# Migrating Legacy Fortran to Python While Retaining Fortran-Level Performance Through Transpilation and Type Hints

Mateusz Bysiek, Aleksandr Drozd, Satoshi Matsuoka

*Tokyo Institute of Technology*
*Meguro, Tokyo, 152–8550 Japan*
*bysiek.m.aa@m.titech.ac.jp, alex@smg.is.titech.ac.jp, matsu@is.titech.ac.jp*

***Abstract*—We propose a method of accelerating Python code by just-in-time compilation leveraging type hints mechanism introduced in Python 3.5. In our approach performance-critical kernels are expected to be written as if Python was a strictly typed language, however without the need to extend Python syntax. This approach can be applied to any Python application, however we focus on a special case when legacy Fortran applications are automatically translated into Python for easier maintenance. We developed a framework implementing two-way transpilation and achieved performance equivalent to that of Python manually translated to Fortran, and better than using other currently available JIT alternatives (up to 5x times faster than Numba in some experiments).**

***Index Terms*—Application migration, gradual typing, interoperability, just-in-time compilation, legacy code, software maintenance, transpilation.**

## 1. Introduction – Traits of Python and Fortran

Python is praised by many because its flexible and dynamic nature makes programming much easier. However, this nature also makes overcoming its various performance issues much harder. Especially when compared to C, C++ and Fortran, Python is computationally very slow.

On the other hand, performance is not and never was an issue in Fortran, indeed it is one of its hallmarks. Additionally, compared to more modern approaches Fortran is hard to use, but despite the difficulties, it is still in use because it has accumulated a remarkable legacy of fast code. Over the years a lot of money was invested to create huge code base that now would be expensive to port – and is increasingly expensive to use [1], [2]. How might we enable continued use and development of old but very efficient computational solutions, implemented using legacy technologies such as Fortran 77? Maybe by migrating it to easy to use Python.

However, at present, by such migration we would lose performance. How might we make high performance in Python more accessible? Specifically, how might we increase performance of performance-critical kernels written in Python, so that their performance matches the performance of equivalent kernels written in Fortran (or other HPC-enabled programming language)?

After looking at strong and weak points of both languages, we observe that Python and Fortran seem to be complementary solutions. Also, because modern Python can be typed, they might be close enough at the language level for source-to-source translation to become feasible.

Our approach to problems of both languages is source-to-source translation employed in the right way at the right time. Cumbersome legacy Fortran application code could be translated (migrated) to type-hinted Python once and permanently. Since, by Amdahl's Law [3], only the performance-critical parts of application require top performance, at the time of execution most of Python code could be interpreted normally without noticeable impact on application performance. The performance-critical kernels, however, would be JIT-translated back to Fortran.

This way, we will take advantage of the best traits of Python and Fortran to create computational solutions that are efficient, maintainable and build upon efficient legacy code. However, no known mapping exists between Python and Fortran. For the purpose of assisting application migration, and to enable JIT transpilation, we need such mapping – at least for a reasonably defined subset of both languages.

## 2. Background

### 2.1. Performance Issues in Python

**2.1.1. Everything is an object – Problems with dynamic indirection.** In many languages there is a distinction between normal types and primitive types (among them usually integers, floating point numbers, and character type) which receive special treatment from the compiler and therefore result in fast machine code. Taking this Python example:

```
i = 0
while i < 123456789:
    i += 1
```

Since in Python everything is an object, even `i = 0` does not declare an integer variable in a sense in which in C++11 `auto i = 0;` would. In Python, an `object` is created, and when `i` is incremented, Python interpreter must check how to increment value of object `i`, and execute the relevant incrementation subroutine. This consumes unnecessary amount of computing resources.

**2.1.2. Duck typing – Static type information is unavailable.** Duck-typing is the principle that follows the saying "If it looks like a duck, it walks like a duck and it quacks like a duck, then it must be a duck." that, in programming, translates to saying "If all objects from class A all have properties a and b, and if some unrelated given object has property a and b, then it must comes from class A." Needless to say, such statement is unreliable.

In general-purpose programming such rule is not a problem by itself, it can be even advantageous sometimes, but in case of computation it will cause an incredible slowdown. The reason for this is that in order to have efficient computation, code needs to be compiled to optimized machine code and static type checking is necessary for many compiler optimizations. Statically, one cannot check what properties a given instance has, because at the moment of checking nothing is yet instantiated. Therefore for every operation, the most generic and robust machine code representation must be chosen, and such representation is necessarily much slower than the version designed for, for example operation on two floating-point numbers of specified precision.

The duck typing problem is, in the context of efficient computation, a consequence of the fact that everything is an object. If there was some short list of types that were not objects, they would have reliable type information, and optimizations would be applicable.

## 2.2. Software Development Issues in Fortran

**2.2.1. Designed for numerics – Problems with generality.** There are perfectly good reasons why Fortran is numerically-oriented. However, to unlock that power of numerical calculations it is essential to have easy access to data on which those calculations are to be performed. We argue that in Fortran, such access is problematic.

Depending on the individual use case, the data might be in a database, in a binary file, in a text file or in some other media – which might be stored locally or remotely. There are many ways and formats in which data is stored in practice in modern heterogeneous world. Fortran does not provide solutions to access that data in many cases. It can be managed using an external tool, however this increases the complexity of the solution and maintenance effort.

It is important to note that the issue of generality is still present in modern Fortran.

**2.2.2. Fixed form – Problems with extensibility.** Fortran code in general has two forms: fixed and free form. Legacy Fortran code is often written in fixed form, because the new free form was only introduced in Fortran 90. Form, in this context, is in other words a set of constraints on how the source code must be formatted.

The definition of fixed form contains very strict rules about what can be in each column[1] of the source code file:

- Column 1: Blank, or a "C" or "*" for comments – however many compilers allow other characters.
- Columns 1-5: Optional statement label.
- Column 6: Continuation of previous line, optional.
- Columns 7-72: Statements.
- Columns 73-80: Sequence number, optional and rarely used today.

Therefore, line of code cannot have more than 80 characters, and only 65 of those can be used for instructions.

An example application of some of those rules would be:

```
      statement
C     comment
  100 statement with label
      sum = a + b + c + d
     & + e + f
```

The free format, on the other hand, does not impose any very strict rules on code formatting, it is in fact almost identical to the coding style seen in many modern languages such as C++ or Java. Therefore, in case of modern Fortran applications the problem of code format should not exist. Despite that, the standard practice for modern Fortran programs is to reuse existing legacy Fortran code as is – therefore although some parts of modern Fortran programs are relatively easier to build upon, depending on the case, a large fraction of code base might be in fixed form.

**2.2.3. Command-line, environment variables and problems with interoperability.** The ability to use command-line arguments[2] or environment variables[3] in Fortran programs was introduced into the language standard in year 2003. Before that, apart from extensions available in some compilers, the only way to provide data to Fortran program was through a file. In case of modern command-line argument parsing, there is a Fortran library[4] that was inspired by Python's built-in argument parsing library – in fact even its usage looks surprisingly similar to that of Python's version.

Fortran-C interoperability was standardized in the 2003 language edition, and extended in 2008. One of the important features of the upcoming Fortran 2015 specification is the improvement of C language interoperability. Fortran interoperability with other programming languages and tools is still an issue today, although not nearly at the scale at which it was in older Fortran.

## 2.3. Gradual Typing with Type Hints

After enumerating various examples of issues in Python and Fortran, let us go forward by describing how type information can be conveyed in a modern Python program.

1. Stanford University, Fortran 77 Tutorial – Basics: https://web.stanford.edu/class/me200c/tutorial_77/03_basics.html

2. Fortran Wiki, Command-line arguments: http://fortranwiki.org/fortran/show/Command-line+arguments

3. GET_ENVIRONMENT_VARIABLE – The GNU Fortran Compiler: https://gcc.gnu.org/onlinedocs/gfortran/GET_005fENVIRONMENT_005fVARIABLE.html

4. FLAP: Fortran command Line Arguments Parser for poor people: https://github.com/szaghi/FLAP

The method adopted by Python is called gradual typing. In this method, type information might be but does not have to be provided.

Python language version 3.5, published in September 2015, in a Python Enhancement Proposal (PEP) numbers 0483 and 0484, introduced so-called type hints [4], [5]. Assuming one has the following function (remark: the example code is superfluous on purpose):

```
1 def sum(a, b):
2     c = None
3     c = a + b
4     return c
```

Then, one can annotate it using type hints:

```
1 def sum(a: int, b: int) -> int:
2     c = None # type: int
3     c = a + b
4     return c
```

There are two kinds of type hints: type annotations (line no.1 contains 3 annotations), and type comments (line no.2 contains a single type comment). The annotations are present in Python grammar since version 3.0 of the language. They were originally meant as a means of function signature documentation, which did not have to contain type information and if it did its format was not standardized.

With Python 3.5 a unified approach to annotations was introduced but for backwards compatibility those type annotations, although parsed into the abstract syntax tree (AST), are completely ignored by CPython interpreter during execution. Type comments are treated like regular comments, and are discarded by the parser, thus making type hints backwards-compatible to Python 3.0. Apart from lack of support for them in built-in CPython modules, also almost all other packages take no notice. This situation has much potential for improvement.

Type hints can be used to convey the intent of the author of the code, but that intent disappears and Python neither does take advantage of it, neither it is troubled by it. In the above case, the 2nd implementation might as well be used to add floating point numbers, concatenate strings of characters, or even concatenate lists – exactly like 1st implementation. This can be considered a good thing, or a bad thing. In our opinion, this flexible nature is one of core characteristics of Python, and is inherently a good thing. The intent of the original author of the code is not forced in any way, which leaves increased possibility for code reuse.

When 1st and 2nd version of the `sum()` function is parsed using built-in modules, the resulting AST looks slightly different, because the function signature annotations are retained. Execution, however, is not affected even if in principle it could be affected. And we think that for performance reasons, in some cases it should be affected.

## 3. Related Work

### 3.1. Migrating Legacy Fortran

For converting legacy Fortran code, there are several solutions available: f2c [6] converts it to C, Fable [7] to C++, there are ways to convert it to Java bytecode [8], or partially to CUDA [9]. The issue with those solutions is that in many cases they do not attempt to generate human-readable code, and even if they do the target language cannot offer programming flexibility at the Python level, and thus connecting from high-level orchestrating code to low-level implementation might be necessary.

### 3.2. Improving Python Performance

As improving performance of Python is a very popular research topic, there is a lot of work being done in that area. We shall not attempt to give a detailed survey of all available solutions in this section. Instead, we will familiarize the reader with solutions that are most related.

**3.2.1. NumPy.** NumPy [10] is a BLAS-compliant numerical library for Python, which, when used correctly, can achieve very good performance. NumPy's approach to higher performance in Python is very straightforward. NumPy is partially implemented in Python, but it mostly consists of a big collection of low-level language implementations which are interfaced with Python through the CPython API. The implementations often contain manually unrolled loops to support SIMD compiler optimizations, or contain code that is preprocessed by the C compiler at installation time to generate the code.

Those low-level implementation are very efficient, but are not useful in cases when they would need to be tailored for some specific use. NumPy cannot be used to accelerate any given algorithm.

**3.2.2. f2py.** In the context of this work, one very notable part of NumPy is f2py [11], a Python wrapper generator for Fortran code. It scans Fortran code for modules and function signatures and creates Python interface for them. It delegates compilation of Fortran to chosen compiler available in the system, and couples together the compiled Fortran code with the Python interface for it into a single binary file that can be imported in Python using standard `import module_compiled_with_f2py` statement.

Code compiled using f2py benefits from outsourcing the compilation process because of the compiler optimizations available in many mature Fortran compilers. Unfortunately, it suffers from significant compilation overhead. Compilation process heavily involves the file system as many intermediate files need to be created and then discarded, which slows it down significantly. This overhead is especially visible in case of compiling relatively small pieces of Fortran code.

**3.2.3. Numba.** Numba [12] is a JIT compiler for Python. It does not compile Python directly, but instead transforms it into LLVM Intermediate Representation, and delegates the compilation to the LLVM toolchain. Through an easy-to-use API, Numba enables JIT compilation of selected parts of Python code. The compiled code sections have to contain only a restricted subset of Python syntax. Additionally, Numba cannot shed all layers of indirection present in Python, because it is not capable of complete type analysis.

An interesting advantage of using Numba is a relatively short compilation time, which is achieved by performing compilation completely in memory. The lack of involvement of the file system can have significant compilation time benefits in case of compiling single Python functions.

As mentioned in the introduction, there are inherent unavoidable problems with unavailability of reliable type inference in Python, and with dynamic indirection – Numba runs into those problems.

**3.2.4. Cython.** Cython [13] is a language derived from Python, and also a software solution that translates Cython language to C with certain extensions. Reason is performance, especially the case of numerical loops [14]. Cython can usually outperform NumPy in cases of construction of sparse matrices, data transformation, repacking, equation solving, among others [15].

Cython language is very similar to Python in a sense that a subset of Python is also valid Cython code. However, Cython extends Python syntax by adding few C-related constructs. This makes Cython code backwards-incompatible with Python – once the code is converted to Cython so as to benefit from its performance boost, it is no longer valid Python. For example: variable types have to be defined in a way which is not compatible with Python, and, there is a separate import system for C-related constructs[5] which is not compatible with Python.

Cython provides a so-called "pure mode" via which the original Python code can be left untouched, and a separate file with static type information for that code needs to be created instead [15]. This additional file is ignored by Python interpreter, but used by Cython framework, which provides some level of compatibility, however this lowers maintainability of the code because two files have to be kept in sync manually.

Cython framework, apart from providing application performance boost, incurs a significant compilation overhead, because Cython framework delegates the compilation of C code to an external compiler (as available in the system) [15], exactly as it is in the case of f2py.

## 4. Our Solution

This work contributes a two-way transpiler operating on subsets of Fortran 77/90/95 and Python 3 that is able to handle:

- fundamental types, basic syntax, selected array operations,
- some idiomatic statements (command-line printing, basic file I/O),
- internal API calls (selected Fortran intrinsics, Python built-ins and stdlib functions) and
- external APIs (MPI to a limited extent).

5. Cython documentation – Faster code via static typing: http://docs.cython.org/src/quickstart/cythonize.html

We also contribute a workflow design for migrating legacy Fortran applications to Python without sacrificing their performance. The workflow consists of 3 main steps:

1) transpilation of legacy Fortran to Python already annotated with type hints – because type information is available in Fortran;
2) performance-critical functions in the resulting Python code have to be manually marked using decorators (a standard Python language feature);
3) decorated kernels are translated at runtime to Fortran, compiled, interfaced with Python using f2py and executed instead of their Python counterparts.

Workflow also supports boosting performance of any Python code as long as it is translatable to Fortran.

The workflow does not aim at full automation when it comes to translating Fortran to Python, because with a complete legacy application translation in mind the only feasible aim can be significant simplification of migration process – with some manual work still required.

Moreover, workflow leverages existing Python tools as much as possible to decrease functional overlap.

Workflow is designed with two main use cases in mind.

### 4.1. Use Case 1

User has high-performing Fortran 77/90/95 source code that she wants to migrate to Python, possibly change some things, and still be able to run it with equivalent efficiency as the original Fortran implementation. At the same time, after migrating to Python, user wants for her application to remain in Python.

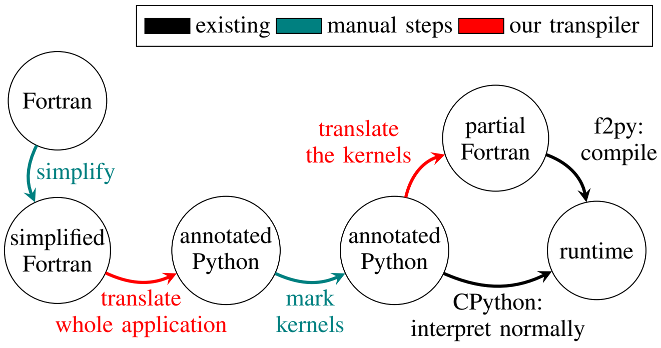

Figure 1. Translation of Fortran to Python creates Fortran-like (i.e. compatible with static type system) Python code.

The process is as follows:

- Fortran source code is translated to Python 3.5 code automatically augmented with type annotations and type comments.
- User can annotate selected functions of resulting Python code with a special decorator.
- This decorator, at runtime, triggers an automatic translation of Python code into Fortran, compilation of Fortran code, creation of Python-Fortran interface,

and substitution of original Python function with that interface.
- Whenever a function is executed, the call is forwarded to wrapped Fortran function and return value, if any, is forwarded from Fortran to Python.

User thus benefits from high performance of Fortran while maintaining all of her code in Python.

### 4.2. Use Case 2

User has a computational application written Python, and wishes to enhance its performance while keeping the source code in Python.

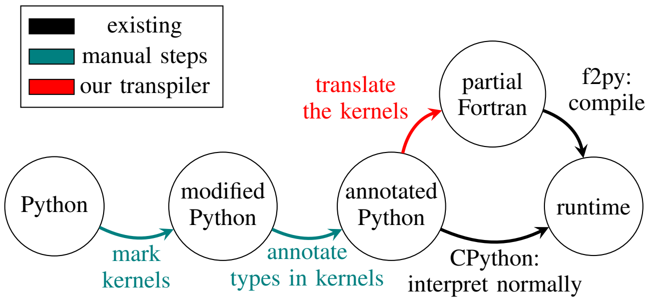

Figure 2. Performance of Fortran-like Python code can be enhanced by leveraging the similarities between subsets of Python and Fortran.

The process is as follows:

- User can extract the computing kernels into separate functions – this is most probably done already.
- User can annotate selected kernel functions with a special decorator.
- Additionally, user can provide type information for those functions using Python type hints – describing function argument types and return type (both with type annotations) and types of local variables (with type comments).
- The decorator, at runtime, triggers the same process as described in the last 2 steps of Use Case 1.

Again: user benefits from high performance of Fortran while maintaining all of her code in Python.

## 5. Implementation

### 5.1. Mapping Between Fortran and Python

Languages like Python and Fortran, although very different, have some common syntax. Not exhaustive list includes: numeric types such as integer and floating point numbers, arrays, integer-indexed for loops, binary operator for exponentiation, branching statements and routines that can receive data by reference. It follows that some parts of a given Python or Fortran program's source code might look very similar to each other.

Translation is straightforward when a 1-to-1 mapping exists between syntax elements of each language. Prominent examples of such syntax would be: simple mathematical expressions, boolean formulas, some assignments, comments, integer-indexed for loops, while loops, etc. These syntactic structures are very simple and vary relatively little across both programming languages.

Non-straightforward translation occurs when there is no exact mapping and the relationship is more complex. Still, there are many such structures which occur commonly. File operations, printing to command-line, etc. In the scientific context some cases of array indexing, array memory layout and usage of various APIs, e.g. MPI, are not so trivial to translate.

**5.1.1. Fundamental types.** Translating type names between programming languages effectively requires a 1-to-1 predefined mapping, and all types not present in the mapping simply cannot be translated. Moreover, in case of certain types, only approximate translation is available. For example, Python's `str` can be mapped only approximately, by assuming some reasonable upper bound on its length.

```
character*1024 :: s = 'hello'
s = 'hello' # type: str
```

**5.1.2. Basic syntax.** Among things trivially translated are many of the binary operators like addition, subtraction, multiplication, division and exponentiation as well as boolean operators.

With static type information available, Python's true division operator can be reliably translated. The same applies to translating to Python: the truncating behaviour of Fortran's division in cases when dividend is integer and Fortran's string concatenation operator.

Translation complexity of while loops depends on the complexity of the expression of loop exit condition, and the syntactic overhead of the while loop itself is nearly nonexistent.

The for loop handling is currently limited to those with with integer index variable. It would also be possible to do a straightforward translation of Python sequence enumeration – in cases where one can depend on sequence having a measurable length that does not change at runtime.

Python's usage of unnamed entities can be directly reflected in Fortran in many cases, however in case of boolean operators applied on arrays there is the same problem as in the case of intrinsic functions.

```
A .ne. 0 ! logical array
nonzero = count(A .ne. 0)
A[A != 0] # sub-array
nonzero = len(A[A != 0])
```

The variable name must be duplicated in case of translating to Python, and the duplicate must be detected and eliminated in case of translating to Fortran. Moreover, since the actual result type of such operations is different, they are translatable only in specific context of use as arguments for intrinsic functions.

**5.1.3. Array operations.** Translation of array access is surprisingly easy, because NumPy arrays and Fortran arrays are addressed very similarly. Arrays in Fortran are indexed from 1 by default, but with care the array section access and assignment are translatable.

Since assignment in Python works differently in case of immutable and mutable objects, its translation depends on the type of translated object and array assignment has to reflect that.

Also, some array operations look entirely different in Fortran and Python. Translation of those operations must sometimes be done on case-by-case basis.

Function call and array element access is in some cases be indistinguishable from one another in Fortran without use of name resolution. The heuristic that we propose in our approach is to assume that a name is a call to a function unless it is found to be a declared variable. Since variable declarations are necessary to be given first, the call/array distinction can be done by the parser provided that the variable declarations are cached and accessible when parsing subsequent statements.

**5.1.4. Idiomatic statements.** The assignment translation becomes complex if value of assigned variable cannot be easily copied.

Fortran's variable declarations have no direct translation in Python because in Python variables are never declared. As such, the translator from Fortran to Python must convert all Fortran variable declarations to assignments, but translator from Python to Fortran must generate extra statements at the beginning of the function after all local variables from the function and their types are known. This is not a straightforward 1-to-1 translation, because in case of two-way translation, the translator cannot know for sure if a given assignment originated as an assignment, or as a variable declaration. Such simplistic approach might create superfluous assignments at the beginning of the function body with each two-way translation iteration. It is, however, not a problem for the intended workflow.

The Fortran's `implicit none` statement has no direct translation in Python. The heuristic we propose is to assume that all generated Fortran functions start with this statement. In perspective, the information about presence or lack of any idiomatic statements can be embedded in Python source code in specially-formatted comments, which are only activated and expressed if a function is transpiled to a language matching the comment's format.

Python's `import` and Fortran's `include` or `use` statements require special treatment not because of their syntax, which is rather simple, but because they add functionality to the code, as explained above in an MPI API example. This added functionality depends completely on the content hiding behind the path/name that is included/imported.

There are many ways to introduce the exact same functionality, and sometimes the mere inclusion of some functionality has side effects. For example, when importing mpi4py in Python, in some cases the `MPI_Init()` is executed automatically, while in other cases it is not.

A heuristic that we propose in our approach is to disallow non-canonical inclusion statements. This ensures generation of a reliable translation for cases it is possible according to the implementation, and signaling a problem for other cases.

**5.1.5. Internal API calls.** Some Fortran's intrinsics have a corresponding Python's stdlib function, others are a combination of several functions. In general, they must translated on case-by-case basis. We provide mappings for a small subset of Fortran's intrinsic functions. Other functions are translated as-is. Such translation might seem useless until we consider that our aim is to support one-way whole-migration from Fortran so that only the kernels are to be translated back to Fortran. In this context, faithful translation functions for accessing environment variables or reading input data from files during application setup is not necessary, because almost always these are better expressed in object-oriented APIs which are available only in Python, and initialization has no effect on computational performance.

**5.1.6. External APIs: MPI.** We consider it as the most important external API to be handled by the transpiler, because it is sometimes used in computational kernels to synchronize progress of many processes, or to overlap communication with computation. After analysis of conventions followed by native Fortran API compared with Python's object-oriented mpi4py, we determined what transformations need to be applied to many commonly used functions.

**5.1.7. Unsupported Python features.** `class` keyword and any concepts related to classes, instantiation etc. are not supported. That is simply because those concepts have limited use in the context of high-performing numerical kernels.

`with` statement is not supported.

`async` keyword is not supported.

Dynamic type change is what occurs in Python when a variable that initially had some type is assigned a value of a different, incompatible type. Such behaviour is by definition illegal in a statically typed language.

The Python's dynamic behaviour cannot be expressed in Fortran directly, but variable renaming, approach used by compilers in optimization, can be used in deterministic assignment cases to resolve the problem. If, from a control flow graph, we determine that a subsequent assignment invalidates the variable for all control flow paths, we can safely create a renamed duplicate and from that point on use the renamed duplicate instead of the original variable. Such approach will not work in all cases, and therefore in our solution we propose to forbid dynamic retyping – i.e. we assume that once a value of specific type is given to a variable, the variable retains that type through its lifetime.

**5.1.8. Unsupported Fortran features.** Fortran's indication of variable memory length via `kind` attribute, as well as `kind`-related intrinsic functions support is missing. Still, most of the time there is a very straightforward workaround for this. Specifically, instead of: `integer(kind=8)` one can use `integer*8` or `integer(8)`.

Fortran 90 and later supports arrays with assumed shape – meaning that the sizes of dimensions of an input array do not have to be predefined in the subroutine. Support for this is missing in the current implementation of the translator.

It is possible to transpile two or more Python functions from the same module to Fortran, and in theory they could as well call each other without any issues. However, in the current implementation, each transpiled function is compiled to a separate shared library object, and therefore currently all translated kernels have to be separate computational entities – one may not call any other.

We do not support Fortran's n-dimensional assignment expression, however it can be reexpressed using equivalent multi-level `do` loops before translation:

```
forall(i=1:ni,j=1:nj) B(i, j) = i * j
```

We currently do not support pragmas for compiler extensions such as OpenMP and OpenACC:

```
!$acc kernels
!$omp parallel
do ...
```

## 5.2. Technologies Used

We developed a Python 3 package that provides all aforementioned features. To parse code and store Python's AST we use a recent `typed_ast`[6] package. To generate code from it we use `typed_astunparse`[7] package. For migrating Fortran, we use our own transpiler implementation and a custom designed AST. We transform between our AST and Python's AST as necessary.

## 6. Case Study 1: DGEMM

To test the transpiler and the workflow's use cases, we did several case studies that highlight the characteristics of the approach, and characteristics of the current transpiler implementation. For all our experiments, we are using the machine with the following hardware and software specification:

| | |
|---|---|
| Operating system | Linux, Ubuntu 14.04.1 x64 |
| Linux kernel | 4.2.0-41 |
| Python | 3.5.1 |

Table 1. SYSTEM OF THE EVALUATION ENVIRONMENT.

| | |
|---|---|
| ATLAS | 3.10.1-4 |
| LAPACK | 3.5.0-1 |
| numpy | 1.11.1 |
| llvmlite | 0.12.0 |
| numba | 0.26.0 |
| mpi4py | 2.0.0 |

Table 2. VERSIONS OF PACKAGES.

In our first case study, we assume that user has a C-like matrix-matrix multiplication implementation in Python, and wants to accelerate it. The starting code is as follows:

6. https://pypi.python.org/pypi/typed-ast
7. https://pypi.python.org/pypi/typed-astunparse

```
def my_matmul(a, b, a_width, a_height, b_width):
    c = [0 for _ in range(b_width * a_height)]
    for y in range(a_height):
        for i in range(a_width):
            for x in range(b_width):
                c[y * b_width + x] += \
                    a[y * a_width + i] * b[i * b_width + x]
    return c
```

### 6.1. Currently Available Solutions

What approaches can she use to boost performance, and what results do they yield? We compare NumPy [10], Numba [12], f2py [11] and our framework.

**6.1.1. NumPy.** This is the most obvious solution in this particular scenario. User can abandon her code and simply use a routine provided by NumPy. This, however, requires restructuring the data. After reshaping the arrays, the multiplication is simply `a @ b`.

This solution, however, is inapplicable in case that the user would like to change the function even a little bit.

**6.1.2. Numba.** Instead of using NumPy like this, user can change her data format from plain Python lists to NumPy arrays. This has several benefits: for example NumPy arrays are more compact (i.e. they use less memory) and there are many convenient and well-implemented (i.e. efficient) functions that one can use to operate on them.

After that, user can decorate her NumPy-enabled function with `@numba.jit` to get some performance boost though just-in-time compilation.

**6.1.3. f2py.** A very drastic alternative solution that could increase the performance boost is to manually translate all of the implementation to a Fortran 77 subroutine and create a Python interface for that subroutine using f2py tool provided with NumPy. The resulting interface will accept NumPy arrays as input and return a NumPy array.

```
      subroutine my_matmul(a, b, c, a_width, a_height,
     &    b_width)

      integer*4, parameter :: max_width = 200
      integer*4, parameter :: max_height = 200
      real*8, intent(in) :: a(max_width * max_height)
      real*8, intent(in) :: b(max_height * max_width)
      real*8, intent(out) :: c(max_height * max_height)
      integer*4, intent(in) :: a_width
      integer*4, intent(in) :: a_height
      integer*4, intent(in) :: b_width
      integer*4 :: y, i, x

      c = 0
      do y = 1, a_height
          do i = 1, a_width
              do x = 1, b_width
                  c((y - 1) * b_width + x) =
     &                c((y - 1) * b_width + x) +
     &                a((y - 1) * a_width + i) *
     &                b((i - 1) * b_width + x)
              end do
          end do
      end do
      return

      end subroutine my_matmul
```

The resulting code is much longer – it contains as many extra lines for variable and constant declarations, as the original Python function counted in total. Moreover, since unfortunately in Fortran 77 arrays cannot have dynamically defined sizes, we need to set a limit for matrix size.

This solution is non-trivial and requires additional knowledge about Fortran and f2py interface in order to create a function that can be later interfaced with Python so that it has, for example, the same signature.

Finally, this solution also requires boilerplate code that would compile the Fortran file, create the Python interface for it and import the interface in a desired place in Python.

**6.1.4. This work.** Finally, the user can opt to use our work, which requires her to decorate the kernel, and annotate variable types.

```python
@reexpress('Fortran77')
def my_matmul(
        a: np.ndarray((200 * 200,), dtype=np.double),
        b: np.ndarray((200 * 200,), dtype=np.double),
        a_width: np.int32, a_height: np.int32,
        b_width: np.int32
        ) -> np.ndarray((200 * 200,), dtype=np.double):
    c = np.zeros(b_width * a_height, dtype=np.double)
    for y in range(a_height): # type: np.int32
        for i in range(a_width): # type: np.int32
            for x in range(b_width): # type: np.int32
                c[y * b_width + x] += \
                    a[y * a_width + i] * b[i * b_width + x]
    return c
```

Type of $c$ is implied by the fact that it is returned.

Since the code is going to be translated to Fortran 77, the same constraints apply to it as to target language. For example, arrays need to have predefined sizes, and choice of the right size can be non-obvious. In this specific case, the maximum array length was set to 40000.

## 6.2. Compilation Overhead Analysis

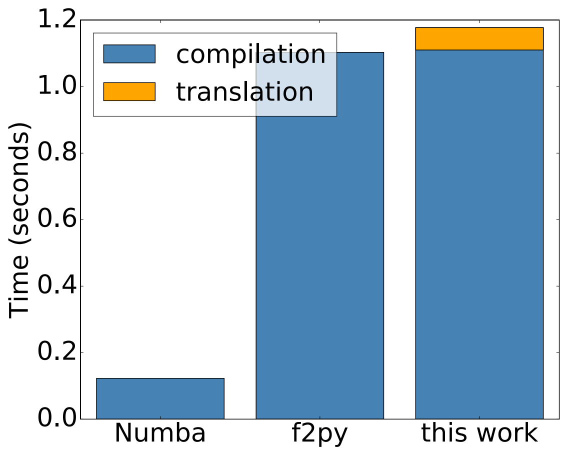

Figure 3. Compilation and/or translation overhead comparison for existing approaches and this work. Numba JIT-compiles completely in memory, whereas f2py and we (because we use f2py as part of the workflow) use intermediate files and create a shared library file.

In traditional computing, compiling cost is paid with each source code update, therefore under ordinary circumstances it doesn't count towards time measurements. On the other hand, in interactive computing, if the whole application were to be compiled with each small change of the code, it would pose a significant problem. However, when the source code is expected to change rapidly, compilation cost may be mitigated by modularizing the application and re-compiling only the necessary parts – approach of Numba and this work.

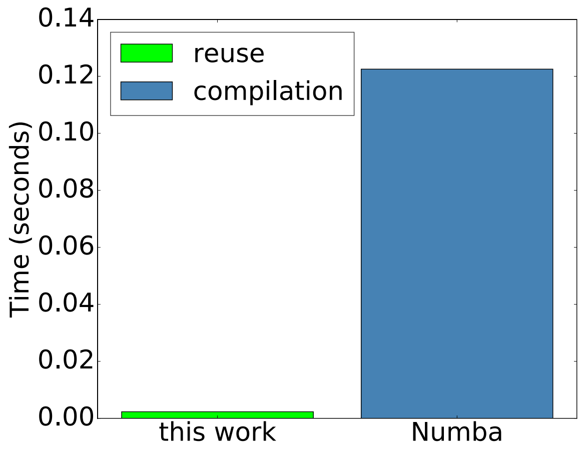

Figure 4. Binary object reuse in our framework and Numba's compilation time. Numba needs to recompile with each application launch, while we can simply load the compiled file.

**6.2.1. Numba.** The JIT compiler Numba compiles the function completely in memory. Although code is translated to LLVM IR, and then to machine code, in-memory approach yields short compilation times – as seen in Figure 3.

Compilation is initiated at first call to the decorated function, so if in a given application run the function is not called, there is no compilation overhead. After the initial call that includes the JIT compilation, the binary object is reused without any visible overhead for subsequent call.

The binary is not stored on disk, so recompilation is needed for each application launch – as seen in Figure 4.

**6.2.2. f2py and this work.** On the other hand, f2py (and thus, this work) uses the file system, creating Fortran source code file even when it is given a string of characters via its Python interface, additional files related to building the Fortran library and some more for building a Python extension module containing the Fortran routine. Then, it launches the Python toolchain to create the library. All this happens behind the scenes in a temporary folder which is discarded at the end of the process with a single exception - the Python extension module that is copied to the destination directory. All this file-juggling takes considerable time, even for a very simple Fortran routine – as seen in Figure 3.

The advantage of this approach is that the compiled binary object can be reused even in-between application launches – as seen in Figure 4.

## 6.3. Computational Performance Analysis

Let us compare the computational performance of each DGEMM implementation. Apart from the manual Fortran reimplementation of the routine, and its complete scraping in favour of matrix multiplication routine provided by NumPy, remaining solutions do not differ that much from the original code. Despite that, their performance varies enormously.

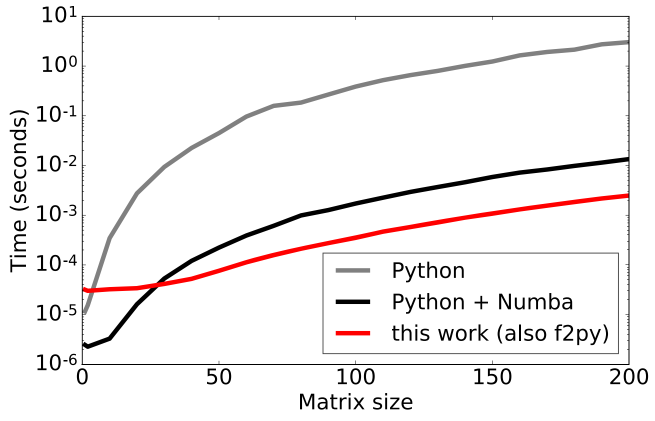

Figure 5. Computational performance of pure Python while using our framework is the same as launching Fortran implementation through f2py.

Overall, interpreted implementation is unsurprisingly the slowest. Let us consider it as a baseline and go down according to running time for matrix size 200.

Numba achieves 230× improvement over the baseline implementation.

User will see another factor 5 improvement after reimplementing it all in Fortran, or over 1000× improvement over the initial version. The same boost is also achieved by automatic translation provided by our transpiler, with which the user didn't have to abandon her Python implementation.

Finally, not shown in the Figure, but achieving factor 2.5 speedup over the simple Fortran code and nearly factor 2900 over initial code, is NumPy. This solution, however, is using a different algorithm entirely, because NumPy behind the scenes delegates matrix multiplication to a high-performing BLAS library. In principle, however, after changing the GEMM algorithm to a better one, the performance difference should be smaller or non-existent.

## 7. Case Study 2: Miranda IO

Miranda IO[8] is a parallel file system benchmarking application developed in Lawrence Livermore National Laboratory (LLNL). It is written in pure Fortran, and the latest intrinsic functions it uses were introduced in Fortran 95.

The benchmark that the application performs is as follows. Miranda performs 100 iterations, and in every iteration it makes very heavy reads and writes and validates that the data was stored and retrieved correctly. It also uses MPI, although mainly for synchronizing the processes so that reads and writes occur concurrently, and to broadcast initialization data from the master process to all processes.

8. Scalable I/O Benchmark Downloads: https://computing.llnl.gov/?set=code&page=sio_downloads

Before migrating to Python, we have refactored the code in order to simplify some Fortran constructs currently unsupported by the transpiler. Additionally, the resulting Python code was not entirely correct. The numerical parts of code were translated correctly, however the orchestrating code had to be adjusted for conformance with Python standard library, specifically: the environment variable access code had to be adjusted and the I/O filename generating formulas were changed. Also, the computational kernel had to be extracted to a separate function.

### 7.1. Translation Overhead Analysis

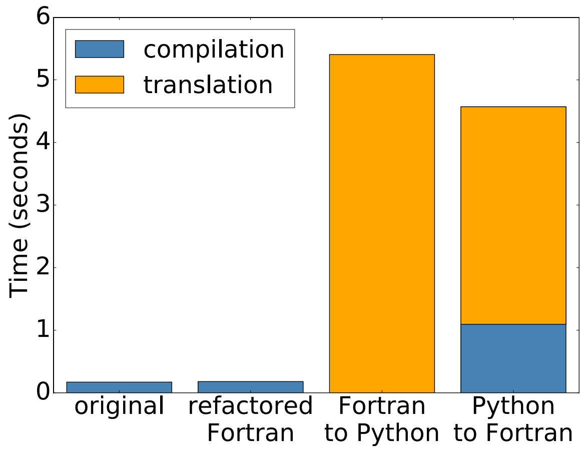

Figure 6. Binary object reuse overhead for approaches that support it – Numba and *any-lang*.

Translation from Python to Fortran is fully automatic. Necessity of manual source code adjustments specifically so that it can work in Fortran, but not anymore in Python, would defeat the purpose of maintaining the code in Python and making it runnable from within Python.

Figure 6 indicates that even if for small functions translation efficiency doesn't matter, for larger translations performance improvements are necessary before the framework becomes useful in a highly interactive environment, where the kernel is supposed to change frequently.

If computing kernel is less likely to change, the cost of translation is lowered by binary object reuse. In case of MPI applications, like Miranda IO, the kernel needs to be transpiled only once per computing node, and then all MPI processes on that node can reuse it.

### 7.2. Computational Performance Analysis

We measured computational performance at 4 stages of migration process:

- original – stands for original Miranda 1.0.1 code;
- refactored – is the refactored Miranda 1.0.1 code;
- python – stands for Python version that was first automatically generated from the refactored Fortran,

and then refactored manually to make up for details missed by current translator implementation; and

- this work – stands for Python code with benchmark kernel annotated with transpilation decorator.

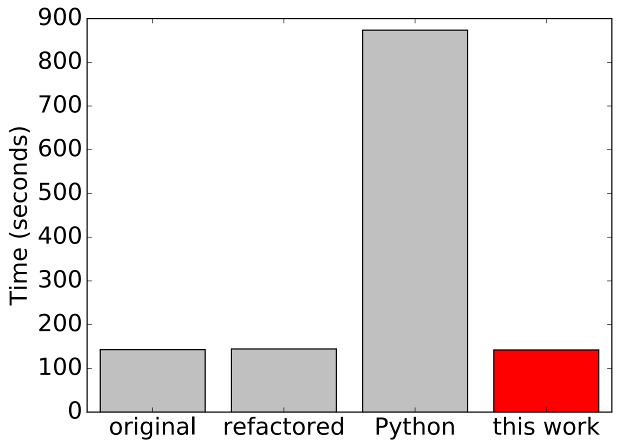

Figure 7. Our framwork completely recovers from computational performance drop of Python version of Miranda IO while maintaining the code in Python.

Original Miranda 1.0.1 code and the refactored version exhibit identical performance, illustrating that rewriting Fortran code without the modern `forall` syntax did not incur any performance penalties.

Code auto-translated to Python version and then refactored displays factor of 6 slowdown. This is a relatively good result for Python code. Although Miranda IO is an I/O benchmarking application, in does not consist entirely of reads and writes. To prepare non-trivial data to write and verify the data that was read, certain amount of calculations is necessary. Those calculations are the primary reason for the slowdown.

## 8. Conclusion

We described a workflow design for migration of legacy Fortran code and acceleration of Python code satisfying certain criteria. The key idea is that Python code can be efficiently compiled if the code is annotated with type hints and written as if the language was static – and the transpiled legacy Fortran code automatically meets these requirements.

We have implemented two-way transpiler that achieves tolerable translation overhead (mitigated by reusability of binary objects between launches) and maximizes the computational speed-ups. Overall, we show that maintainability, extensibility and interoperability can be improved without sacrificing performance.

We have evaluated the work on compute-intensive and I/O-intensive cases. We demonstrated that performance of DGEMM written in Python equals that of Fortran: there is no computational overhead from the framework; Python code can be as fast as Fortran when it is compiled to well-optimized machine code.

Also, we showed that benchmark written in Fortran retains original performance after migration to Python: two-way translation approach is not only feasible, but also useful.

Our final thought is that type hints are not only a static analysis tool, but can also be used as a reliable source of information for runtime performance optimization.

## Acknowledgments

This research was supported by JST, CREST (Research Area: Advanced Core Technologies for Big Data Integration).

## References

[1] K. Bennett, “Legacy systems: coping with success,” *IEEE Software*, vol. 12, no. 1, pp. 19–23, Jan 1995.

[2] F. G. Tinetti and M. Méndez, “Fortran legacy software: Source code update and possible parallelisation issues,” *SIGPLAN Fortran Forum*, vol. 31, no. 1, pp. 5–22, Mar. 2012. [Online]. Available: http://doi.acm.org/10.1145/2179280.2179281

[3] G. M. Amdahl, “Validity of the single processor approach to achieving large scale computing capabilities,” in *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, ser. AFIPS ’67 (Spring). New York, NY, USA: ACM, 1967, pp. 483–485. [Online]. Available: http://doi.acm.org/10.1145/1465482.1465560

[4] G. van Rossum and I. Levkivskyi. (2014, Dec.) Pep 483 – the theory of type hints. [Online]. Available: https://www.python.org/dev/peps/pep-0483/

[5] G. van Rossum, J. Lehtosalo, and . Langa. (2014, Sep.) Pep 484 – type hints. [Online]. Available: https://www.python.org/dev/peps/pep-0484/

[6] S. I. Feldman, “A fortran to c converter,” in *ACM SIGPLAN Fortran Forum*, vol. 9, no. 2. ACM, 1990, pp. 21–22.

[7] R. W. Grosse-Kunstleve, T. C. Terwilliger, N. K. Sauter, and P. D. Adams, “Automatic fortran to c++ conversion with fable,” *Source code for biology and medicine*, vol. 7, no. 1, p. 1, 2012.

[8] K. Seymour and J. Dongarra, “Automatic translation of fortran to jvm bytecode,” *Concurrency and Computation: Practice and Experience*, vol. 15, no. 3-5, pp. 207–222, 2003.

[9] A. Corrigan, F. Camelli, R. Löhner, and F. Mut, “Semi-automatic porting of a large-scale fortran cfd code to gpus,” *International Journal for Numerical Methods in Fluids*, vol. 69, no. 2, pp. 314–331, 2012.

[10] T. E. Oliphant, *A guide to NumPy*. Trelgol Publishing USA, 2006, vol. 1.

[11] P. Peterson, “F2py: a tool for connecting fortran and python programs,” *International Journal of Computational Science and Engineering*, vol. 4, no. 4, pp. 296–305, 2009. [Online]. Available: http://cens.ioc.ee/~pearu/papers/IJCSE4.4_Paper_8.pdf

[12] S. K. Lam, A. Pitrou, and S. Seibert, “Numba: A llvm-based python jit compiler,” in *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, ser. LLVM ’15. New York, NY, USA: ACM, 2015, pp. 7:1–7:6. [Online]. Available: http://doi.acm.org/10.1145/2833157.2833162

[13] S. Behnel, R. Bradshaw, D. Seljebotn, G. Ewing *et al.* Cython: C-extensions for python, 2008. [Online]. Available: http://cython.org/

[14] D. S. Seljebotn, “Fast numerical computations with cython,” in *Proceedings of the 8th Python in Science Conference*, vol. 37, 2009.

[15] S. Behnel, R. Bradshaw, C. Citro, L. Dalcin, D. S. Seljebotn, and K. Smith, “Cython: The best of both worlds,” *Computing in Science & Engineering*, vol. 13, no. 2, pp. 31–39, 2011. [Online]. Available: http://scitation.aip.org/content/aip/journal/cise/13/2/10.1109/MCSE.2010.118