PyKokkos: Performance Portable Kernels in Python

Nader Al Awar / Neil Mehta / Steven Zhu / George Biros / Milos Gligoric

ECP 2022 DEMO

NERSC

TEXAS
The University of Texas at Austin

Performance Portability

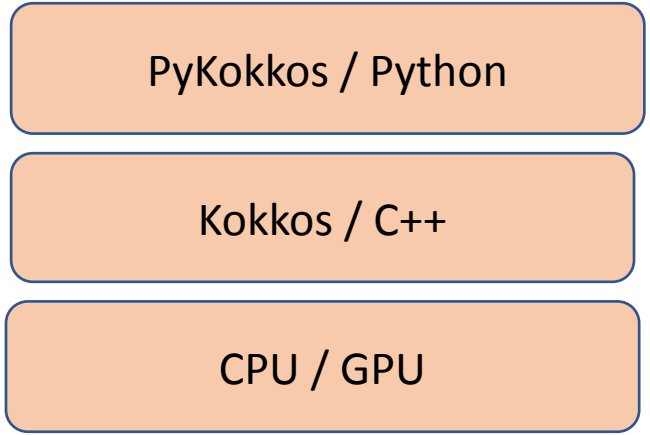
- Different HPC frameworks (OpenMP, CUDA, HIP, etc.)
  - Different APIs
  - Different targeted architectures
- Code is tightly coupled to API + architecture, meaning code is not portable
- Moving to another framework requires:
  - Rewriting significant parts of the application.
  - Learning new architecture-specific details.
  - Maintaining two versions of the same application.
- We want performance portability: run the same code on different devices with good performance.

Motivation

- Kokkos is a powerful framework
- BUT, not everybody wants to write C++:
  - Cryptic error messages
  - Complicated build processes
- Many scientific applications nowadays are written in Python:
  - NumPy, SciPy, TensorFlow, PyTorch, etc.
  - Quick prototyping that with small sacrifices in performance
- How do we bridge the gap?

PyKokkos: Kokkos abstraction in Python

- Enables developers to leverage a performance portability programming ecosystem (Kokkos) from Python
- Provides Kokkos-like abstractions through a domain specific language embedded in Python:
  - Looks like a statically-typed subset of Python
- Implemented as a Python Framework:
  - Partially connected via bindings (PyKokkos-Base)
  - Partially translated to C++ Kokkos (PyKokkos)



PyKokkos: Example

- To import pykokkos: **import pykokkos as pk**
- PyKokkos arrays, called Views, are created as follows: `view = pk.View([n], dtype=int)`
- PyKokkos also makes use of decorators and type annotations
  - **@pk.functor**
  - **@pk.workunit**
  - pk.View1D[int]
- See README.md for a “hello, world!” example
- We will now show an example of a matrix-weighted inner product kernel in PyKokkos

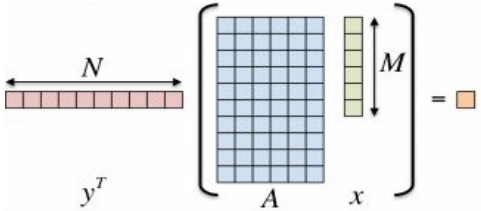
PyKokkos: Example

- To run this example, go to examples/kokkos-tutorials/functor/
- Command: python 02.py
  - Initialize with fill(): python 02.py --fill

- Translates Python code (@pk.workunit) to C++ and then compiles it
  - Compiled kernels are saved on the file system to avoid re-compilation overhead
- Limits the Python features that can be used for kernels only, but the rest of the application can use any Python feature/library

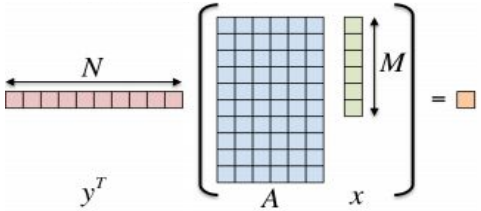
PyKokkos: Example

```python
@pk.functor
class Workload:
    def __init__(self, N: int, M: int):
        self.N: int = N
        self.M: int = M
        self.y: pk.View1D[float] = pk.View([N], float)
        self.x: pk.View1D[float] = pk.View([M], float)
        self.A: pk.View2D[float] = pk.View([N, M], float)

        self.y.fill(1)
        self.x.fill(1)
        self.A.fill(1)

    @pk.workunit
    def yAx(self, j: int, acc: pk.Acc[float]):
        temp2: float = 0
        for i in range(self.M):
            temp2 += self.A[j][i] * self.x[i]

        acc += self.y[j] * temp2


def run() -> None:
    pk.set_default_space(pk.OpenMP)
    N: int = 10
    M: int = 10
    w = Workload(N, M)
    p = pk.RangePolicy(pk.Default, 0, N)
    result = pk.parallel_reduce(p, w.yAx)
```



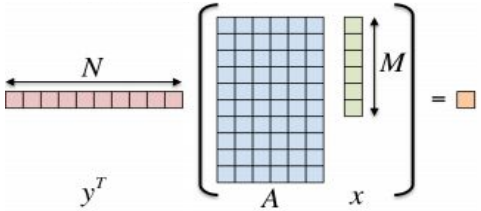
PyKokkos: Example

```python
@pk.functor
class Workload:
    def __init__(self, N: int, M: int):
        self.N: int = N
        self.M: int = M
        self.y: pk.View1D[float] = pk.View([N], float)
        self.x: pk.View1D[float] = pk.View([M], float)
        self.A: pk.View2D[float] = pk.View([N, M], float)

        self.y.fill(1)
        self.x.fill(1)
        self.A.fill(1)

    @pk.workunit
    def yAx(self, j: int, acc: pk.Acc[float]):
        temp2: float = 0
        for i in range(self.M):
            temp2 += self.A[j][i] * self.x[i]

        acc += self.y[j] * temp2


def run() -> None:
    pk.set_default_space(pk.OpenMP)
    N: int = 10
    M: int = 10
    w = Workload(N, M)
    p = pk.RangePolicy(pk.Default, 0, N)
    result = pk.parallel_reduce(p, w.yAx)
```



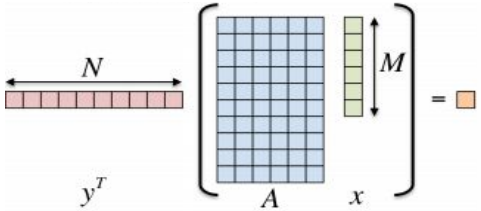
PyKokkos: Example

```python
@pk.functor
class Workload:
    def __init__(self, N: int, M: int):
        self.N: int = N
        self.M: int = M
        self.y: pk.View1D[float] = pk.View([N], float)
        self.x: pk.View1D[float] = pk.View([M], float)
        self.A: pk.View2D[float] = pk.View([N, M], float)

        self.y.fill(1)
        self.x.fill(1)
        self.A.fill(1)

    @pk.workunit
    def yAx(self, j: int, acc: pk.Acc[float]):
        temp2: float = 0
        for i in range(self.M):
            temp2 += self.A[j][i] * self.x[i]

        acc += self.y[j] * temp2


def run() -> None:
    pk.set_default_space(pk.OpenMP)
    N: int = 10
    M: int = 10
    w = Workload(N, M)
    p = pk.RangePolicy(pk.Default, 0, N)
    result = pk.parallel_reduce(p, w.yAx)
```



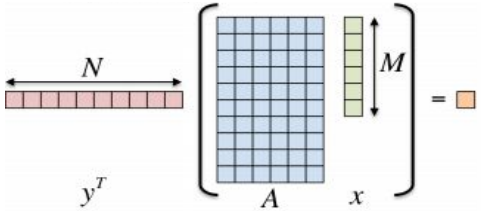
PyKokkos: Example

```python
@pk.functor
class Workload:
    def __init__(self, N: int, M: int):
        self.N: int = N
        self.M: int = M
        self.y: pk.View1D[float] = pk.View([N], float)
        self.x: pk.View1D[float] = pk.View([M], float)
        self.A: pk.View2D[float] = pk.View([N, M], float)

        self.y.fill(1)
        self.x.fill(1)
        self.A.fill(1)

    @pk.workunit
    def yAx(self, j: int, acc: pk.Acc[float]):
        temp2: float = 0
        for i in range(self.M):
            temp2 += self.A[j][i] * self.x[i]

        acc += self.y[j] * temp2


def run() -> None:
    pk.set_default_space(pk.OpenMP)
    N: int = 10
    M: int = 10
    w = Workload(N, M)
    p = pk.RangePolicy(pk.Default, 0, N)
    result = pk.parallel_reduce(p, w.yAx)
```



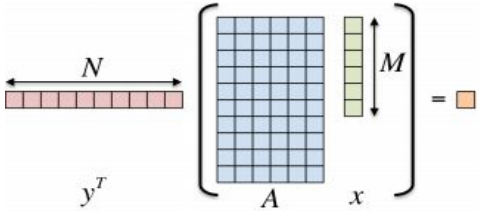
PyKokkos: Example

```python
@pk.functor
class Workload:
    def __init__(self, N: int, M: int):
        self.N: int = N
        self.M: int = M
        self.y: pk.View1D[float] = pk.View([N], float)
        self.x: pk.View1D[float] = pk.View([M], float)
        self.A: pk.View2D[float] = pk.View([N, M], float)

        self.y.fill(1)
        self.x.fill(1)
        self.A.fill(1)

    @pk.workunit
    def yAx(self, j: int, acc: pk.Acc[float]):
        temp2: float = 0
        for i in range(self.M):
            temp2 += self.A[j][i] * self.x[i]

        acc += self.y[j] * temp2


def run() -> None:
    pk.set_default_space(pk.OpenMP)
    N: int = 10
    M: int = 10
    w = Workload(N, M)
    p = pk.RangePolicy(pk.Default, 0, N)
    result = pk.parallel_reduce(p, w.yAx)
```



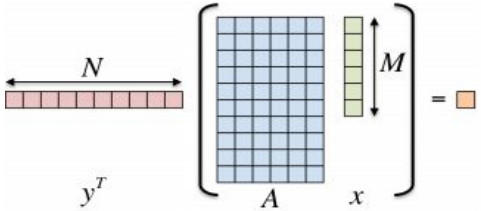
PyKokkos: Example

```python
@pk.functor
class Workload:
    def __init__(self, N: int, M: int):
        self.N: int = N
        self.M: int = M
        self.y: pk.View1D[float] = pk.View([N], float)
        self.x: pk.View1D[float] = pk.View([M], float)
        self.A: pk.View2D[float] = pk.View([N, M], float)

        self.y.fill(1)
        self.x.fill(1)
        self.A.fill(1)

    @pk.workunit
    def yAx(self, j: int, acc: pk.Acc[float]):
        temp2: float = 0
        for i in range(self.M):
            temp2 += self.A[j][i] * self.x[i]

        acc += self.y[j] * temp2

def run() -> None:
    pk.set_default_space(pk.OpenMP)
    N: int = 10
    M: int = 10
    w = Workload(N, M)
    p = pk.RangePolicy(pk.Default, 0, N)
    result = pk.parallel_reduce(p, w.yAx)
```



PyKokkos: Example

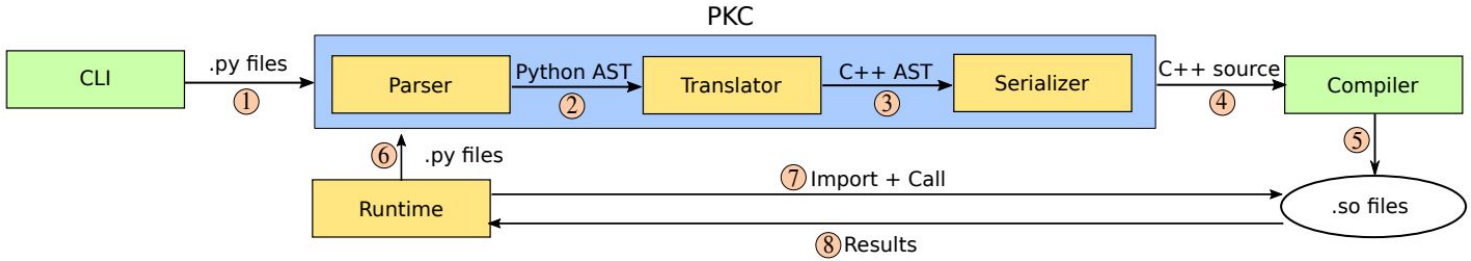
```python
@pk.functor
class Workload:
    def __init__(self, N: int, M: int):
        self.N: int = N
        self.M: int = M
        self.y: pk.View1D[float] = pk.View([N], float)
        self.x: pk.View1D[float] = pk.View([M], float)
        self.A: pk.View2D[float] = pk.View([N, M], float)

        self.y.fill(1)
        self.x.fill(1)
        self.A.fill(1)

    @pk.workunit
    def yAx(self, j: int, acc: pk.Acc[float]):
        temp2: float = 0
        for i in range(self.M):
            temp2 += self.A[j][i] * self.x[i]

        acc += self.y[j] * temp2


def run() -> None:
    pk.set_default_space(pk.OpenMP)
    N: int = 10
    M: int = 10
    w = Workload(N, M)
    p = pk.RangePolicy(pk.Default, 0, N)
    result = pk.parallel_reduce(p, w.yAx)
```



PyKokkos: Internals

- Translates Python code (@pk.workunit) to C++ and then compiles it
  - Compiled kernels are saved on the file system to avoid re-compilation overhead
- Limits the Python features that can be used for kernels only, but the rest of the application can use any Python feature/library

PyKokkos: Internals



1. The Python files containing the kernels are passed to PKC (PyKokkos compiler)
2. PKC extracts the kernels into an Abstract Syntax Tree (AST)
3. PKC then translates the Python AST into a C++ AST and generates Python bindings using pybind11
4. PKC serializes the AST and writes it to a cpp file
5. PKC then calls a compiler (g++, nvcc, …) to generate a shared object (.so) file
6. At run-time, when the user calls a kernel, PyKokkos calls PKC if it does not find the corresponding .so file
7. PyKokkos then imports the .so file and calls the kernel
8. Finally, the results of the kernel execution are returned to Python

PyKokkos: Features

Feature	Details
Views	Multi-Dimensional Views, Subviews, Dual Views
Memory Spaces	HostSpace, CudaSpace, CudaUVMSpace, HIPSpace
Memory Layouts	LayoutRight, LayoutLeft
Memory Traits	Atomic, RandomAccess, Restrict, Unmanaged
Execution Spaces	OpenMP, CUDA, HIP, Threads, Serial
Execution Patterns	parallel_for, parallel_reduce, parallel_scan
Execution Policies	RangePolicy, MDRangePolicy, TeamPolicy, TeamThreadRange, ThreadVectorRange, WorkTag
Hierarchical Parallelism	Team Loops, Vector Loops
Atomic Operations	All atomic_fetch_[op] operations
Other	Kokkos functions, BinSort, Timer, printf, RNG, Scratch Memory

PyKokkos: Evaluation

- Extensively evaluated (but never-ending story)
- Subjects:
  - Kokkos tutorial exercises (each demoing a specific feature)
  - Various kernels written by scientists (the ParRes GitHub repository)
  - ExaMiniMD mini app
  - Ongoing: ArborX
- Platform
  - CPU: 6-core Intel i7-8700 3.20GHz 64GB RAM
  - GPU: GeForce RTX 2080, GPU NVIDIA V100, AMD MI100

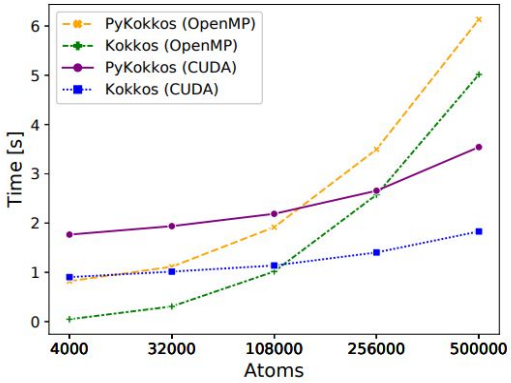
PyKokkos: Profiling tutorial cases

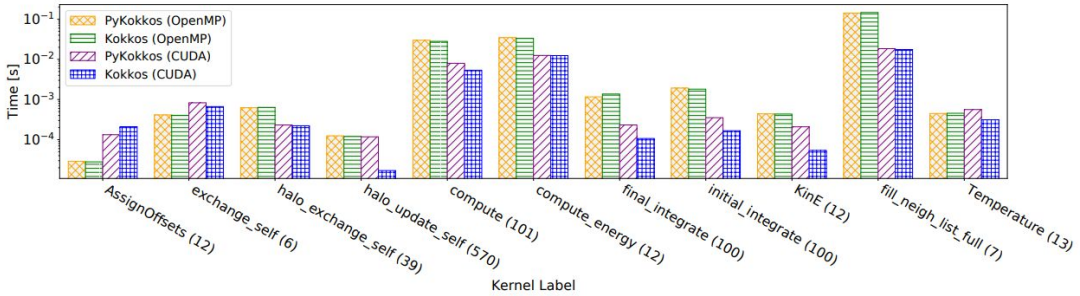
Application	PyKokkos / Kokkos (OpenMP)	PyKokkos / Kokkos (CUDA)
02	1.01x	1.00x
03	1.00x	1.00x
04	1.00x	1.00x
mdrange	1.01x	1.00x
subview	1.01x	1.00x
team_policy	1.00x	1.00x
team_vector_loop	1.00x	1.00x

PyKokkos: Profiling ParRes kernels

Application	PyKokkos / Kokkos (OpenMP)	PyKokkos / Kokkos (CUDA)
nstream	1.00x	1.00x
stencil	1.02x	0.98x
transpose	1.00x	1.00x

PyKokkos: Profiling ExaMiniMD





Metric	OpenMP		Cuda	
	PyKokkos	Kokkos	PyKokkos	Kokkos
Loop time(s)	4.90	4.51	2.15	0.86
Total time(s)	6.12	5.02	3.60	1.83
Atomstep/s(1/s)	1.02e+7	1.11e+7	2.33e+7	5.78e+7

Conclusions

- PyKokkos
  - Bridges the gap between Python and Kokkos/C++
  - Enables use of Views (multi-dimensional arrays) available in Kokkos (via PyKokkos-Base)
  - Enables entire applications to be written in Python (PyKokkos)
- PyKokkos is open-source and is available on GitHub as part of the Kokkos organization
- https://github.com/kokkos/pykokkos
  - Nader Al Awar, https://naderalawar.github.io/, nader.alawar@utexas.edu
  - Neil Mehta, neilmehta@lbl.gov

Thank you!

NERSC

TEXAS
The University of Texas at Austin

Kokkos

- Kokkos is a programming model that enables performance portability by providing layers of abstraction over existing “lower level” frameworks
- It is implemented as a C++ library on top of OpenMP, CUDA, etc.
- Applications written in Kokkos can be run on different devices by configuring them at compile-time.

Application

C++ | Kokkos

NERSC

TEXAS
The University of Texas at Austin