Dynamic Generation of Python Bindings for HPC Kernels

Steven Zhu, Nader Al Awar, Mattan Erez, and Milos Gligoric
The University of Texas at Austin
{stevenzhu,nader.alawar,mattan.erez,gligoric}@utexas.edu

***Abstract*—Traditionally, high performance kernels (HPKs) have been written in statically typed languages, such as C/C++ and Fortran. A recent trend among scientists—prototyping applications in dynamic languages such as Python—created a gap between the applications and existing HPKs. Thus, scientists have to either reimplement necessary kernels or manually create a connection layer to leverage existing kernels. Either option requires substantial development effort and slows down progress in science. We present a technique, dubbed WAYOUT, which automatically generates the entire connection layer for HPKs invoked from Python and written in C/C++. WAYOUT performs a hybrid analysis: it statically analyzes header files to generate Python wrapper classes and functions, and dynamically generates bindings for those kernels. By leveraging the type information available at run-time, it generates only the necessary bindings. We evaluate WAYOUT by rewriting dozens of existing examples from C/C++ to Python and leveraging HPKs enabled by WAYOUT. Our experiments show the feasibility of our technique, as well as negligible performance overhead on HPKs performance.**

***Index Terms*—bindings, high performance kernels, dynamic program analysis, Python**

I. INTRODUCTION

Traditionally, high-performance computing (HPC) applications are written in statically typed (and low-level) programming languages, such as C/C++ and Fortran [1]–[3]. These languages are the de facto standard in the HPC area due to the excellent performance of the resulting applications.

HPC applications spend most of their execution time in so-called *high-performance kernels* (HPKs), such as linear algebra operations and solvers [4]. Over the last several years, the number of HPKs has been steadily growing and existing HPKs are constantly optimized and updated to support new hardware platforms.

Recently, several frameworks were introduced to enable developers to write performance portable HPKs. Namely, a developer can write an HPK only once and the framework automatically enables the execution of that HPK on a variety of hardware platforms (e.g., Intel CPUs, Nvidia GPUs, and AMD GPUs). Some of the most notable frameworks that support performance portability include Kokkos [5], [6] and RAJA [7]. These frameworks enable the rapid development of new HPKs, although they are still based on C/C++.

Meanwhile, scientists are transitioning to dynamically typed languages, such as Python [8], Julia [9], or Lua, for writing their applications. In order to obtain good performance, scientists have to either: (a) implement HPKs in their language of choice (using high-performance libraries like Numba [10] or PyKokkos [11]), or (b) create *bindings* to existing HPKs implemented in C/C++ or one of the frameworks that supports performance portability (using libraries like pybind11 [12]). In either case, substantial work is required [13], [14]. Maintenance of manually written bindings (as HPKs evolve) introduces additional challenges.

We present WAYOUT, a novel approach to automatically generating connection layers for existing (performance portable) HPKs to be used by Python applications. WAYOUT is the first approach that combines static and dynamic program analysis. Specifically, for a given header file, WAYOUT performs static analysis to create: (1) *wrapper classes and functions*, i.e., a Python API provided to scientists that reflect the given header file, and (2) *header files* with templated bindings that will be instantiated at run-time. When a Python application is executed and one of the wrapper functions is invoked, WAYOUT intercepts the call, instantiates and generates the *bindings* for the given types, and invokes an existing HPK. One of the key insights behind WAYOUT is that it postpones binding generation until it has the types needed (which are not available statically in Python). WAYOUT also caches generated bindings, so only the very first invocation of each function (with one set of type arguments) introduces some overhead; the cache is saved across application runs.

We designed WAYOUT to overcome the limitations of cppyy [15] and pyximport [16], which target the same task, but take very different approaches. Unfortunately, neither of the two mentioned approaches could be used to invoke existing HPKs from within a Python application. Cppyy depends on a powerful but immature tool chain, including PyPy [17], an alternative implementation of the Python interpreter, and Cling [18], an interactive C++ interpreter. On the other hand, pyximport does not support dynamic instantiation of templates and thus is unable to instantiate bindings if types are known only at run-time.

We overcome a set of critical challenges to realize WAYOUT, including: (1) the lack of function and method overloading in Python; (2) concurrent use of multiple template instantiations of the same class; (3) inferring types of returned objects; and (4) ambiguously typed template arguments.

We evaluate WAYOUT by automatically generating bindings for Kokkos Kernels [4], one of the most popular frameworks for HPKs, and Thrust [19], a powerful template library containing parallel algorithms. We rewrote a number of existing examples (that use Kokkos Kernels and Thrust) from C/C++

to Python. Our experiments show the feasibility of our technique, as well as its negligible performance overhead on HPK performance. In our experiments, we also show that WAYOUT does not impact the performance portability of HPKs: we were able to execute all the examples on both CPUs and GPUs.

This paper makes the following key contributions:

- Design of WAYOUT, a novel approach for automatically generating a connection layer for existing HPKs to be used in Python applications. WAYOUT uses a hybrid approach—a combination of static and dynamic program analysis—to instantiate the connection layer.
- Implementation of WAYOUT for Python. The design of WAYOUT is modular and others could use our processing of header files to support connection layer with other programming languages, e.g., Lua. Source code of WAYOUT is available at https://github.com/EngineeringSoftware/wayout.
- Evaluation of WAYOUT by rewriting a number of existing examples from C/C++ to Python and using existing HPKs from Kokkos Kernels and Thrust. We chose Kokkos because it is a popular performance portability framework and it currently has only a few manually written bindings; we chose Thrust to demonstrate the generality of our approach.

II. MOTIVATION

In this section, we provide some background on HPKs and binding generation, as well as motivation for WAYOUT.

A. HPKs

The usage of hand-optimized HPKs in scientific computing is extremely common. Typically, these kernels are written using high performance C/C++ frameworks that can exploit parallelism on multi-core processors, such as OpenMP [20] for CPUs and CUDA [21] for GPUs. More recently, frameworks such as Kokkos [5], [6] and RAJA [7] build abstractions on top of these device-specific frameworks to enable performance portability, i.e., code that is portable across devices while still achieving good performance. As such, these frameworks are a natural choice for writing high-performance kernels. Kokkos, for example, is used by numerous applications and packages for large-scale scientific computing, such as Trilinos [22], LAMMPS [23], Albany [24], Empire [25], and others.

Kokkos Kernels [4] is a collection of performance portable kernels written in Kokkos. It includes a large variety of math kernels and data structures commonly used in linear algebra and graph algorithms. One such example of a linear algebra kernel is the *sparse matrix vector multiply kernel*, or SpMV for short. The following code snippet shows how SpMV can be called in a Kokkos (C++) application, where `A` is a sparse matrix, `alpha` and `beta` are scalars, and `x` and `y` are vectors.

```
KokkosSparse::spmv("N", alpha, A, x, beta, y);
```

B. Binding Generation

The target audience for these HPKs is largely composed of scientists [4], [22] who need them for simulations and experiments. However, these scientists typically do not have formal training in programming, so using C++, which is notorious for its poor error messages and complicated build systems, can be a huge deterrent. Instead, they prefer higher level languages with "batteries included" [8], such as Python.

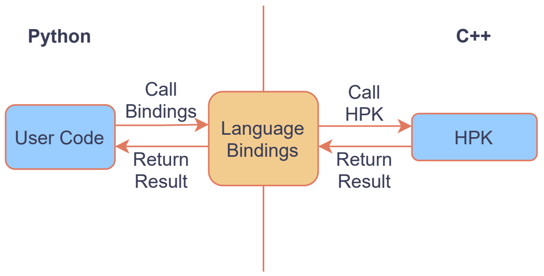


Fig. 1: An illustration of language bindings.

Several attempts have been made to expose these libraries and kernels to other languages [11], [26], [27]. This requires the use of *language bindings*, which allow for interoperability between different languages. Figure 1 shows a high-level illustration of language bindings between Python and C++. Numerous frameworks have been implemented to provide Python bindings to C++ code, such as Boost.Python [28], pybind11 [12], and SWIG [29]. The following code snippet shows what a call to SpMV *could look* like once it has been exposed to Python through one of the binding frameworks.

```
spmv(char_ptr("N"), alpha, A, x, beta, y)
```

However, manually writing these bindings can be tedious and challenging. For example, the Python bindings for creating a Kokkos View [13], the main multi-dimensional data structure in Kokkos, are written in pybind11. Despite only binding a small part of Kokkos, the total lines of code for these bindings is over 900, as they make heavy use of C++ macros and compile-time *template instantiation* to generate all the different combinations of template arguments. For Kokkos Views, this includes different data types (`int16_t`, `int32_t`, `double`, etc.), dimensions (one through eight), memory layouts, memory spaces, and memory traits. Each combination of these arguments forms a single template instantiation. The following code snippet shows one such instantiation.

```
Kokkos::View<int*, LayoutLeft, HostSpace>;
```

Besides being hard to write, compiling the bindings takes a large amount of time (around 6 hours on our machines for a commonly used subset of all combinations) due to the large number of template instantiations that need to happen. In addition to the time overhead, compilation occasionally runs out of memory due to the large number of template argument combinations, meaning that the process will not terminate successfully on some machines.

Prior work on automatic generation of Python bindings for C++ code [30], [31] extracts library APIs by parsing header files for class and function declarations. While this simplifies writing the bindings, it requires that the user manually adds code to instantiate all the needed template arguments since these frameworks employ static analysis. Also, this does not solve the compilation issues for large numbers of template instantiations. Therefore, such an approach does not work well for templated libraries such as Kokkos Kernels and Thrust.

As a result, we propose generating these bindings *dynamically*, i.e., on demand at run-time such that only the necessary template instantiations are created. This allows types to be passed at run-time, removing the need for the user to manually add template instantiation. It also reduces the cost of compilation by compiling bindings only when needed. We show that this approach can achieve performance comparable to manually written bindings.

III. WayOut Overview

In this section, we show an example of high performance kernel (HPK) from Kokkos Kernels, and then use this HPK to demonstrate the workflow of WayOut.

We encountered multiple challenges during the design and implementation of WayOut. We highlight these challenges ☆ *like so*, and then outline our design choices and how we solved these challenges.

A. Example

Figure 2 shows the function signature of the SpMV HPK `spmv` (line 4) and the class declaration of `CrsMatrix`, the sparse matrix data structure it operates on (line 13). This kernel performs the operation $y = beta * y + alpha * A * x$.

The template parameters of `spmv` are used to set the types used in the kernel at compile-time: `AlphaType`, `BetaType` are the scalar types, `XVector` and `YVector` are the vector types, and `AMatrix` is the sparse matrix type, which can be set to `CrsMatrix` in this example. The template arguments for `CrsMatrix` are as follows: `ScalarType` is the type of entries contained in the matrix, `OrdinalType` is the type of the matrix index, `Device` specifies on which device's memory (e.g., GPU) the matrix is located in, `MemoryTraits` specifies the Kokkos memory access trait to be used (Atomic, RandomAccess, etc.), and `SizeType` specifies the type of the row offset.

Figure 3 shows an example using `spmv` and `CrsMatrix`. To call the kernel, the user first defines `mat_t` to alias the instantiated `CrsMatrix` type (line 3) and instantiates the matrix and vectors (lines 7-10). The `CrsMatrix` constructor takes in as arguments the number of rows, columns, and elements, followed by Views containing the matrix entries, row offsets, and column indices. Views `y` and `x` represent the one-dimensional vectors, and their constructor specifies the size of the View. The View constructor is templated on the datatype and dimensionality (one-dimensional `double` in this case). Finally, the user can call the `spmv` kernel (as shown on line 12). The arguments passed to the call are a string specifying the operation mode (no transpose, transpose, or conjugate transpose), the scalar $alpha$, the matrix A, the vector x, the scalar $beta$, and the vector y. The latter is passed by reference and will hold the result of the operation upon return from the function.

```
1  template <class AlphaType, class AMatrix,
2      class XVector, class BetaType,
3      class YVector>
4  void spmv(const char mode[], const AlphaType &alpha,
5            const AMatrix &A, const XVector &x,
6            const BetaType &beta, const YVector &y);
7    /* ... */
8
9  template <class ScalarType, class OrdinalType,
10   class Device, class MemoryTraits = void,
11   class SizeType = typename Kokkos::ViewTraits
12     <OrdinalType *, Device, void, void>::size_type>
13 class CrsMatrix;
14   /* ... */
```

Fig. 2: An example of a kernel and data structure declaration from Kokkos Kernels.

```
1  int main() {
2    /* ... */
3    using mat_t = KokkosSparse::CrsMatrix<
4      double, int,
5      Kokkos::DefaultExecutionSpace, void, int>;
6
7    mat_t A = mat_t(numRows, numCols, nnz, val, ptr, in);
8
9    View<double *> y(N);
10   View<double *> x(N);
11
12   KokkosSparse::spmv("N", alpha, A, x, beta, y);
13 }
```

Fig. 3: An example using a kernel and data structure from Kokkos Kernels.

B. Workflow

Figure 4 shows a high-level overview of WayOut; in this section we highlight the user workflow. There are two main steps to WayOut's workflow. First, the user provides the path to the header files or the include directory. WayOut then generates a Python API consisting of wrappers for the C++ API, which was declared in the passed header files. The user can then access the C++ API using the Python API exposed by the generated wrappers.

1) Header Files: The first step in using WayOut is passing in the header files containing the required class and function declarations that together constitute the API (step ① in Figure 4).

WayOut can then generate Python wrappers that mirror the C++ API (`kernel.py` in Figure 4).

2) Python Wrappers: Once the Python wrappers have been generated, they can be imported (step ②) and called (step ③) by the user. Calling a wrapper for the first time will generate the templated bindings which will then be compiled into a shared library ④. Figure 5 shows the SpMV example using the generated wrappers. Similar to the C++ version, we first alias the matrix type (line 3), and then define the matrix and vectors (lines 5-9). We call the `CrsMatrix` class method `nnz`, which returns the number of entries in the matrix, to demonstrate how a class method can be called (line 6). Finally, we call the `spmv` kernel (line 10).

WayOut generates wrappers for both function and class declarations, as well as wrappers for public fields and methods,

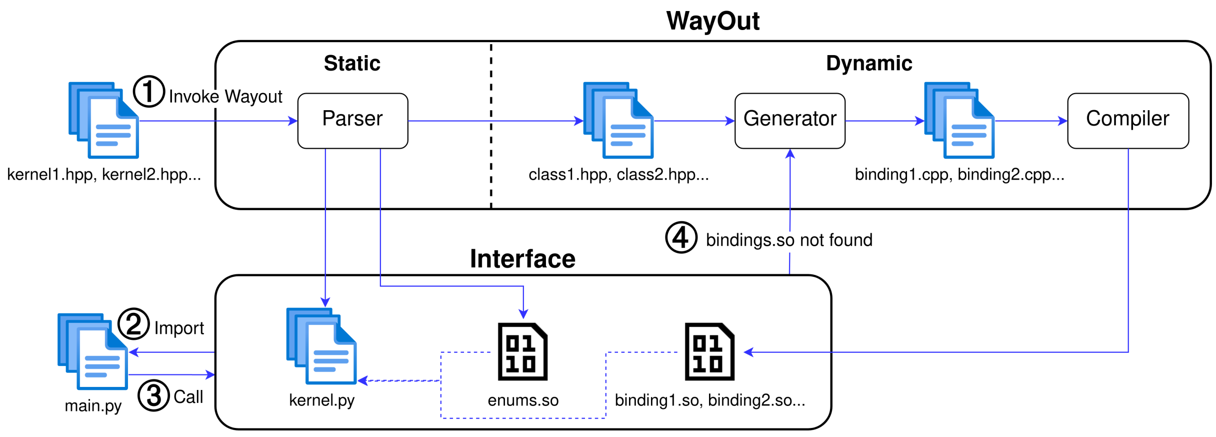


Fig. 4: An overview of WAYOUT's workflow.

```
1  if __name__ == "__main__":
2      # assume constructor arguments are initialized
3      mat_t = CrsMatrix(
4          float, int, "Kokkos::DefaultExecutionSpace", None, int)
5      A = mat_t(numRows, numCols, nnz, val, ptr, ind)
6      print("num_elem:", A.nnz())
7
8      y = View("double *")(N)
9      x = View("double *")(N)
10     spmv(char_ptr("N"), alpha, A, x, beta, b)
```

Fig. 5: Python WAYOUT example using `spmv` and `CrsMatrix`.

```
1  if __name__ == "__main__":
2      # assume constructor arguments are initialized
3      crsmat_t = CrsMatrix(
4          float, int, "Kokkos::DefaultExecutionSpace", None, int)
5      A = generate_structured_matrix2D(
6          "FD", structure, template_args=[crsmat_t])
```

Fig. 6: Python WAYOUT example using `template_args`.

in the original C++ API.

Functions. One Python wrapper function is generated for each C++ function. An issue that arises here is overloaded functions. ☆ *Python does not allow overloaded functions* i.e., redefining a function with a different number and different types of arguments. To account for this, WAYOUT instead generates a single wrapper function with a variable number of arguments for each unique function name. At run-time, if an overloaded function is used, the correct instance will be called based on the number and types of the arguments passed by the user.

Users can call templated functions normally because the template arguments can be deduced from the argument types at run-time in most cases. ☆ *In some cases, these types cannot be deduced*, and so have to be explicitly specified by the user. For example, Figure 6 shows a code snippet taken from a Kokkos Kernels tutorial using `CrsMatrix`. Instead of calling the constructor directly, it calls the `generate_structured_matrix2D` kernel to initialize the matrix. In C++, this kernel is templated on the type of the matrix to be initialized. The two arguments are for stencil type and matrix structure. These arguments do not hint at what the type of the generated matrix should be, so the users must pass the template argument to WAYOUT; otherwise, C++ compilation fails. These arguments can be passed in via the keyword argument `template_args`.

When the user calls a function, the arguments are passed to the underlying kernels by reference. However, there are cases where a kernel expects an argument as a pointer. To support this, WAYOUT provides a simple class named `ptr` which the user can use to wrap their object and indicate that the argument should be passed as a pointer. A similar issue occurs with string arguments: some functions require the standard C++ `string` whereas others accept character pointers. To support this, Python strings are cast to standard strings by default, and arguments that are character pointers use the `char_ptr` wrapper class. Line 10 in Figure 5 shows an example of this.

If a function returns a pointer, the default behavior is to treat it as a reference, i.e., assume that C++ retains ownership of the object. This means that when the resource is freed, Python would not attempt to garbage collect the object and assume the C++ run-time would do so. To override this behavior, the user can set the boolean keyword argument `take_ownership` so that Python is responsible for freeing memory.

Classes. One wrapper class is generated for each C++ class. The `__init__` method (i.e., the constructor in Python) of the wrapper class is used to pass in template arguments, creating a *type object* that can also be used as a type alias (Figure 5 line 3). To create an instance of the class, the user calls the type object, passing in the constructor arguments to the `__call__` method (line 5).

Wrapper classes can accept a variable number of templates to support optional template arguments. Additionally, if the template argument is a primitive data type (i.e., `int`, `float`, etc.), the corresponding Python data type can be used. If the template argument is a class type, it can be set to a type alias or it can be passed as a string. The latter is useful for referring to `typedefs` defined in the header files. For instance, in Kokkos the `DefaultExecutionSpace` type is simply a `typedef` that changes depending on compile-time flags, but we can still use it as a template argument in Python by passing it as a string to the class constructor (line 3). This can also be used to specify pointer types (e.g., `double*`) for template arguments (line 8).

Once an object has been created, it can be used like any Python object. The wrapper class contains all class fields and methods present in the C++ version. Private fields and methods are not accessible. As WAYOUT supports inheritance, attributes from the parent class are accessible as well. Any object returned from a function call will be automatically wrapped using the correct wrapper class.

Figure 7 shows the generated Python wrappers for the SpMV example. The `spmv` wrapper is defined on line 1 and the `CrsMatrix` wrapper is defined on line 8; the contents of these wrappers are explained in the next section.

IV. TECHNIQUES

In this section, we describe our binding generation approach, including both static and dynamic phases. In the static phase (Section IV-A), WAYOUT parses C++ header files to generate Python wrappers and templated bindings. In the dynamic phase (Section IV-B), WAYOUT intercepts calls to the Python wrappers. Then, it instantiates, compiles, and imports the templated bindings based on the types known only at run-time, completing the link between Python and C++. We then describe the casting mechanisms used to move arguments from Python to C++ and vice versa (Section IV-C). Next, we describe our techniques to support inheritance (Section IV-D) and operator overloading (Section IV-E). Finally, we discuss GPU support (Section IV-F) and integration with manually written bindings (Section IV-G).

There are two highlights to our approach: first, generating Python code in the form of wrapper classes and functions allows the user to easily use and potentially modify the generated bindings; second, the lazy approach to binding instantiation and compilation reduces the otherwise high computational cost of binding and compiling everything ahead of time. Once a binding has been compiled, it is cached on the file system for later use.

A. Static Generation

We use Clang [32] to parse the header files and pybind11 [12] as the bindings library. We chose pybind11 due to its popularity, flexibility, and ease of use. Writing bindings using pybind11 involves defining a Python *module object* which is used to register classes and functions so that they can be accessed from Python.

When WAYOUT is invoked by a user, it uses the Clang Python API to parse the header files and return the root node of the corresponding Abstract Syntax Tree (AST). WAYOUT can then extract the API from header files by traversing the AST recursively to discover classes and functions. One issue with this approach is that ☆ *Python does not allow function or method overloading*, both of which are used heavily in HPKs, such as Kokkos Kernels, especially for constructors. To deal with this, WAYOUT first stores function names in a set so that only one wrapper function is generated, even if other overloaded instances exist. Inside the wrapper functions for overloaded functions, WAYOUT adds code that selects the appropriate overloaded instance at run-time based on the types of the arguments. These types are extracted from the arguments using the Python built-in function `type()`.

```
1  def spmv(*args, template_args=None, take_ownership=False):
2      mod, name=generate_func_binding(”spmv”, ”KokkosSparse”,
3          args, _includes, template_args, take_ownership)
4      args = [get_handle(arg) for arg in args]
5      res = getattr(mod, name)(*args)
6      return cast_return(res)
7
8  class CrsMatrix:
9      ”””Compressed sparse row implementation of a sparse matrix.”””
10     _namespace = ”KokkosSparse”
11     def __init__(self, *template_args, _handle=None):
12         self._handle = _handle
13         self._cpp_name = _handle._cpp_type if _handle else \
14             register_class(”CrsMatrix”, self._namespace, template_args)
15     def __call__(self, *args):
16         if self._handle:
17             if hasattr(self, ’__cpp_call__’):
18                 return self.__cpp_call__(*args)
19             raise RuntimeError(
20                     ”Error: can’t call constructor on instance!”)
21         mod,name = generate_constructor(self._cpp_name, args, _includes)

22         args = [get_handle(arg) for arg in args]
23         inst = _copy.copy(self)
24         inst._handle = getattr(mod, name)(*args)
25         return inst
26     def nnz(self, *args, take_ownership=False):
27         ”””//! The number of stored entries in the sparse matrix.”””
28         mod,name = generate_class_func_binding(self, ”nnz”, args,
29             _includes, take_ownership)
30         args = [get_handle(arg) for arg in args]
31         res = getattr(mod, name)(self._handle, *args)
32         return cast_return(res)
33     ”””/* ... */”””
```

Fig. 7: Python wrapper generated by WAYOUT for `spmv` and `CrsMatrix`.

```
#include <pybind11/pybind11.h>
#include <KokkosSparse_CrsMatrix.hpp>
template <class T>
void generate_class(pybind11::module &_mod,
    const char *name, const char *cpp_type) {
  pybind11::class_<T> _class(_mod, name);
  _class.def_property_readonly_static("_cpp_type",
    [cpp_type](const pybind11::object&) {
      return cpp_type;
    });
  _class.def_readwrite("graph", &T::graph);
  _class.def_readwrite("values", &T::values);
  /* ... */
}
```

Fig. 8: Generated C++ templated header for the `CrsMatrix`.

WAYOUT then generates Python wrappers mirroring the original C++ API. Figure 7 partially shows the generated wrappers for the `spmv` function and `CrsMatrix` class, with the latter also containing wrapper methods for its corresponding C++ class methods.

In addition to Python wrappers, WAYOUT generates one C++ header file for each class encountered during AST traversal. Figure 8 shows the header generated for the `CrsMatrix` class. The header file contains a function templated on `T`, where `T` is the type to be registered via pybind11. The function registers the type T with pybind11, as well as all the class fields. Since all instances of a templated class have the same members, the header file can be reused by different instantiations of the templated class at run-time e.g., `CrsMatrix<double,...>` or `CrsMatrix<int,...>`.

B. Dynamic Generation

At run-time, the user imports and calls the generated Python wrappers (shown in Figure 7). Internally, the wrappers call WAYOUT to instantiate the templated functions based on the types passed, generating a C++ source file that uses the templated binding header files generated in the static phase. WAYOUT then compiles the C++ source into a shared object file (or simply DSO) that can be imported and used by the wrapper. Later calls to the same wrappers will reuse the existing DSO if the types are unchanged.

1) Wrapper: In Figure 7, the `spmv` wrapper calls the WAYOUT function `generate_function_binding` (line 2) to generate the function binding. This call captures information such as function name (`"spmv"`) and namespace (`"KokkosSparse"`) which are needed to uniquely identify the C++ function that needs to be bound. This is needed in combination with the arguments and optionally the template arguments to generate a hash that uniquely identifies the binding instantiation. Similarly, the methods of `CrsMatrix` call WAYOUT to generate instantiated bindings. The `generate` functions check to see if a module matching the hash has been imported. If so, it simply returns the module object containing the function. If the module has not been imported, WAYOUT attempts to import it from the file system. If the corresponding DSO does not exist, then WAYOUT generates the binding instantiation source code for the function.

2) Binding Generation: There are two main types of bindings. One is for registering classes so pybind11 knows how to cast objects between Python and C++, while the other is for binding an instantiated templated function. For class registration, the binding source code first includes the class header (shown in Figure 8) generated during the static phase and uses it to register classes. For function bindings, WAYOUT generates intermediate C++ functions that cast arguments from Python types to the corresponding C++ types and internally call the API function.

Figure 9 shows examples for both types of bindings. During class registration, a Python module object is first created using the `PYBIND11_MODULE` (line 4). The first argument is the name of the kernel which is set to the unique hash corresponding to that instantiation. The second argument is a handle to the module object that is used to register functions for that module. Then, the class is registered in pybind11 (line 5).

WAYOUT defines an intermediate function for each method (lines 15, 26, and 35) which accepts as input an argument of type `pybind11::args` containing a list of arguments. We use `auto` as the return type of the intermediate functions and rely on the compiler to deduce it from the argument types.

Each intermediate function explicitly casts each argument to its corresponding C++ type (e.g., lines 16-17) and then calls

```
 1 /*==================================================*/
 2 /* generated binding code for registering CrsMatrix */
 3 #include "CrsMatrix.hpp"
 4 PYBIND11_MODULE(f_f8ee838d9c3174dc82a, k) {
 5   generate_class<KokkosSparse::CrsMatrix<
 6     double, int,
 7     Kokkos::DefaultExecutionSpace, void, int>>(
 8       k, "f_f8ee838d9c3174dc82a",
 9       "KokkosSparse::CrsMatrix<double,int,"
10       "Kokkos::DefaultExecutionSpace,void,int>");
11 }
12
13 /*==================================================*/
14 /* generated binding code for CrsMatrix constructor */
15 auto func(pybind11::args args) {
16   auto a0 = args[0].cast<std::string>();
17   auto a1 = args[1].cast<int>();
18   /* ... */
19   return new KokkosSparse::CrsMatrix<double, int,
20          Kokkos::DefaultExecutionSpace, void, int>
21          {a0, a1, a2, a3, a4, a5, a6};
22 }
23
24 /*==================================================*/
25 /* generated binding code for nnz method of CrsMatrix */
26 auto func(pybind11::args args) {
27   auto &a0 = args[0].cast<
28     KokkosSparse::CrsMatrix<double, int,
29       Kokkos::DefaultExecutionSpace, void, int> &>();
30   return a0.nnz();
31 }
32
33 /*==================================================*/
34 /* generated binding code for spmv */
35 auto func(pybind11::args args) {
36   auto a0 = args[0].cast<std::string>();
37   auto a1 = args[1].cast<double>();
38   /* ... */
39   return KokkosSparse::spmv(a0.c_str(), a1, a2, a3, a4, a5);

40 }
```

Fig. 9: Generated C++ binding instantiation code for the SpMV example.

the C++ API function. The first function calls the `CrsMatrix` constructor (line 19), the second function calls the `nnz` class method (line 30), and the third function calls the standalone `spmv` function (line 39).

The bindings are then compiled into object files. Intuitively, WAYOUT would then link the files containing all the instantiations into one single DSO file and import it. Whenever a new instantiation is generated and linked, WAYOUT would reload the DSO. However, this will not work because ☆ *Python does not provide support for dynamically reloading DSOs unless their reference count reaches zero and they are garbage collected.* Waiting for the garbage collector to run is unreliable and might not even happen before the application completes. Our solution is to generate a separate DSO for each template instantiation of every class and function. This has the added benefit of avoiding the extra linking overhead when new bindings are generated. It also allows WAYOUT to elegantly support overloading and templates by separating them into different modules and avoiding re-definition errors in Python, since each combination of arguments would correspond to a different module.

The generated Python wrapper can then access and call functions registered in the module using the built-in `getattr` function (Figure 7, lines 5, 24, and 31).

C. Casting

When the user calls a bound function (such as `spmv` in Figure 5, line 10), WAYOUT casts the passed arguments from types that are valid in Python to types that are valid in C++. Once control returns to the Python side, the returned binding object is also cast to the correct wrapper class. WAYOUT uses three forms of casting: explicit, implicit, and autocasting.

1) Explicit Casting: As mentioned previously, intermediate functions accept as input a list of arguments (`args`). Explicit casting refers to calling the pybind11 `cast` method on elements of `args` to convert them into types that can be used in C++, storing them in local variables (Figure 9, lines 16-17). These variables can then be passed to the C++ function call.

The type to be cast to is passed as a template argument. Since the binding instantiation is generated at run-time, these types are chosen based on the types of the passed arguments. This form of casting works fine if the argument is a primitive (e.g., `int`). However, if the argument type is one of the wrapper classes (e.g., `CrsMatrix`), an additional implicit cast may be required.

2) Implicit Casting: In heavily templated classes, it is common for objects with slightly different template instantiations to be semantically equivalent. For instance, the Kokkos View object has an execution space template argument, which can either be of type `Device` or `MemorySpace`, which are interchangeable. In the SpMV example, `spmv` can accept both `Kokkos::View<double *, HostSpace>` and `Kokkos::View<double *, Device<OpenMP, HostSpace>` for its View arguments, even if they are different types, because Kokkos internally implements implicit casting between the two.

In order for pybind11's `cast` to work on non-primitive types, WAYOUT must use the type that was obtained during class registration, as that is the type that pybind11 recognizes. Otherwise, `cast` throws an exception for an illegal cast.

In some cases, different parts of a C++ API depend on different template instantiations of the same class, even if they are semantically equivalent. ☆ *This is a challenge for* WAYOUT *since it uses pybind11 to cast objects to the exact type needed by functions, which will result in an exception if there is any difference in types.*

To solve this, WAYOUT caches information about the C++ type of a binding object by adding an extra `_cpp_type` field during class registration. This extra field is a string set to the fully qualified C++ type name. Therefore, during binding generation, WAYOUT can use this stored name to cast the argument to the appropriate type.

3) Autocasting: ☆ *When an object is returned from a function, pybind11 does not cast it to one of* WAYOUT*'s wrapper classes, so it cannot be used to access the fields and methods.* Ideally, the functions would return objects of the same type as the generated wrapper class.

WAYOUT therefore wraps these objects in the appropriate wrapper class so the class fields and methods can still be accessed normally (Figure 7, line 6). To do so, WAYOUT first checks if the returned object has the `_cpp_type` field. If not, then the returned object is a primitive and no casting is needed. Otherwise, WAYOUT initializes a wrapper object using the binding object as the handle.

Additional complications occur when the return type has not been registered with pybind11. For example, assume the user calls a function that returns a matrix type that has not been instantiated before. To solve this, we also generate *dummy* functions which return empty instances of the return type. When a module is imported, WAYOUT also calls the dummy function. If the class is not registered, a `TypeError` will be thrown by pybind11, which we catch and parse to extract the class that needs to be registered. Since this only needs to be done once when a module is imported, the overhead is minimal and guarantees that all return types are registered.

D. Inheritance

Inheritance is a commonly used feature in C++ to facilitate code reuse. While it is not used much in Kokkos Kernels, Thrust [19] extensively utilizes inheritance in its various structures. WAYOUT supports inheritance during the static phase, where the name of the parent can be extracted from the AST. Then we can naturally emulate the C++ inheritance relationship by having the Python wrapper class of a C++ child class inherit from the Python wrapper class of the parent.

E. Operator Overloading

Operator overloading in C++ is used to implement the built in operators for custom datatypes, e.g., using the [ ] operator to access elements in a data structure. WAYOUT supports operator overloading by treating them as class methods, with the caveat that the method name is mapped to the corresponding Python magic method name (e.g., `operator[]` to `__setitem__` and `__getitem__`). Since WAYOUT already uses the `__call__` magic method for invoking the constructor, we map the C++ call operator to a new `__cpp_call__` method which is invoked when a class instance is called (e.g., Figure 7 line 17). WAYOUT currently supports the C++ addition, subtraction, bracket, call, and dereference operators, although support for others is planned.

F. GPU Support

As most HPKs support heterogeneous systems, it is important for WAYOUT to support GPUs as well. Code that runs on GPUs (e.g., CUDA or HIP) typically cannot be compiled using a regular C++ compiler such as g++. Instead, it needs to be compiled with a specific compiler (e.g., NVCC for CUDA). This is easy to do in WAYOUT, as the only modification needed is to switch to the right compiler. Additionally, since one of the main targets of our work is Kokkos, the kernel interface does not change when running with a GPU, so no further modifications to WAYOUT are needed.

G. Integration of Manually Written Bindings

There are instances where it is still beneficial to use manually written bindings for convenience reasons. For instance, the Kokkos View object is a general purpose n-dimensional data structure. It overloads the parentheses operator for reading and modifying data instead of the commonly used square brackets (e.g., `int x = view(1);`). This does not work well with pybind11 since the parentheses operator returns a reference to a primitive, which pybind11 handles by passing by value to Python, meaning that modification of the contents is not possible. However, Kokkos does have Python bindings (manually written) for Views [13]. These bindings leverage a pybind11 feature that allows the Python buffer protocol [33] to be implemented for the raw data buffer contained in Views, which allows the internal data to be accessed normally from Python. Since they are implemented using pybind11, these bindings can be used seamlessly with WAYOUT.

V. EVALUATION

We evaluate WAYOUT by answering the following four research questions:

RQ1. How effective is WAYOUT at generating bindings for Kokkos Kernels and CUDA Thrust?

RQ2. What is the run-time performance overhead of the bindings generated by WAYOUT?

RQ3. How does the run-time performance of the automatically generated bindings compare to handwritten bindings?

RQ4. What is the time needed to generate the bindings?

We ran all experiments on an Ubuntu 18.04 machine with a 6-core Intel Core i7-8700 3.20GHz CPU and 64GB of RAM, and an Nvidia GeForce 1080 GPU with 8GB of memory. We used Python 3.8.5, GCC 7.5, OpenMP 4.5, and CUDA 10.2. We used Kokkos 3.1.01, and Kokkos Kernels from the "develop" branch (commit 62985984). Finally, we used Thrust 1.12.0.

All data presented are averaged over 3 runs and the Thrust subjects were run for 100 iterations.

A. Results

RQ1: How effective is WAYOUT *at generating bindings for Kokkos Kernels and CUDA Thrust?*

Using WAYOUT, we automatically generated bindings for *all* the kernels in the Kokkos Kernels framework. We verified that WAYOUT is able to run all 39 kernels present in the Kokkos Kernels wiki [34], as well as the sparse matrix container `CrsMatrix` and numerous other helper functions used for memory allocation and initialization.

We then ported existing C++ programs that use these kernels to Python. Specifically, we implemented 7 applications from the official Kokkos repository [35] in Python:

- **CGSolve**: Implements a conjugate gradient algorithm for solving systems of linear equations of the form $Ax = b$.
- **CGSolve_SpILUKprecond**: Similar to CGSolve, but uses preconditioning for faster convergence.
- **GaussSeidel**: Implements the Gauss-Seidel method for solving a system of linear equations.
- **GraphColoring**: Assigns colors to elements of a graph such that no neighboring nodes have the same color.
- **InnerProduct**: Calculates the inner product of the form $\langle y, A * x\rangle = y^T * A * x$.
- **SpGEMM**: Implements sparse matrix-matrix multiplication in two phases: symbolic followed by numeric, with a kernel for each phase.
- **SpILUK**: Implements sparse k-level incomplete LU factorization.

We also need Python bindings for Kokkos Views as they appear frequently in our test subjects and in Kokkos Kernels. In our subjects, we used both the manually written Python bindings and bindings automatically generated by WAYOUT. As mentioned before, Views use the C++ parentheses operator to modify data, meaning that they cannot be directly modified in Python using the automatically generated bindings, so we implement only four of our subjects using the latter.

To demonstrate the generality of our approach, we also generated bindings for kernels in the Thrust library. We ported 7 examples from the official Thrust repository [36] to Python: histogram, mode, saxpy, set_operations, sort, sparse, and sum.

In summary, WAYOUT successfully generated bindings to Kokkos Kernels and Thrust, which we were able to use to port workloads from C++ to Python.

RQ2: What is the run-time performance overhead of the bindings generated by WAYOUT*?*

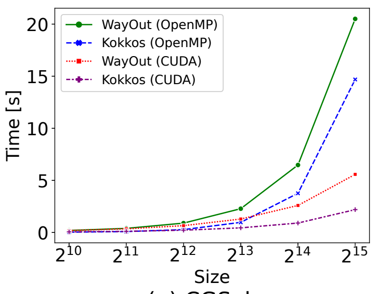
Figure 10 shows plots of computation time (y-axis) vs. input data size (x-axis) for our subjects from Kokkos Kernels and Thrust. For WAYOUT, we show computation time after the bindings have been instantiated and compiled for all types that occur in each subject. We show binding generation time in RQ4. The time shown does not include time spent to initialize the subject, as most subjects initialize arrays in sequential loops, which dominates the running time for larger input sizes. Including that time would mean comparing Python to C++ rather than measuring the overhead of the generated bindings.

For most subjects, our Python implementation can achieve performance comparable to the original C++ implementation.

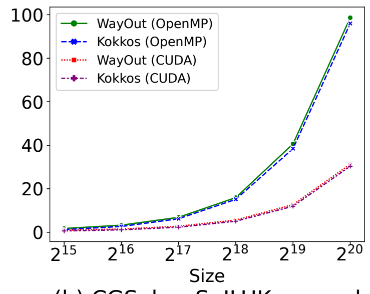
For the CGSolve subject, we observe overhead that scales with the size of the input data. This happens because the subject runs most of its computations in a loop that calls the kernel internally. It also computes a square root in Python using the `math.sqrt()` function. The number of iterations of this loop scales with the size of the input data, increasing the number of calls to `math.sqrt()`, which in turn increases the total time taken compared to the C++ implementation.

We also observe noticeable performance overhead for the set_operations subject (Figure 10k). This subject invokes various functions that each allocates a result vector and calls a different set operation (e.g., merge, union). In C++, the result vector is allocated on the stack, while in Python, the object must be allocated on the heap. Both heap allocation and Python's garbage collector introduce substantial overhead.

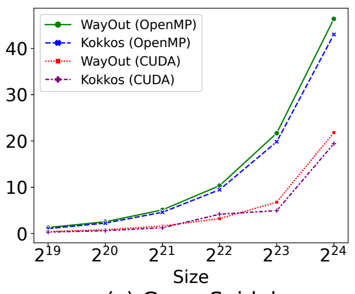
Thus, these two outliers can be attributed to Python itself rather than WAYOUT. In summary, bindings generated by WAYOUT introduce minimal performance overhead.



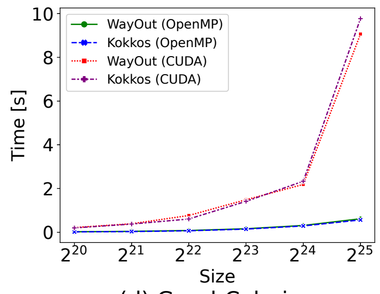
(a) CGSolve



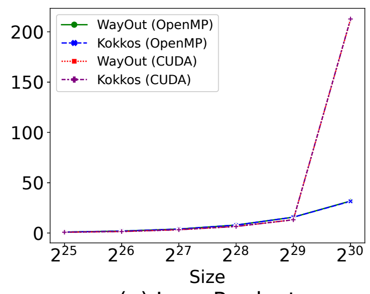
(b) CGSolve_SpILUKprecond



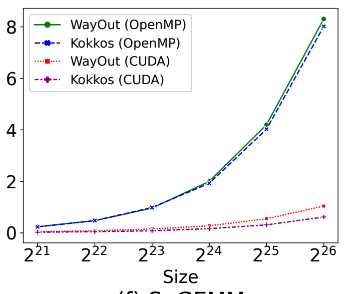
(c) GaussSeidel



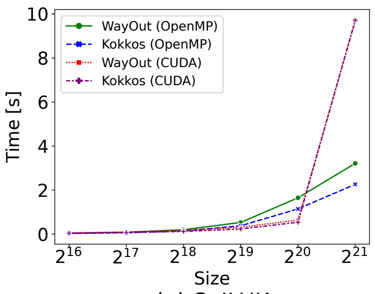
(d) GraphColoring



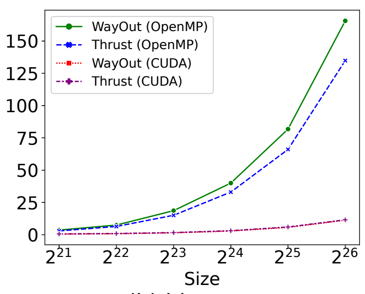
(e) InnerProduct



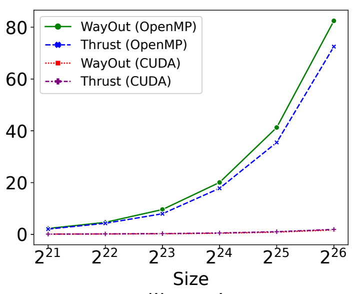
(f) SpGEMM



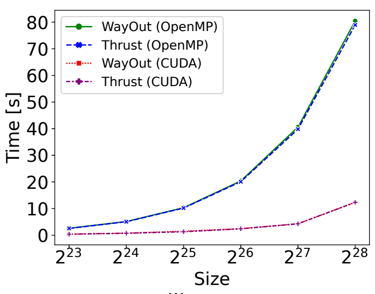
(g) SpILUK



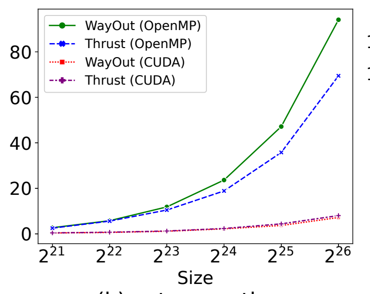
(h) histogram



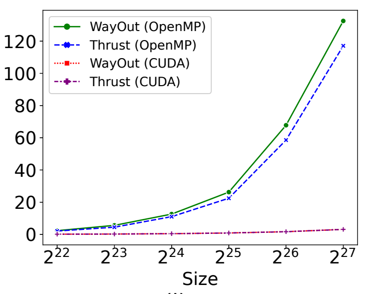
(i) mode



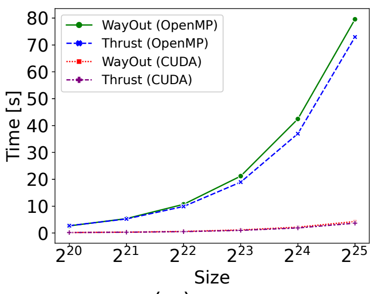
(j) saxpy



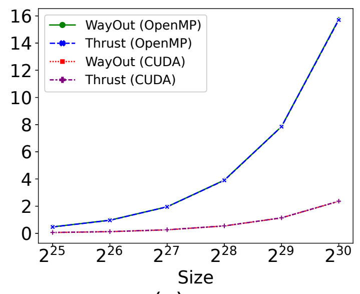
(k) set_operations



(l) sort



(m) sparse



(n) sum

Fig. 10: Kernel time using WAYOUT generated bindings vs. original Kokkos Kernels/Thrust implementation.

TABLE I: Performance of Generated versus Manually Written Bindings.

Subject	Size	OpenMP Time [s]			CUDA Time [s]		
		Manual	Generated	Ratio	Manual	Generated	Ratio
CGSolve_SpILUKprecond	2^{20}	99.14	102.49	1.03	31.46	30.51	0.97
GaussSeidel	2^{24}	43.09	43.33	1.01	18.14	19.76	1.09
InnerProduct	2^{30}	31.65	31.62	1.00	212.82	213.79	1.00
SpILUK	2^{21}	3.18	3.12	0.98	9.66	9.71	1.01

TABLE II: Bindings Build Time (Kokkos Kernels on the left and Thrust on the right).

Subject	Kernels	Modules	Static Phase [s]	Dynamic Phase (g++) [s]	Dynamic Phase (NVCC) [s]	Subject	Kernels	Modules	Static Phase [s]	Dynamic Phase (g++) [s]	Dynamic Phase (NVCC) [s]
CGSolve	7	12	3.43	32.13	82.93	histogram	13	34	4.21	98.84	281.24
CG_SpILUK	23	35	5.94	96.17	248.01	mode	10	28	4.15	81.86	230.83
GaussSeidel	8	15	5.77	43.17	111.63	saxpy	6	17	3.52	49.41	139.40
GraphColoring	11	17	5.13	51.29	130.21	set_operations	11	17	3.68	49.50	144.43
InnerProduct	2	2	3.05	7.59	26.28	sort	5	12	3.61	34.79	98.36
SpGEMM	7	12	4.37	33.23	85.83	sparse	9	33	3.81	96.60	273.38
SpILUK	18	28	5.31	76.49	196.39	sum	4	11	3.50	31.92	90.87

RQ3: How does the run-time performance of the automatically generated bindings compare to manually written bindings?

We compare the manually written Python bindings provided in the Kokkos repository for the `View` class against the bindings generated by WAYOUT. Table I shows the performance of the generated bindings versus the handwritten ones with both OpenMP and CUDA. The first column shows the name of the subject. The second column shows the size of the input data. The rest of the table shows computation time for both the manually written and automatically generated bindings, as well as the ratio of generated time to manual time.

The results show that the performance of the bindings generated by WAYOUT matches that of the manually written bindings. This is expected as both sets of bindings use pybind11, and WAYOUT only generates an additional lightweight Python wrapper which has minimal performance overhead.

RQ4: What is the time needed to generate the bindings?

Table II shows the average time taken to automatically generate the bindings for each library. The columns show the name of the subject, the number of kernels used, the number of modules generated (i.e., DSOs that instantiate the classes and functions), the time taken during the static phase, and the time taken during the dynamic phase for g++ and NVCC respectively.

The results show that WAYOUT has acceptable execution time. The largest cause of performance overhead in either phase is caused by calling the C++ compiler. The time taken during the static phase is mostly caused by compiling the enums DSO file and does not vary greatly across subjects. The time taken during the dynamic phase varies depending on the number of modules generated and the compiler used. More kernel calls with different types results in more template instantiations, and therefore more modules generated. For example, the CGSolve_SpILUKprecond subject has a large dynamic phase execution time, as it calls 23 kernels and generates 35 modules, more than any other subject.

It is important to note that the execution time shown here only occurs once, when the bindings are instantiated for the first time. Later calls of kernels with the same types, and even later runs of the same application would not incur this overhead as the modules are cached on the filesystem.

WAYOUT is also considerably faster than the approach used in the Kokkos View bindings [13], which is a purely static approach that instantiates all combinations of types during compilation. On our machine, compiling those bindings takes over 6 hours, and runs out of memory on another machine.

VI. LIMITATIONS

C++ allows passing arguments and returning values by value, pointer, or reference. Python always passes primitives by value and objects by reference. As such, the Python API generated by WAYOUT will not always exactly match the functionality of the C++ API: primitives are always passed and returned by value, and objects are always passed by reference or pointer. WAYOUT allows passing pointers with `ptr` and character pointers with `char_ptr`.

Another limitation of WAYOUT is that the generated wrappers may not be very "Pythonic". For example, while `ptr` and `char_ptr` are practical solutions to pointer arguments, such constructs will be unfamiliar to Python programmers. Additionally, the generated wrappers do not make use of certain Python features such as keyword arguments (i.e., `**kwargs`) and dynamic typing.

It would be possible to make the generated APIs more Pythonic by adding another layer of abstraction on top of the wrappers generated by WAYOUT. Currently, this would require additional effort from the user, although we plan to explore a way to automate this step in future work.

Some kernels in Thrust accept a function object as an argument in order for the user to define kernel behavior. WAYOUT does not support these kernels as this would require translating Python code to C++; an earlier work, PyKokkos [11], supports translation from Python to C++. However, since the goal of WAYOUT is to bind existing HPKs where the behavior is already defined, this is a minor limitation.

Finally, we focused primarily on Kokkos and Thrust in our evaluation. We chose Kokkos because it is a popular performance portability framework with a large number of kernels, and Thrust is a popular CUDA library.

VII. Related Work

A. Binding Frameworks

Boost.Python [28], pybind11 [12], SWIG [29], and pyximport [16] are frameworks that allow binding C or C++ code so that it can be called from Python. Typically, these frameworks require that the user specify the C++ interface to be bound using some form of domain-specific language or configuration file. WAYOUT only asks the user for the header files containing class and function declarations, and automatically generates the bindings with no extra effort from the user.

B. Static Binding Generation

CFFI [37] is a Python library that can import C code using C-like declarations and generate the necessary bindings in a C file. However, it does not support C++ and requires the user to manually declare the interface. AutoWIG [30] provides a Python API to pass in header files and then generates bindings using Boost.Python. Additionally, the user has to provide a header file that contains all the needed template instantiations for templated classes and functions. Afterwards, the user must compile the generated bindings. Similarly, Binder [31] statically parses header files to obtain all classes and functions. As with AutoWIG, the desired template instantiations must be explicitly used or specified in the header files. In contrast to AutoWIG, it is meant to be used entirely through the command-line. WAYOUT is more flexible and more Pythonic through its dynamic analysis: templates are only instantiated at run-time through types passed to automatically generated Python wrapper classes. The user does not have to specify all the types that they want to use ahead of time.

C. Dynamic Binding Generation

Cppyy [15] dynamically generates bindings to C++ libraries. It uses Cling [18], a C++ interpreter based on Clang and LLVM, to generate C++ code that instantiates and calls classes and functions included in header files, and then binds that code to enable accessing it from Python. The definitions of those classes and functions are loaded at run-time by dynamically linking a shared object library. This presents a problem for libraries such as Kokkos Kernels, which currently can only be compiled to a static library. WAYOUT provides the flexibility of linking a static library during compilation, instead of exclusively requiring shared object libraries as cppyy does. Additionally, WAYOUT's use of pybind11 to interface between Python and C++ allows the user to manually write bindings for some classes to make them more Pythonic if desired.

Furthermore, the dependence on Cling also limits supported libraries to features supported by Cling. For instance, it does not have support for thread level storage symbol relocation, which is used in the shared object for Kokkos. Another example is CUDA support. Since WAYOUT invokes a compiler to compile shared objects, it has flexibility of choosing NVCC rather than g++ as the compiler, whereas Cling support for CUDA is still experimental.

D. High Performance Python

PyKokkos [11] is a framework for writing performance portable kernels in Python. The user writes kernels in a small, statically typed subset of Python, which PyKokkos then translates to C++ (Kokkos) to obtain better performance. Numba [10] is a Python JIT compiler based on LLVM. Cython [16] adds C-like language extensions to Python to improve performance. WAYOUT is not meant for writing kernels. WAYOUT provides access to pre-existing, hand-tuned high-performance kernels.

NumPy [27] and SciPy [26] both contain data structures and kernels used in scientific computing. A significant part of both libraries is implemented in C and C++ and manually wrapped so it can be accessed from Python. WAYOUT automatically generates bindings to interoperate between Python and C++.

VIII. Conclusion

We present WAYOUT, a technique for automatically generating Python bindings for C++ code, specifically high-performance kernels. WAYOUT combines static and dynamic analysis in order to reconcile Python's dynamic nature with C++'s static typing, and is able to support heavily templated classes and functions. We implement WAYOUT by building Python and C++ code generators that produce a connection layer between the two languages. Our evaluation shows that WAYOUT can support Kokkos Kernels framework and CUDA Thrust with minimal performance overhead. Additionally, WAYOUT can generate bindings at an acceptable performance cost, making it more feasible than manually written and statically generated bindings. We believe that WAYOUT enables faster development of scientific applications by connecting Python, a high-level language frequently used by scientists, to existing HPKs written in C++.

Acknowledgment

We thank George Biros, Martin Burtscher, Ian Henriksen, Jonathan R. Madsen, Arthur Peters, Keshav Pingali, Sivasankaran Rajamanickam, Christopher J. Rossbach, 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-2107291, and the Department of Energy, National Nuclear Security Administration under Award Number DE-NA0003969.

References

[1] C. R. Trott, "ExaMiniMD," https://github.com/ECP-copa/ExaMiniMD, 2017.

[2] D. Lebrun-Grandié, A. Prokopenko, B. Turcksin, and S. R. Slattery, "ArborX: A performance portable geometric search library," *ACM Transactions on Mathematical Software*, vol. 47, no. 1, pp. 1–15, 2020.

[3] S. Slattery, "Cabana," https://github.com/ECP-copa/Cabana, 2018.

[4] S. Rajamanickam, S. Acer, L. Berger-Vergiat, V. Dang, N. Ellingwood, E. Harvey, B. Kelley, C. R. Trott, J. Wilke, and I. Yamazaki, "Kokkos kernels: Performance portable sparse/dense linear algebra and graph kernels," https://arxiv.org/abs/2103.11991, 2021.

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

[6] C. Trott, L. Berger-Vergiat, D. Poliakoff, S. Rajamanickam, D. Lebrun-Grandie, J. Madsen, M. Gligoric, N. Al Awar, G. Shipman, and G. Womeldorff, "The Kokkos ecosystem: Comprehensive performance portability for high performance computing," *Computing in Science and Engineering*.

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

[8] T. E. Oliphant, "Python for scientific computing," *Computing in Science and Engineering*, vol. 9, no. 3, pp. 10–20, 2007.

[9] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah, "Julia: A fresh approach to numerical computing," *SIAM Review*, vol. 59, no. 1, pp. 65–98, 2017.

[10] S. K. Lam, A. Pitrou, and S. Seibert, "Numba: A LLVM-based Python JIT compiler," in *Workshop on the LLVM Compiler Infrastructure in HPC*, 2015, pp. 1–6.

[11] N. Al Awar, S. Zhu, G. Biros, and M. Gligoric, "A performance portability framework for Python," in *International Conference on Supercomputing*, 2021, pp. 467–478.

[12] "Pybind11 Documentation," 2020, https://pybind11.readthedocs.io/en/stable/intro.html.

[13] J. R. Madsen, "kokkos-python," https://github.com/kokkos/kokkos-python, 2020.

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

[15] W. T. Lavrijsen and A. Dutta, "High-performance Python-C++ bindings with PyPy and Cling," in *Workshop on Python for High-Performance and Scientific Computing (PyHPC)*, 2016, p. 2735.

[26] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S. J. van der Walt, M. Brett, J. Wilson, K. J. Millman, N. Mayorov, A. R. J. Nelson, E. Jones, R. Kern, E. Larson, C. J. Carey, İ. Polat, Y. Feng, E. W. Moore, J. VanderPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E. A. Quintero, C. R. Harris, A. M. Archibald, A. H. Ribeiro, F. Pedregosa, P. van Mulbregt, and SciPy 1.0 Contributors, "SciPy 1.0:

[16] S. Behnel, R. Bradshaw, C. Citro, L. Dalcin, D. S. Seljebotn, and K. Smith, "Cython: The best of both worlds," in *Computing in Science and Engineering*, 2011, pp. 31–39.

[17] "PyPy," 2021, https://www.pypy.org/.

[18] V. Vassilev, P. Canal, A. Naumann, L. Moneta, and P. Russo, "Cling – the new interactive interpreter for ROOT 6," in *Journal of Physics: Conference Series*, 2012, pp. 52–71.

[19] N. Bell and J. Hoberock, "Chapter 26 - Thrust: A productivity-oriented library for CUDA," in *GPU Computing Gems Jade Edition*, 2012, pp. 359–371.

[20] "OpenMP," 2020, https://www.openmp.org.

[21] "CUDA Zone," 2020, https://developer.nvidia.com/cuda-zone.

[22] The Trilinos Project Team, *The Trilinos Project Website*.

[23] "LAMMPS molecular dynamics simulator," https://lammps.sandia.gov/, 2020.

[24] "Albany multiphysics code," http://snlcomputation.github.io/Albany/, 2020.

[25] M. T. Bettencourt and S. Shields, "EMPIRE: Sandia's next generation plasma tool," Sandia National Lab.(SNL-NM), Albuquerque, NM (United States), Tech. Rep., 2019.
Fundamental Algorithms for Scientific Computing in Python," *Nature Methods*, vol. 17, pp. 261–272, 2020.

[27] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Rio, M. Wiebe, P. Peterson, P. Gerard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant, "Array programming with NumPy," *Nature*, vol. 585, no. 7825, pp. 357–362, 2020.

[28] D. Abrahams and R. W. Grosse-Kunstleve, "Building hybrid systems with Boost.Python," *The C/C++ Users Journal*, vol. 21, 2003.

[29] D. Beazley, "Automated scientific software scripting with SWIG," *Future Generation Computer Systems*, vol. 19, no. 5, pp. 599–609, 2003.

[30] P. Fernique and C. Pradal, "AutoWIG: Automatic generation of Python bindings for C++ libraries," *PeerJ Computer Science*, vol. 4, 2018.

[31] RosettaCommons, "Binder," https://github.com/RosettaCommons/binder, 2016.

[32] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *International Symposium on Code Generation and Optimization*, 2004, pp. 75–86.

[33] T. Oliphant and C. Banks, "Pep 3118 – revising the buffer protocol," https://www.python.org/dev/peps/pep-3118/, 2006.

[34] S. Rajamanickam, "Kokkos kernels wiki," https://github.com/kokkos/kokkos-kernels/wiki/APIReference, 2021.

[35] C. R. Trott, "Kokkos Tutorials," https://github.com/kokkos/kokkos-tutorials, 2015.

[36] "Thrust," https://github.com/NVIDIA/thrust, 2021.

[37] "CFFI documentation," https://cffi.readthedocs.io/en/latest/, 2012.