PyKokkos: Performance Portable Kernels in Python

Nader Al Awar
nader.alawar@utexas.edu
The University of Texas at Austin
Austin, Texas, USA

Neil Mehta
neilmehta@lbl.gov
NERSC
Berkeley, California, USA

Steven Zhu
stevenzhu@utexas.edu
The University of Texas at Austin
Austin, Texas, USA

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

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

ABSTRACT

As modern supercomputers have increasingly heterogeneous hardware, the need for writing parallel code that is both portable and performant across different hardware architectures increases. Kokkos is a C++ library that provides abstractions for writing performance portable code. Using Kokkos, programmers can write their code once and run it efficiently on a variety of architectures. However, the target audience of Kokkos, typically scientists, prefers dynamically typed languages such as Python instead of C++. We demonstrate a framework, dubbed PyKokkos, that enables performance portable code through Python. PyKokkos transparently translates code written in a subset of Python to C++ and Kokkos, and then connects the generated code to Python by automatically generating language bindings. PyKokkos achieves performance comparable to Kokkos in ExaMiniMD, a ~3k lines of code molecular dynamics mini-application. The demo video for PyKokkos can be found at https://youtu.be/1oFvhlhoDaY.

KEYWORDS

PyKokkos, Python, high performance computing, Kokkos

ACM Reference Format:
Nader Al Awar, Neil Mehta, Steven Zhu, George Biros, and Milos Gligoric. 2022. PyKokkos: Performance Portable Kernels in Python. In *44th International Conference on Software Engineering Companion (ICSE '22 Companion), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3510454.3516827

1 INTRODUCTION

Modern high-performance computing (HPC) systems are adopting increasingly heterogeneous hardware: the current TOP500 list [3], which ranks supercomputers based on a standard benchmark, shows that seven of the top ten include more than one kind of processor, typically a CPU and a GPU. This hardware is provided by various semiconductor chip vendors, including Intel, Nvidia, and AMD. This presents a challenge to end users, as targeting each kind of hardware requires that users learn specific programming interfaces and frameworks, such as OpenMP or CUDA, and learn about architecture-specific details to extract optimal performance, such as optimal memory layouts. Consequently, users end up re-writing code to achieve the same functionality on different hardware.

It is therefore desirable to write code once and be able to run it on different hardware without losing performance. Kokkos [10] is a framework and C++ library for writing *performance portable* code. Using Kokkos, users can write parallel, high-performance code that can run efficiently on different hardware without needing to re-write any code. Kokkos achieves this by providing *high-level abstractions* that generalize over different HPC frameworks, providing unified syntax and hiding architecture-specific details.

Python has recently seen widespread use in the machine learning and scientific computing communities [9]. As the main implementation of Python is an interpreter, it's performance is an issue when compared to C++. Python users have therefore turned to libraries and packages such as NumPy [7], which provides a high-performance array type, and SciPy [11], which includes native implementations of algorithms commonly used in scientific computing. These implementations are written in C or C++ and are exposed to Python. However, scientists typically need to write their own implementations of parallel high-performance functions (also known as *kernels*), ideally using Python.

We present PyKokkos, a Python framework for writing performance portable kernels entirely through Python [4, 12]. PyKokkos is a Python implementation of the Kokkos framework, and allows users to write high-performance kernels that can run efficiently on a variety of architectures. PyKokkos provides a domain-specific language (DSL for short) embedded in Python for writing these kernels. It will translate this DSL into C++ and Kokkos, and then automatically generate language bindings to access the generated kernel code from Python.

We evaluated PyKokkos by porting existing Kokkos applications and kernels to Python and PyKokkos [4], finding that PyKokkos applications can achieve performance similar to their Kokkos counterparts, while being more concise (i.e., requiring less lines of code).

PyKokkos is open source and is publicly available on GitHub as part of the official Kokkos organization at:
https://github.com/kokkos/pykokkos.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
ICSE '22 Companion, May 21–29, 2022, Pittsburgh, PA, USA
© 2022 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9223-5/22/05.
https://doi.org/10.1145/3510454.3516827

```
1  import pykokkos as pk
2
3  @pk.functor
4  class InnerProduct:
5      def __init__(self, N: int, M: int):
6          self.N: int = N
7          self.M: int = M
8          self.y: pk.View1D[int] = pk.View([N], dtype=int)
9          self.x: pk.View1D[int] = pk.View([M], dtype=int)
10         self.A: pk.View2D[int] = pk.View([N, M], dtype=int)
11
12     @pk.workunit
13     def yAx(self, j: int, acc: pk.Acc[int]):
14         temp2: int = 0
15         for i in range(self.M):
16             temp2 += self.A[j][i] * self.x[i]
17         acc += self.y[j] * temp2
18
19 # Assume N, M are given on the command line and parsed before use
20 if __name__ == "__main__":
21     pk.set_default_space(pk.OpenMP)
22     t = InnerProduct(N, M)
23     policy = pk.RangePolicy(pk.Default, 0, N)
24     result = pk.parallel_reduce(policy, t.yAx)
```

Figure 1: An example of a matrix-weighted inner product kernel from the Kokkos tutorial written in PyKokkos.

2 EXAMPLE

In this section, we first describe the main abstractions used in Kokkos, and then show an example of a PyKokkos kernel that illustrates these abstractions in Python.

2.1 Kokkos

The main goal of Kokkos is to allow writing high performance code that is portable across different architectures. Consequently, it provides abstractions for *parallel execution* and *data structures* to enable this goal. The main abstractions for parallel execution include execution spaces, which represent the processors on a particular machine, such as CPUs and GPUs; execution patterns, which represent common parallel operations, such as a parallel for, parallel reduce, and parallel scan; and execution policies, which specify *how* a kernel will run (i.e., execution space, number of threads, etc.). The main abstractions for data structures include memory spaces, which represent the memory accessible from these processors, and memory layouts, which specify how memory buffers are arranged in memory, such as row-major or column-major.

2.2 PyKokkos

Figure 1 shows an example of a matrix-weighted inner product kernel written in Python and PyKokkos. This was originally written in C++ and Kokkos in the `03` exercise in the official Kokkos tutorials repository [1], but we ported the example to Python and PyKokkos.

To use PyKokkos from Python, the user must first import the `pykokkos` module (line 1). The `as pk` statement means that `pk` can be used as an alias to `pykokkos`.

PyKokkos provides three styles for writing kernels. The style shown in Figure 1 is an example of the *ClassSty* style. In this style, the user first defines a class with a `@pk.functor` decorator (line 3), referred to as a functor. The user can then write each kernel as a method in the class decorated with `@pk.workunit` (line 12).

Inside the class, the user defines a constructor, which is the `__init__` method in Python (line 5). In the constructor, the user defines all member variables that they wish to access from the kernels. As PyKokkos will translate kernels to C++, the user must specify the types of all variables that will be used in kernel code. This is accomplished through the use of Python's type annotations [2]. Lines 6 and 7 show an example of member variables defined as integers using Python's `int` type annotation. Besides integers, PyKokkos allows other Python primitive types such as `bool`, `float`, as well as NumPy primitive types. Another important datatype used in Kokkos and PyKokkos is the View. A View is an n-dimensional array that serves as the main data structure in Kokkos. PyKokkos provides type annotations for views that include the dimensionality and the datatype (lines 8-10). The View constructor accepts as input a list of dimensions and the datatype of the elements. Crucially, the user does not need to specify the memory layout (i.e. row-major or column-major), as that will be selected by PyKokkos using the currently enabled execution space.

With the member variables defined, the user can begin writing kernels. Recall, a kernel is defined as a method decorated with `@pk.workunit`, `yAx` in this example (line 13). The first argument of a workunit is `self`, which simply refers to the class instance. This argument will not be translated to C++ as `this` is implicit in C++; a type annotation is therefore not needed. The second argument is an integer that represents a thread ID, which will have a unique value per each thread at run-time. Since this kernel will perform a reduction, we will need a third argument to hold the result of that reduction, called an *accumulator*. In C++ and Kokkos, it would be enough to pass a variable by reference to hold the result. Python, however, does not allow passing primitive types by reference. Consequently, we introduce a new type annotation, `pk.Acc`, parameterized on the datatype of the accumulator, i.e. `pk.Acc[int]` which is equivalent to `int&` in C++.

The kernel's body also contains type annotations. We first define a temporary variable (line 14), then perform a sequential reduction (lines 15-16). Finally, we update the accumulator (line 17).

The user can now call the kernel. Starting from `main` (line 20), the user first sets the default execution space to be OpenMP (line 21). This ensures that, by default, all views will be allocated in a memory space accessible from the CPU with the appropriate memory layouts. The user then creates an object of the functor class (line 22) and a `RangePolicy`, specifying the execution space (`pk.Default` will evaluate to OpenMP in this case), the starting thread ID, and the number of threads to launch (line 23). The user can then call `pk.parallel_reduce`, passing in the execution policy and the kernel to be executed. When the kernel finishes execution, the result is returned (line 24).

To run this kernel with CUDA, the only change necessary is passing `pk.Cuda` to `pk.set_default_space` on line 21.

3 TECHNIQUE AND IMPLEMENTATION

In this section, we describe the implementation and workflow of the PyKokkos framework [4, 12]. The workflow of PyKokkos can

be divided into two phases: an ahead-of-time (AOT) phase and a run-time phase. During the AOT phase, PyKokkos translates kernel code to C++ and Kokkos, then generates language bindings code to allow inter-operation between Python and the generated kernel code, and finally compiles the generated code. During the run-time phase, PyKokkos imports the compiled code from Python and calls it. Additionally, PyKokkos makes use of existing Python language bindings for C++ Kokkos views from the PyKokkos-Base repository.

3.1 AOT Phase

Figure 2 [12] shows a high level overview of the implementation and workflow of PyKokkos. First, the user provides the Python files containing the PyKokkos kernel code to PKC (step ① in Figure 2). PKC, short for PyKokkos compiler, is the main component of the framework which handles translation and language binding code generation, accessible through a command line script.

PKC will parse the user-provided Python files to extract a Python abstract syntax tree (AST for short) (step ②)using the Python standard library module `ast`. The translator component of PKC will walk through this tree and translate it to a C++ AST that contains the functor and kernel code (step ③).

Once the kernel code is generated, PKC must do additional work to make it accessible from Python. This is accomplished through the use of language bindings, which allow for inter-operation between different languages. For PyKokkos, we are interested in calling C++ from Python, so we make use of `pybind11`, a library to create Python bindings of C++ code. PKC will generate a wrapper function that instantiates the functor and calls the kernel, and then generate `pybind11` code to bind the wrapper function.

The output of the translator is a C++ AST that includes both the functor and the language binding code. PKC serializes the AST into a C++ source file (step ④) and compiles it into a *shared object* file (step ⑤) that it caches on the filesystem to be used at run-time.

3.2 Run-Time Phase

During the run-time phase, the user calls their kernel code as if it were normal Python (line 24 in Figure 1). At this stage, PyKokkos checks if the kernel code has already been translated and compiled in the AOT phase by looking for the shared object file. If PyKokkos does not find it, it will internally call PKC to generate it at run-time (step ⑥). Note that this will incur significant overhead due to calling the C++ compiler; however, once the shared object file has been generated, subsequent calls to the kernel will simply re-use it instead of re-compiling, even across different runs.

PyKokkos will then import the shared object file and call the requested kernel (step ⑦), returning the result if the kernel performed a parallel reduce or scan operation (step ⑧).

PyKokkos additionally makes use of existing Python language bindings for C++ Kokkos views. These bindings allow calling the C++ constructor of the views, which will return a View object to Python that behaves as a regular NumPy array. As in Kokkos, PyKokkos will automatically select the memory space and layout according to the default execution space, although the user is allowed to manually override these. In case the selected memory space is not accessible from Python (e.g., GPU memory), PyKokkos will instead allocate the View in main memory and automatically copy data to the necessary memory space prior to kernel execution. This saves the user from reasoning about data copying and synchronization and also allows PyKokkos to support any architecture as long as it supports data copying to and from main memory.

4 INSTALLATION

In this section we describe the steps needed to install PyKokkos.

Required software and libraries. PyKokkos requires the Conda [5] package manager and compilers supported by Kokkos (e.g. NVCC for CUDA). Each Kokkos execution space additionally requires the corresponding framework's software (e.g., a CUDA installation).

The first step is to clone the PyKokkos-Base repository and install the necessary dependencies into a new Conda environment.

```
$ git clone https://github.com/kokkos/pykokkos-base/
$ cd pykokkos-base
$ conda create --name pyk --file requirements.txt
```

This will create an environment called `pyk`. Afterwards, the user can install PyKokkos-Base into the environment.

```
$ python setup.py install -- -DKokkos_ENABLE_OPENMP=ON \
    -DKokkos_ENABLE_CUDA=ON -DENABLE_LAYOUTS=ON
```

This command calls the Python setup script, which will compile the C++ View constructor bindings. The arguments after `install` specify the execution spaces to enable, as well as enabling memory layouts in the View constructors. The next step is to clone and install PyKokkos itself.

```
$ git clone https://github.com/kokkos/pykokkos/
$ pip install --user -e .
```

5 USAGE

We briefly describe how PyKokkos applications can be executed. The first step is to invoke `pkc.py` script, passing in one or more files containing the kernels and specifying the execution space. Since the PyKokkos code is embedded in regular Python code, the application can then be launched normally.

```
$ pkc.py 03.py -spaces OpenMP
$ python 03.py
```

Figures 3 and 4 show screenshots of the output of these commands respectively. Alternatively, users can skip the call to `pkc.py` and launch the application directly, causing PyKokkos to translate and compile the kernels at run-time.

6 EVALUATION

In this section, we summarize a performance evaluation of PyKokkos using ExaMiniMD [4], a ~3k lines of code molecular dynamics mini-application. ExaMiniMD was originally written in C++ and Kokkos, but we ported it to Python and PyKokkos.

Figure 5 shows a plot the number of atoms (x-axis) and total ExaMiniMD execution time (y-axis). We show data for both PyKokkos and Kokkos, using both OpenMP and CUDA. The plots show that Python and PyKokkos with OpenMP only introduces minimal, constant overhead that does not scale with the size of the input data, even as the number of atoms increases. For CUDA, we do observe extra overhead. By profiling ExaMiniMD further, we found that the PyKokkos kernels themselves achieved performance identical to the original Kokkos kernels. The additional constant overhead can

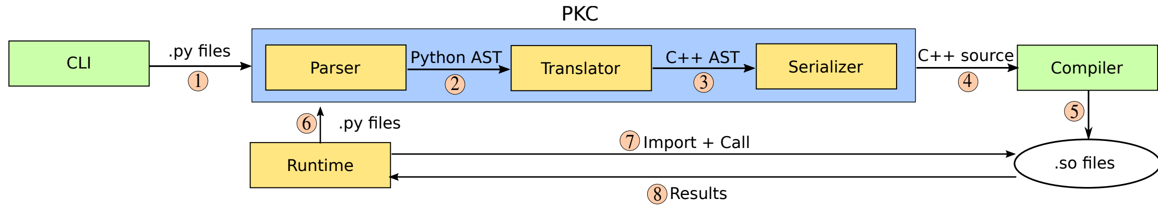

Figure 2: An overview of the PyKokkos framework implementation.

```
(pyk_final) nalawar@seoul:~/demo$ pkc.py 03.py -spaces OpenMP
INFO:root:Path 03.py
INFO:root:0 workloads
INFO:root:1 functors
INFO:root:0 workunits
INFO:root:0 classtypes
INFO:root:translation 0.001s
INFO:root:compilation 7.386s
```

Figure 3: Screenshot of using PKC from the command line.

```
(pyk_final) nalawar@seoul:~/demo$ python 03.py
Total size S = 262144 N = 256 M = 1024
Computed result for 256 x 1024 is 262144.0
N(256) M(1024) nrepeat(100) problem(MB) time(2.12) bandwidth(GB/s)
```

Figure 4: Screenshot of running the 03 exercise.

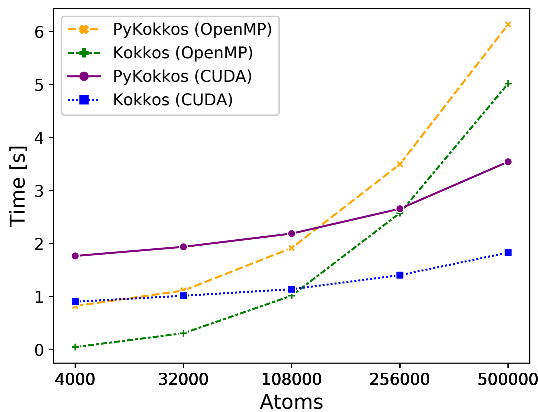

Figure 5: ExaMiniMD total execution time.

be attributed to the startup time of the Python interpreter. Furthermore, the extra overhead for CUDA can be attributed to Kokkos prefetching memory, which is currently not available in PyKokkos (although support for this is being added currently).

In summary, PyKokkos achieves performance on par with Kokkos with only small overhead. Our ICS'21 paper [4] includes a more extensive evaluation on numerous smaller kernels, showing similar results, as well as a study of code complexity that shows that PyKokkos code is more concise and less verbose than Kokkos.

7 CONCLUSION

We presented PyKokkos, a framework for writing performance portable kernels using Python. Existing approaches include Cython [6], which provides C-like language extensions and statically compiles code for better performance; Cython, however, currently has limited support for parallelism. Numba [8] is a just-in-time compiler that compiles a subset of Python to LLVM IR. Numba supports parallelism, but does not provide performance portability. WayOut [12] automatically generates language bindings for existing C++ code; the developers were able to generate bindings for a library of pre-existing kernels written in C++ and Kokkos. PyKokkos allows users to write new kernels entirely through Python. Our evaluation showed that PyKokkos can match Kokkos for performance, even for larger applications such as ExaMiniMD.

ACKNOWLEDGMENTS

We thank Martin Burtscher, Mattan Erez, Ian Henriksen, Damien Lebrun-Grandie, Jonathan R. Madsen, Arthur Peters, Keshav Pingali, David Poliakoff, Sivasankaran Rajamanickam, Christopher J. Rossbach, Joseph B. Ryan, Karl W. Schulz, and Christian Trott. This work was partially supported by the US National Science Foundation under Grant Nos. CCF-1652517 and CCF-1817048, and the Department of Energy, National Nuclear Security Administration under Award Number DE-NA0003969.

REFERENCES

[1] 2015. Kokkos Tutorials. https://github.com/kokkos/kokkos-tutorials.
[2] 2020. typing - Support for type hints. https://docs.python.org/3/library/typing.html.
[3] 2021. Top 500 November 2021. https://www.top500.org/lists/top500/2021/11/.
[4] Nader Al Awar, Steven Zhu, George Biros, and Milos Gligoric. 2021. A Performance Portability Framework for Python. In *Proceedings of the ACM International Conference on Supercomputing*. 467–478.
[5] Inc. Anaconda. 2021. Conda. https://docs.conda.io/projects/conda/en/latest/.
[6] Stefan Behnel, Robert Bradshaw, Craig Citro, Lisandro Dalcin, Dag Sverre Seljebotn, and Kurt Smith. 2011. Cython: The Best of Both Worlds. In *Computing in Science and Engineering*. 31–39.
[7] Charles R. Harris, K. Jarrod Millman, Stefan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernandez del Rio, Mark Wiebe, Pearu Peterson, Pierre Gerard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. 2020. Array programming with NumPy. *Nature* 585, 7825 (2020), 357–362.
[8] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. 2015. Numba: A LLVM-Based Python JIT Compiler. In *Workshop on the LLVM Compiler Infrastructure in HPC*. 1–6.
[9] Travis E. Oliphant. 2007. Python for Scientific Computing. *Computing in Science and Engineering* 9, 3 (2007), 10–20.
[10] Christian Trott, Luc Berger-Vergiat, David Poliakoff, Sivasankaran Rajamanickam, Damien Lebrun-Grandie, Jonathan Madsen, Nader Al Awar, Milos Gligoric, Galen Shipman, and Geoff Womeldorff. 2021. The Kokkos EcoSystem: Comprehensive Performance Portability for High Performance Computing. *Computing in Science Engineering* 23, 5 (2021), 10–18.
[11] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stefan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. 2020. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods* 17 (2020), 261–272.
[12] Steven Zhu, Nader Al Awar, Mattan Erez, and Milos Gligoric. 2021. Dynamic Generation of Python Bindings for HPC Kernels. In *International Conference on Automated Software Engineering (ASE)*. 92–103.