SC26
CHICAGO NOV 15–20

Interactive Debugger for Performance Portable Python HPC Kernels


Ivan Grigorik


Gabriel Kosmacher


George Biros


Milos Gligoric

TEXAS
The University of Texas at Austin

AMD

NNSA
National Nuclear Security Administration
DE-NA0003969

NSF
CCF-2217696
CCF-2313027
CCF-2403036

Motivation

- Python is increasingly used for scientific computing
  - Easy to write, read and maintain
  - Rich ecosystem of libraries: NumPy, CuPy, machine-learning workflows

Motivation

- Python is increasingly used for scientific computing
  - Easy to write, read and maintain
  - Rich ecosystem of libraries: NumPy, CuPy, machine-learning workflows
- Many libraries support performance portability: PyKokkos, Triton, Numba, ...
  - Python kernels are compiled just-in-time to C++, MLIR, ...
  - One source runs on OpenMP-threaded CPUs and various GPUs

Motivation

- Python is increasingly used for scientific computing
  - Easy to write, read and maintain
  - Rich ecosystem of libraries: NumPy, CuPy, machine-learning workflows
- Many libraries support performance portability: PyKokkos, Triton, Numba, . . .
  - Python kernels are compiled just-in-time to C++, MLIR, . . .
  - One source runs on OpenMP-threaded CPUs and various GPUs
- One big issue: the lack of support for developers who use these frameworks
  - Native debuggers like pdb cannot enter eDSL kernels
  - Developers fall back to impractical methods for eDSL debugging

Motivation

- Python is increasingly used for scientific computing
  - Easy to write, read and maintain
  - Rich ecosystem of libraries: NumPy, CuPy, machine-learning workflows
- Many libraries support performance portability: PyKokkos, Triton, Numba, ...
  - Python kernels are compiled just-in-time to C++, MLIR, ...
  - One source runs on OpenMP-threaded CPUs and various GPUs
- One big issue: the lack of support for developers who use these frameworks
  - Native debuggers like pdb cannot enter eDSL kernels
  - Developers fall back to impractical methods for eDSL debugging

Key question

How can we develop and efficient debugger for Performance Portable Python HPC kernels?

State of practice

1. pdb • 2. `print` statements • 3. Lowering to sequential execution

Python: `yax.py`

```
    1 import cupy as cp
    2 import pykokkos as pk
    3
    4 @pk.workunit
    5 def yAx(j, acc, cols, y_view, x_view, A_view):
    6     temp2: float = 0
    7     for i in range(cols):
    8       temp2 += A_view[j][i] * x_view[i]
B   9     acc += y_view[j] * temp2
   10
   11 if __name__ == "__main__":
   12     N = 256   # Rows
   13     M = 1024  # Cols
   14     y = cp.random.rand(N)
   15     x = cp.random.rand(M)
   16     A = cp.random.rand(N, M)
B  17     space = pk.RangePolicy(
   18         pk.ExecutionSpace.Cuda, 0, N)
B  19     pk.parallel_reduce(space, yAx, cols=M,
   20         y_view=y, x_view=x, A_view=A)
```

`@pk.workunit` marks a kernel that runs in parallel; the rest is plain Python

State of practice

1. pdb • 2. `print` statements • 3. Lowering to sequential execution

Python: `yax.py`

```
   1 import cupy as cp
   2 import pykokkos as pk
   3
   4 @pk.workunit
   5 def yAx(j, acc, cols, y_view, x_view, A_view):
   6     temp2: float = 0
   7     for i in range(cols):
   8       temp2 += A_view[j][i] * x_view[i]
 B 9     acc += y_view[j] * temp2
  10
  11 if __name__ == "__main__":
  12     N = 256    # Rows
  13     M = 1024   # Cols
  14     y = cp.random.rand(N)
  15     x = cp.random.rand(M)
  16     A = cp.random.rand(N, M)
-> 17     space = pk.RangePolicy(
  18         pk.ExecutionSpace.Cuda, 0, N)
 B 19     pk.parallel_reduce(space, yAx, cols=M,
  20         y_view=y, x_view=x, A_view=A)
```

Debugging session: `yax.py`

```
(Pdb) run
-> A = cp.random.rand(N , M)
(Pdb) continue
-> space = pk.RangePolicy(
```

[ok] pdb stops on the `space` line, like any Python statement

State of practice

1. pdb • 2. `print` statements • 3. Lowering to sequential execution

Python: `yax.py`

```
   1 import cupy as cp
   2 import pykokkos as pk
   3
   4 @pk.workunit
   5 def yAx(j, acc, cols, y_view, x_view, A_view):
   6     temp2: float = 0
   7     for i in range(cols):
   8       temp2 += A_view[j][i] * x_view[i]
B  9     acc += y_view[j] * temp2
  10
  11 if __name__ == "__main__":
  12     N = 256    # Rows
  13     M = 1024   # Cols
  14     y = cp.random.rand(N)
  15     x = cp.random.rand(M)
  16     A = cp.random.rand(N, M)
B 17     space = pk.RangePolicy(
  18         pk.ExecutionSpace.Cuda, 0, N)
-> 19     pk.parallel_reduce(space, yAx, cols=M,
  20         y_view=y, x_view=x, A_view=A)
```

Debugging session: `yax.py`

```
(Pdb) run
-> A = cp.random.rand(N , M)
(Pdb) continue
-> space = pk.RangePolicy(
(Pdb) continue
-> pk.parallel_reduce(space, yAx, cols=M,
```

`[ok]` at the call site every variable is still inspectable

State of practice

1. pdb • 2. `print` statements • 3. Lowering to sequential execution

Python: `yax.py`

```
  -> 1 import cupy as cp
     2 import pykokkos as pk
     3
     4 @pk.workunit
     5 def yAx(j, acc, cols, y_view, x_view, A_view):
     6     temp2: float = 0
     7     for i in range(cols):
     8       temp2 += A_view[j][i] * x_view[i]
  B  9     acc += y_view[j] * temp2
    10
    11 if __name__ == "__main__":
    12     N = 256    # Rows
    13     M = 1024   # Cols
    14     y = cp.random.rand(N)
    15     x = cp.random.rand(M)
    16     A = cp.random.rand(N, M)
  B 17     space = pk.RangePolicy(
    18         pk.ExecutionSpace.Cuda, 0, N)
  B 19     pk.parallel_reduce(space, yAx, cols=M,
    20         y_view=y, x_view=x, A_view=A)
```

Debugging session: `yax.py`

```
(Pdb) run
-> A = cp.random.rand(N , M)
(Pdb) continue
-> space = pk.RangePolicy(
(Pdb) continue
-> pk.parallel_reduce(space, yAx, cols=M,
(Pdb) continue
The program finished and will be restarted
-> import cupy as cp
```

[error!] `step` cannot enter `yAx`: it jumps over the whole call

State of practice

1. pdb • **2. `print` statements** • 3. Lowering to sequential execution

Kernel with a print

```
4  @pk.workunit
5  def yAx(j, acc, cols, y_view, x_view, A_view):
6      temp2: float = 0
7      for i in range(cols):
8          temp2 += A_view[j][i] * x_view[i]
9          pk.printf("j=%d i=%d\n", j, i)
10     acc += y_view[j] * temp2
```

Try to output every inspected data

State of practice

1. pdb • **2. `print` statements** • 3. Lowering to sequential execution

Kernel with a print

```
4  @pk.workunit
5  def yAx(j, acc, cols, y_view, x_view, A_view):
6      temp2: float = 0
7      for i in range(cols):
8          temp2 += A_view[j][i] * x_view[i]
9          pk.printf("j=%d i=%d\n", j, i)
10     acc += y_view[j] * temp2
```

Output

```
$ python yax.py
j=0 i=0
j=1 i=0
j=2 i=0
j=3 i=0
...
j=254 i=1023
j=255 i=1023
```

Try to output every inspected data

! One `printf`: $256 \times 1024 = 262{,}144$ lines per launch.

State of practice

1. pdb • 2. `print` statements • **3. Lowering to sequential execution**

Fall back to the host

```
      space = pk.RangePolicy(
-         pk.ExecutionSpace.Cuda, 0, N)
+         pk.ExecutionSpace.Serial, 0, N)
```

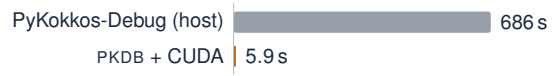
State of practice

1. pdb • 2. `print` statements • **3. Lowering to sequential execution**

Fall back to the host

```
      space = pk.RangePolicy(
-         pk.ExecutionSpace.Cuda, 0, N)
+         pk.ExecutionSpace.Serial, 0, N)
```

ExaMiniMD debug wall time, 100–500 atoms (max)



The host fallback edits the program and is up to 120× slower.

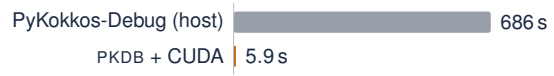
State of practice

1. pdb • 2. `print` statements • **3. Lowering to sequential execution**

Fall back to the host

```
      space = pk.RangePolicy(
-         pk.ExecutionSpace.Cuda, 0, N)
+         pk.ExecutionSpace.Serial, 0, N)
```

ExaMiniMD debug wall time, 100–500 atoms (max)



The host fallback edits the program and is up to 120× slower.

Ewald on OpenMP (CPU)

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

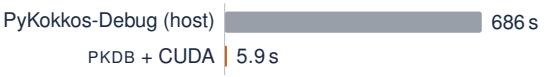
State of practice

1. pdb • 2. `print` statements • **3. Lowering to sequential execution**

Fall back to the host

```
      space = pk.RangePolicy(
-         pk.ExecutionSpace.Cuda, 0, N)
+         pk.ExecutionSpace.Serial, 0, N)
```

ExaMiniMD debug wall time, 100–500 atoms (max)



The host fallback edits the program and is up to 120× slower.

Ewald on OpenMP (CPU)

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

Ewald on CUDA (GPU)

```
cudaStreamSynchronize(stream) error( cudaErrorIllegalAddress): an illegal memory access ../Cuda/
     Kokkos_Cuda_Instance.cpp:165
Backtrace:
                          ...
[0x72..39] reshuffle_particles_fp64(*kwargs)
                          ...
Aborted                     (core dumped) python celllist.py --device Cuda
```

Same bug: the CPU run is a bare segfault, only the GPU run names the kernel.

State of practice

[issue!]

1. pdb

skips the kernel

breakpoints inside `yAx` never fire

[issue!]

2. `print` statements

262,144 lines

from one `pk.printf` per launch

[issue!]

3. Sequential execution

up to 120× slower

and it can hide device-only bugs

None of them debugs the kernel while it runs on the device.

Our contributions

1. **Design** of PKDB, a debugger for performance-portable Python kernels
2. **Implementation** of PKDB for PyKokkos

Our contributions

1. **Design** of PKDB, a debugger for performance-portable Python kernels
2. **Implementation** of PKDB for PyKokkos
3. **Standard debugging features**: breakpoints, stepping, `continue`, variable inspection

Our contributions

1. **Design** of PKDB, a debugger for performance-portable Python kernels
2. **Implementation** of PKDB for PyKokkos
3. **Standard debugging features**: breakpoints, stepping, `continue`, variable inspection
4. **Live code evaluation**: run Python expressions and whole kernels at a stop

Our contributions

1. **Design** of PKDB, a debugger for performance-portable Python kernels
2. **Implementation** of PKDB for PyKokkos
3. **Standard debugging features**: breakpoints, stepping, `continue`, variable inspection
4. **Live code evaluation**: run Python expressions and whole kernels at a stop
5. **Kernel call site substitution**: swap a kernel without restarting the application

Our contributions

1. **Design** of PKDB, a debugger for performance-portable Python kernels
2. **Implementation** of PKDB for PyKokkos
3. **Standard debugging features**: breakpoints, stepping, `continue`, variable inspection
4. **Live code evaluation**: run Python expressions and whole kernels at a stop
5. **Kernel call site substitution**: swap a kernel without restarting the application
6. **Approach evaluation** (3 benchmarks, 3 machines) and case studies (3 bugs found)

Our contributions

1. **Design** of PKDB, a debugger for performance-portable Python kernels
2. **Implementation** of PKDB for PyKokkos
3. **Standard debugging features**: breakpoints, stepping, `continue`, variable inspection
4. **Live code evaluation**: run Python expressions and whole kernels at a stop
5. **Kernel call site substitution**: swap a kernel without restarting the application
6. **Approach evaluation** (3 benchmarks, 3 machines) and case studies (3 bugs found)

Key Insights

PKDB brings interactive debugging to Python HPC kernels.

Background

Python: `yax.py`

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

@pk.workunit
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

if __name__ == "__main__":
    N = 256   # Rows
    M = 1024  # Cols
    y = cp.random.rand(N)
    x = cp.random.rand(M)
    A = cp.random.rand(N, M)
    space = pk.RangePolicy(
        pk.ExecutionSpace.Cuda, 0, N)
    pk.parallel_reduce(space, yAx, cols=M,
        y_view=y, x_view=x, A_view=A)
```

PyKokkos is a Python-embedded eDSL for writing parallel kernels.

Background

Python: `yax.py`

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

@pk.workunit
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

if __name__ == "__main__":
    N = 256    # Rows
    M = 1024   # Cols
    y = cp.random.rand(N)
    x = cp.random.rand(M)
    A = cp.random.rand(N, M)
    space = pk.RangePolicy(
        pk.ExecutionSpace.Cuda, 0, N)
    pk.parallel_reduce(space, yAx, cols=M,
        y_view=y, x_view=x, A_view=A)
```

`@pk.workunit` is the decorator that marks a function as a kernel.

Background

Python: `yax.py`

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

@pk.workunit
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

if __name__ == "__main__":
    N = 256    # Rows
    M = 1024   # Cols
    y = cp.random.rand(N)
    x = cp.random.rand(M)
    A = cp.random.rand(N, M)
    space = pk.RangePolicy(
        pk.ExecutionSpace.Cuda, 0, N)
    pk.parallel_reduce(space, yAx, cols=M,
        y_view=y, x_view=x, A_view=A)
```

The first argument is the thread ID; `acc` accumulates the reduction.

Background

Python: `yax.py`

```
1  import cupy as cp
2  import pykokkos as pk
3
4  @pk.workunit
5  def yAx(j, acc, cols, y_view, x_view, A_view):
6      temp2: float = 0
7      for i in range(cols):
8          temp2 += A_view[j][i] * x_view[i]
9      acc += y_view[j] * temp2
10
11 if __name__ == "__main__":
12     N = 256    # Rows
13     M = 1024   # Cols
14     y = cp.random.rand(N)
15     x = cp.random.rand(M)
16     A = cp.random.rand(N, M)
17     space = pk.RangePolicy(
18         pk.ExecutionSpace.Cuda, 0, N)
19     pk.parallel_reduce(space, yAx, cols=M,
20         y_view=y, x_view=x, A_view=A)
```

The remaining arguments pass data into the kernel.

Background

Python: `yax.py`

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

@pk.workunit
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

if __name__ == "__main__":
    N = 256    # Rows
    M = 1024   # Cols
    y = cp.random.rand(N)
    x = cp.random.rand(M)
    A = cp.random.rand(N, M)
    space = pk.RangePolicy(
        pk.ExecutionSpace.Cuda, 0, N)
    pk.parallel_reduce(space, yAx, cols=M,
        y_view=y, x_view=x, A_view=A)
```

`pk.parallel_reduce` launches the kernel; `parallel_for` and `parallel_scan` also exist.

Background

Python: `yax.py`

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

@pk.workunit
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

if __name__ == "__main__":
    N = 256    # Rows
    M = 1024   # Cols
    y = cp.random.rand(N)
    x = cp.random.rand(M)
    A = cp.random.rand(N, M)
    space = pk.RangePolicy(
        pk.ExecutionSpace.Cuda, 0, N)
    pk.parallel_reduce(space, yAx, cols=M,
        y_view=y, x_view=x, A_view=A)
```

translate, compile, run on GPU

Kokkos C++: generated at call time

```
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; };
```

The kernel is translated to C++, compiled, bound with pybind11 and run.

Example: PKDB

Python: `yax.py`

```
   1 import cupy as cp
   2 import pykokkos as pk
   3
   4 @pk.workunit
   5 def yAx(j, acc, cols, y_view, x_view, A_view):
   6     temp2: float = 0
   7     for i in range(cols):
   8       temp2 += A_view[j][i] * x_view[i]
 B 9     acc += y_view[j] * temp2
  10
  11 if __name__ == "__main__":
  12     N = 256    # Rows
  13     M = 1024   # Cols
  14     y = cp.random.rand(N)
  15     x = cp.random.rand(M)
  16     A = cp.random.rand(N, M)
-> 17     space = pk.RangePolicy(
  18         pk.ExecutionSpace.Cuda, 0, N)
 B 19     pk.parallel_reduce(space, yAx, cols=M,
  20         y_view=y, x_view=x, A_view=A)
```

[ok] `run` stops on the `space` line

Debugging session: `yax.py`

```
(pkdb) run
-> space = pk.RangePolicy(
```

Example: PKDB

Python: `yax.py`

```
   1 import cupy as cp
   2 import pykokkos as pk
   3
   4 @pk.workunit
   5 def yAx(j, acc, cols, y_view, x_view, A_view):
   6     temp2: float = 0
   7     for i in range(cols):
   8       temp2 += A_view[j][i] * x_view[i]
 B 9     acc += y_view[j] * temp2
  10
  11 if __name__ == "__main__":
  12     N = 256    # Rows
  13     M = 1024   # Cols
  14     y = cp.random.rand(N)
  15     x = cp.random.rand(M)
  16     A = cp.random.rand(N, M)
 B 17     space = pk.RangePolicy(
  18         pk.ExecutionSpace.Cuda, 0, N)
-> 19     pk.parallel_reduce(space, yAx, cols=M,
  20         y_view=y, x_view=x, A_view=A)
```

[ok] at the call site every variable is still inspectable

Debugging session: `yax.py`

```
(pkdb) run
-> space = pk.RangePolicy(
(pkdb) continue
-> pk.parallel_reduce(space, yAx, cols=M,
```

Example: PKDB

Python: `yax.py`

```
    1 import cupy as cp
    2 import pykokkos as pk
    3
    4 @pk.workunit
    5 def yAx(j, acc, cols, y_view, x_view, A_view):
    6     temp2: float = 0
    7     for i in range(cols):
    8       temp2 += A_view[j][i] * x_view[i]
 ->  9     acc += y_view[j] * temp2
   10
   11 if __name__ == "__main__":
   12     N = 256    # Rows
   13     M = 1024   # Cols
   14     y = cp.random.rand(N)
   15     x = cp.random.rand(M)
   16     A = cp.random.rand(N, M)
B  17     space = pk.RangePolicy(
   18         pk.ExecutionSpace.Cuda, 0, N)
B  19     pk.parallel_reduce(space, yAx, cols=M,
   20         y_view=y, x_view=x, A_view=A)
```

Debugging session: `yax.py`

```
(pkdb) run
-> space = pk.RangePolicy(
(pkdb) continue
-> pk.parallel_reduce(space, yAx, cols=M,
(pkdb) continue
-> acc += y_view[j] * temp2
```

[ok] `continue` enters `yAx`: the kernel body is a real stop

Example: PKDB

Python: `yax.py`

```
    1 import cupy as cp
    2 import pykokkos as pk
    3
    4 @pk.workunit
    5 def yAx(j, acc, cols, y_view, x_view, A_view):
    6     temp2: float = 0
    7     for i in range(cols):
    8       temp2 += A_view[j][i] * x_view[i]
 -> 9     acc += y_view[j] * temp2
   10
   11 if __name__ == "__main__":
   12     N = 256    # Rows
   13     M = 1024   # Cols
   14     y = cp.random.rand(N)
   15     x = cp.random.rand(M)
   16     A = cp.random.rand(N, M)
B  17     space = pk.RangePolicy(
   18         pk.ExecutionSpace.Cuda, 0, N)
B  19     pk.parallel_reduce(space, yAx, cols=M,
   20         y_view=y, x_view=x, A_view=A)
```

Debugging session: `yax.py`

```
(pkdb) run
-> space = pk.RangePolicy(
(pkdb) continue
-> pk.parallel_reduce(space, yAx, cols=M,
(pkdb) continue
-> acc += y_view[j] * temp2
(pkdb-cuda) print j, temp2
j, temp2 = (0, 253.51438426667102)
  (type: tuple)
```

[ok] the transition to the CUDA side changes the prompt to `pkdb-cuda`: it shows the framework being debugged

Example: PKDB

Python: `yax.py`

```
   1 import cupy as cp
   2 import pykokkos as pk
   3
   4 @pk.workunit
   5 def yAx(j, acc, cols, y_view, x_view, A_view):
   6     temp2: float = 0
   7     for i in range(cols):
   8       temp2 += A_view[j][i] * x_view[i]
-> 9     acc += y_view[j] * temp2
  10
  11 if __name__ == "__main__":
  12     N = 256    # Rows
  13     M = 1024   # Cols
  14     y = cp.random.rand(N)
  15     x = cp.random.rand(M)
  16     A = cp.random.rand(N, M)
B 17     space = pk.RangePolicy(
  18         pk.ExecutionSpace.Cuda, 0, N)
B 19     pk.parallel_reduce(space, yAx, cols=M,
  20         y_view=y, x_view=x, A_view=A)
```

[ok] kernel state is inspectable while running on the GPU

Debugging session: `yax.py`

```
(pkdb) run
-> space = pk.RangePolicy(
(pkdb) continue
-> pk.parallel_reduce(space, yAx, cols=M,
(pkdb) continue
-> acc += y_view[j] * temp2
(pkdb-cuda) print j, temp2
j, temp2 = (0, 253.51438426667102)
  (type: tuple)
(pkdb-cuda) print x_view
x_view = [0.7480483992564797, 0.5964216165148513, '...']
  (type: list)
```

Example: PKDB

Python: `yax.py`

```
-> 1  import cupy as cp
   2  import pykokkos as pk
   3
   4  @pk.workunit
   5  def yAx(j, acc, cols, y_view, x_view, A_view):
   6      temp2: float = 0
   7      for i in range(cols):
   8        temp2 += A_view[j][i] * x_view[i]
B  9      acc += y_view[j] * temp2
  10
  11  if __name__ == "__main__":
  12      N = 256    # Rows
  13      M = 1024   # Cols
  14      y = cp.random.rand(N)
  15      x = cp.random.rand(M)
  16      A = cp.random.rand(N, M)
B 17      space = pk.RangePolicy(
  18          pk.ExecutionSpace.Cuda, 0, N)
B 19      pk.parallel_reduce(space, yAx, cols=M,
  20          y_view=y, x_view=x, A_view=A)
```

Debugging session: `yax.py`

```
(pkdb) run
-> space = pk.RangePolicy(
(pkdb) continue
-> pk.parallel_reduce(space, yAx, cols=M,
(pkdb) continue
-> acc += y_view[j] * temp2
(pkdb-cuda) print j, temp2
j, temp2 = (0, 253.51438426667102)
  (type: tuple)
(pkdb-cuda) print x_view
x_view = [0.7480483992564797, 0.5964216165148513, '...']
  (type: list)
(pkdb-cuda) continue
The program finished and will be restarted
-> import cupy as cp
```

[ok] `continue` hands control back to the Python host: one session spans both sides

Architecture



Three cooperating processes, joined by two pseudo-terminals.

Architecture



`python -m pkdb prog` starts pdb* on the Python host.

Architecture




```
pk.parallel_reduce(pk.RangePolicy(pk.Cuda, 0, N), yAx, cols=M, y_view=y, x_view=x, A_view=A)
```

Passed to the controller

- Execution space: `Cuda`

pdb* reads the execution space and recompiles the kernel with `-G -g -O0`.

Architecture




```
pk.parallel_reduce(pk.RangePolicy(pk.Cuda, 0, N), yAx, cols=M, y_view=y, x_view=x, A_view=A)
```

Passed to the controller

- Execution space: `Cuda`
- Line mapping: Python → C++

Python: `yax.py` → **Kokkos C++**: generated functor

```
6 temp2:  float = 0                  →  4 double temp2 = 0;
7 for i in range(cols):              →  5 for (int32_t i = 0; i < cols; i += 1) {
8   temp2 += A_view[j][i] * x_view[i] →  6   temp2 += A_view(j, i) * x_view(i);
                                         7 }
9 acc += y_view[j] * temp2           →  8 acc += y_view(j) * temp2;
```

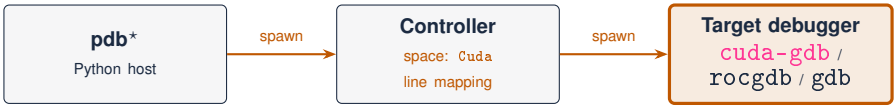
Breakpoints set on Python lines map to the generated C++.

Architecture



A parallel dispatch hands the kernel over to the controller.

Architecture

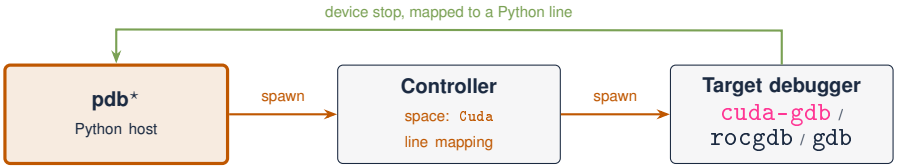


Controller

- Spawn `cuda-gdb` for `Cuda`
- Break at kernel launch
- Set the user's breakpoints

The controller spawns the target debugger and breaks at launch.

Architecture

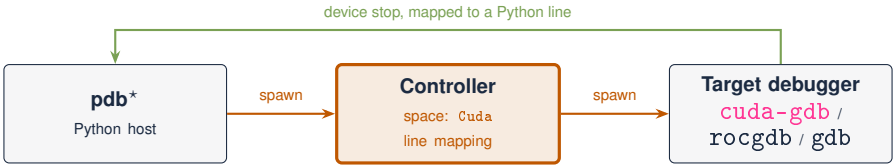


Target debugger

- Attaches to the pdb* process
- Continues to the next breakpoint

Target debugger attaches to the Python process.

Architecture

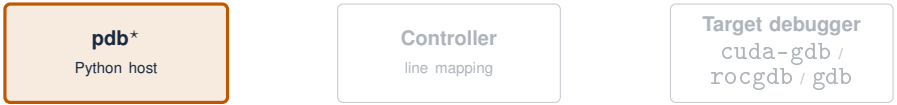


Controller

- Handles kernel stops and commands
- `GDBController`: OpenMP
- `AcceleratorGDBController`: CUDA, HIP

Only the controller is platform-specific; the rest is portable across devices.

Architecture



Execution finishes: both controller and debugger terminates, control returns to pdb*.

Feature: Kernel call site substitution

Python: `yax.py`

```
5 def yAx(j, acc, cols, y_view, x_view, A_view):
6     temp2: float = 0
7     for i in range(cols):
8       temp2 += A_view[j][i] * x_view[i]
9     acc += y_view[j] * temp2
```

Call site

```
19     result = pk.parallel_reduce(
20         space, yAx, cols=M,
21         y_view=y, x_view=x, A_view=A)
22     print(result)
```

Feature: Kernel call site substitution

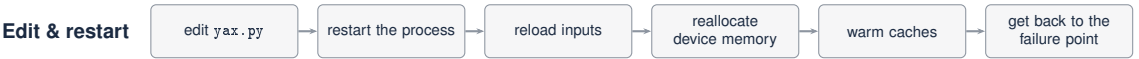
Python: `yax.py`

```
5  def yAx(j, acc, cols, y_view, x_view, A_view):
6      temp2: float = 0
7      for i in range(cols):
8        temp2 += A_view[j][i] * x_view[i]
9      acc += y_view[j] * temp2
```

Call site

```
19     result = pk.parallel_reduce(
20         space, yAx, cols=M,
21         y_view=y, x_view=x, A_view=A)
22     print(result)
```

Why it matters



Feature: Kernel call site substitution

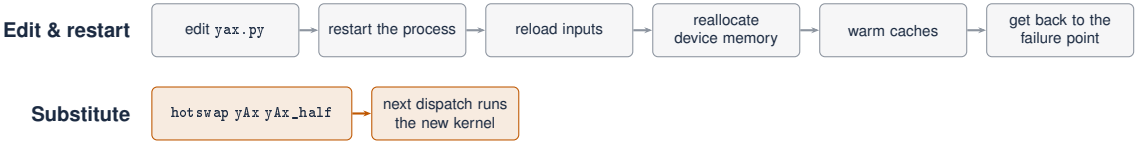
Python: `yax.py`

```
5  def yAx(j, acc, cols, y_view, x_view, A_view):
6      temp2: float = 0
7      for i in range(cols):
8        temp2 += A_view[j][i] * x_view[i]
9      acc += y_view[j] * temp2
```

Call site

```
19     result = pk.parallel_reduce(
20         space, yAx, cols=M,
21         y_view=y, x_view=x, A_view=A)
22     print(result)
```

Why it matters



Feature: Kernel call site substitution

Python: `yax.py`

```
5 def yAx(j, acc, cols, y_view, x_view, A_view):
6     temp2: float = 0
7     for i in range(cols):
8       temp2 += A_view[j][i] * x_view[i]
9     acc += y_view[j] * temp2
```

Call site

```
19     result = pk.parallel_reduce(
20         space, yAx, cols=M,
21         y_view=y, x_view=x, A_view=A)
22     print(result)
```

Debugging session

```
(pkdb) continue
253.51
The program finished and will be restarted
-> import cupy as cp
```

Feature: Kernel call site substitution

Python: `yax.py`

```
5 def yAx(j, acc, cols, y_view, x_view, A_view):
6     temp2: float = 0
7     for i in range(cols):
8       temp2 += A_view[j][i] * x_view[i]
9     acc += y_view[j] * temp2
```

Call site

```
19    result = pk.parallel_reduce(
20        space, yAx, cols=M,
21        y_view=y, x_view=x, A_view=A)
22    print(result)
```

Replacement kernel

```
@pk.workunit
def yAx_half(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 * 0.5
```

- Every later dispatch of `yAx` runs `yAx_half`
- The source file on disk is never edited
- PKDB checks for a substitution at each parallel dispatch

Debugging session

```
(pkdb) continue
253.51
The program finished and will be restarted
-> import cupy as cp
(pkdb) hotswap yAx yAx_half
Registered: yAx -> yAx_half (all call sites)
```

Feature: Kernel call site substitution

Python: `yax.py`

```
5 def yAx(j, acc, cols, y_view, x_view, A_view):
6     temp2: float = 0
7     for i in range(cols):
8       temp2 += A_view[j][i] * x_view[i]
9     acc += y_view[j] * temp2
```

Call site

```
19     result = pk.parallel_reduce(
20         space, yAx_half, cols=M,
21         y_view=y, x_view=x, A_view=A)
22     print(result)
```

Replacement kernel

```
@pk.workunit
def yAx_half(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 * 0.5
```



Debugging session

```
(pkdb) continue
253.51
The program finished and will be restarted
-> import cupy as cp
(pkdb) hotswap yAx yAx_half
Registered: yAx -> yAx_half (all call sites)
(pkdb) continue
126.75
The program finished and will be restarted
-> import cupy as cp
```

Same session: the next dispatch runs the new kernel.

Feature: Live code evaluation

Stopped inside the kernel

```
    4 @pk.workunit
    5 def yAx(j, acc, cols, y_view, x_view, A_view):
    6     temp2: float = 0
    7     for i in range(cols):
    8       temp2 += A_view[j][i] * x_view[i]
-> 9     acc += y_view[j] * temp2
```

- Inspect and modify program state without restarting the application

Feature: Live code evaluation

Stopped inside the kernel

```
   4 @pk.workunit
   5 def yAx(j, acc, cols, y_view, x_view, A_view):
   6     temp2: float = 0
   7     for i in range(cols):
   8       temp2 += A_view[j][i] * x_view[i]
-> 9     acc += y_view[j] * temp2
```

- Inspect and modify program state without restarting the application
- Run any Python expression, or a whole new kernel, at a kernel stop

Feature: Live code evaluation

Stopped inside the kernel

```
   4 @pk.workunit
   5 def yAx(j, acc, cols, y_view, x_view, A_view):
   6     temp2: float = 0
   7     for i in range(cols):
   8       temp2 += A_view[j][i] * x_view[i]
-> 9     acc += y_view[j] * temp2
```

- Inspect and modify program state without restarting the application
- Run any Python expression, or a whole new kernel, at a kernel stop
- `cuda-gdb` and `rocgdb` target native C++ only

Feature: Live code evaluation

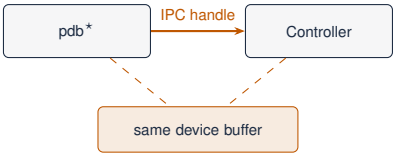
Stopped inside the kernel

```
 4 @pk.workunit
 5 def yAx(j, acc, cols, y_view, x_view, A_view):
 6     temp2: float = 0
 7     for i in range(cols):
 8       temp2 += A_view[j][i] * x_view[i]
-> 9     acc += y_view[j] * temp2
```

Debugging session

```
(pkdb-cuda) eval numpy.linalg.norm(x_view)
18.372943...
  (type: float64)
```

- The Python environment is captured at every parallel kernel call, so it is reachable from any stop



Feature: Live code evaluation

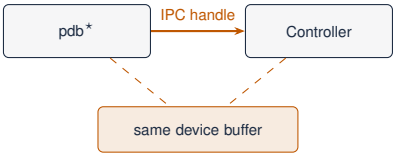
Stopped inside the kernel

```
    4 @pk.workunit
    5 def yAx(j, acc, cols, y_view, x_view, A_view):
    6     temp2: float = 0
    7     for i in range(cols):
    8       temp2 += A_view[j][i] * x_view[i]
-> 9     acc += y_view[j] * temp2
```

Debugging session

```
(pkdb-cuda) eval numpy.linalg.norm(x_view)
18.372943...
  (type: float64)
(pkdb-cuda) eval pk.parallel_reduce(
                 pk.RangePolicy(0, N), yAxFixed, cols=M,
                 y_view=y, x_view=x, A_view=A)
126.75719213333551
  (type: float)
```

- The Python environment is captured at every parallel kernel call, so it is reachable from any stop
- Any stop works: on the host, or inside a kernel on the device



Feature: Live code evaluation

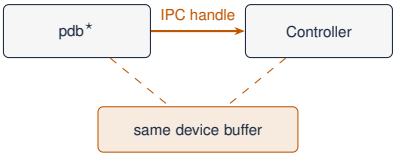
Stopped inside the kernel

```
    4 @pk.workunit
    5 def yAx(j, acc, cols, y_view, x_view, A_view):
    6     temp2: float = 0
    7     for i in range(cols):
    8       temp2 += A_view[j][i] * x_view[i]
-> 9     acc += y_view[j] * temp2
```

Debugging session

```
(pkdb-cuda) eval numpy.linalg.norm(x_view)
18.372943...
  (type: float64)
(pkdb-cuda) eval pk.parallel_reduce(
                 pk.RangePolicy(0, N), yAxFixed, cols=M,
                 y_view=y, x_view=x, A_view=A)
126.75719213333551
  (type: float)
```

- The Python environment is captured at every parallel kernel call, so it is reachable from any stop
- Any stop works: on the host, or inside a kernel on the device
- Arrays are passed by IPC handle, never copied



Feature: concurrent kernel launch

- Run several kernels from one stop, not one at a time



Feature: concurrent kernel launch

- Run several kernels from one stop, not one at a time
- Running them concurrently from one point can give more insights about program flow.



Feature: concurrent kernel launch

- Run several kernels from one stop, not one at a time
- Running them concurrently from one point can give more insights about program flow.
- Differential testing: check that two implementations agree on the live data



Feature: concurrent kernel launch

Debugging session

```
(pkdb-cuda) parallel_launch
    yAx pk.Cuda   pk.RangePolicy(0,N) y x A;
    yAx pk.OpenMP pk.RangePolicy(0,N) y x A
```

one stop

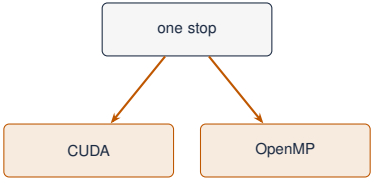
Each clause

`yAx`	`pk.Cuda`	`pk.RangePolicy(0,N)`	`y x A`
kernel	execution space (optional)	number of threads	arguments

Feature: concurrent kernel launch

Debugging session

```
(pkdb-cuda) parallel_launch
    yAx pk.Cuda   pk.RangePolicy(0,N) y x A;
    yAx pk.OpenMP pk.RangePolicy(0,N) y x A
clause 1  Cuda     253.51438426667102
clause 2  OpenMP   253.51438426667096
```



Each clause

`yAx` kernel

`pk.Cuda` execution space (optional)

`pk.RangePolicy(0,N)` number of threads

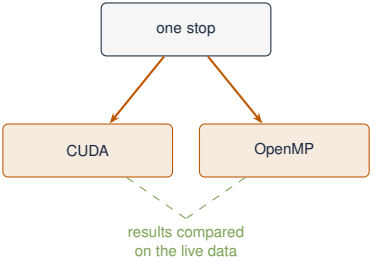
`y x A` arguments

Parallel launch and parallel result getting are inspectable and can be used in expressions.

Feature: concurrent kernel launch

Debugging session

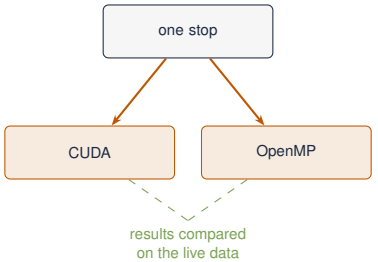
```
(pkdb-cuda) parallel_launch
    yAx pk.Cuda   pk.RangePolicy(0,N) y x A;
    yAx pk.OpenMP pk.RangePolicy(0,N) y x A
clause 1  Cuda    253.51438426667102
clause 2  OpenMP  253.51438426667096
```

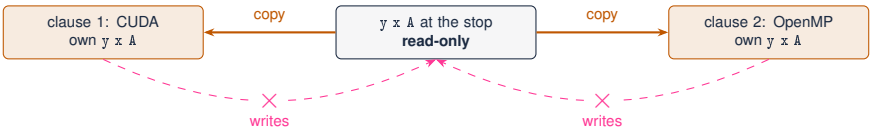


Feature: concurrent kernel launch

Debugging session

```
(pkdb-cuda) parallel_launch
    yAx pk.Cuda   pk.RangePolicy(0,N) y x A;
    yAx pk.OpenMP pk.RangePolicy(0,N) y x A
clause 1  Cuda    253.51438426667102
clause 2  OpenMP  253.51438426667096
```

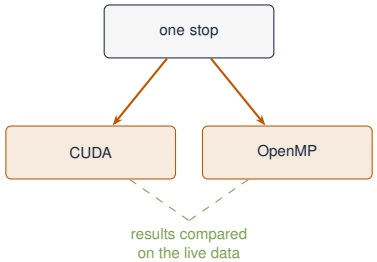


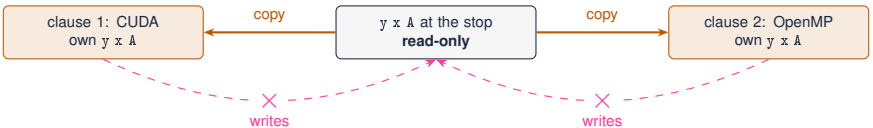


Feature: concurrent kernel launch

Debugging session

```
(pkdb-cuda) parallel_launch
    yAx pk.Cuda   pk.RangePolicy(0,N) y x A;
    yAx pk.OpenMP pk.RangePolicy(0,N) y x A
clause 1  Cuda    253.51438426667102
clause 2  OpenMP  253.51438426667096
```





! `mut:x` shares `x` between clauses: races are possible, and avoiding them is up to the developer.

Evaluation

- **PKDB vs pdb**: debug overhead on CUDA, HIP and OpenMP, 3 benchmarks across 3 machines

Evaluation

- **PKDB vs pdb**: debug overhead on CUDA, HIP and OpenMP, 3 benchmarks across 3 machines
- **PKDB vs PyKokkos Debug**: on-device debugging vs sequential, Python-only execution

Evaluation

- **PKDB vs pdb**: debug overhead on CUDA, HIP and OpenMP, 3 benchmarks across 3 machines
- **PKDB vs PyKokkos Debug**: on-device debugging vs sequential, Python-only execution
- **Feature evaluation**: cost of kernel call site substitution vs editing the source and restarting

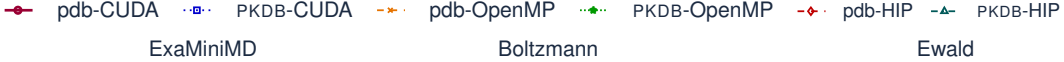
Evaluation

- **PKDB vs pdb**: debug overhead on CUDA, HIP and OpenMP, 3 benchmarks across 3 machines
- **PKDB vs PyKokkos Debug**: on-device debugging vs sequential, Python-only execution
- **Feature evaluation**: cost of kernel call site substitution vs editing the source and restarting
- **Case studies**: debugging the Boltzmann and Ewald applications

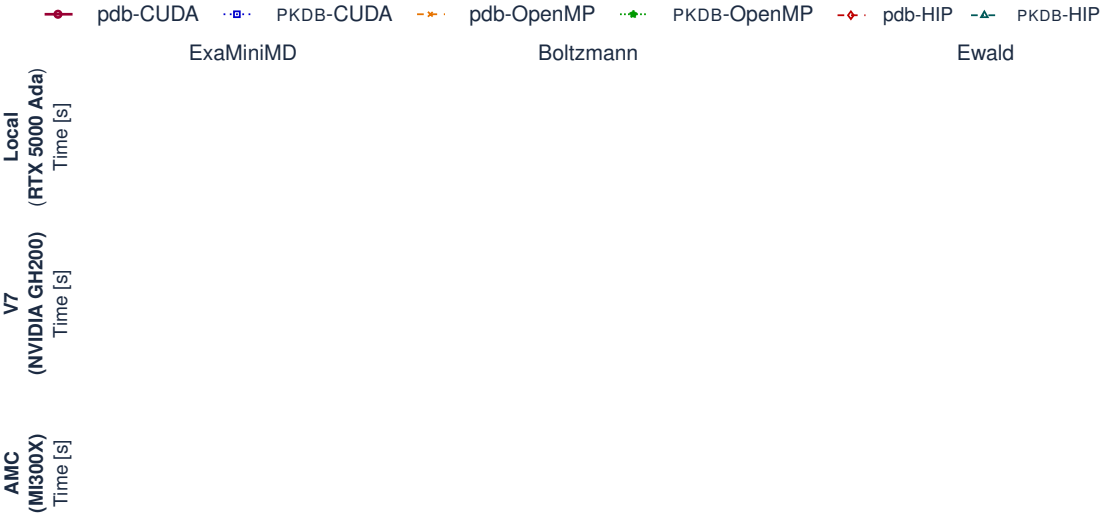
Results: pdb vs PKDB



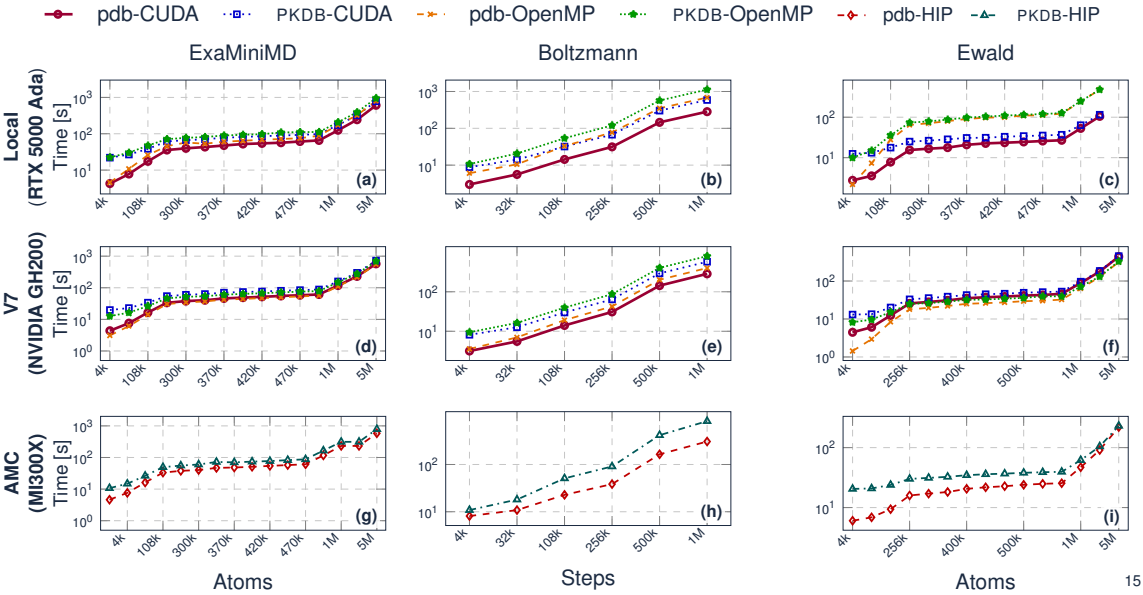
Results: pdb vs PKDB



Results: pdb vs PKDB



Results: pdb vs PKDB



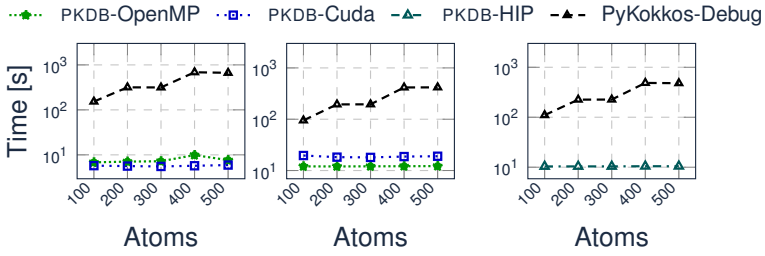
Results: pdb vs PKDB

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

ExaMiniMD, worst mean slowdown per backend: OpenMP 1.75× CUDA 1.95× HIP 1.54×

Interactive on-device debugging costs under 2× across every backend.

Results: PyKokkos Debug[1] vs PKDB



ExaMiniMD debug wall time (Local, RTX 5000).

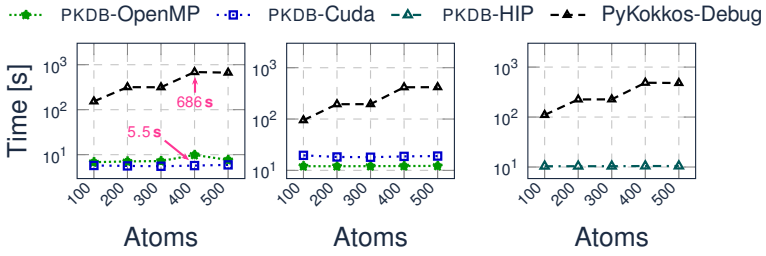
ExaMiniMD debug wall time (V7, GH200).

ExaMiniMD debug wall time (AMC, HIP).

Baseline: PyKokkos Debug lowers every kernel to sequential execution on the host.

[1] Same as Triton and Numba debug mode: lowered to sequential execution

Results: PyKokkos Debug[1] vs PKDB



ExaMiniMD debug wall time (Local, RTX 5000).

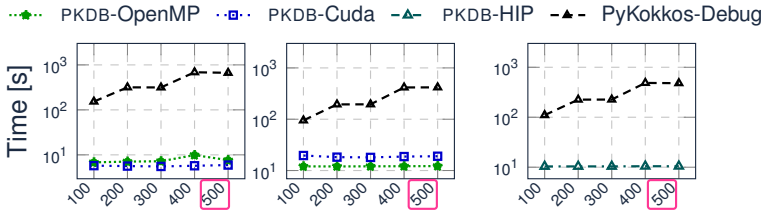
ExaMiniMD debug wall time (V7, GH200).

ExaMiniMD debug wall time (AMC, HIP).

Two orders of magnitude faster than sequential host-only debugging.

[1] Same as Triton and Numba debug mode: lowered to sequential execution

Results: PyKokkos Debug[1] vs PKDB



ExaMiniMD debug wall time (Local, RTX 5000).

ExaMiniMD debug wall time (V7, GH200).

ExaMiniMD debug wall time (AMC, HIP).

Beyond 500 atoms, PyKokkos Debug becomes impractical.

[1] Same as Triton and Numba debug mode: lowered to sequential execution

Results: kernel call site substitution

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

4 alternative ExaMiniMD kernels, each swapped in at the same call site

Results: kernel call site substitution

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

Edit-and-debug: edit the source and relaunch a pdb session per variant

Results: kernel call site substitution

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

Call site substitution: one PKDB session, one `hotswap` per variant

Swapping beats editing on disk and relaunching the session.

Case study: Ewald

- During Ewald timing runs, some runs fail
- The failure is non-deterministic: it appears only once in a while
- CUDA builds report a cryptic error; OpenMP builds fail silently

Case study: Ewald

- During Ewald timing runs, some runs fail
- The failure is non-deterministic: it appears only once in a while
- CUDA builds report a cryptic error; OpenMP builds fail silently

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

OpenMP: a bare segmentation fault, no Python traceback.

Case study: Ewald

```
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
```

CUDA: an opaque abort, with the workunit on the stack.

Case study: Ewald

```
1  @pk.function
2  def _get_cell_fp64(p, ...) -> int:
3      ...
4      cell_xyz = [p[0][i] / c_size[0], ..., ...]
5      return get_idx(cell_xyz)
6
7  @pk.workunit
8  def reshuffle_particles_fp64(i, counter, p, ...):
9      cell: int = _get_cell_fp64(p, ...)
10     offset: int = pk.atomic_add(counter, [cell], 1)
11     ...
```

The suspect: a computed cell index written into an array.

Case study: Ewald

```
1  @pk.function
2  def _get_cell_fp64(p, ...) -> int:
3      ...
4      cell_xyz = [p[0][i] / c_size[0], ..., ...]
5      return get_idx(cell_xyz)
6
7  @pk.workunit
8  def reshuffle_particles_fp64(i, counter, p, ...):
9      cell: int = _get_cell_fp64(p, ...)
10     offset: int = pk.atomic_add(counter, [cell], 1)
11     ...
```

The suspect: a computed cell index written into an array.

Case study: Ewald

Debugging session

```
(pkdb) print _grid_shape
array([9, 8, 7])
```

Source: `celllist_workunits.py`

```
   1 @pk.function
   2 def _get_cell_fp64(p, ...) -> int:
   3     ...
B  4     cell_xyz = [p[0][i] / c_size[0], ..., ...]
   5     return get_idx(cell_xyz)
   6
   7 @pk.workunit
   8 def reshuffle_particles_fp64(i, counter, p, ...):
B  9     cell: int = _get_cell_fp64(p, ...)
B 10     offset: int = pk.atomic_add(counter, [cell], 1)
  11     ...
```

Host side: the cell grid is $9 \times 8 \times 7$.

Case study: Ewald

Debugging session

```
(pkdb) print _grid_shape
array([9, 8, 7])
(pkdb) continue
Workunit breakpoint, celllist_workunits.py:9
(pkdb-cuda) print len(counter)
int(504)
```

Source: `celllist_workunits.py`

```
    1 @pk.function
    2 def _get_cell_fp64(p, ...) -> int:
    3     ...
B   4     cell_xyz = [p[0][i] / c_size[0], ..., ...]
    5     return get_idx(cell_xyz)
    6
    7 @pk.workunit
    8 def reshuffle_particles_fp64(i, counter, p, ...):
->  9     cell: int = _get_cell_fp64(p, ...)
B  10     offset: int = pk.atomic_add(counter, [cell], 1)
   11     ...
```

`counter` has one slot per cell: $9 \times 8 \times 7 = 504$.

Case study: Ewald

Debugging session

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

Source: `celllist_workunits.py`

```
    1 @pk.function
    2 def _get_cell_fp64(p, ...) -> int:
    3     ...
-> 4     cell_xyz = [p[0][i] / c_size[0], ..., ...]
    5     return get_idx(cell_xyz)
    6
    7 @pk.workunit
    8 def reshuffle_particles_fp64(i, counter, p, ...):
B   9     cell: int = _get_cell_fp64(p, ...)
B  10     offset: int = pk.atomic_add(counter, [cell], 1)
   11     ...
```

`parallel_print` shows `cell_xyz` for every device thread.

Case study: Ewald

Debugging session

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

Source: `celllist_workunits.py`

```
   1 @pk.function
   2 def _get_cell_fp64(p, ...) -> int:
   3     ...
B  4     cell_xyz = [p[0][i] / c_size[0], ..., ...]
   5     return get_idx(cell_xyz)
   6
   7 @pk.workunit
   8 def reshuffle_particles_fp64(i, counter, p, ...):
B  9     cell: int = _get_cell_fp64(p, ...)
-> 10     offset: int = pk.atomic_add(counter, [cell], 1)
  11     ...
```

At the atomic add, the computed `cell` index is 528.

Case study: Ewald

Debugging session

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

Source: `celllist_workunits.py`

```
    1 @pk.function
    2 def _get_cell_fp64(p, ...) -> int:
    3     ...
B   4     cell_xyz = [p[0][i] / c_size[0], ..., ...]
    5     return get_idx(cell_xyz)
    6
    7 @pk.workunit
    8 def reshuffle_particles_fp64(i, counter, p, ...):
B   9     cell: int = _get_cell_fp64(p, ...)
-> 10     offset: int = pk.atomic_add(counter, [cell], 1)
   11     ...
```

`counter` holds 504 entries; the kernel indexes 528.

Conclusion

- **Novelty:** PKDB is the first debugger that provides interactive debugging for Python's eDSL code;
- **Technique:** PKDB composes existing debuggers instead of replacing them;
- **Benchmark results**: PKDB introduces minimal overhead compared to non-debug execution and speeds up the existing PyKokkos debugging up to 120×;
- **Case studies**: we found 3 bugs in existing PyKokkos benchmarks during PKDB development;

Code: `github.com/EngineeringSoftware/pkdb` • **Contact:** `grigorik@utexas.edu`