the call site; (5) record wall-clock intervals and verify debugger outputs.

Results are reported in Table VI. We target kernels that run early in each ExaMiniMD timestep so both cases reach the inspection point after modest work[3]. Both cases exit immediately after the kernel returns.

As expected, **edit-and-debug** is more expensive than **call site substitution**, as it launches a second PKDB session and reruns the program up to the call sites after each edit. The `hotswap` command itself adds only tens of milliseconds of wall time (Table VI, t_{hs}), a negligible overhead.

F. Case studies

We use PKDB to debug the Boltzmann and Ewald PyKokkos applications.

1) Boltzmann: While developing and evaluating PKDB on the Boltzmann benchmark, we encountered a bug, which we then debugged using PKDB. The Boltzmann benchmark failed during CUDA execution: the run aborted with `cudaErrorIllegalAddress` on the line 1 in Figure 8. Here, the argument memory spaces and layouts looked consistent with a legal launch, yet we could only stop on the first line of the workunit and step through to line 8 (Figure 9), after which the benchmark crashed with the error messages in Figure 10.

```
1  cudaDeviceSynchronize() error( cudaErrorIllegalAddress)
       : an illegal memory access .../Cuda/
       Kokkos_Cuda_Instance.cpp:154
2  Backtrace:
3  [0x7d...c9] Kokkos::Impl::save_stacktrace()
4  [0x7d...50] Kokkos::Impl::host_abort(char const*)
5  [0x7d...bb] Kokkos::Impl::cuda_internal_error_abort
       (...)
6  [0x7d...1a] Kokkos::Impl::cuda_device_synchronize(...)
7  [0x7d...0d] Kokkos::Impl::ExecSpaceManager::
       static_fence(...)
8  [0x7d...d1] void Kokkos::deep_copy<...>(...)
9  [0x7d...0e] run(...)
10                              ...
11 [0x5a...3e] _start
12 Aborted (core dumped)
```

Fig. 8: Boltzmann benchmark: `cudaErrorIllegalAddress` at `cudaDeviceSynchronize` (Kokkos `deep_copy` between `CudaSpace` and `CudaUVMSpace`).

```
1  @pk.workunit(scratch=[(float, lambda p, s: s.scratch_s)
       ])
2  def redcheck(self, t: pk.TeamMember):
3      i: int = t.league_rank() * t.team_size() + t.
           team_rank()
4      tid: int = t.team_rank()
5      if i < self.M:
6          bb: pk.ScratchView1D[float] = pk.ScratchView1D(t
               .team_scratch(0))
7          for j in range(self.r):
8              bb[self.r*tid + j] = self.w[self.r*i + j]
9          t.team_barrier()
```

Fig. 9: Boltzmann `redcheck` workunit definition under debugging.

This error message suggested that the scratch size was not correctly specified; we discovered that the PyKokkos scratch size specification via the lambda definition on line 1 was broken: PyKokkos correctly registered the scratch size, but failed to translate the scratch specification to C++. As `pk.TeamMember` on line 2 was translated to the C++, we used native debugger-oriented command `send print this->team_size`, which shows size of current team of threads, and observed that the team size was set to 0. Thus, the cache was not allocated for the team, since there is no team at all. After resolving this issue in PyKokkos, the Boltzmann benchmark ran successfully.

2) Ewald: During Ewald benchmark execution for PKDB timing comparison, we encountered a non-deterministic failure with a cryptic error message for CUDA builds and silent fails for OpenMP builds. Figure 11 shows console logs (11a, 11b), the relevant `celllist_workunits.py` listing with marked lines (11c), and a PKDB transcript (11d). The OpenMP run (line 1 in Figure 11a) terminates with only a shell-level segmentation fault, whereas the CUDA run (Figure 11b) surfaces `cudaErrorIllegalAddress` at `cudaStreamSynchronize`

[2]We choose one call site as **edit-and-debug** scales linearly while **call site substitution** has constant asymptotic time (given infinite processes).

[3]Setting a breakpoint later in the execution would increase **edit-and-debug**'s runtime but have no effect on **call site substitution** runtime.

```
[pkdb-cuda] accelerator debugger: CUDA Exception: Warp Illegal Address
[pkdb-cuda] accelerator debugger: The exception was triggered at PC 0x7c..10 pk_functor_Scale<Kokkos::Cuda>::operator()(...) const (functor.hpp:37)
[pkdb-cuda] accelerator debugger: Thread 15 "cuda-EvtHandlr" received signal CUDA_EXCEPTION_14, Warp Illegal Address.
```

Fig. 10: Boltzmann session: pkdb CUDA exception reporting warp illegal address at generated functor code (`functor.hpp:37`).

with `reshuffle_particles_fp64` on the stack (l ine 6).

Guided by the CUDA backtrace in Figure 11b, we set a breakpoint inside of `reshuffle_particles_fp64` workunit on the line 9 in Figure 11c, and stepped through the launch path. Evaluating Python-side expressions in PKDB (e.g., applying `len(<arg>)` to the arguments) revealed out-of-bounds indexing. Lines 5, 6 in Figure 11d show the length of array, while lines 17, 18 show that Ewald is trying access element with 528 index.

Continuing, we discovered a bug in `_get_cell_fp64` (Figure 11c, line 4), which computes the cell index; the function mishandles the case where a particle lies exactly on a cell boundary. Figure 11d shows a representative PKDB trace: after printing `_grid_shape`, we stop first at line 9, Figure 11c (the `_get_cell` call in `reshuffle_particles_fp64`) and then at line 4, Figure 11c inside the helper. We issue the `parallel_print` instruction at the device breakpoint to print `cell_xyz`. The `parallel_print` evaluates the expression once per accelerator thread and prints the resulting values as a thread-indexed list. We then continue to the following line in the reshuffle workunit (Figure 11c, line 10), where `pk.atomic_add` updates `counter` at the computed cell index, and observe that this index is out of bounds for `counter` (yellow highlight in the figure).

PKDB was able to help us find a bug in the Ewald research code, not a synthetic regression harness; leading the developers to the exact location of the bug to be fixed.

This case study is a natural fit for HPC-oriented debugging: PKDB can issue device-side instructions (e.g., `parallel_print <args>`), and inspect live per-thread state in one session, rather than being limited to a single active host thread when a parallel kernel misbehaves.

V. Limitations and Future Work

Like most full-featured debuggers, PKDB trades raw performance for observability: end-to-end runs with the debugger-instrumented Kokkos build are slower than with the paired release build. However, the performance overhead, based on our evaluation (IV-C), is moderate.

We selected PyKokkos as a proof-of-concept framework. However, we believe that the core ideas behind PKDB generalize to other eDSLs (such as Triton [5], Numba [22], Pallas [23]) and other mixed-language stacks, e.g., Python front ends that invoke native CUDA, HIP, or OpenMP code through pybind11 or similar bindings, but substantial engineering might be required to implement a controller and breakpoint handling.

```
Segmentation fault (core dumped) python celllist.py --device OpenMP
```

(a) Host OpenMP: `celllist.py` aborts with no Python traceback.

```
cudaStreamSynchronize(stream) error(cudaErrorIllegalAddress): an illegal memory access
  ../Cuda/Kokkos_Cuda_Instance.cpp:165
Backtrace:
[0x72..19] Kokkos::Impl::save_stacktrace()
[0x72..d0] Kokkos::Impl::host_abort(char const*)
                        ...
[0x72..39] reshuffle_particles_fp64(*kwargs)
                        ...
[0x72..40] __libc_start_main
[0x5d..3e] _start
Aborted (core dumped) python celllist.py --device Cuda
```

(b) CUDA: Kokkos abort at `cudaStreamSynchronize` after `reshuffle_particles_fp64`.

```
@pk.function
def _get_cell_fp64(p, ...) →int:
    ...
    cell_xyz = [p[0][i] / c_size[0], ..., ...]
    return get_idx(cell_xyz)

@pk.workunit
def reshuffle_particles_fp64(i, counter, p, ...):
    cell: int = _get_cell_fp64(p, ...)
    offset: int = pk.atomic_add(counter, [cell], 1)
    ...
```

(c) Illustrative part of (`celllist_workunits.py`): cell indexing and reshuffle path.

```
(pkdb) print _grid_shape
array([9, 8, 7])
(pkdb) continue
Workunit breakpoint, celllist_workunits.py:9
(pkdb-device) print len(counter)
int(504)
(pkdb-device) continue
Workunit breakpoint, celllist_workunits.py:4
(pkdb-device) parallel_print cell_xyz
tid | cell_xyz
--------------
0 | [9, 7, 0]
...
9 | [2, 8, 2]
(pkdb-device) continue
Workunit breakpoint, celllist_workunits.py:10
(pkdb-device) print cell
int(528)
```

(d) pkdb session: host-sided `print _grid_shape`, then workunit stops at `celllist_workunits.py` lines 4 and 9, and `parallel_print` of `cell_xyz` per thread.

Fig. 11: Ewald case study: failure logs (a–b); broken `reshuffle/_get_cell` code with lines of interest marked (c); and a PKDB transcript stopping at those lines (d).

Our future work involves several research directions. The first is user experience, which requires conducting user study against existing debugging methods, such as print-debugging and pdb-based debugging with no access to kernel code. Another direction is focused on developing an eDSL-agnostic

layer of PKDB, which would provide the flexible support without eDSL dependencies.

VI. Related Work

We survey prior work relevant to PKDB: (1) interactive debugging for HPC and GPU programs; (2) debugging for Python eDSL and multi-language stacks; (3) dynamic software updating and kernel call site substitution; and (4) performance-portable programming frameworks.

HPC and GPU debugging. Interactive debugging at scale has a long history in the HPC community [24]. Prior works [6, 25, 26] study the challenges of debugging massively parallel applications without disrupting program execution. [27] surveys debugging tools for GPU applications, including the native debuggers CUDA-GDB [8] and ROCgdb [9] that PKDB uses internally; [28] provides Kokkos hooks for the aforementioned device debuggers. All of these tools target native C++ or CUDA code; none provide an interactive debugging interface for Python eDSL kernels. PKDB fills this gap by bridging pdb Python debugging with preexisting device debuggers with uniform interface to the developer, and bringing novel debugging features (e.g., live code evaluation, kernel call site substitution).

Python eDSL and multi-language debugging. A common debugging approach for Python GPU eDSLs is to fall back to CPU execution. Triton [5, 10] and Numba [4, 21] provide CPU simulators to step through kernels on the CPU; masking device-specific bugs [29, 30] and requiring alteration of the program or its data to keep execution time manageable. Alternatively, PyCuda [3] allows users to step through generated CUDA kernels using default CUDA-GDB, but all Python context, LOC mapping, and variable translation must be handled manually by the user,

Multi-language debugging is a closely related challenge [31, 32, 33]. [34] compose a debugger across Java and C; similar tools exist for Cython [35], a compiled language providing C-extensions for Python. [36] develops a framework for debugging more general extensible C languages, like mbeddr C [37]. PKDB follows a similar composition strategy: pdb* interposes on PyKokkos dispatch, handing control to a platform-specific controller that manages the target debugger. Unlike [34, 35, 36], PKDB targets performance-portable kernels, supports concurrent execution on accelerators and introduces capabilities such as live code evaluation and kernel call site substitution.

Dynamic software updating and kernel call site substitution. Dynamic software updating (DSU) [17, 38] [39] replaces running code without stopping a live process. These concepts are crucial to modern debugging; e.g., the Wolverine debugger [40] uses these tools to diagnose and repair C programs without restarting. Recent work extends DSU to active long-running functions [41] and whole-kernel subsystems [42]. PKDB's kernel call site substitution is motivated by the same desire to avoid costly restarts, but targets PyKokkos kernels specifically: replacements are recorded as call-site substitutions that take effect at the next `parallel_<op>` dispatch, and the substitution semantics are tied to original call sites rather than to inter-component dependencies [43, 41, 44] (Section III-C).

Performance-portable and Python HPC frameworks. PKDB targets PyKokkos [2], which translates a Python eDSL to Kokkos C++ [1, 12] and dispatches kernels to device backends. Related performance-portable frameworks include RAJA [45] and Leonid [46]. Numba [22] and JAX [47] represent alternative approaches to Python HPC, JIT-compiling kernels to LLVM or XLA; both lack interactive on-device debugging. Legion [48] and its Python eDSL Pygion [49] provide a performance-portable task-based parallelization framework for *distributed* computations. The Scout DSL [50] is built on top of Legion and supports a custom debugger based off LLDB, but as far as we are aware this debugger has *not* been extended to support Legion or Pygion itself. PKDB's architecture is designed to be extensible: adding support for a new backend—whether another PyKokkos execution space or a different Python eDSL—requires implementing the controller interface described in § III-A.

VII. Conclusions

We presented PKDB, the first interactive debugger for performance-portable Python HPC kernels. PKDB brings familiar `pdb`-style debugging—breakpoints, stepping, and variable inspection—to kernels executing natively on CPUs and GPUs, without source modification. Beyond standard debugging, PKDB introduces live code evaluation, which lets developers execute arbitrary code directly on a live device during a paused session, and kernel call site substitution, which allows kernels to be updated and reloaded without restarting the application. We hope PKDB lowers the barrier to productive development of performance-portable HPC software in Python, and serves as a foundation for richer tooling in this space, which traditionally has not received sufficient attention by researchers.

Acknowledgment

We thank Jakob Bludau, Damien Lebrun-Grandie, and the anonymous reviewers for their advice and helpful feedback. This work was partially funded by the U.S. Department of Energy, National Nuclear Security Administration Award Number DE-NA0003969; the U.S. National Science Foundation (NSF) Nos. CCF-2217696, CCF-2313027, CCF-2403036; and AMD (University Program AI & HPC Cluster). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the funding entities.

References

[1] H. Carter Edwards, C. R. Trott, and D. Sunderland, "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," *Journal of Parallel and Distributed Computing*, vol. 74, no. 12, pp. 3202–3216, 2014.

[2] N. A. Awar, N. Mehta, S. Zhu, G. Biros, and M. Gligoric, "PyKokkos: performance portable kernels in Python," in *International Conference on Software Engineering: Companion Proceedings*, 2022, pp. 164–167.

[3] A. Klöckner, "Frequently Asked Questions about PyCUDA," 2025. [Online]. Available: https://wiki.tiker.net/PyCuda/FrequentlyAskedQuestions/#frequently-asked-questions-about-pycuda

[4] I. Anaconda *et al.*, "Troubleshooting and tips – Numba documentation," 2025-12-24, 2012-2020. [Online]. Available: https://numba.pydata.org/numba-doc/dev/user/troubleshoot.html#debugging-cuda-python-code

[5] P. Tillet, H. T. Kung, and D. Cox, "Triton: an intermediate language and compiler for tiled neural network computations," in *International Workshop on Machine Learning and Programming Languages*, 2019, pp. 10–19.

[6] I. Laguna, D. H. Ahn, B. R. de Supinski, T. Gamblin, G. L. Lee, M. Schulz, S. Bagchi, M. Kulkarni, B. Zhou, Z. Chen, and F. Qin, "Debugging high-performance computing applications at massive scales," *Commun. ACM*, vol. 58, no. 9, pp. 72–81, 2015.

[7] K. Ida, Y. Ohno, S. Inoue, and K. Minami, "Performance profiling and debugging on the K computer," *Fujitsu Scientific & Technical Journal*, vol. 48, no. 3, pp. 331–339, 2012.

[8] NVIDIA Corporation, "CUDA-GDB: The NVIDIA CUDA debugger," 2026, version 13.2. [Online]. Available: https://docs.nvidia.com/cuda/debugger-api/index.html

[9] Advanced Micro Devices, Inc., "ROCgdb documentation," 2024. [Online]. Available: https://rocm.docs.amd.com/projects/ROCgdb/en/latest/

[10] P. Tillet, "Debugging Triton – Triton documentation," 2020. [Online]. Available: https://triton-lang.org/main/programming-guide/chapter-3/debugging.html#using-the-interpreter

[11] Python Software Foundation, "pdb - the Python debugger," https://docs.python.org/3/library/pdb.html, 2025, accessed: 2026-04-07.

[12] C. Trott, L. Berger-Vergiat, D. Poliakoff, S. Rajamanickam, D. Lebrun-Grandie, J. Madsen, N. Al Awar, M. Gligoric, G. Shipman, and G. Womeldorff, "The Kokkos ecosystem: Comprehensive performance portability for high performance computing," *Computing in Science Engineering*, vol. 23, no. 5, pp. 10–18, 2021.

[13] A. Meurer, A. Reines, R. Gommers, Y.-L. L. Fang, J. Kirkham, M. Barber, S. Hoyer, A. Müller, S. Zha, S. Shanabrook, S. J. Gacha, M. Lezcano-Casado, T. J. Fan, T. Reddy, A. Passos, H. Kwon, T. Oliphant, and C. f. P. D. A. Standards, "Python array API standard: Toward array interoperability in the scientific Python ecosystem," *Scipy*, 2023.

[14] Python Software Foundation., "cmd - support for line-oriented command interpreters," https://docs.python.org/3/library/cmd.html, 2025, [Online; accessed 2026-04-07].

[15] J.-B. Boric, "Reverse-engineering part 5: old-school binary patching," 2023. [Online]. Available: https://boricj.net/reverse-engineering/2023/06/05/part-5.html

[16] B. Li, J. Li, T. Wo, C. Hu, and L. Zhong, "A VMM-based system call interposition framework for program monitoring," in *International Conference on Parallel and Distributed Systems*, USA, 2010, pp. 706–711.

[17] M. Hicks and S. Nettles, "Dynamic software updating," *ACM Trans. Program. Lang. Syst.*, vol. 27, no. 6, pp. 1049–1096, 2005.

[18] E. Project, "ExaMiniMD: An exascale proxy application for molecular dynamics," 2021. [Online]. Available: https://github.com/ECP-copa/ExaMiniMD

[19] J. Almgren-Bell, N. A. Awar, D. S. Geethakrishnan, M. Gligoric, and G. Biros, "A multi-GPU Python solver for low-temperature non-equilibrium plasmas," in *International Symposium on Computer Architecture and High Performance Computing*, 2022, pp. 140–149.

[20] G. Kosmacher, Z. Du, J. Bagge, and G. Biros, "A performance portable fast Ewald summation for Stokes flow," 2026. [Online]. Available: https://arxiv.org/abs/2606.19059

[21] I. Anaconda *et al.*, "Debugging CUDA Python with the the CUDA Simulator – Numba documentation," 2012-2020. [Online]. Available: https://numba.pydata.org/numba-doc/dev/cuda/simulator.html#using-the-simulator

[22] S. K. Lam, A. Pitrou, and S. Seibert, "Numba: a LLVM-based Python JIT compiler," in *Workshop on the LLVM Compiler Infrastructure in HPC*. Association for Computing Machinery, 2015.

[23] W. Li, B. Butun, T. Chu, M. Fiore, and P. Patras, "Pallas: A data-plane-only approach to accurate persistent flow detection on programmable switches in high-speed networks," in *International Conference on Network Protocols*. IEEE Computer Society, 2025, pp. 1–11.

[24] J. M. Francioni and C. M. Pancake, "A debugging standard for high-performance computing," *Scientific Programming*, vol. 8, no. 2, p. 971291, 2000.

[25] G. L. Lee, D. H. Ahn, D. C. Arnold, B. R. de Supinski, M. Legendre, B. P. Miller, M. Schulz, and B. Liblit, "Lessons learned at 208k: Towards debugging millions of cores," in *SC '08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, 2008, pp. 1–9.

[26] R. Wismüller, M. Oberhuber, J. Krammer, and O. Hansen, "Interactive debugging and performance analysis of massively parallel applications," *Parallel Computing*, vol. 22, no. 3, pp. 415–442, 1996.

[27] M. Knobloch and B. Mohr, "Tools for GPU computing - debugging and performance analysis of heterogenous HPC applications," *Supercomputing Frontiers and Innovations*, vol. 7, no. 1, pp. 91–111, 2020.

[28] S. D. Hammond, C. R. Trott, D. Ibanez, and D. Sunderland, "Profiling and debugging support for the Kokkos programming model," in *High Performance Computing*, 2018, pp. 743–754.

[29] R. Rathnasuriya, N. Majoju, Z. Song, and W. Yang,

"An investigation on numerical bugs in GPU programs towards automated bug detection," *Proc. ACM Softw. Eng.*, vol. 2, no. ISSTA, 2025.

[30] Q. Zhan, X. Hu, Y. Lin, T. Xu, X. Xia, and S. Li, "When allclose fails: Round-off error estimation for deep learning programs," in *International Conference on Automated Software Engineering*, 2025, pp. 91–103.

[31] M. Grichi, M. Abidi, F. Jaafar, E. E. Eghan, and B. Adams, "On the impact of interlanguage dependencies in multilanguage systems empirical case study on java native interface applications (JNI)," *Transactions on Reliability*, vol. 70, no. 1, pp. 428–440, 2021.

[32] W. Li, L. Li, and H. Cai, "On the vulnerability proneness of multilingual code," in *Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 847–859.

[33] H. Yang, Y. Nong, S. Wang, and H. Cai, "Multi-language software development: Issues, challenges, and solutions," *Transactions on Software Engineering*, vol. 50, no. 3, pp. 512–533, 2024.

[34] B. Lee, M. Hirzel, R. Grimm, and K. S. McKinley, "Debug all your code: portable mixed-environment debugging," in *Conference on Object Oriented Programming Systems Languages and Applications*, 2009, pp. 207–226.

[35] S. Behnel, R. Bradshaw, D. S. Seljebotn, G. Ewing, W. Stein, G. Gellner *et al.*, "Debugging your Cython program," https://cython.readthedocs.io/en/latest/src/userguide/debugging.html, 2026, [Online; accessed 2026-04-08].

[36] D. Pavletic and K. Haßlbauer, "Interactive debugging for extensible languages in multi-stage transformation environments." in *EXE@ MoDELS*, 2016, pp. 19–25.

[37] M. Völter, D. Ratiu, B. Schätz, and B. Kolb, "Mbeddr: An extensible C-based programming language and IDE for embedded systems," 10 2012, pp. 121–140.

[38] C. Hofmeister and J. Purtilo, "Dynamic reconfiguration in distributed systems: adapting software modules for replacement," in *International Conference on Distributed Computing Systems*, 1993, pp. 101–110.

[39] J. Appavoo, K. Hui, C. A. N. Soules, R. W. Wisniewski, D. M. Da Silva, O. Krieger, M. A. Auslander, D. J. Edelsohn, B. Gamsa, G. R. Ganger, P. McKenney, M. Ostrowski, B. Rosenburg, M. Stumm, and J. Xenidis, "Enabling autonomic behavior in systems software with hot swapping," *IBM Systems Journal*, vol. 42, no. 1, pp. 60–76, 2003.

[40] S. Verma and S. Roy, "Synergistic debug-repair of heap manipulations," in *Joint Meeting on Foundations of Software Engineering*, 2017, pp. 163–173.

[41] F. Strömbäck, D. Varró, J. Edwards, and M. Taeumel, "Active DSU: Dynamic software updates for active functions," in *International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*, 2024, pp. 26–37.

[42] T. Ma, S. Chen, Y. Wu, E. Deng, Z. Song, Q. Chen, and M. Guo, "Efficient scheduler live update for Linux kernel with modularization," in *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2023, pp. 194–207.

[43] C. M. Hayden, E. K. Smith, M. Hicks, and J. S. Foster, "Kitsune: Efficient, general-purpose dynamic software updating for C," *Transactions on Programming Languages and Systems*, vol. 36, no. 4, pp. 13:1–13:38, 2014.

[44] A. Barbalace, B. Ravindran, D. Katz, and J. Cesnjevar, "Mvedsua: Higher availability dynamic software updates via multi-version execution," in *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019, pp. 573–585.

[45] D. A. Beckingsale, J. Burmark, R. Hornung, H. Jones, W. Killian, A. J. Kunen, O. Pearce, P. Robinson, B. S. Ryujin, and T. R. Scogland, "RAJA: Portable performance for large-scale scientific applications," in *International Workshop on Performance, Portability and Productivity in HPC*, 2019, pp. 71–81.

[46] C. Zhang, H. Luo, and C. Yang, "Leonid: Exploring automated kernel fusion in performance-portable programming models for scientific computation," in *International Conference on Supercomputing*, 2025, pp. 928–942.

[47] J. Bradbury, R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, G. Necula, A. Paszke, J. VanderPlas, S. Wanderman-Milne, and Q. Zhang, "JAX: composable transformations of Python+NumPy programs," 2018. [Online]. Available: http://github.com/jax-ml/jax

[48] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, "Legion: expressing locality and independence with logical regions," in *International Conference on High Performance Computing, Networking, Storage and Analysis*, 2012, pp. 1–11. [Online]. Available: https://dl.acm.org/doi/10.5555/2388996.2389086

[49] E. Slaughter and A. Aiken, "Pygion: Flexible, scalable task-based parallelism with Python," in *Parallel Applications Workshop, Alternatives To MPI*, 2019, pp. 58–72.

[50] P. McCormick, C. Sweeney, N. Moss, D. Prichard, S. K. Gutierrez, K. Davis, and J. Mohd-Yusof, "Exploring the construction of a domain-aware toolchain for high-performance computing," in *International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing*, 2014, pp. 1–10.