A Performance Portability Framework for Python

Nader Al Awar  
nader.alawar@utexas.edu  
The University of Texas at Austin  
Austin, Texas, 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
Kokkos is a programming model for writing performance portable applications for all major high performance computing platforms. It provides abstractions for data management and common parallel operations, allowing developers to write portable high performance code with minimal knowledge of architecture-specific details. Kokkos is implemented as a heavily-templated C++ library. However, C++ is not ideal for rapid prototyping and quick algorithmic exploration. An increasing number of developers use Python for scientific computing, machine learning, and data analytics. In this paper, we present a new Python framework, dubbed PyKokkos, for writing performance portable applications entirely in Python. PyKokkos provides Kokkos-like abstractions that are easier to use and more concise than the C++ interface. We implemented PyKokkos by building a translator from a subset of Python to C++. Kokkos and bridging necessary function calls via automatically generated Python bindings. PyKokkos is also compatible with NumPy, a widely-used high performance Python library. By porting several existing Kokkos applications to PyKokkos, including ExaMiniMD (∼3k lines of code in C++), we show that the latter can achieve efficient execution with low performance overhead.

CCS CONCEPTS
• Software and its engineering → Source code generation;  
• Computing methodologies → Parallel programming languages.

KEYWORDS
PyKokkos, Python, high performance computing, Kokkos

1 INTRODUCTION
Traditionally, parallel, high-performance code for scientific applications is written in low-level, architecture-specific high performance computing (HPC) frameworks such as OpenMP [28], CUDA [14], and others. These frameworks require that the user be aware of architecture-specific details in order to write efficient code. For example, the optimal data layout of a two-dimensional array differs across different hardware devices: row-major on a CPU (OpenMP) to enable cached memory accesses vs. column-major on a GPU (CUDA) for coalesced memory accesses [18]. Additionally, each framework has its own syntax for expressing parallel execution patterns. This results in code that is closely coupled to a framework’s syntax and idioms. Once an HPC application is implemented using a specific framework, it cannot easily be ported to run on other frameworks and devices.

Recently, there has been a paradigm shift in HPC programming models to account for the issues mentioned above. Kokkos [18] and RAJA [7] are two models that provide layers of abstraction over existing HPC frameworks to enable writing performance portable code, i.e., code that runs on different architectures with good performance. Both models include high-level abstractions for expressing common parallel execution patterns and memory layouts, and hide low-level details about the target framework or device from the user. Kokkos and RAJA are both implemented in C++, and applications written in either of the two can run on multiple devices with minimal or no code changes required.

While Kokkos and RAJA have achieved their goal of performance portability [20], general usability remains an issue. Templates, cryptic error messages, manual memory management, complicated build processes, and other aspects of C++ make for a high barrier of entry for scientists with limited backgrounds in computer science and programming, despite scientific computing being an important use-case of the Kokkos model.

Due to these shortcomings, dynamic languages such as Python and Julia [9] are preferred to C++ in the scientific computing and machine learning communities [27], both for algorithmic exploration but also increasingly for production. In the past decade, numerous libraries have been developed for writing high-performance Python code [6, 21, 30, 39]. For example, the NumPy library [21] provides a high-performance multi-dimensional array type that is at the core of scientific computing in Python.

While these libraries provide Python APIs, their performance critical functions (also commonly called kernels) are implemented
in C or C++ for performance and portability reasons. These kernels are then wrapped in manually written language bindings for interoperability with other languages, including Python. This is commonly done in practice and can be seen in some of the most popular Python packages, including SciPy [39], a Python library for scientific computing, and machine learning libraries such as TensorFlow [6] and PyTorch [30]. However, if a kernel is not available, developers have to look for alternatives.

Numba [25] is a just-in-time compiler for Python that targets LLVM [26]. Numba can target a number of devices but does not provide high-level abstractions to hide device-specific code, so portability remains an issue. Cython [8] is a static compiler that extends Python with C-like syntax to achieve better performance. However, these extensions make Cython a superset of Python, which may not be desirable, and Cython supports only OpenMP for parallelism at this point.

We present PyKokkos, the first framework for writing performance portable applications in a subset of Python. PyKokkos is an implementation of the Kokkos programming model. It provides an API that enables developers to write high-performance, device-portable code entirely in Python. Additionally, PyKokkos interoperates with NumPy arrays, allowing for easy integration with existing scientific applications written in Python.

PyKokkos translates Python kernel code to C++ Kokkos. Furthermore, it automatically generates the necessary Python language bindings. It also makes use of existing (manually-written) Kokkos bindings for memory allocations. Crucially, PyKokkos makes no changes to the Python language or its interpreter. We evaluated PyKokkos by manually porting a number of kernels from C++ Kokkos to PyKokkos, as well as ExaMiniMD [4], a scientific application for molecular dynamics.

The main contributions of this paper include:

- Design of a framework, dubbed PyKokkos, for writing performance portable Python code. PyKokkos is designed to closely follow the Kokkos programming model while being more concise and easier to use than C++ Kokkos.
- Implementation of the framework by combining code translation and automatic binding generation. PyKokkos supports three styles to write PyKokkos applications and can currently run on both CPUs and Nvidia GPUs.
- Evaluation of PyKokkos using a number of applications, including existing high-performance kernels and ExaMiniMD, which is a large-scale molecular dynamics application. Our results show that the kernels generated by PyKokkos can match the performance of manually written C++ kernels.

PyKokkos source code and applications that we wrote are available at https://github.com/kokkos/pykokkos.

2 BACKGROUND AND EXAMPLE

In this Section, we first provide some background on Kokkos (Section 2.1), then we introduce PyKokkos via an example (Section 2.2).

2.1 Kokkos

Kokkos is a programming model that provides abstractions for writing performance portable HPC code. The two major components of the Kokkos model are execution spaces and memory spaces. Given a computing node, the processors are modeled as execution space instances, and the different memory locations are modeled as memory spaces. For example, on a machine with a CPU and a GPU, there could be two (or more) execution spaces, the CPU and the GPU, and two corresponding memory spaces, main memory and GPU memory. Other main Kokkos abstractions include:

- **Execution patterns**: an execution pattern represents a parallel operation, including parallel for, parallel reduce, and parallel scan, as well as task-based programming abstractions.
- **Execution policies**: an execution policy specifies how a parallel operation runs. The simplest policy is RangePolicy, which specifies that an operation will run for all values in a range. Another policy is the TeamPolicy that can be used for hierarchical (also known as nested) parallelism. The execution policy can also be used to set the execution space.
- **Memory layouts**: the memory layout specifies how data buffers are laid out in memory. For example, Kokkos supports column-major and row-major layouts among others.
- **Memory traits**: the memory trait specifies access properties of data buffers. For example, this could be set to Atomic, so that all accesses to elements of the data buffer are atomic.

The C++ Kokkos library (Kokkos for short) is a concrete instantiation of the programming model described above. The main data structure is a multi-dimensional array referred to as a `View`. It is implemented as a C++ class templated on the data type, number of dimensions, memory space, memory layout, and memory trait. It maintains a memory buffer internally and uses reference counting for automatic deallocation. The following code snippet shows an example of a one-dimensional `View` of size $N$ holding elements of type int.

```cpp
Kokkos::View<int*> v("v", N);
```

Kokkos uses C++ functors to define the computational body, also known as a workunit, of parallel operations. Functors are classes or structs that define `operator()` as an instance method. The body of this method represents the operation that will be executed by the threads. The following code shows a simple example of a functor that performs a reduction over all the elements of a `View`.

```cpp
struct Functor {
    Kokkos::View<int*> v;
    Functor(Kokkos::View<int*> v) { this->v = v; }

    KOKKOS_FUNCTION
    void operator() (int tid, int& acc) const {
        acc += this->v(tid);
    }
};
```

KOKKOS_FUNCTION is a macro that abstracts framework-specific function type qualifiers for portability (e.g., `__host__ __device__` for CUDA). A work index (tid in the example above) parameter representing the thread ID is included in the operator() method signature. Since this is a reduction operation, a scalar result must be returned, so the definition includes an additional parameter, called an accumulator, that is passed by reference to hold that result. The scan operation additionally requires a boolean parameter to indicate whether the scan operation is on its final pass; the final pass is used
to update the elements of a View. The parallel for operation only requires a work index as a parameter.

All the variables and Views needed by a functor are defined as instance variables (see \texttt{v} in the snippet above). An alternative to functors is C++ \texttt{lambda}s, or anonymous functions. Instead of instance variables, lambdas capture all the variables they need from the scope they are defined in. Lambdas are commonly more concise than functors, but the two are otherwise equivalent.

Kokkos provides a different function for each parallel operation: \texttt{parallel_for}, \texttt{parallel_reduce}, and \texttt{parallel_scan}. These functions accept as input an execution policy (or simply the number of threads) as the first argument and a functor object or a lambda as the second argument. As mentioned before, reduce and scan return a scalar result, so their functions accept as input a third argument passed by reference to hold that result. The following code shows how the functor defined earlier is used to call \texttt{parallel_reduce}, where \texttt{N} represents the number of elements of the View.

\begin{verbatim}
Functor \( f(v); \text{int} \ acc = 0; \)
Kokkos::parallel_reduce(
    Kokkos::RangePolicy<>(0, N), f, acc);
\end{verbatim}

Kokkos implements these operations for all the HPC backends it supports, including OpenMP, CUDA, and others. The user selects which backends to enable when invoking the compiler. During compilation, Kokkos selects the default execution spaces from the enabled backends, the corresponding memory spaces, and the optimal memory layouts for those spaces. An application can be ported to other devices by re-compiling with the needed execution spaces.

2.2 PyKokkos via an Example

PyKokkos is a Python implementation of the Kokkos model that enables developers to write performance portable Python applications. It is implemented as a Python framework and provides an API that is similar in structure to the Kokkos API, but is as easy to use as regular Python (based on our experience). Internally, PyKokkos translates certain parts of the application into Kokkos and C++, automatically generates Python bindings for interoperability, and compiles and imports them. It also makes use of existing bindings to Kokkos to perform memory allocation.

Figure 1 shows an example written entirely in Python using PyKokkos. This example is taken from the team_vector_loop exercise in the Kokkos tutorials repository [2], and is used to demonstrate hierarchical parallelism in Kokkos. It calculates a matrix-weighted inner product \( y^T Ax \). We manually ported the example from Kokkos to PyKokkos.

The first step in writing a PyKokkos application is to import the pykokkos package (line 1). The as \texttt{pk} statement added after the import statement indicates that \texttt{pk} is an alias for pykokkos.

A PyKokkos functor is defined by decorating a class definition with \texttt{pk functor} (line 3). The functor includes a constructor \texttt{__init__} (line 5) which defines member variables and Views. All class members that are meant to be used in PyKokkos code have to be defined with type annotations [5] in the constructor. PyKokkos provides type annotations for Views that include the number of dimensions, i.e., \texttt{View1D}, \texttt{View2D}, etc. up to eight dimensions (the maximum allowed by Kokkos) as well as the data type. Additional

```python
1 import pykokkos as pk
2 @pk.functor
3 class TeamVectorLoop:
4     def __init__(self, N: int, M: int,........
9     self.M: int = M
10     self.y: pk.View2D[int] = y
11     self.x: pk.View2D[int] = x
13     @pk.workunit
14     def yAx(self, m: pk.TeamMember, acc: pk.Acc[int]):
15         e: int = m.league_rank()
16         def team_reduce(j: int, team_acc: pk.Acc[int]):
17             def vector_reduce(i: int, vector_acc: pk.Acc[int]):
18                 def single():
19                     def team_thread_reduce(m: pk.TeamThreadRange(m, self.N), team_reduce)
20                     def parallel_reduce(policy, t.yAx)
21                     def parallel_scan(policy, t.yAx)
22                     def parallel_for(policy, t.yAx)
23                     def parallel_scan(policy, t.yAx)
24                     def parallel_for(policy, t.yAx)
25                     def parallel_scan(policy, t.yAx)
26                     def parallel_for(policy, t.yAx)
27                     def parallel_scan(policy, t.yAx)
28                     def single():
29                         def nonlocal acc
30                         acc += tempN
31                         pk.single(pk.PerTeam(m), single)
32                         if __name__ == "__main__":
33                             pk.set_default_space(pk.OpenMP)
34                             y = pk.View2D([E, N], dtype=int)
35                             x = pk.View2D([E, M], dtype=int)
36                             A = pk.View3D([E, N, M], dtype=int)
37                             t = TeamVectorLoop(N, M, y, x, A)
38                             policy = pk.TeamPolicy(pk.Default, E, pk.AUTO, M)
39                             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.
```
default execution space. The Views are then passed to a functor object through the constructor (line 40).

The execution policy of the functor is a TeamPolicy (line 41) since it uses hierarchical parallelism. The first argument is the execution space. OpenMP in this case since it was set as the default. The second argument is the number of thread teams. In Kokkos, a single thread team is a group of threads that share a common team index. The third argument is the size of each team; AUTO tells Kokkos to select the appropriate team size based on the target architecture. The final argument is the vector length i.e., the number of threads on the final level of parallelism.

To run the functor, parallel_reduce is called with the execution policy and workload passed as arguments (line 42). When the workload finishes execution, parallel_reduce returns the result of the reduction operation. This is in contrast to Kokkos, which places the result in a variable passed by reference.

The body of the parallel operation is defined as a method decorated with @pk.workunit (line 14). Since this is a reduction operation, the workload has two parameters: a work index and an accumulator variable. The work index for this workload has to be of type pk.TeamMember since it uses hierarchical parallelism. Since the accumulator is modified in the workload, it cannot be a primitive type in Python, so we use the pk.Acc class type parameterized with a specific data type.

On the outermost team level, each thread obtains its team index via league_rank() (line 15), a value shared across threads in the same level. The second level is the thread level and the third and final level is the vector level. The operations in the inner levels are defined using nested functions (lines 17 and 18). Nested functions capture the variables that are in scope when they are defined. In this case, both functions capture e (the team index), and the innermost function captures j (the thread index). The nested functions can then be invoked by calling parallel_reduce with the appropriate execution policy (lines 22 and 26). Finally, one thread per team member updates the outermost accumulator variable (line 31). The nonlocal statement is needed in Python so that acc is not redefined in the nested function. Once all threads are finished executing, the reduction result is returned through the original parallel_reduce on line 42.

This example can be executed with CUDA by simply changing the default execution space (line 35). PyKokkos takes care of setting the proper memory spaces and layouts in the View constructors. It is also possible to set the default execution space externally in a constructor. Views and other members of the class are defined with type-annotations in the constructor and can be used in workunits.

In the ClassSty style (used in Figure 1), workunits are defined as methods, and a single class can contain one or more workunits. Each class is similar in style to a Kokkos functor, with the major difference being that workunits are annotated with @pk.workunit instead of the operator() method in C++. Only Views and other member variables that are defined with type-annotations in the constructor can be used in workunits. Additionally, Kokkos functions can be defined as methods inside a PyKokkos class using the @pk.function decorator. These methods can then be called from any workunit within the class.

3.1.2 ClassStyWithMain. The ClassStyWithMain style is similar to the ClassSty style except that it also contains a special method decorated with @pk.main, which we refer to as the PyKokkos main method. This method allows us to use parts of the Kokkos API for which we currently do not have bindings, such as BinSort. We add Python endpoints similar to the Kokkos API and translate those calls directly to the corresponding C++ version. This can also be used to call parallel operations, which similarly get translated to Kokkos. To execute the main method, the user calls pk.execute(execution_space, instance), where instance is an instance of a pk.workload class.

3.1.3 FunctionSty. With this style, PyKokkos attempts to mimic C++ lambda usage in Kokkos. (Using Python lambdas is not an option since they are limited to a single expression unlike lambdas in C++) The FunctionSty style allows standalone workunits that are defined as global functions (outside any class). In addition to the specific arguments required by each operation (e.g., accumulator for reduction), all Views and variables needed by the workunit are passed as type-annotated arguments. These arguments are passed to the workunit when the parallel operation is called.

Figure 2: Visual summary of the three code styles supported in PyKokkos; the highlighted boxes represent the code that is translated to C++.
All major loop-based execution patterns are supported. There is CUDA GPU unified memory. The supported memory layouts are (row-major) and LayoutLeft. The supported Kokkos backends are OpenMP, CUDA, Threads, Serial, and CudaUVMSpace. PyKokkos annotated code has to conform to the C++ scoping rules in this regard.

3.3 Syntax Rules
PyKokkos translates all functions and classes that are annotated with @pk.functor, @pk.workunit, and @pk.function, which we collectively refer to as annotated code, to C++ Kokkos. This forces restrictions on what is allowed in annotated code. In this Section, we describe these restrictions in detail.

Python is a dynamically typed language, meaning that variable types can change at run-time. On the other hand, C++ is statically typed, meaning that all variable types need to be known at compile-time and cannot be altered at run-time. Therefore, annotated code must have type annotations for all variables and Views; this includes both local and instance variables. Additionally, these variables cannot be assigned to values of a different type. These restrictions do not apply outside annotated code.

Another characteristic of Python that affects translation is scoping. Whenever a function is called in Python, it creates a new local scope. Variables defined inside control blocks like if and for are scoped to the containing function. If the body of a control block contains a variable definition, then that variable can be accessed after the control block provided that it is executed. If the body of the control block is not executed, accessing the variable results in a run-time error. In C++, variables defined in control blocks go out of scope at the end of those blocks. Attempting to access these variables outside the block they were defined in results in a compile-time error. Therefore, PyKokkos annotated code has to conform to the C++ scoping rules in this regard.

Finally, not all variable types are allowed in annotated code. As of now, the types allowed are int, float, bool, C++ integer and floating point types of different sizes (e.g., int32_t, double, etc.), pk.View, and some NumPy primitive types. PyKokkos also allows user-defined classtypes that can be used in annotated code. These classtypes are Python classes with constructors and methods decorated with @pk.function (classtypes are therefore also considered as annotated code). Other types are not supported either because they are not necessary (strings), there is no clear C++ equivalent, or the C++ equivalent cannot be used in Kokkos code. Additionally, using modules from the Python Standard Library is not allowed in annotated code, except for several functions from the math module that can be mapped to C++ cmath functions.

In summary, PyKokkos annotated code is a subset of Python that adds restrictions to its dynamic typing, scoping rules, and allowed types in order to enable translation to C++.

4 PYKOKKOS internals
In this Section, we describe the PyKokkos framework internals. We implemented PyKokkos entirely in Python in order to allow for easy integration into existing Python codebases. Additionally, the Python Standard Library contains modules for working with the Python AST.

At a high level, PyKokkos first translates annotated code written in Python into C++ Kokkos code, compiles that code into a shared object file that can be imported as a Python module, and finally

<table>
<thead>
<tr>
<th>Feature</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Views</td>
<td>Multi-dimensional Views, Subviews, Dual Views</td>
</tr>
<tr>
<td>Memory Spaces</td>
<td>HostSpace, CudaSpace, CudaUVMSpace</td>
</tr>
<tr>
<td>Memory Layouts</td>
<td>LayoutRight, LayoutLeft</td>
</tr>
<tr>
<td>Memory Traits</td>
<td>Atomic, RandomAccess, Restrict, Unmanaged</td>
</tr>
<tr>
<td>Execution Spaces</td>
<td>OpenMP, CUDA, Threads, Serial</td>
</tr>
<tr>
<td>Execution Patterns</td>
<td>parallel_for, parallel_reduce, parallel_scan</td>
</tr>
<tr>
<td>Execution Policies</td>
<td>RangePolicy, MDRangePolicy, TeamPolicy, TeamThreadRange, ThreadVectorRange, WorkTag</td>
</tr>
<tr>
<td>Hierarchical Parallelism</td>
<td>Team Loops, Vector Loops</td>
</tr>
<tr>
<td>Atomic Operations</td>
<td>All atomic_fetch_[op] operations</td>
</tr>
<tr>
<td>Other</td>
<td>Kokkos Functions, BinSort, Timer, printf</td>
</tr>
</tbody>
</table>

if we were to write the example in Figure 1 in the FunctionSty style, the variables and Views would have been passed through the call to parallel_reduce() on line 42.
The next step is to translate the AST nodes into C++. The Translator proceeds by extracting all PyKokkos class members, functions, and workunits. First, it extracts all type information for the class members from type annotations. C++ Views are templated on data type, dimensionality, memory layout, memory space, and memory traits, so PyKokkos has to collect this information per View. The data type and dimensionality are extracted from the View type annotation. Non-default memory layouts and memory traits for each View can be passed in as arguments to the PyKokkos decorator, otherwise the default values set by Kokkos are used. Since the memory space depends on the execution space, PyKokkos needs to generate a different template argument per memory space. To avoid generating multiple types per memory space, we use a macro that is defined based on the enabled execution space.

Note that regardless of the PyKokkos style used, annotated code is always translated into Kokkos functors and not lambdas, as this simplifies the translation process. The member variables of the generated C++ Kokkos functor are the class members extracted in the previous step.

The final step is to generate bindings to call the translated workunits. Since there are no existing bindings for invoking the parallel operations, we cannot call them directly from Python. To solve this, the Translator creates wrapper functions that call the parallel operations internally. Figure 4 shows the wrapper function generated for the example shown in Figure 1. The arguments of the wrapper are the members extracted in the previous step and are passed to the functor constructor (line 8). The wrapper then calls parallel reduce (line 11) and returns the result (line 15). The Translator then binds these wrappers using the C++ pybind11 library [32]. The Translator passes the C++ AST to a Serializer (step 3) which generates a source file and passes it to a C++ compiler (step 4) which compiles it into a shared object file (step 5).

During compilation, PKC calls the C++ compiler once for each supported backend (although a user can select only a subset of backends), from which it selects a default execution space in a manner similar to the default selection that occurs during Kokkos compilation. It writes this execution space to a file that is read at run-time and used to set the default execution space as a substitute for the user doing so explicitly (line 35 in Figure 1).

4.2 Runtime

The PyKokkos API can be divided into two groups: an interface for executing code and an interface for Views. First, we show how the PyKokkos Runtime (and by extension Kokkos) is initialized. Second, we show how the Runtime invokes parallel operations. Third, we discuss how Views are created and shared between Python and C++. Finally, we describe how annotated code can be run sequentially in Python, which can help debug kernels.
4.2.1 Initialization. PyKokkos is initialized when the import pykokkos statement is executed. This creates all the necessary entities that are needed by PyKokkos at run-time: the Runtime, Parser, Translator, and Serializer. Additionally, PyKokkos internally calls Kokkos::initialize(). This initializes all Kokkos internal objects and acquires hardware resources. PyKokkos also registers Kokkos::finalize() to be called when Python terminates.

4.2.2 Parallel Execution. To call a parallel operation, the user has to pass in a workunit and execution policy. This workunit can either be a method in an initialized object i.e., ClassSty, or a free function i.e., FunctionSty. For the latter, the user also passes in all the necessary arguments. For the former, the Runtime automatically extracts these arguments from the class members. The ClassStyWithMain style does not require an execution policy since it executes multiple workunits, each of which could potentially have a different policy.

The Runtime then checks whether a module (i.e., the shared object file) corresponding to the workunit has already been generated with PKC. If not, this means that the compile-time phase was skipped by the user, so the Runtime has to call PKC (step 6). The Runtime can then import the module and call the necessary wrapper function (step 7). If any View type or primitive type does not match the C++ type in the translated code, an error message is printed. This could happen if the type was changed in Python at run-time. For ClassSty and FunctionSty the execution policy passed by the user provides additional arguments that are passed on to the wrapper function, where they are used to construct the execution policy object (e.g., line 12 in Figure 4).

The wrapper function instantiates the Kokkos functor and execution policies, and then calls the necessary parallel operations. After execution terminates, the Runtime transfers the results of all reduction and scan operations back to Python. For FunctionSty and ClassSty there is only a single result that will be returned directly by the wrapper function (step 8). For ClassStyWithMain, there could be multiple calls to parallel reduce or scan, so the result of each operation is added to a View that the Runtime can access.

4.2.3 Views. PyKokkos Views are classes created through regular constructor calls (see lines 36-38 in Figure 1). Similar to Kokkos, the user is not expected to set the memory space and layout of a PyKokkos View for portability reasons. Instead, PyKokkos selects these based on the current default execution space. For the CPU execution spaces (such as OpenMP), the memory space is always set to HostSpace. For CUDA, PyKokkos does not use CudaSpace since it is not accessible from Python. It has to select a host accessible memory space i.e., HostSpace or CudaUVMSpace (Unified Virtual Memory [13]). At run-time, HostSpace Views are copied to CudaSpace as needed. This approach allows the user to switch between different execution and memory spaces without worrying about where the data is located in memory. It can also be applied to execution spaces that PyKokkos will support in the future (e.g., AMD GPUs). The only drawback is the overhead introduced by copying data between different memory spaces.

When the PyKokkos View constructor is called, it invokes the C++ Kokkos View constructor internally through the available Python bindings [32]. This constructor allocates the memory for the View data buffer and the binding returns a Python object that provides access to the underlying data buffer through a NumPy array. The returned object can be passed by reference between C++ and Python through pybind11.

The PyKokkos View type is therefore a wrapper over a NumPy array. Its purpose is to provide an interface that is similar to the Kokkos View interface, specifically the constructor. Otherwise, it behaves as a regular NumPy array in Python. This allows PyKokkos to be easily added to existing Python codebases.

4.2.4 Pure Python Execution. Since valid annotated code is a subset of valid Python code, PyKokkos supports execution of workunits in Python. This is especially helpful for debugging logic-based errors in Python rather than C++ due to the dynamic nature of Python.

We implement calls to parallel operations using sequential for loops. In every iteration, we pass the current iteration counter to the workunit as the thread ID. To support hierarchical parallelism, we pass an object which provides access to the thread and team ID. MDRangePolicy iterates over multiple ranges, so we loop over a combination of two thread IDs. In reduce and scan operations, the px.Acc object wraps the result as a substitute for Python’s lack of reference types for primitives. We overloaded the arithmetic operators of px.Acc so it can behave like a regular primitive type without any extra function calls.

5 EVALUATION

In this Section, we present the results of our evaluation of PyKokkos. First, we show how PyKokkos performance compares to C++ Kokkos for smaller applications where the running time is dominated by kernel execution. Second, we compare PyKokkos and Kokkos performance for a larger application. Third, we report the cost of pure Python execution of PyKokkos (i.e., Python sequential execution). Fourth, we compare the PyKokkos code to Kokkos code in terms of the lines of code and number of characters. Finally, we briefly compare PyKokkos with Numba.

5.1 Evaluation Setup

We ran all experiments on an Ubuntu 18.04.5 machine with a 6-core Intel i7-8700 3.20GHz CPU and 64GB RAM and an Nvidia GeForce RTX 2080 GPU with 8GB of memory. For all our experiments, we used Python 3.8.3, Kokkos 3.1.01, OpenMP 4.5, CUDA 10.2, GCC 7.5, and Numba 0.51.

5.2 Subjects

For the purposes of our experiments, we ported existing C++ Kokkos applications to PyKokkos. We implemented 7 exercises from the official Kokkos tutorials repository [2]. All exercises follow a structure similar to the example in Figure 1: calculate a matrix-weighted inner product using an outer loop and inner loop, each of which performs a reduction operation. Each exercise introduces a feature that improves on the previous exercise. A couple of exercises that are not ported use features that we do not currently support, while a number of them are not relevant to PyKokkos, e.g., mdrange, team_policy, and team_vector_loop:

- 02: Introduces Views and uses the View constructors instead of malloc() in 01.
Table 2: Comparison of Execution Time of PyKokkos and Kokkos Applications with OpenMP and CUDA.

<table>
<thead>
<tr>
<th>Application</th>
<th>Size</th>
<th>OpenMP Time [s]</th>
<th>CUDA Time [s]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>PyKokkos</td>
<td>Kokkos</td>
</tr>
<tr>
<td>02</td>
<td>$2^{18} \times 2^{10}$</td>
<td>70.3</td>
<td>69.5</td>
</tr>
<tr>
<td></td>
<td>$2^{19} \times 2^{10}$</td>
<td>140.5</td>
<td>139.2</td>
</tr>
<tr>
<td>03</td>
<td>$2^{18} \times 2^{10}$</td>
<td>69.8</td>
<td>69.5</td>
</tr>
<tr>
<td></td>
<td>$2^{19} \times 2^{10}$</td>
<td>139.5</td>
<td>139.2</td>
</tr>
<tr>
<td>04</td>
<td>$2^{18} \times 2^{10}$</td>
<td>69.6</td>
<td>69.5</td>
</tr>
<tr>
<td></td>
<td>$2^{19} \times 2^{10}$</td>
<td>139.5</td>
<td>139.2</td>
</tr>
<tr>
<td>mrange</td>
<td>$2^{18} \times 2^{10}$</td>
<td>70.2</td>
<td>69.5</td>
</tr>
<tr>
<td></td>
<td>$2^{19} \times 2^{10}$</td>
<td>141.2</td>
<td>139.2</td>
</tr>
<tr>
<td>subview</td>
<td>$2^{18} \times 2^{10}$</td>
<td>69.7</td>
<td>69.5</td>
</tr>
<tr>
<td></td>
<td>$2^{19} \times 2^{10}$</td>
<td>139.9</td>
<td>139.2</td>
</tr>
<tr>
<td>team_policy</td>
<td>$2^{18} \times 2^{10}$</td>
<td>69.8</td>
<td>69.6</td>
</tr>
<tr>
<td></td>
<td>$2^{19} \times 2^{10}$</td>
<td>139.9</td>
<td>139.4</td>
</tr>
<tr>
<td>team_vector_loop</td>
<td>$2^{8} \times 2^{10} \times 2^{10}$</td>
<td>70.6</td>
<td>70.4</td>
</tr>
<tr>
<td></td>
<td>$2^{9} \times 2^{10} \times 2^{10}$</td>
<td>141.0</td>
<td>140.5</td>
</tr>
<tr>
<td>nstream</td>
<td>$2^{27} \times 1$</td>
<td>143.8</td>
<td>144.6</td>
</tr>
<tr>
<td></td>
<td>$2^{28} \times 1$</td>
<td>286.7</td>
<td>287.9</td>
</tr>
<tr>
<td>stencil</td>
<td>$2^{12} \times 2^{12}$</td>
<td>15.9</td>
<td>15.7</td>
</tr>
<tr>
<td></td>
<td>$2^{13} \times 2^{13}$</td>
<td>63.1</td>
<td>62.1</td>
</tr>
<tr>
<td>transpose</td>
<td>$2^{12} \times 2^{12}$</td>
<td>23.9</td>
<td>24.0</td>
</tr>
<tr>
<td></td>
<td>$2^{13} \times 2^{13}$</td>
<td>95.4</td>
<td>95.8</td>
</tr>
<tr>
<td>bytes_and_flops</td>
<td>$2^{12} \times 2^{10}$</td>
<td>127.2</td>
<td>129.8</td>
</tr>
<tr>
<td></td>
<td>$2^{13} \times 2^{10}$</td>
<td>254.4</td>
<td>259.5</td>
</tr>
<tr>
<td>gather</td>
<td>$2^{11} \times 2^{5}$</td>
<td>112.4</td>
<td>111.2</td>
</tr>
<tr>
<td></td>
<td>$2^{12} \times 2^{5}$</td>
<td>223.3</td>
<td>222.7</td>
</tr>
<tr>
<td>gups</td>
<td>$2^{27} \times 1$</td>
<td>104.0</td>
<td>104.0</td>
</tr>
<tr>
<td></td>
<td>$2^{28} \times 1$</td>
<td>207.2</td>
<td>204.9</td>
</tr>
<tr>
<td>BabelStream</td>
<td>$2^{24} \times 1$</td>
<td>71.3</td>
<td>71.5</td>
</tr>
<tr>
<td></td>
<td>$2^{23} \times 1$</td>
<td>143.0</td>
<td>144.2</td>
</tr>
</tbody>
</table>

- 03: Introduces device (i.e., GPU) Views and shows how memory is copied between host and device.
- 04: Introduces memory spaces, layouts, and RangePolicy.
- mrange: Introduces MRangePolicy to initialize matrix A.
- subview: Introduces subview to split each column of A into a one-dimensional View.
- team_policy: Introduces two-level hierarchical parallelism by replacing the inner sequential reduction with a parallel version that uses TeamPolicy.
- team_vector_loop: Increases the dimensionality of each view and introduces three-level hierarchical parallelism using TeamThreadRange (shown in Figure 1).

We also implemented the nstream, stencil, and transpose kernels from the Parallel Research Kernels (or PRK) repository [24]; the bytes_and_flops, gups, and gather benchmarks from the official Kokkos repository; and BabelStream [15]. Finally, we ported ExaMiniMD [4], a ~3k lines of code molecular dynamics application, entirely to Python (and PyKokkos). We excluded code from the original implementation (which is written entirely in C++) that was not executed by the inputs provided in the repository. For all PyKokkos code, we used the ClassStyWithMain style. All kernel execution times were collected with the Simple Kernel Timer from the kokkos-tools repository [3].
5.3 Performance: Small Applications

In this Section, we compare the performance of PyKokkos to Kokkos for smaller applications where the running time is dominated by kernel execution. All values shown (e.g., execution time) represent the mean of three runs. Additionally, each application runs the kernel 1,000 times. All CUDA execution times are using CUDA device memory (i.e., CudaSpace).

Table 2 shows execution time for all applications. The first column shows the name of the application. The second column shows the size of the largest $\text{View}$ used in our experiments. For the tutorial exercises, this $\text{View}$ is $A$. The rest of the table shows execution time of the main kernel and total execution time of PyKokkos and Kokkos using both OpenMP and CUDA backends. The Ratio columns show PyKokkos kernel execution time relative to Kokkos.

The results show that PyKokkos can achieve performance parity with Kokkos for these applications. By comparing kernel execution time for both PyKokkos and Kokkos across both backends, it can be seen that kernel code generated by PyKokkos can match the corresponding Kokkos version for performance. Any slight difference can likely be attributed to the overhead caused by running the Python interpreter concurrently with the kernels. For the CUDA backend, this effect is less pronounced since GPU execution is not as affected by the Python interpreter.

To measure the overhead introduced by PyKokkos, we compare the total running time to kernel execution time. It can be seen that for these applications, the overhead introduced by the PyKokkos Runtime and Python itself is minimal. (The stencil application total time is much longer than kernel time for both PyKokkos and Kokkos since it calls a different kernel to increment the input $\text{View}$ each iteration.) Additionally, the overhead introduced by the Python interpreter on total execution time is minimal, as these applications spend very little time in non-PyKokkos Python code.

In summary, PyKokkos can match Kokkos for smaller applications dominated by kernel execution time. We expect this solid performance for all applications where kernel execution time dominates the time spent inside the Python interpreter.

5.4 Performance: ExaMiniMD

In this Section, we compare the performance of PyKokkos to Kokkos for ExaMiniMD. ExaMiniMD first reads an input file and initializes the position, velocity, and force $\text{Views}$ in a sequential for loop. The size of these $\text{Views}$ is $\#\text{atoms} \times 3$. It then executes another sequential for loop for 100 time steps, updating the position, velocity, and force $\text{Views}$ and calculating the temperature, potential energy, and kinetic energy values by calling parallel kernels.

In our initial PyKokkos implementation of ExaMiniMD we observed relatively large execution times, around 18s using OpenMP for the largest size (x-axis) shown in Figure 5. We profiled our implementation and discovered that the total execution time was dominated by the sequential for loop that initializes the $\text{Views}$, not the kernels written in PyKokkos. Since Python is an interpreted language, sequential loops with large iteration counts (e.g., $\#\text{atoms}$ in ExaMiniMD) have significantly more overhead than in C++. We rewrote the initialization loop using Numba [25], a JIT compiler that translates Python to LLVM, to optimize the for loop. This resulted in performance comparable to the C++ for loop.

Figure 5 shows a plot of the total execution time vs. number of atoms. We used Unified Memory for all CUDA runs. For both OpenMP and CUDA, we observe performance comparable to Kokkos. The extra performance overhead in the PyKokkos implementation does not substantially increase as the size increases.

To understand this overhead, we first look at the kernel execution times shown in Figure 6. For all PyKokkos kernels, we observe minimal to no overhead compared to Kokkos. This is in agreement with the results observed for the kernels in Table 2.

Table 3 shows performance metrics collected during execution: loop time is the amount of time spent in the main loop (that runs for 100 time steps), total time is end-to-end execution time, and atomsteps per second is the number of atoms multiplied by time steps per second. In addition to kernel execution time, these metrics include time spent during Python execution. Here, we observe larger performance differences between PyKokkos and Kokkos than in the kernels themselves. Thus the additional overhead observed in the loop time and total time can be attributed to time spent in the Python interpreter, outside of the generated kernels.

5.5 Pure Python Execution

In this Section, we report the cost of pure Python execution in PyKokkos (Section 4.2.4). Since all kernels are executed using Python sequential loops, we expect substantial performance overhead. We use the tutorial exercises to highlight the cost of each feature individually. Table 4 shows a comparison of total execution time using different PyKokkos backends. We set the timeout to 300s and show the largest size that completes within this budget. Clearly, this mode should be used only for debugging logical errors, as it
5.6 Code Characteristics

Table 5 shows basic code characteristics of the applications used in our experiments. The first column shows the source of the applications. We do not use the benchmarks since they include additional boilerplate for initialization or BabelStream since it includes code for other frameworks. The second and third columns show the lines of code (LOC) and number of characters (NOC) for Kokkos and PyKokkos, respectively. For the Tutorials and PRK rows, we show a single entry that is the summation of the values for each individual application. The fourth column shows the reduction in code size of the PyKokkos implementation compared to Kokkos.

Table 5: Code Characteristics of PyKokkos and Kokkos Applications. Numbers for Tutorials and PRK Show Total for all Applications in those Groups.

<table>
<thead>
<tr>
<th>Application</th>
<th>LOC</th>
<th>NOC</th>
<th>LOC</th>
<th>NOC</th>
<th>LOC</th>
<th>NOC</th>
<th>Reduction [%]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tutorials</td>
<td>503</td>
<td>15758</td>
<td>592</td>
<td>18627</td>
<td>15</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>PRK</td>
<td>290</td>
<td>10004</td>
<td>385</td>
<td>11379</td>
<td>24</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>ExaMiniMD</td>
<td>2846</td>
<td>94811</td>
<td>3269</td>
<td>113210</td>
<td>12</td>
<td>16</td>
<td></td>
</tr>
</tbody>
</table>

Table 6: Comparison of Execution Time of PyKokkos and Numba Applications with OpenMP and CUDA.

<table>
<thead>
<tr>
<th>Application</th>
<th>Size</th>
<th>OpenMP Time [s]</th>
<th>CUDA Time [s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>PyKokkos</td>
<td></td>
<td>Numba</td>
<td>PyKokkos</td>
</tr>
<tr>
<td>nstream</td>
<td>$2^8 \times 1$</td>
<td>289.0</td>
<td>290.2</td>
</tr>
<tr>
<td>stencil</td>
<td>$2^{13} \times 2^{13}$</td>
<td>102.5</td>
<td>106.1</td>
</tr>
<tr>
<td>transpose</td>
<td>$2^{13} \times 2^{13}$</td>
<td>96.8</td>
<td>103.1</td>
</tr>
</tbody>
</table>

Table 5 shows that PyKokkos code is more concise than Kokkos. We identify several reasons. First, Kokkos applications have to add code to initialize and finalize the Kokkos context. In PyKokkos, this is hidden from the user. Second, C++ naturally tends to be more verbose than Python. Static typing in particular contributes significantly to code clutter, even more so when templates and nested namespaces are involved. Some Kokkos applications include typedef and using declarations to avoid repeating long types, but even that still adds to the clutter. In contrast, type annotations are optional in Python (outside of PyKokkos annotated code), and dynamic typing subsumes the need for templates. Third, in C++, header files need to be included for string manipulation, IO, and other functionality, most of which is available in Python without any imports. Parsing command line arguments in C++ needs to be done through string comparison and large contiguous blocks of if statements, while in Python, this can be done with the argparse module from the Standard Library.

5.7 Numba Comparison

In this Section, we compare PyKokkos to Numba. Specifically, we are interested in examining the effort required to write kernels targeting CPUs and GPUs in each framework. Of all of our test subjects, only the PRK applications have existing Numba implementations. However, the kernels do not make use of the parallelism features in Numba, so we modified them by setting parallel=True and using prange. We also made further changes to get performance closer to the PyKokkos implementation, but we note once again that our goal is not to provide a complete performance comparison between the two, and that both implementations could be optimized further. For stencil and transpose, we manually implemented tiling in the Numba kernels to get better performance. This was not
needed in the PyKokkos implementations due to the availability of MDRangePolicy, which provides a multi-dimensional iteration space with tiling.

We also implemented the kernels using CUDA through Numba. This required us to use syntax specific to CUDA and to manually set the number of threads and blocks at each kernel launch.

Table 6 shows a comparison of total execution times. For all kernels, we observe similar execution times. All PyKokkos kernels use one common code for both OpenMP and CUDA, while for Numba, we had to re-implement the kernels for each device and add loop tiling for the CPU kernel. PyKokkos kernels are therefore more performance portable.

6 LIMITATIONS AND FUTURE WORK

PyKokkos currently supports a subset of the Kokkos API so additional work is needed to add other Kokkos features, such as scratch memory, scatter views, etc. So far, we have focused on the most commonly used features. In the future, we plan to add higher level abstractions (i.e., extended API in Python) that allow for the same level of performance while being more familiar to Python programmers. We also plan on adding support for Kokkos Kernels, a library containing Kokkos implementations of commonly used linear algebra and graph kernels [34].

We selected Kokkos instead of similar libraries, such as RAJA, due to Kokkos being older and more established in the community. Additionally, the availability of bindings for views creation in Kokkos was a plus. However, it would also be possible to develop an abstraction layer over both libraries to allow for translation to target both Kokkos and RAJA.

Current support for debugging PyKokkos applications is limited to execution in Python. This approach is helpful for finding logic-based bugs but not concurrency bugs. In the future, we plan to add support for running PyKokkos with a debugger by adding line number information to the generated C++ code. Optimizing pure Python execution would also improve debugging experience.

7 DISCUSSION

Dynamic compilation. One additional benefit of PyKokkos over Kokkos is that the translation to Kokkos can happen dynamically during the execution of a program. This, for example, enables a user to build a kernel during the execution of a program and execute it in the appropriate execution space. So far, we have focused on migrating existing kernels to PyKokkos. However, it would be interesting to see how we can benefit further from dynamic compilation, and if such a style would lead to a novel way for writing kernels.

Existing kernels. In our examples, we (manually) migrated existing kernels to PyKokkos. As stated earlier in the paper, using existing (manually-written) Python bindings one can invoke existing kernels written in C++. Thus, our migration from C++ to Python was performed only with the goal to evaluate PyKokkos styles and performance. We envision PyKokkos being used for writing new kernels, and existing kernels being invoked via bindings.

8 RELATED WORK

There has been a significant effort to improve high performance Python. Numba [25] compiles a subset of the language to LLVM IR and provides support for parallelism. Cython [8] extends Python with C types and translates code to C; at this point Cython supports only OpenMP for several parallel constructs. Shed Skin [37] compiles pure Python 2 programs to C++ but only supports a restricted subset of Python. Unlike prior work, PyKokkos enables performance portability across HPC frameworks by targeting the C++ Kokkos library and supports the latest version of Python. Pygion [38] enable distributed task-based programming in Python. PyKokkos focuses on shared-memory parallelism instead.

There has been previous work on higher level abstractions to facilitate programmability and portability. PyTorch [30] and TensorFlow [6] are high performance libraries that provide abstractions for tensor computing and machine learning. Halide [33] is a domain specific language (DSL) embedded in C++ for writing portable, high performance image processing code. DiffTaichi [23] is a high-performance framework embedded in Python for building differentiable physical simulators. IrGL [29] is an intermediate representation for parallel graph algorithms that is compiled to CUDA. PyKokkos closely follows the Kokkos model for performance portability without necessarily specializing in a specific application domain.

Java has seen an increase in popularity for GPU computing [16]. Lime [17] and HJ-OpenCL [22] are Java-based DSLs that can access GPUs while providing limited support for various Java features. Lime is a Java-compatible object-oriented language capable of generating GPU code for OpenCL or CUDA. HJ-OpenCL generates OpenCL kernels from the Habanero-Java language, and further work [19] adds support for dynamic object allocation. Rootbeer [31] translates Java code that implements a specific kernel interface to CUDA workloads. Jacc [12] is another framework that translates native annotated Java code, but takes a different approach by directly generating Nvidia PTX rather than OpenCL or CUDA. GVM [10] is a Java interpreter that runs entirely on GPUs. TornadoVM [11] is a Java framework for high-performance heterogeneous programming. PyKokkos is embedded in Python rather than Java, and is not limited to GPU execution since it targets Kokkos instead of device specific frameworks.

A recent approach to transcompiling is unsupervised translation by training on monolingual source code [36]. PyKokkos takes a more traditional approach to translation that does not include machine learning. Combining the two approaches is worth exploring.

9 CONCLUSION

We presented PyKokkos, a new Python framework for writing performance portable applications entirely in Python. PyKokkos provides Kokkos-like abstractions that are easier to use and more concise than the C++ interface. We implemented PyKokkos by building a translator from PyKokkos annotated code to C++ Kokkos and bridging necessary function calls via automatically generated Python bindings. Our results showed that PyKokkos can obtain performance close to Kokkos for applications that dominated by kernel execution time. PyKokkos applications are more concise than their Kokkos counterparts, and can achieve comparable performance in most cases. Kokkos provides a performance portability programming ecosystem, and we believe that PyKokkos enables developers to utilize such an ecosystem.
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, Christian Trott, and the anonymous reviewers for their feedback on this work. 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

Nader Al Awar, Steven Zhu, George Biros, and Miloš Gligorijevic